INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

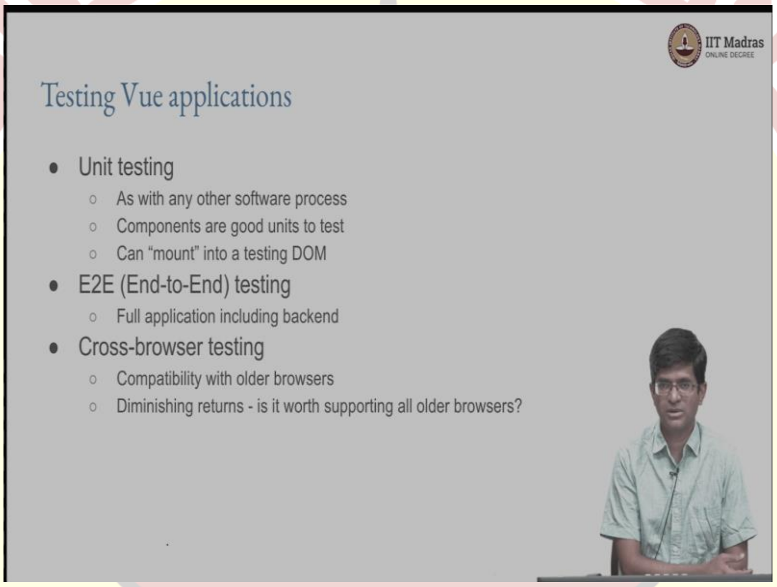
IIT Madras

ONLINE DEGREE

Modern Application Development - II
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Vue - Testing

Hello, everyone, welcome to Modern Application Development Part 2. Now, finally, let us look at one topic, which is very important, and has been stressed upon multiple times throughout this course at different stages, which is testing. How do you test? What a given component can or cannot do? For example.

(Refer Slide Time: 0:32)



Testing Vue applications

- Unit testing
  - As with any other software process
  - Components are good units to test
  - Can "mount" into a testing DOM
- E2E (End-to-End) testing
  - Full application including backend
- Cross-browser testing
  - Compatibility with older browsers
  - Diminishing returns - is it worth supporting all older browsers?

Now, testing view applications can be done in a few different ways. There is, of course, the unit testing, which should be done just as with any other software process. And if you think about it, a component in view, by definition, more or less is probably a good thing level at which to unit test.

But, the question is, how do you test a component, because after all, components are heavily visually oriented, meaning that they are modifying the DOM, they are actually inserting elements into the DOM, they are displaying error messages, they are, you know, certain things get seen only under certain conditions, and so on.

But the interesting thing is the test mechanisms, in view allow you to sort of mount the entire component into, what is called a testing DOM. It is a virtual DOM, Document Object Model. Where you can inject certain kinds of data into the system and get different kinds of behaviour associated with it. We will take a look at that not actually running it in the code, but you know, we will take a look at that briefly, how that would look.

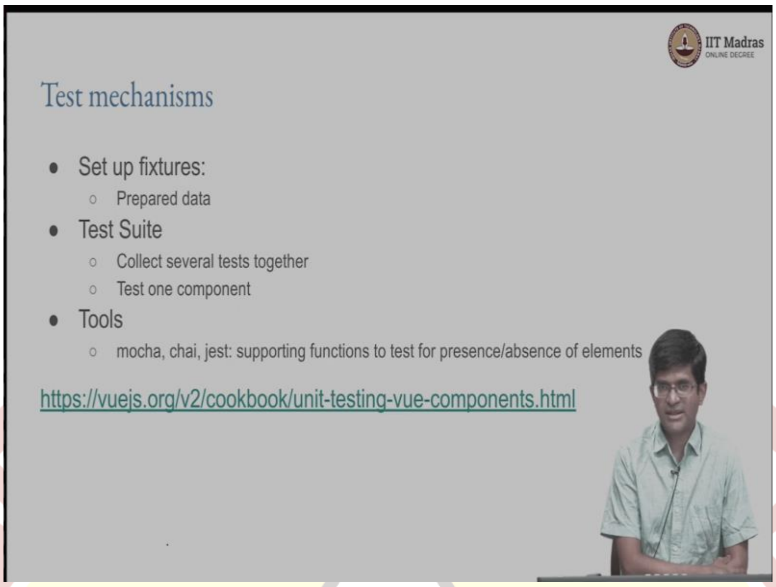
Now, in addition to the unit testing there is also so-called end-to-end testing, where you actually need to test the full application, including the backend and how it behaves under certain cases. This is similar to what you would do with a flask application, you need to be able to create a dummy test database, you need to be able to create requests, you need to see how the system behaves under practical use cases.

Now, there is also one more part which actually needs to be tested, which is since view deals with the frontend, you also probably need to test for cross-browser behaviour. Does it work only with Chrome? Does it work only with Firefox? Does it require a specific version of Chrome or Firefox or Edge or Safari in order to work? How do you sort of handle that?

Now, at some point, this sort of gets you into a stage of diminishing returns, because maintaining full backwards compatibility with all older browsers is nearly an impossible task. So, you need to take a call at some stage. You have the option of either saying, this programme; this app will only work with certain versions of certain browsers. Or you could at least use some kind of, you know, a compiler like Babel to resolve those difficulties to some extent, and you know, push the problem there.

As long as Babel is able to generate code for different systems, they should work reasonably well. It does raise the question about whether the Babel compiler itself is completely accurate. In other words, let us say you write a specific kind of code, and the Babel compiler translates it into something else. Are they both doing exactly what you had originally intended? Okay. But it is that is true of any compiler. So, it is a call that you have to take and decide, is this the approach that you want, in terms of achieving your final compatibility requirements.

(Refer Slide Time: 3:32)



Test mechanisms

- Set up fixtures:
  - Prepared data
- Test Suite
  - Collect several tests together
  - Test one component
- Tools
  - mocha, chai, jest: supporting functions to test for presence/absence of elements

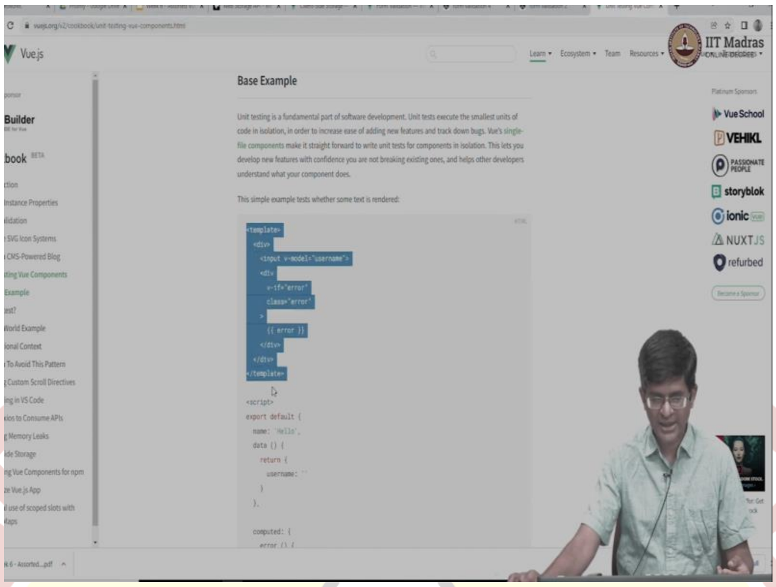
https://vuejs.org/v2/cookbook/unit-testing-vue-components.html

So, test mechanisms in view, are just like the test mechanisms that we would have, for example, in Python unit test, you can set up fixtures with prepared data, you can have a test suite that collects several tests together. And maybe perhaps these are several tests that, in turn are testing one particular component.

