INDIAN INSTITUTE OF TECHNOLOGY MADRAS

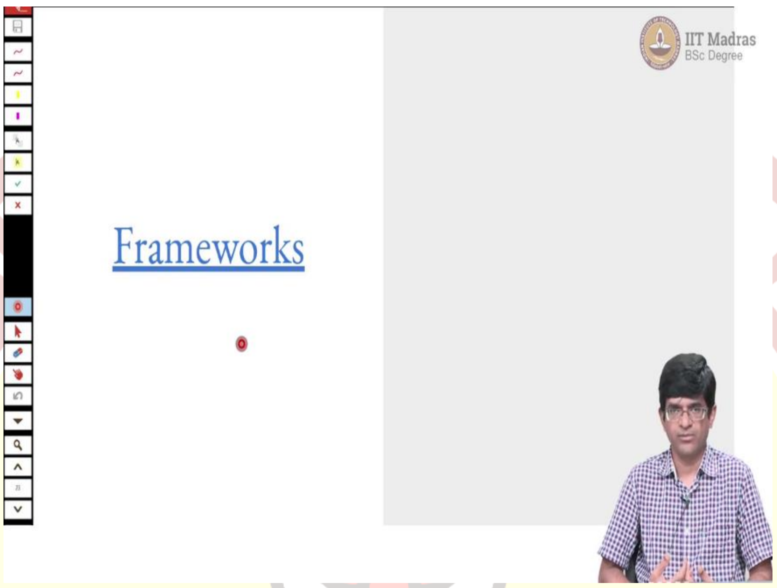
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Modern Application Development-I
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Frameworks

(Refer Slide Time: 00:16)



Frameworks

Hello, everyone, and welcome to this course on modern application development. So, with all of this web components and so on in place, it finally leads us to the topic of frameworks frontend frameworks as they are called.

So, the first thing that we would ask is what is the purpose of a framework? We are looking at some kind of a context where some kind of basic functionality is already available. So, the Python programming language can already, it has as part of its standard library, it has like network interfaces, it can create a function that will listen on a port. It can manipulate strings. So, it can append strings together, it can create new strings, it can even do some kind of basic templating and filling in text and so on.

Similarly, JavaScript has the functionality for extending elements. So, custom element is already defined and it has the DOM manipulation APIs. So, that sounds great. It means, that in principle, I can already create any kind of web-based application using Python or I can create any kind of web component using JavaScript.

The problem is, there will be a lot of code reputation and this word boilerplate is something that you would have probably come across in multiple places. Boilerplate basically means it is a standard repeating thing. It is the same thing that you are going to do multiple places. So, for example, you might find that anytime you are writing a new Python program, the first thing you need to do is import a bunch of things.

And to some extent, you might find that it is so repetitive that look why do not I just have sort of boilerplate starter code, which allows me to get started without having to type in all those inputs one by one. I know that I am always going to be importing the same things. So, that is boilerplate code repetition, we would like to avoid it if possible. It also means, that there will be a lot of reinventing of the wheel, because people have different coding styles, they have different coding techniques, and ultimately, that results in many different ways of creating components, using components, mixing them together, and so on.

So, the solution, that it is not unique to JavaScript or Python or anything else of that sort, it is a standard technique for you try and identify standard techniques for solving common problems, and this ultimately, is related to the concept of design patterns that I have been talking about repeatedly through the course. Model view controller was, in some sense, a form of design pattern.

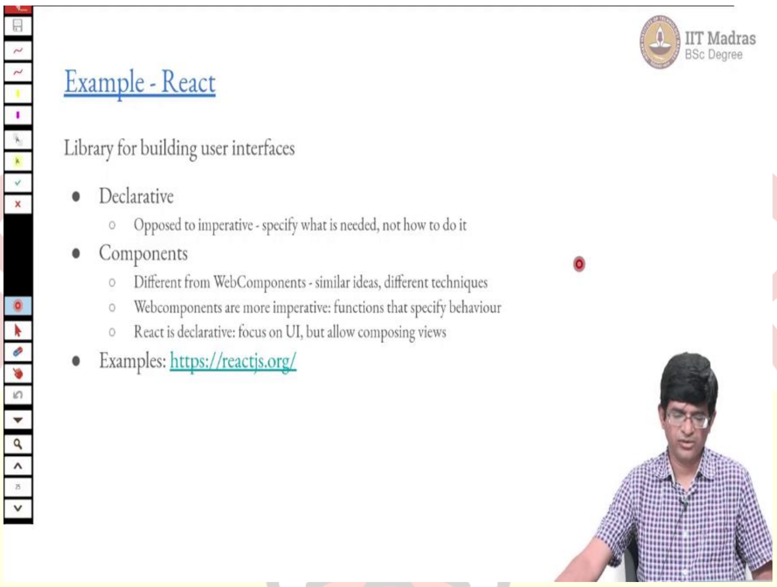
Ultimately, all that it comes about is there is experience that people have come up, come up with over years of trying to do things in a certain way, they try and distill that and take out the essence of it, and put it in a form that can then be reused by others. And when that is done nicely, it is really powerful and a lot of people tend to use such things. That is what a framework is. And flask, for example, is a framework for creating Python web applications. React is one of the frameworks that is available for creating JavaScript components.