Now, the test utilities from view also provide interfaces to a number of tools. So, there are tools such as Mocha, Chai, jest, and so on, which are supporting functions, they basically allow you to check for the presence or absence of certain elements and to write the tests themselves in a way that are almost self-documenting, and easy to understand.

So finally, let us take a look at some examples of what unit testing view components would be like. But we are not really going to run through these, because in order to actually run these examples, you need to have the entire, you know, CLI and everything else set up, which at least for this present video, I am not going to cover those aspects of it.

(Refer Slide Time: 04:44)

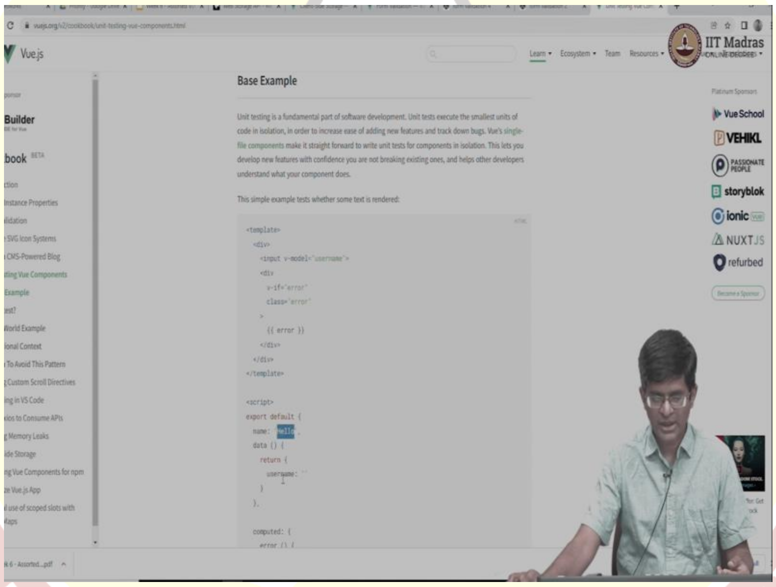


Base Example

Unit testing is a fundamental part of software development. Unit tests execute the smallest units of code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

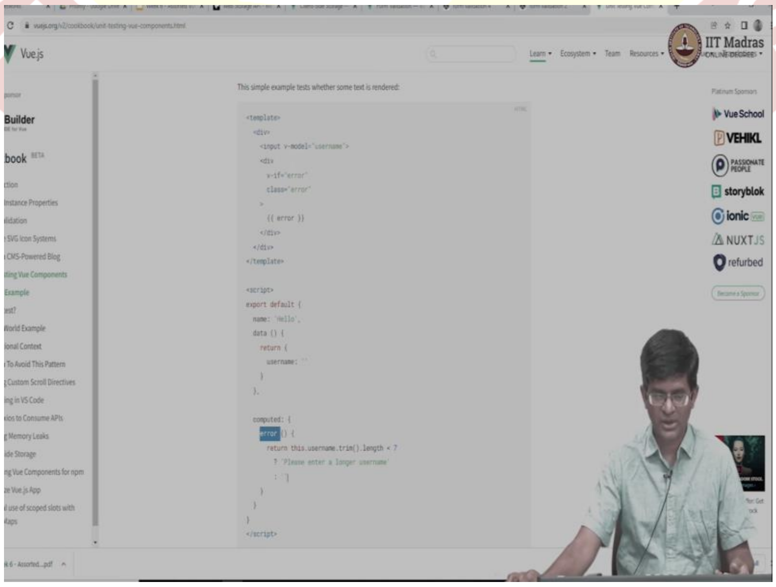
```html
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
```



Base Example

Unit testing is a fundamental part of software development. Unit tests execute the smallest units of code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:



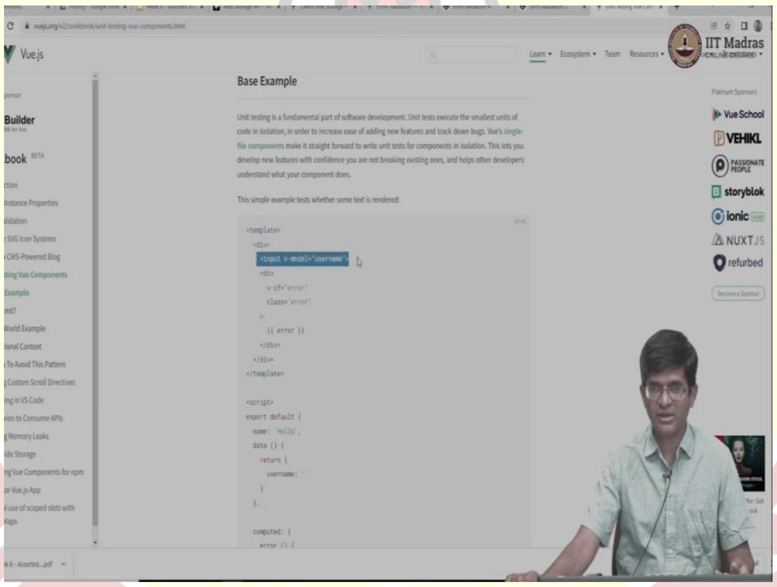
This simple example tests whether some text is rendered:

```html
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

So, once again, going back to the cookbook, you know, there is quite a lot of detail on what unit testing view components looks like. Now, since I am not actually going to run this code. What I want to focus on over here is the concepts, how would you actually go about testing a component that is going to get displayed on a screen.

And, of course, you know, the component itself looks something like this. It is a very trivial thing. It is basically a hello component. And what it says is, there is an error, which is a computed function, which says that, unless the user name is at least seven characters long, it is going to basically set this error parameter to be true. And when not true, it is going to either have this or an empty string. Now, in the code itself of the component, it basically says that if error is present, it sets the class error and displays this message. And that is pretty much all that it does.

(Refer Slide Time: 05:50)



Base Example

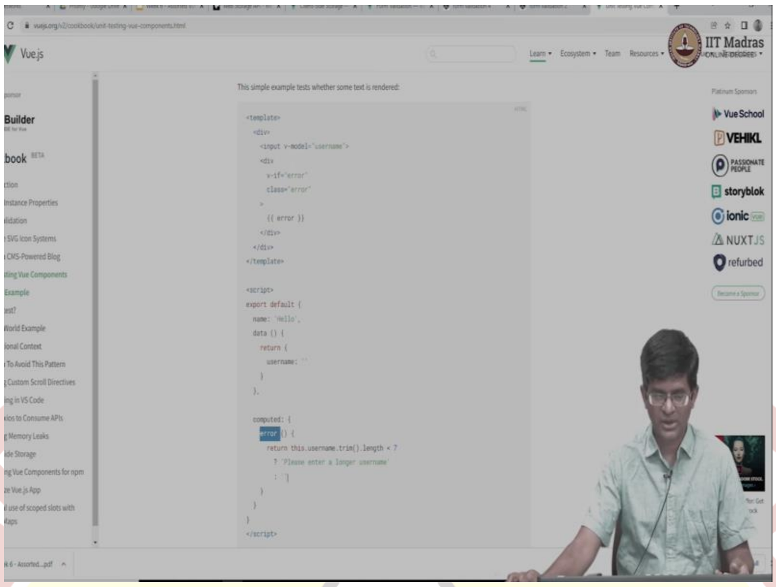
Unit testing is a fundamental part of software development. Unit tests execute the smallest units of code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
```

It has something for the input, user name. And if the user name does not match the requirements, because after all, this is computed, which means that any time that user name changes, after we have got a two-way binding, because of this V model.

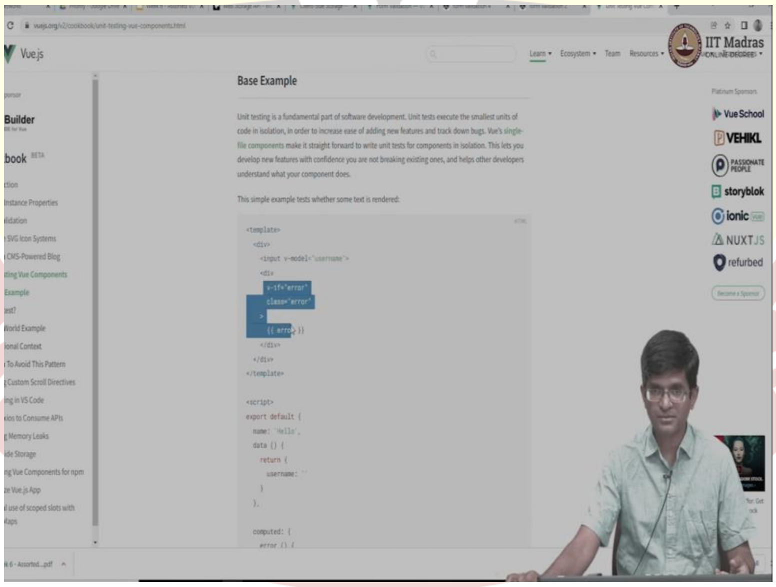
(Refer Slide Time: 06:11)


This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

The moment I type any character, change user name, I will find that this computed error value changes. And until I have crossed 7 characters, it will basically say, please enter a longer user name.

(Refer Slide Time: 06:18)


Base Example

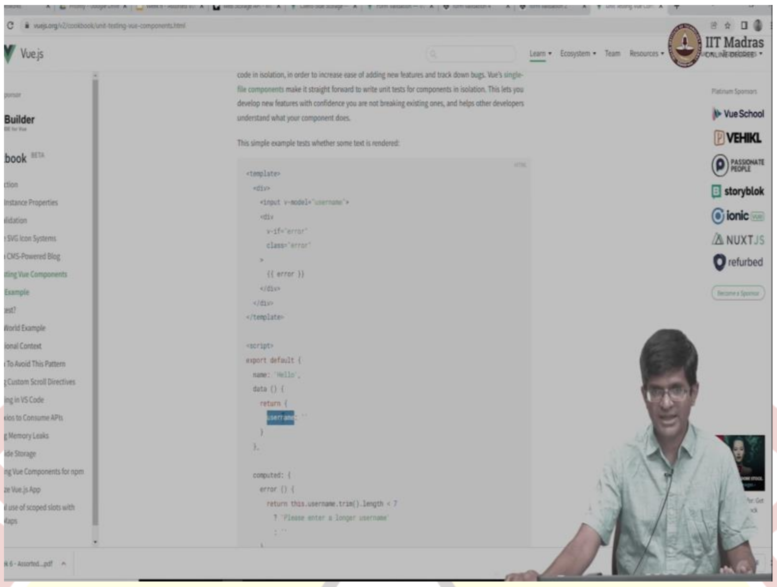
Unit testing is a fundamental part of software development. Unit tests execute the smallest units of code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
```

And that message will show about here. Now, how would I test something like this? What we could do is, ultimately, when I am testing in practice, what I would do is, I would probably bring up the page, look at what it displays. It should have an input component. So far, so good. That looks like a test; at least the component should have an input.

(Refer Slide Time: 06:45)

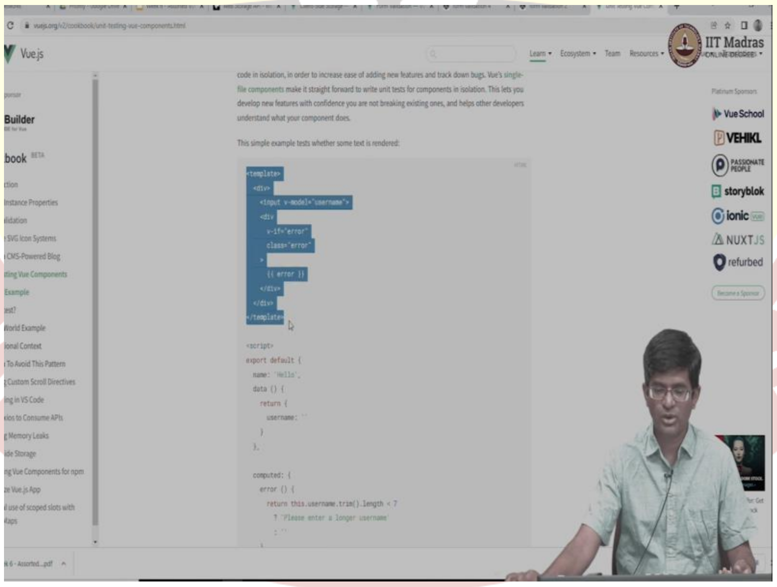

code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
```