Not exactly JavaScript web components. There is a difference between react components and JavaScript web components. Once again, it is partly different styles of doing things, but they are all trying to solve similar problems, react focuses only on the user interface, whereas, web components does a bit more slightly, including the state of the system and so on. There are some differences, there are some similarities as well. Once again, you can see that, it ultimately comes down to is it popular? Is it easy to use? People will use it.

Now, in the context of JavaScript a lot of the frameworks that have come up are built around this concept of single page applications. And many JavaScript frontend frameworks are focused on enabling this in different ways. Part of the reason for the popularity of single page applications is because the single page application effectively becomes an app by itself, which does not need like multiple different web pages that you need to navigate through.

All the sort of communication happens in the back with JavaScript, a lot of the computation may also happen with JavaScript, which means, that the user experience becomes a lot more smooth than if you need to keep on clicking on pages and waiting for a web page to load. And that was the focus of many of the JavaScript front ends, at least at least to start with.

(Refer Slide Time: 04:50)



IIT Madras
BSc Degree

Example - React

Library for building user interfaces

- Declarative
  - Opposed to imperative - specify what is needed, not how to do it
- Components
  - Different from WebComponents - similar ideas, different techniques
  - Webcomponents are more imperative: functions that specify behaviour
  - React is declarative: focus on UI, but allow composing views
- Examples: https://reactjs.org/

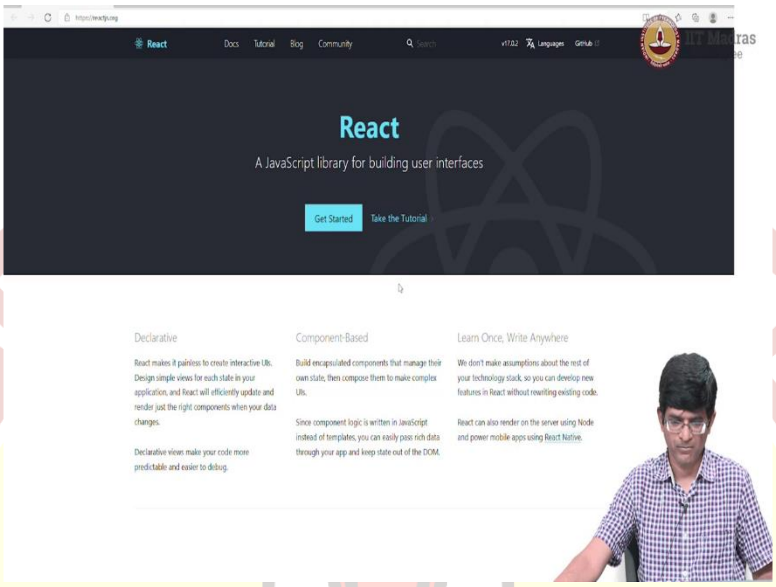
So, one example that I am just going to use in order to demonstrate a little bit about how you might use a framework is react. It is one of the most popular ones that are available today. But like I said this is I am sure there will be like very strong opinions in other direction saying that look not react, you should use angular or you should use view or you should use something else or all of these are a bad idea.

So, I do not want to get into the whether it is a good idea or a bad idea I am just saying this is one possibility. It is a library, which has its primary purpose as building user interfaces. It is declarative. Remember SGML and declarative syntax, once again, you try and specify what you want, rather than how to draw it on the screen. The how to draw also has to be specified, but the react, once you have defined a bunch of react components, it allows you to basically say what you want to create with them.

And the components themselves contain internal details of how to go about getting that result. They are different from web components, but somewhat similar ideas, different techniques for achieving sort of similar results. Web components tend to be more imperative, they actually attach the functionality that needs to be done directly as functions inside the component, whereas, react is more declarative, it focuses on composing a user interface by putting different components together.

Not a trivial thing to understand, but fairly straightforward when you get into, I mean, there are differences and both can be handled in different ways. Let us take a couple of examples of react, which is directly available from their main webpage itself.