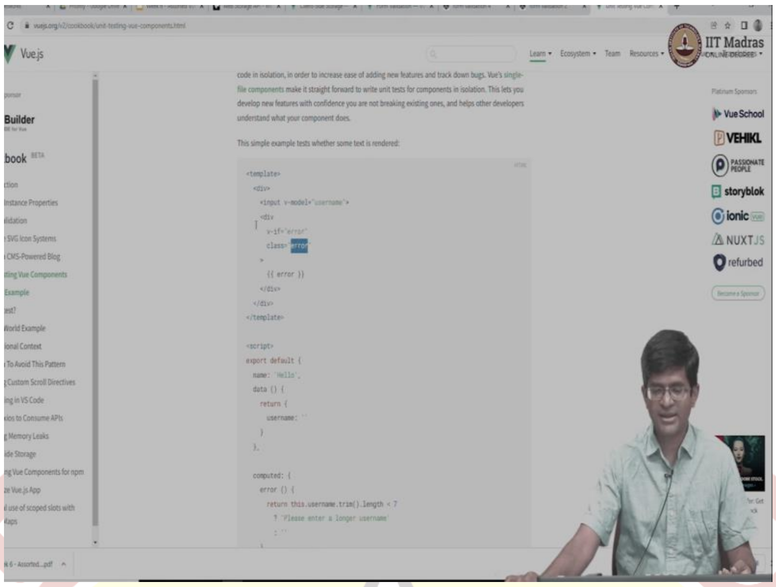
But in addition to that, there is also this thing of, what am I going to do with the variables that are present in the JavaScript? And the tests that I could have over there, could. For example, be what happens is, if the username is empty. What do I expect the output to look like?

(Refer Slide Time: 06:55)


code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

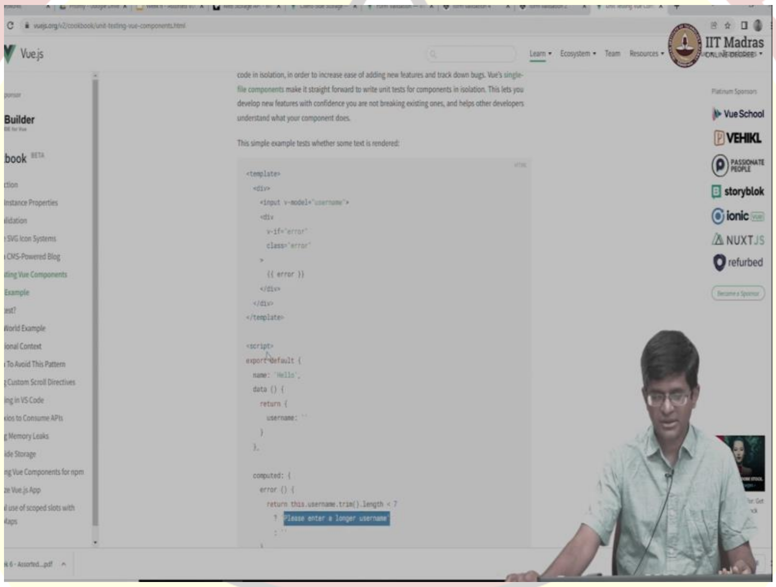
```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
```



code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
```



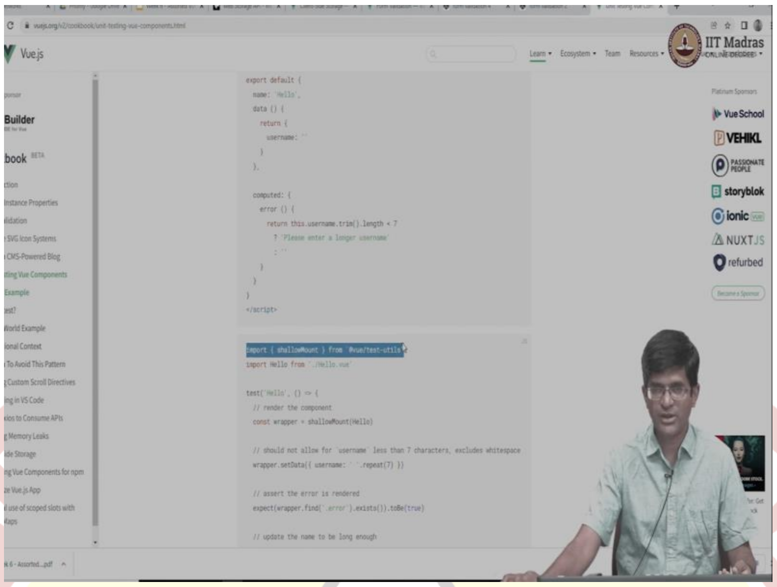
code in isolation, in order to increase ease of adding new features and track down bugs. Vue's single-file components make it straight forward to write unit tests for components in isolation. This lets you develop new features with confidence you are not breaking existing ones, and helps other developers understand what your component does.

This simple example tests whether some text is rendered:

```
<template>
  <div>
    <input v-model="username">
    <div
      v-if="error"
      class="error"
    >
      {{ error }}
    </div>
  </div>
</template>

<script>
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
```

I should see that there is a div with error in it, I could also say, maybe I need to look for the error message, which is, you know, please enter a longer username. But I have to choose I mean, you know, probably does not make sense to look for both. Maybe if I just look for whether this class error is present on a dev, somewhere in this module, that is good enough actually.

(Refer Slide Time: 07:26)

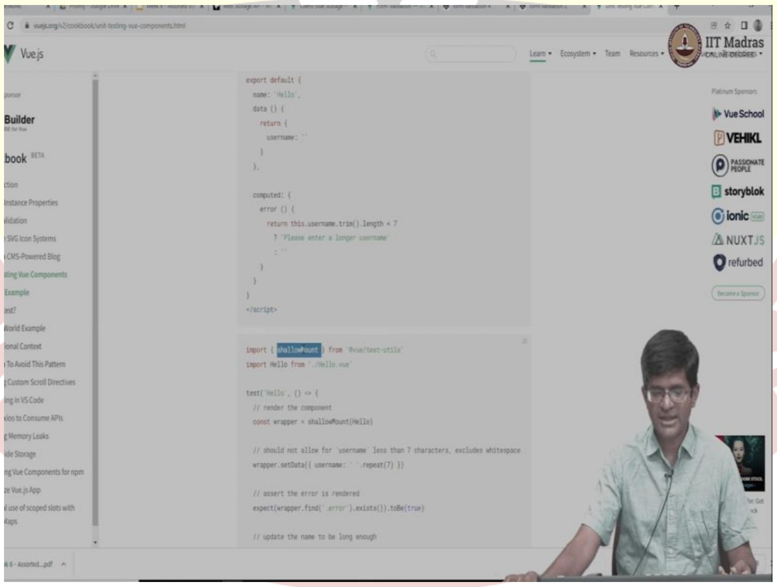


```
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
```

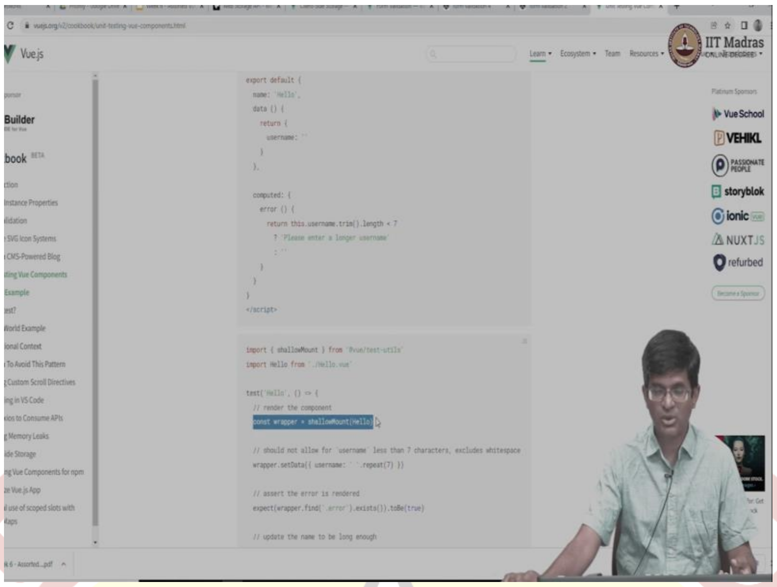
So, let us look at what the test utility looks like. It imports something called Shallow Mount from view test utils. So, the view test utils has, what is called the official view-testing library. It provides a number of different functions that allow you to write tests for view components.

(Refer Slide Time: 07:41)



```
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
```



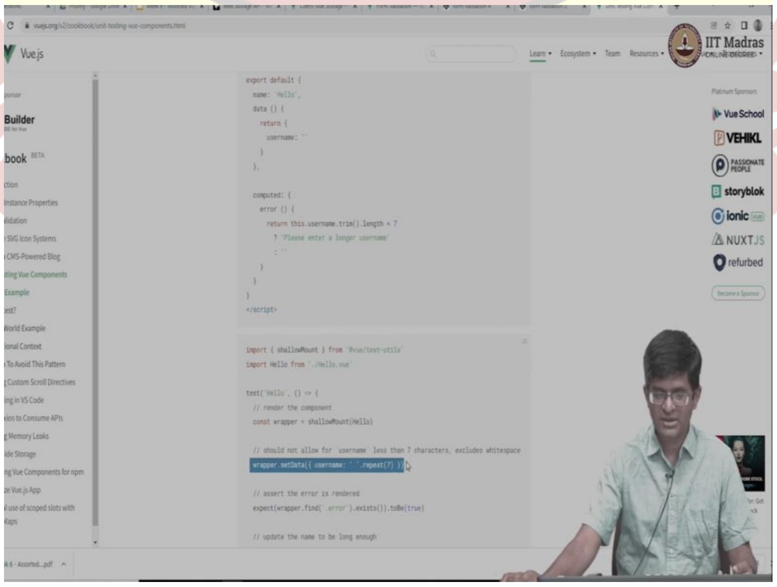
```
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
```

Shallow mount is something, which sort of emulates the process of mounting a component within actual DOM. So, it is effectively in other words, what happens when I do this construct equals shallow mount of Hello, is I have created a kind of DOM, where this component Hello has been mounted.

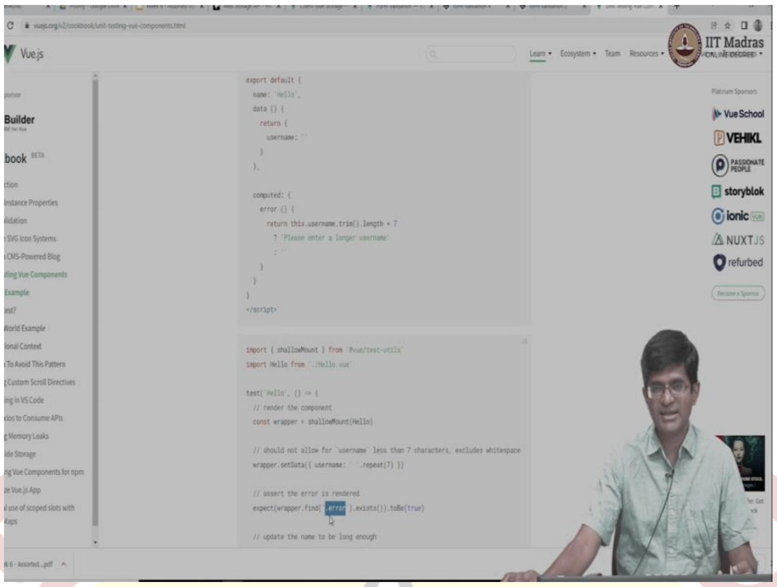
Now, I can do things like I should be able to set the data, user name, to some value over here. And what happens is, when I wrapper dot setData user name, and I am setting it, you know, I am basically giving it a name, which is seven characters, but those characters are spaces, which is basically going to be rejected.

(Refer Slide Time: 08:30)



```
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
```



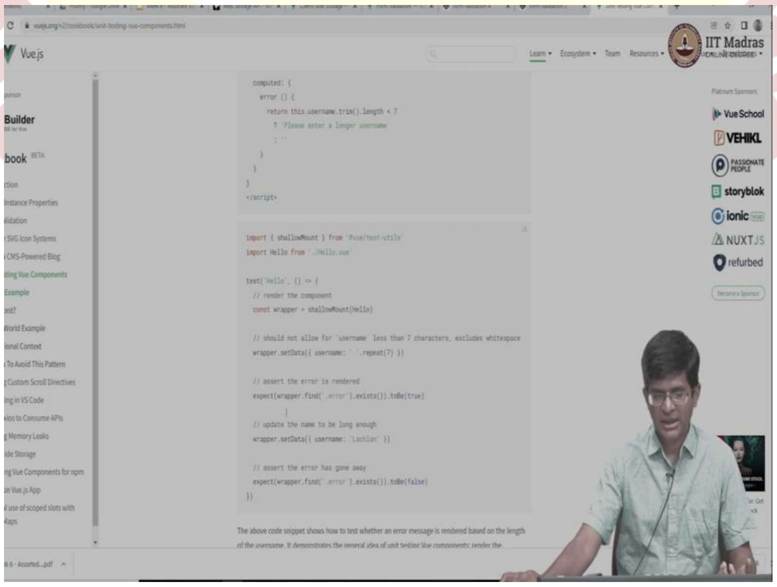
```
export default {
  name: 'Hello',
  data () {
    return {
      username: ''
    }
  },