(Refer Slide Time: 06:44)



React

A JavaScript library for building user interfaces

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

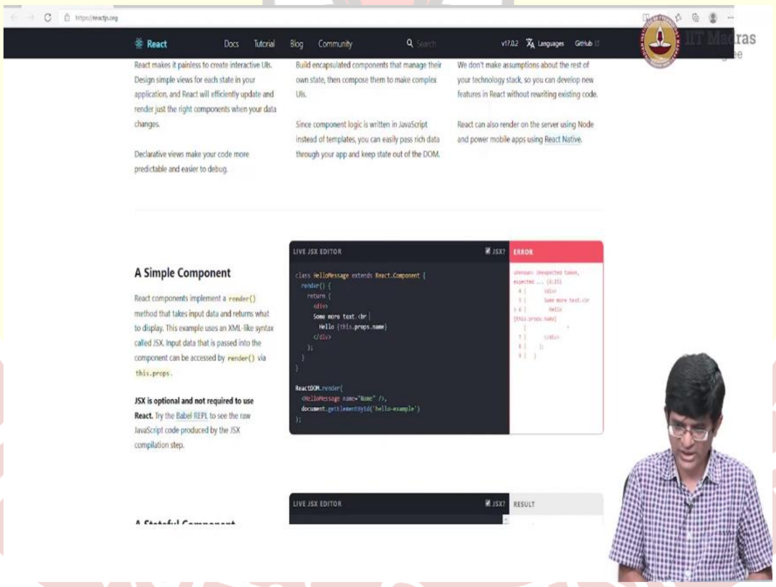
Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using React Native.



React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using React Native.

A Simple Component

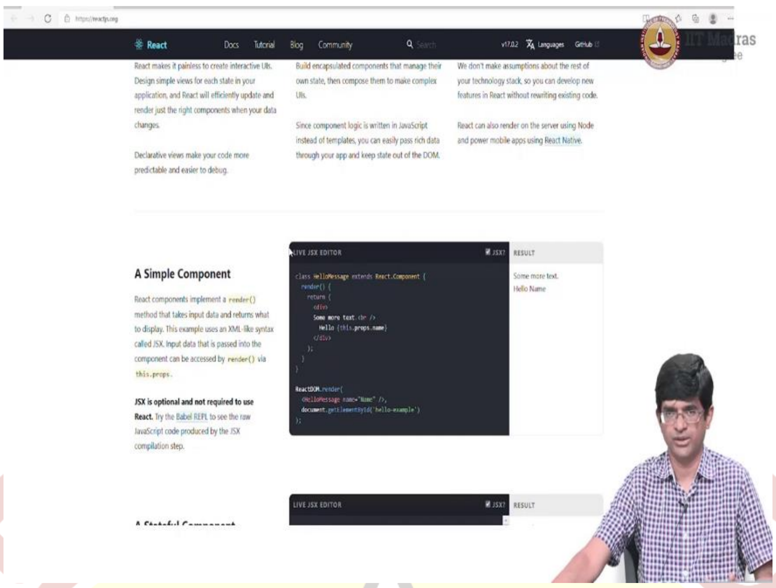
React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Some more text.<br
          Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Name" />,
  document.getElementById('hello-example')
);
```

```
unknown: Unexpected token, expected ... (4:15)
  4 |       <div>
  5 |         Some more text.<br
> 6 |           Hello
{this.props.name}
    |              ^
  7 |       </div>
  8 |     );
  9 |   }
```

A Stateful Component



React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using React Native.

A Simple Component

React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

LIVE JSX EDITOR ☑ JSX?

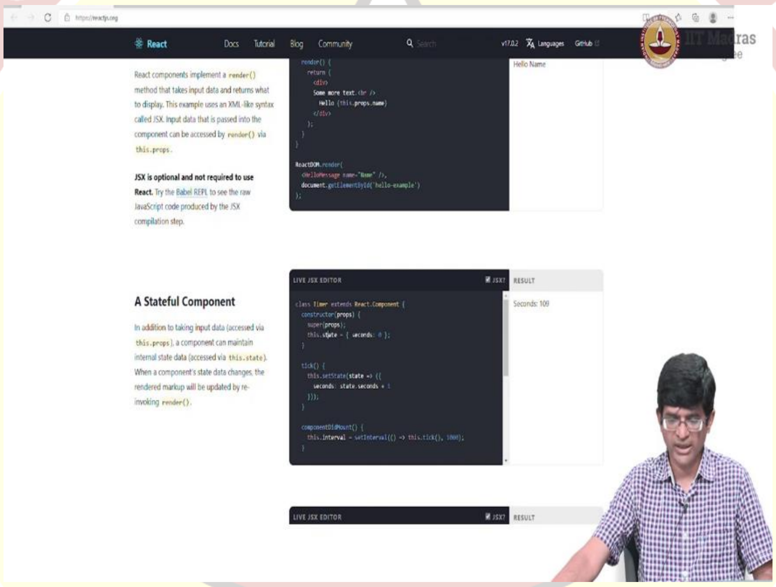
```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Some more text.<br />
          Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Name" />,
  document.getElementById('hello-example')
);
```

RESULT