  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
```

So, what happens is that when I set wrapper to these seven spaces, I should find that I should expect that inside the wrapper, I should be able to find one element, at least with the dot error class associated with it. So, look at the way that the test is written expect wrapper dot find dot error exists. And the whole thing to be true, so, expects this to be true.

Now, this kind of behaviour is sort of, what you will find in a number of different libraries that support testing. They are helping you to write your test in a way that looks reasonably easy to read. So, when I read this line, I can sort of figure out what it is trying to check, expect that, you know, inside the wrapper, I should find dot error, it should exist. And I expect that to be true.

(Refer Slide Time: 09:28)



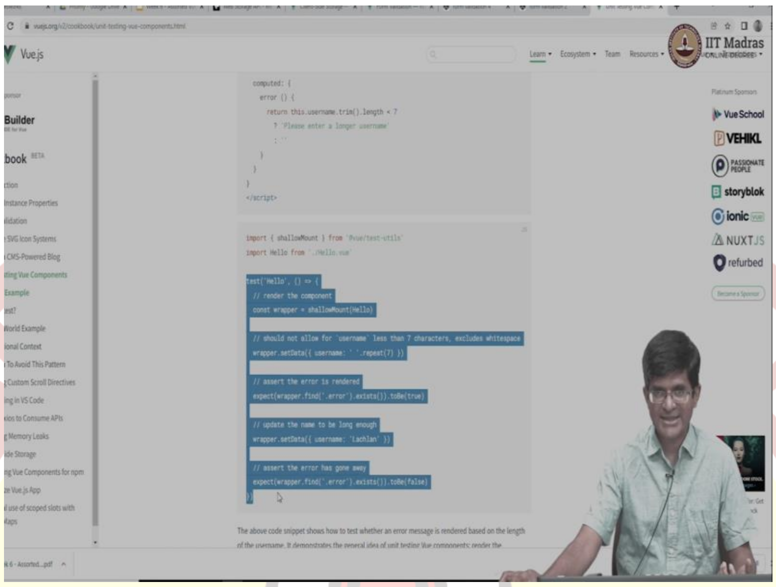
```
  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
  wrapper.setData({ username: 'Lachlan' })

  // assert the error has gone away
  expect(wrapper.find('.error').exists()).toBe(false)
})
```

The above code snippet shows how to test whether an error message is rendered based on the length of the username. It demonstrates the general idea of unit testing Vue components: render the

Now, you can go further and basically say, you know, now what happens, change the data, to an actual seven-character name. And at this point, I should find that the dot error is false.

(Refer Slide Time: 09:43)



```
  computed: {
    error () {
      return this.username.trim().length < 7
        ? 'Please enter a longer username'
        : ''
    }
  }
}
</script>
```

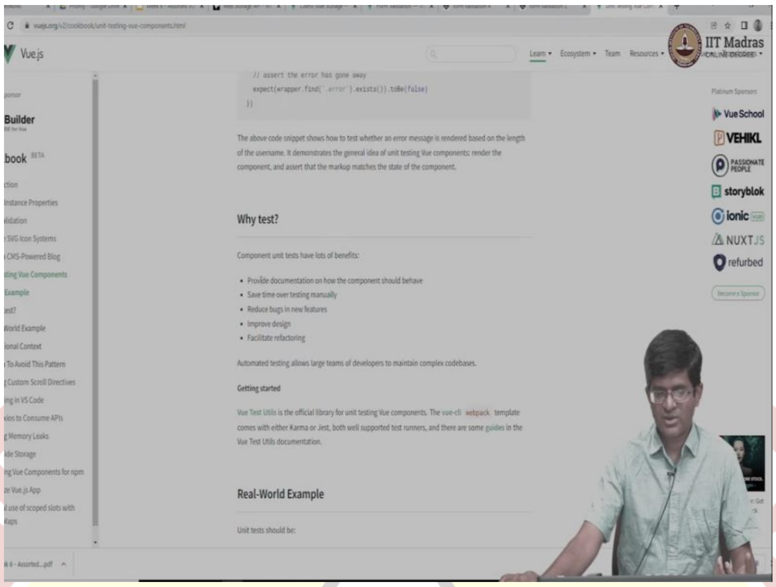
```
import { shallowMount } from '@vue/test-utils'
import Hello from './Hello.vue'

test('Hello', () => {
  // render the component
  const wrapper = shallowMount(Hello)

  // should not allow for `username` less than 7 characters, excludes whitespace
  wrapper.setData({ username: ' '.repeat(7) })

  // assert the error is rendered
  expect(wrapper.find('.error').exists()).toBe(true)

  // update the name to be long enough
  wrapper.setData({ username: 'Lachlan' })

  // assert the error has gone away
  expect(wrapper.find('.error').exists()).toBe(false)
})
```

The above code snippet shows how to test whether an error message is rendered based on the length of the username. It demonstrates the general idea of unit testing Vue components: render the

So, if you think about it, you know, what is happening is this is a single test, because it is after all defined as a single test out here. But, it is testing two things, it is all it is testing whether the, you know, when the name is less than 7 characters, is it turning out to be true? And if it is more than 7, more than or equal to 7 characters does it go away? The point is that writing such a test becomes, at least, you know, now you can sort of understand, how you might go about testing it.

So, let us say you are writing a weather display component, you should be able to inject a certain set of cities into it, and sort of see what the display looks like; does it actually have the cards corresponding to each of those? You know, call an update function on one of those and then see what happens. Does it actually get the data? What happens when you call it with an invalid city name? All of those things could be automated those test processes.

(Refer Slide Time: 10:40)



```
  // assert the error has gone away
  expect(wrapper.find('.error').exists()).toBe(false)
})
```

The above code snippet shows how to test whether an error message is rendered based on the length of the username. It demonstrates the general idea of unit testing Vue components: render the component, and assert that the markup matches the state of the component.

Why test?

Component unit tests have lots of benefits:

- Provide documentation on how the component should behave
- Save time over testing manually
- Reduce bugs in new features
- Improve design
- Facilitate refactoring

Automated testing allows large teams of developers to maintain complex codebases.

Getting started

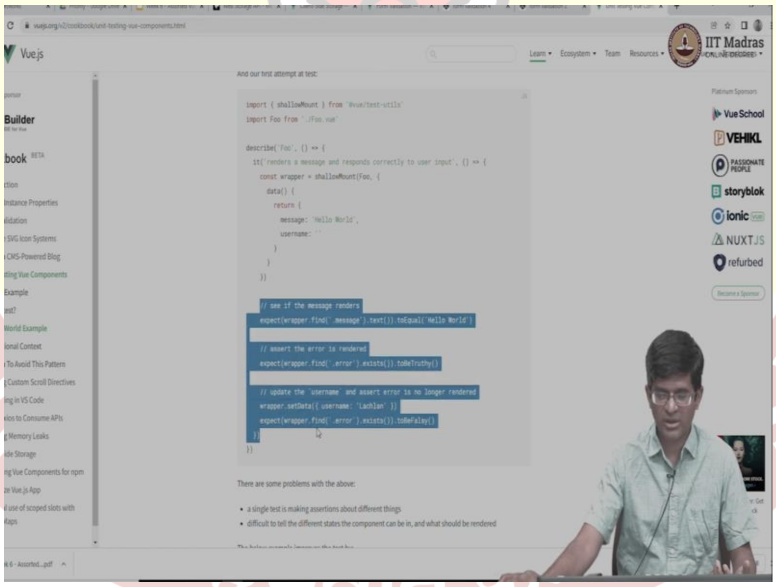
Vue Test Utils is the official library for unit testing Vue components. The vue-cli webpack template comes with either Karma or Jest, both well supported test runners, and there are some guides in the Vue Test Utils documentation.

Real-World Example

Unit tests should be:

And, you know, this page also says, why test, which hopefully, at this point, you know, most of you are convinced that testing is a good thing. So, I am not going to go into that anymore.

(Refer Slide Time: 11:01)



And our first attempt at test:

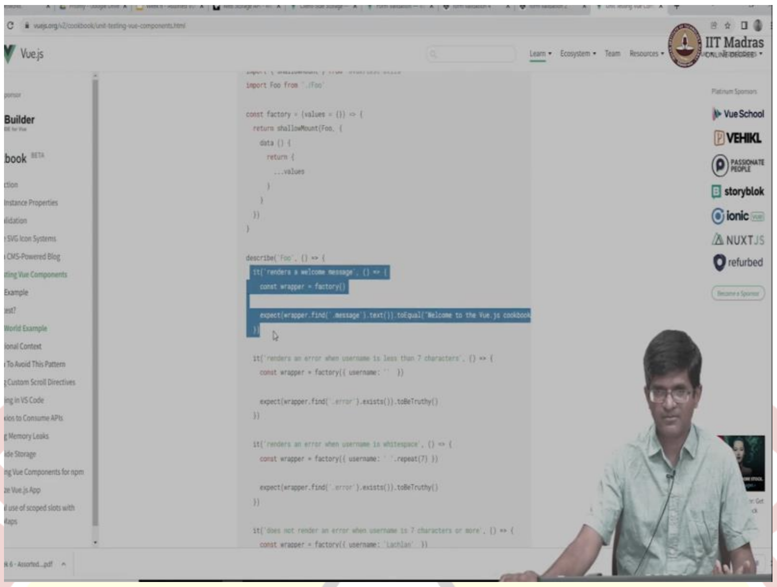
```
import { shallowMount } from '@vue/test-utils'
import Foo from './Foo.vue'

describe('Foo', () => {
  it('renders a message and responds correctly to user input', () => {
    const wrapper = shallowMount(Foo, {
      data() {
        return {
          message: 'Hello World',
          username: ''
        }
      }
    })

    // see if the message renders
    expect(wrapper.find('.message').text()).toEqual('Hello World')

    // assert the error is rendered
    expect(wrapper.find('.error').exists()).toBeTruthy()

    // update the `username` and assert error is no longer rendered
    wrapper.setData({ username: 'Lachlan' })
    expect(wrapper.find('.error').exists()).toBeFalsy()
  })
})
```

There are some problems with the above:

- a single test is making assertions about different things
- difficult to tell the different states the component can be in, and what should be rendered

Now, of course, there is more that can be done with this, which is that the next example that they have basically says that, you know, you could have something, which has like multiple different tests put in one place over here.

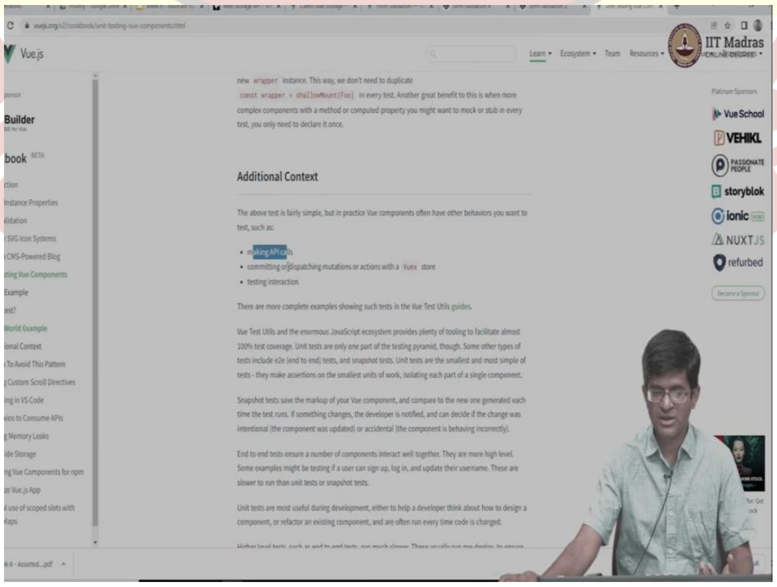
(Refer Slide Time: 11:09)



```
import Foo from './Foo'

const factory = (values = {}) => {
  return shallowMount(Foo, {
    data () {
      return {
        ...values
      }
    }
  })
}

describe('Foo', () => {
  it('renders a welcome message', () => {
    const wrapper = factory()

    expect(wrapper.find('.message').text()).toEqual('Welcome to the Vue.js cookbook
  })

  it('renders an error when username is less than 7 characters', () => {
    const wrapper = factory({ username: '' })

    expect(wrapper.find('.error').exists()).toBeTruthy()
  })

  it('renders an error when username is whitespace', () => {
    const wrapper = factory({ username: ' '.repeat(7) })

    expect(wrapper.find('.error').exists()).toBeTruthy()
  })