```
Some more text.
Hello Name
```

LIVE JSX EDITOR ☑ JSX? RESULT

A Stateful Component



React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

```
  render() {
    return (
      <div>
        Some more text.<br />
          Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Name" />,
  document.getElementById('hello-example')
);
```

```
Hello Name
```

A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

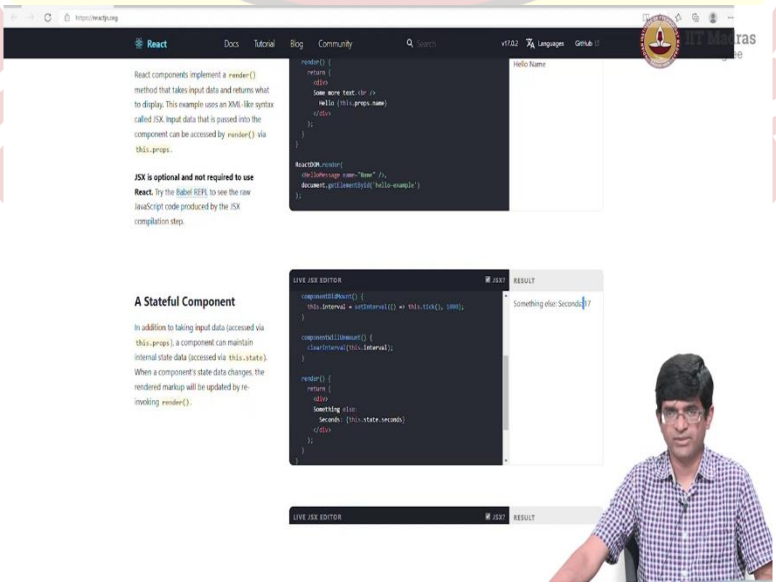
LIVE JSX EDITOR ☑ JSX?

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(state => ({
      seconds: state.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }
```

RESULT

```
Seconds: 109
```

LIVE JSX EDITOR ☑ JSX? RESULT



React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

```
  render() {
    return (
      <div>
        Some more text.<br />
          Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Name" />,
  document.getElementById('hello-example')
);
```

```
Hello Name
```

A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

LIVE JSX EDITOR ☑ JSX?

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Something else:
          Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

RESULT

```
Something else: Seconds: 17
```

LIVE JSX EDITOR ☑ JSX? RESULT

So, this is the main react webpage. And you can see that it contains all the information about it. It also has a very nice demonstration of how react works. And in fact, all of these are basically some kind of react components that are being used in order to display all of these demo applications. And you can actually go in here and make changes directly in the code.

And you can literally see that, as you type instantaneously, it is updating on the site. It is not even like the code pen examples where the HTML, CSS JavaScript, and then it will go and update, the react components are pretty much instantaneous. You type and the reaction is immediate. That is because of the fact that they are using react on both sides, whereas, the code pen page was a more generic sort of solution. It allows you to type in certain things, and then update later.

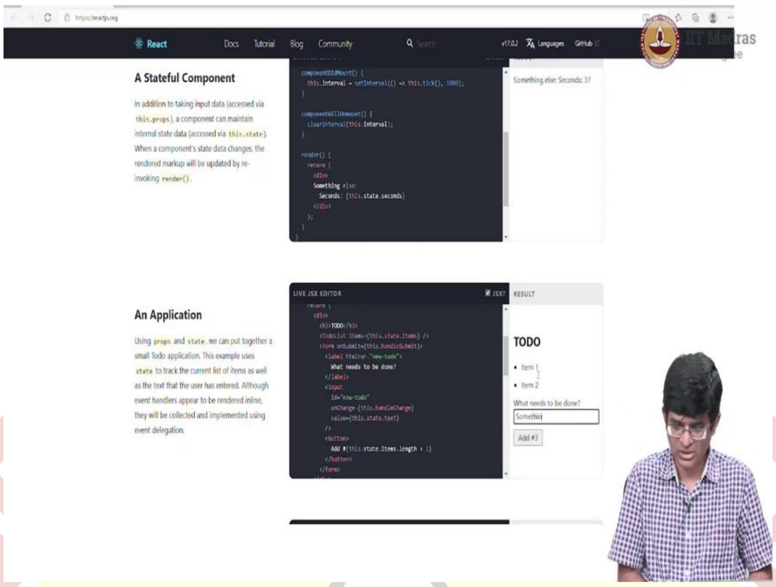
So, what we have over here is that you have a component, you modify it, and it immediately goes and changes something out here. And what has happened is, basically, by creating this class, hello message, which extends react.component, has told you what needs to be displayed that it should be a div like this. And within that div, I could go and add more information.

I could just go in there and type some more text, and I could, for example, say put a line break. And all of that immediately gets rendered on the other side, because it is all being done in JavaScript on your browser, there is no going back to the server out here. Now, we also have something which is a stateful component. So, for example, this timer out here it started out by initializing the value of seconds to 0. Ever since then, every second this number is ticking away.

And all that this component does is it just basically displays the state dot seconds. And of course, like before, I can add more text through here and it would update the screen. And you can see that all, but the interesting thing is, as soon as I made a change over here, effectively, the component was reinitialized, which is why the seconds went back to 0, and started counting again, from there. So, that is one thing to keep in mind.

I mean, this is not normally the way that you are expected to use it. You do not change the component and expect it to update. Whenever you make a change in the component, yes, it has to reevaluate from scratch.

(Refer Slide Time: 09:23)



A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Something else:
          Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

Something else: Seconds: 37

An Application

Using `props` and `state`, we can put together a small Todo application. This example uses `state` to track the current list of items as well as the text that the user has entered. Although event handlers appear to be rendered inline, they will be collected and implemented using event delegation.

```
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <label htmlFor="new-todo">
            What needs to be done?
          </label>
          <input
            id="new-todo"
            onChange={this.handleChange}
            value={this.state.text}
          />
          <button>
            Add #{this.state.items.length + 1}
          </button>
        </form>
```

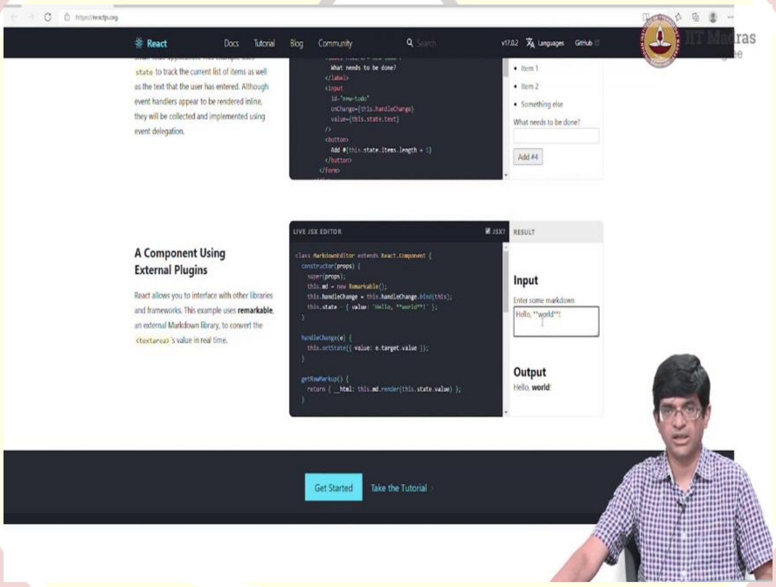
TODO

- Item 1
- Item 2

What needs to be done?

Somethin

Add #3



`state` to track the current list of items as well as the text that the user has entered. Although event handlers appear to be rendered inline, they will be collected and implemented using event delegation.

```
            What needs to be done?
          </label>
          <input
            id="new-todo"
            onChange={this.handleChange}
            value={this.state.text}
          />
          <button>
            Add #{this.state.items.length + 1}
          </button>
        </form>
```

- Item 1
- Item 2
- Something else

What needs to be done?

Add #4

A Component Using External Plugins

React allows you to interface with other libraries and frameworks. This example uses **remarkable**, an external Markdown library, to convert the `<textarea>`'s value in real time.

```
class MarkdownEditor extends React.Component {
  constructor(props) {
    super(props);
    this.md = new Remarkable();
    this.handleChange = this.handleChange.bind(this);
    this.state = { value: 'Hello, **world**!' };
  }

  handleChange(e) {
    this.setState({ value: e.target.value });
  }

  getRawMarkup() {
    return { __html: this.md.render(this.state.value) };
  }
```

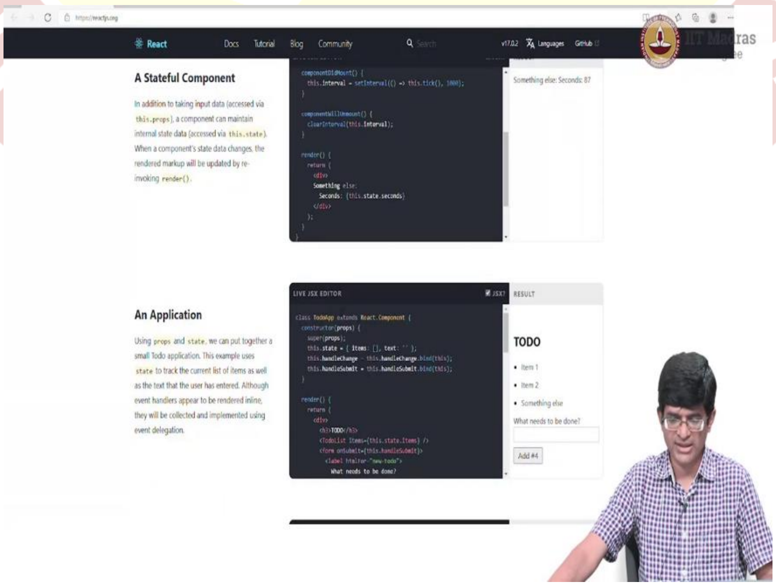
Input

Enter some markdown

Hello, **world**!