  it('does not render an error when username is 7 characters or more', () => {
    const wrapper = factory({ username: 'Lachlan' })
```

But a better way of doing it is actually to break it out into a suite of tests. So, you have one test, which is defined by it, and this other one, which is another it and other it, and so on. And this is, why this particular name it, and so on, it is just something, which supposedly helps with helps with readability. Maybe it does. It depends on what you find to be comfortable and easy to understand. But when you look at these things, one thing that is sure is, you know, if tests are written this way, it is fairly easy to sort of figure out what they are trying to do.

(Refer Slide Time: 11:45)



new `wrapper` instance. This way, we don't need to duplicate `const wrapper = shallowMount(Foo)` in every test. Another great benefit to this is when more complex components with a method or computed property you might want to mock or stub in every test, you only need to declare it once.

Additional Context

The above test is fairly simple, but in practice Vue components often have other behaviors you want to test, such as:

- making API calls
- committing or dispatching mutations or actions with a `Vuex` store
- testing interaction

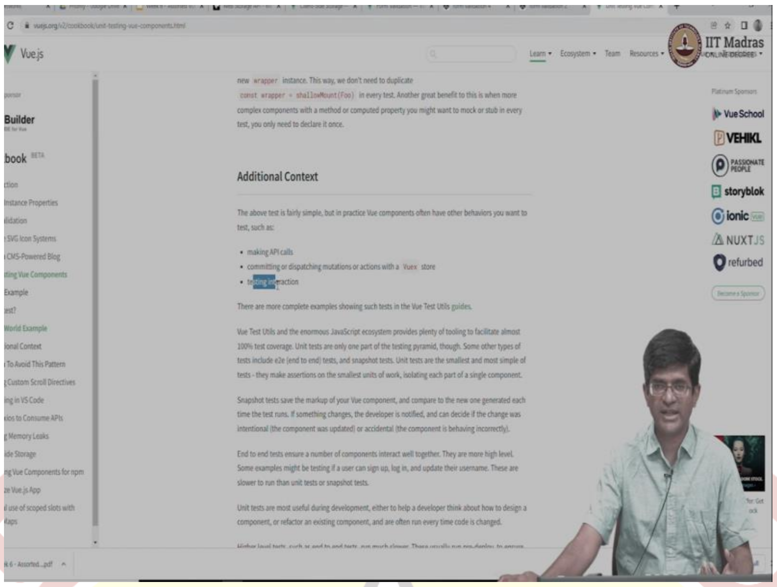
There are more complete examples showing such tests in the Vue Test Utils guides.

Vue Test Utils and the enormous JavaScript ecosystem provides plenty of tooling to facilitate almost 100% test coverage. Unit tests are only one part of the testing pyramid, though. Some other types of tests include e2e (end to end) tests, and snapshot tests. Unit tests are the smallest and most simple of tests - they make assertions on the smallest units of work, isolating each part of a single component.

Snapshot tests save the markup of your Vue component, and compare to the new one generated each time the test runs. If something changes, the developer is notified, and can decide if the change was intentional (the component was updated) or accidental (the component is behaving incorrectly).

End to end tests ensure a number of components interact well together. They are more high level. Some examples might be testing if a user can sign up, log in, and update their username. These are slower to run than unit tests or snapshot tests.

Unit tests are most useful during development, either to help a developer think about how to design a component, or refactor an existing component, and are often run every time code is changed.



new `wrapper` instance. This way, we don't need to duplicate `const wrapper = shallowMount(Foo)` in every test. Another great benefit to this is when more complex components with a method or computed property you might want to mock or stub in every test, you only need to declare it once.

Additional Context

The above test is fairly simple, but in practice Vue components often have other behaviors you want to test, such as:

- making API calls
- committing or dispatching mutations or actions with a `Vuex` store
- testing interaction

There are more complete examples showing such tests in the Vue Test Utils guides.

Vue Test Utils and the enormous JavaScript ecosystem provides plenty of tooling to facilitate almost 100% test coverage. Unit tests are only one part of the testing pyramid, though. Some other types of tests include e2e (end to end) tests, and snapshot tests. Unit tests are the smallest and most simple of tests - they make assertions on the smallest units of work, isolating each part of a single component.

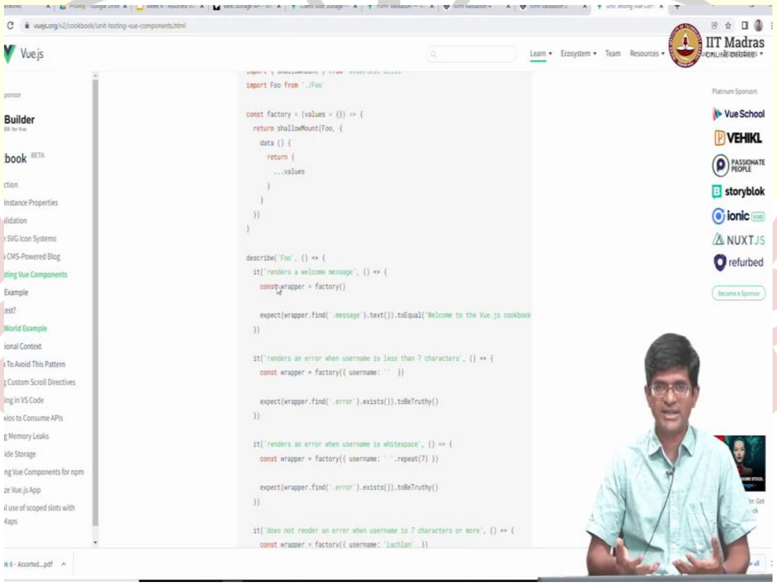
Snapshot tests save the markup of your Vue component, and compare to the new one generated each time the test runs. If something changes, the developer is notified, and can decide if the change was intentional (the component was updated) or accidental (the component is behaving incorrectly).

End to end tests ensure a number of components interact well together. They are more high level. Some examples might be testing if a user can sign up, log in, and update their username. These are slower to run than unit tests or snapshot tests.

Unit tests are most useful during development, either to help a developer think about how to design a component, or refactor an existing component, and are often run every time code is changed.

So, you go through this entire process. And you could go even further, you could even make API calls or you know, have like, further things that are done with vuex is not something that we have covered so far, but we will be looking at in one of the screencasts, at least later. Testing interaction, which means that you can actually generate clicks, and input events and various other things and different parts of the essentially the component.

(Refer Slide Time: 12:15)



```
import Foo from './Foo'

const factory = (values = {}) => {
  return shallowMount(Foo, {
    data () {
      return {
        ...values
      }
    }
  })
}

describe('Foo', () => {
  it('renders a welcome message', () => {
    const wrapper = factory()

    expect(wrapper.find('.message').text()).toEqual('Welcome to the Vue.js cookbook
  })

  it('renders an error when username is less than 7 characters', () => {
    const wrapper = factory({ username: '' })

    expect(wrapper.find('.error').exists()).toBeTruthy()
  })

  it('renders an error when username is whitespace', () => {
    const wrapper = factory({ username: ' '.repeat(7) })

    expect(wrapper.find('.error').exists()).toBeTruthy()
  })

  it('does not render an error when username is 7 characters or more', () => {
    const wrapper = factory({ username: 'Lachlan' })
```

So, like I said, in order to actually implement and demonstrate all of this, you need to have it with the CLI implemented. The reason why I have brought it up over here is, first thing to show that such a thing can be done within the view framework. Secondly, how would you go about testing? What are the things that you can test?

Typically, it would be, are certain elements present in the DOM? And the good thing is that, you know the test utilities allow you to set or change the values of certain JavaScript variables, and run tests corresponding to those different kinds of behaviours. It is also possible to construct more complicated tests that depend on multiple parts of the state of the system. And all of those ideally should be done. You need to have a good set of tests, so that you are reasonably comfortable that your component works in all the scenarios that you might have thought of.