Output

Hello, **world**!



A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Something else:
          Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

Something else: Seconds: 87

An Application

Using `props` and `state`, we can put together a small Todo application. This example uses `state` to track the current list of items as well as the text that the user has entered. Although event handlers appear to be rendered inline, they will be collected and implemented using event delegation.

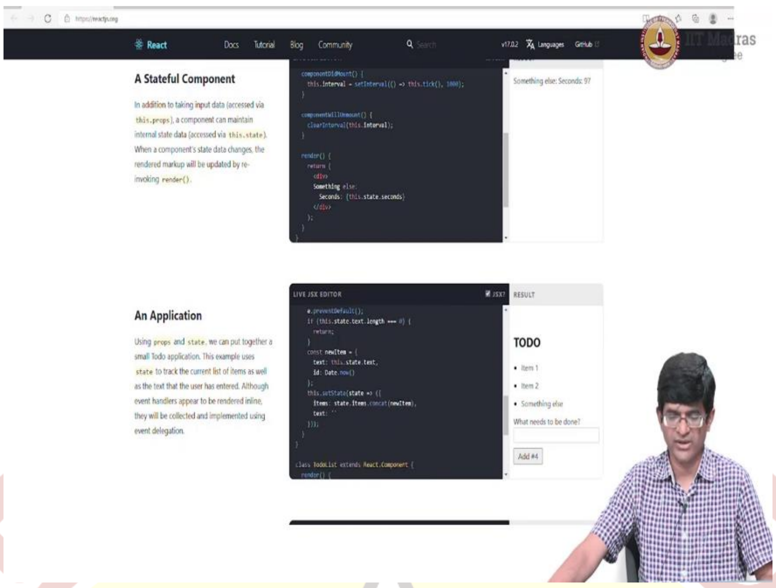
```
class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <label htmlFor="new-todo">
            What needs to be done?
```

TODO

- Item 1
- Item 2
- Something else

What needs to be done?

Add #4



A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
      Something else:
        Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

Something else: Seconds: 97

An Application

Using `props` and `state`, we can put together a small Todo application. This example uses `state` to track the current list of items as well as the text that the user has entered. Although event handlers appear to be rendered inline, they will be collected and implemented using event delegation.

LIVE JSX EDITOR ☑ JSX?

```
    e.preventDefault();
    if (this.state.text.length === 0) {
      return;
    }
    const newItem = {
      text: this.state.text,
      id: Date.now()
    };
    this.setState(state => ({
      items: state.items.concat(newItem),
      text: ''
    }));
  }
}

class TodoList extends React.Component {
  render() {
```

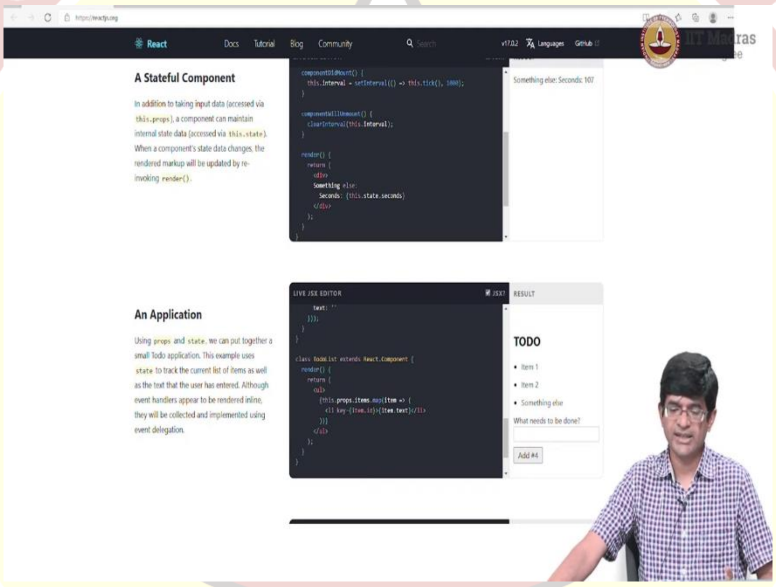
RESULT

TODO

- Item 1
- Item 2
- Something else

What needs to be done?

Add #4



A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
      Something else:
        Seconds: {this.state.seconds}
      </div>
    );
  }
}
```

Something else: Seconds: 107

An Application

Using `props` and `state`, we can put together a small Todo application. This example uses `state` to track the current list of items as well as the text that the user has entered. Although event handlers appear to be rendered inline, they will be collected and implemented using event delegation.

LIVE JSX EDITOR ☑ JSX?

```
      text: ''
    }));
  }
}

class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

RESULT

TODO

- Item 1
- Item 2
- Something else

What needs to be done?

Add #4

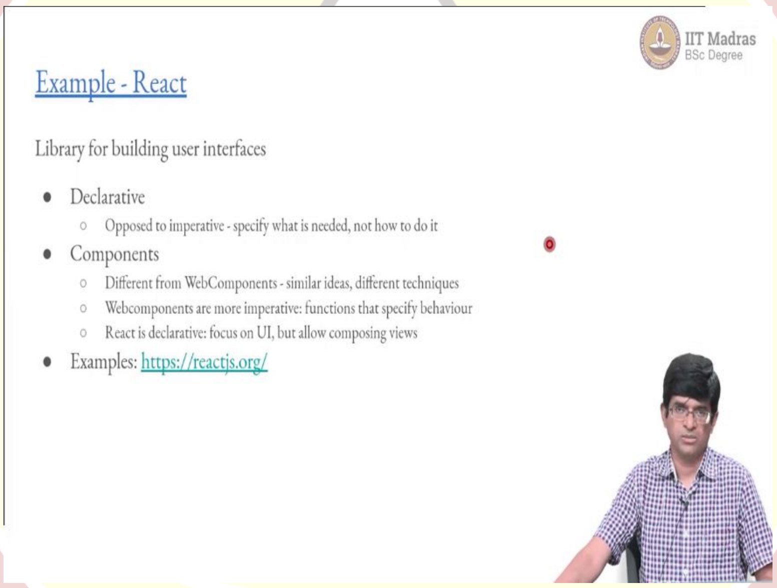
It is even more powerful. I mean, I can actually create something like a to do list right here. I can just basically say item 1, item 2, something else and so on and it is just basically adding all of that out here. Now, is it actually creating a to do list? No, it is not saving it to a database, it is not connecting somewhere else.

It is not sort of allowing you to delete or update or anything else with the to do list, but you can see that as far as the user interface is concerned, it is very clean, it is very fast and it has a nice way of basically just encapsulating the functionality that you need. And there are other. So, for example, you could even directly have markdown as one of the components over here, you could have this text.

And, as you can see, it renders the output by actually calling a markdown processor. So, what is the bottom line? I mean, why is all this useful? Because ultimately, what react is allowing you to do is to create these kinds of components. I have a to do app, which extends react component, it has certain properties, it has state, and so on all of which are coded properly into the JavaScript out here. But once you have that, it just is able to create this entire list and allow you to use that for the user interface. It does not tell you anything about the backend.

It can be combined with other kinds of backends. In fact, you could potentially even combine a react front end possibly with a PHP back end if necessary. Although, there are also ways of doing it such that even the backend is handled by other things, which are similar in JavaScript itself. So, all of that are different in that all of those things are different ways of handling the separation between the front end and back end. React as per their definition, at least they try and stick as much as possible just to the front end.

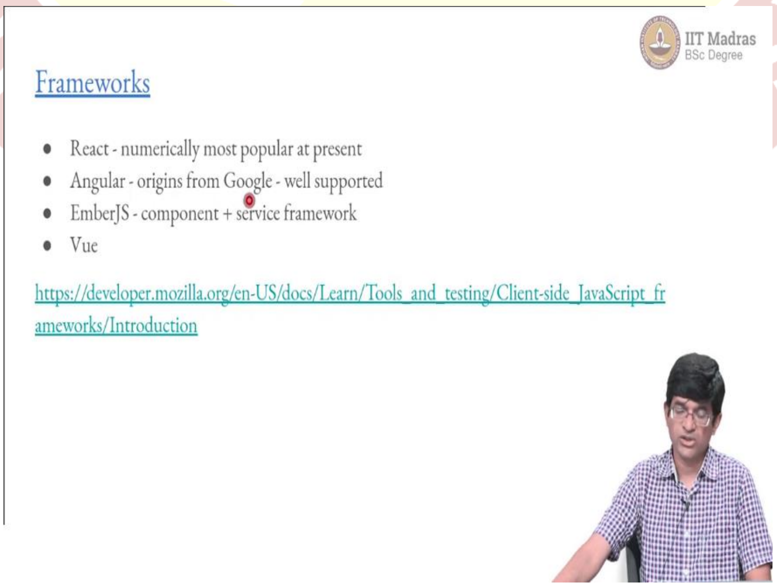
(Refer Slide Time: 11:18)



Example - React

Library for building user interfaces

- Declarative
  - Opposed to imperative - specify what is needed, not how to do it
- Components
  - Different from WebComponents - similar ideas, different techniques
  - Webcomponents are more imperative: functions that specify behaviour
  - React is declarative: focus on UI, but allow composing views
- Examples: https://reactjs.org/



Frameworks

- React - numerically most popular at present
- Angular - origins from Google - well supported
- EmberJS - component + service framework
- Vue

https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction

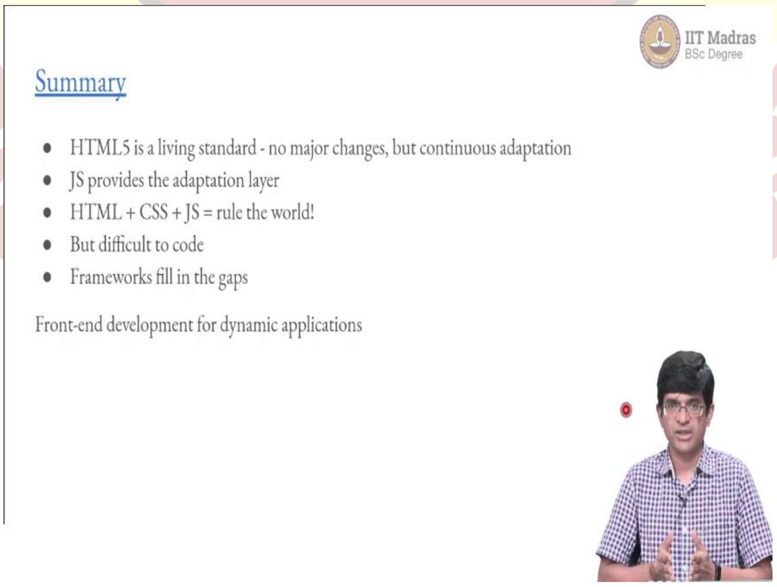
So, react is one of the more popular frontend frameworks that's available today. Is it the only one? By no means there are plenty. Numerically it is probably the most popular at present. Angular with its origins from Google is also very well supported. It has a lot of capabilities similar, in terms of creating reusable components. EmberJS is something which both provides components as well as a service framework.

So, it also allows you to define services and routes and so on. Vue is another way, so EmberJS, is probably a bit more than just a frontend framework, it also does a little bit of the some of the backend part. Vue view, for example, is something similar to react, it is very often it comes up as an alternative to react, which is supposed to be simpler in some ways, and so on. It also has sort of a lot of commonalities with Angular.

Many of these things, what you will find is that, the initial versions have certain functionality, they tend to get more and more complicated. And then there is a sort of simplification that happens and people come up with new designs that are simplified or faster versions of what was there before. And that is part of the natural learning process, if you think about it.

Once again, the Mozilla Developer Network does have a lot of information about these things, including tutorials on how to build certain kinds of apps using this using several different frameworks, so you can even compare the different frameworks by trying all of them out.

(Refer Slide Time: 12:48)



Summary

- HTML5 is a living standard - no major changes, but continuous adaptation
- JS provides the adaptation layer
- HTML + CSS + JS = rule the world!
- But difficult to code
- Frameworks fill in the gaps

Front-end development for dynamic applications

So, to summarize, everything that we have so far, HTML5 ultimately has sort of settled as a kind of living standard. There will be no major changes in the standard, but is going to be continuously adapted as time goes on. So, minor changes in the document that maintains the living standard are always going to be there.

Now, the adaptation layer that allows you to sort of put in new functionality or extend the functionality of HTML5 comes from JavaScript. And JavaScript basically means that HTML plus CSS, for styling, plus JavaScript, you can pretty much create any kind of app that you want. Basic, raw HTML plus CSS plus JavaScript can be difficult to code in by itself, which is where frameworks come in to fill in the gaps. They provide ways of taking out a lot of the boilerplate, making certain things easier, while at the same time providing their own opinions on how certain things should be done.

Now, ultimately, what this means is that this is probably the way to go in terms of the frontend development, if you want dynamic applications, things that are highly responsive to users. And because a lot of the work is done by JavaScript at the users end itself, before even requiring a network communication. The network very often ends up being the slowest part of the entire chain, which means, that if you can do a lot of work using JavaScript at the users end then it simplifies it.

Now, this does not mean, that you should try and overuse JavaScript and you do not make it very fancy web pages with a lot of animations. As far as that is concerned, you still need to go back to the basics of UI design and aesthetics and all of those principles that we have already discussed multiple times in the past. So, HTML plus CSS plus JavaScript is probably the basis of frontend design. It is the basis of frontend design today and very likely to remain so for quite a while in the future.