**Build Apps Using React: Introducing React for Web Applications**

This 18-video course focuses on the React library for component-driven development and basic principles involved in creating React applications. Begin with an introduction to the React library and what is it used for, then discover how to design components with React. Identify features that make React powerful, and examine the concepts underlying important React features. Also explore the virtual document object model (DOM) and how it is used. Learn how to implement a simple web page by using React web application programming interfaces (APIs); recognize basic components of the web page built using React; and use minified productionized React libraries. A two-part tutorial focuses on creating nested elements: creating a DOM hierarchy with nested components, and using key attribute to identify unique sibling elements. Delve into JavaScript XML (JSX) and the functionality of the Babel compiler when used with JSX; then create elements using JSX and the Babel compiler, and render a DOM hierarchy using JSX. Conclude the course by focusing on evaluating simple expressions with JSX, and referring to JavaScript variables within JSX.

**Course Overview**

[Video description begins] *Topic title: Course Overview* [Video description ends]

Hi and welcome to this course Introducing React for Web Applications. My name is Janani Ravi and I will be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a Software engineer and big data expert.* [Video description ends]

A little about myself first, I did my Master's from Stanford University, and have worked at various companies, including Google. I presently work for Loonycorn, a studio for high-quality video content. React is a declarative component-based JavaScript library which originated at Facebook.

And is now very widely used and supported by a vibrant open-source community. React makes use of a clever architectural design known as the virtual DOM, which makes it ideal for apps that work with fast-changing data.

In addition, React uses a declarative extension to the JavaScript language known as JSX, making it easy to control component rendering. This course introduces the React library for component-driven development, and discusses the basic principles involved in creating React applications. It explains important React concepts and features, such as the virtual DOM and JSX.

You'll see how the virtual DOM helps React improve the performance of rendering DOM elements by updating only those properties that have actually changed. The virtual DOM is the fundamental design concept on which React is built.

You will also work with JSX to specify the HTML structure of your page and use the Babel compiler to convert JSX to pure JavaScript. Once you're done with this course, you will be able to implement a simple web page using React APIs, recognize the basic components of the web page built using React. And use the key attribute to identify unique sibling elements.

You will also be able to create elements using JSX and the Babel compiler and describe the functionality of the Babel compiler when used with JSX.

**Introducing React**

[Video description begins] *Topic title: Introducing React. Your host for this session is Janani Ravi. Screen title: What is React?* [Video description ends]

Back in the 1990s and maybe the early 2000s we had static web pages where web pages just had HTML and CSS. You couldn't really do much other than navigate between pages using links. That was still cool, but that was cool then.

As the 2000s rolled in, we started looking for interactivity in our web pages. Interactivity involved running code within our browser to build up dynamic web components, and this was done using JavaScript.

But as our web pages got more complex, we needed something else, and that brings us to React. What exactly is React? Let's get introduced. Using plain JavaScript to build really complex web applications is very difficult. Your web application will simply not be maintainable, which is why there are frameworks and libraries to help you.

React is a library, a cool new technology that helps web developers to build interactive web applications. React is a library that you'll use to build user interfaces in a structured and maintainable way.

React was originally developed at Facebook and it's extensively used to build the website that powers Facebook.com. Other Facebook projects also use React under the hood. React is some of the secret sauce that lies behind Facebook's snappy web interface.

Even though it was originally developed at Facebook, React is currently an open-source project and is maintained by a community of engineers. React still has extensive support from Facebook, which makes it a very stable and robust library.

It's also a library where new features are constantly added. React is used by other Facebook apps as well, Instagram is entirely built on React. When you work in the world of JavaScript, there are just a huge number of a libraries and frameworks available.

jQuery is one such and it's very popular. React is relatively new. But as developers discover its utility and see how easy it is to build component-based user interfaces that are robust, easy to maintain, and great to set up interactions, well, interest in React has only grown in recent years.

Dynamic websites built using JavaScript has been around for about 20 years at this point in time, and the initial libraries are more popular, such as jQuery. React is fairly new in that it was introduced only around May 2013, so its adoption is continuously growing as new developers discover it.

If you were to compare React to, say, jQuery well, jQuery has been around since 2006. Today, React is a JavaScript library of choice for most web developers, whether they are building a web application from scratch, or extending web applications that have already been deployed.

So a good question to ask at this point in time, So What is React? Now let's flip this around. Let's first understand what React is not. React is NOT a Framework. A software framework is essentially like a platform where you write little bits of code that are invoked by the frameworks at the right point in time.

So the framework handles most of the functionality, your custom code hooks into the framework. You have no control over when your code will be called. So React is NOT a framework, React is a JavaScript library.

React is a user-interface library, allowing you to build UI components, or user-interfaces, very easily. A library provides a set of helper functions, objects or modules which your application code calls for specific functionality.

When you use React, you are in charge of the control flow of your code, and you are in charge for how your different UI widgets interact with one another. React simply provides a bunch of libraries to help you build up this interaction in an intuitive way.

React is a declarative, efficient, and flexible JavaScript library that allows you to build reusable, stateful, and interactive components. So you build React one component at a time, and you put them all together, add interactions between components to get your entire application.

[Video description begins] *Screen title: Consider the MVC Programming Paradigm* [Video description ends]

If you've built user interfaces, you know that user interfaces typically follow the Model View Controller design pattern. MVC is a programming paradigm very commonly used to build robust user-interfaces. So what does MVC stand for?

M stands for Model. User-interfaces have to work with data, may be used to create, read, update, and delete data. In the MVC programming paradigm, model refers to what you use to represent the data that you're working with.

The data in your database is represented using a Model, and when you want to interact with the data, you'll need to work with the Model. In the MVC paradigm, View refers to what the user sees. The data that you're working with is represented or visualized using the View. This is what is shown to the user, this is what the user interacts with.

All of the user interface widgets that you create, any charts that you display, sliders, text boxes, these make up the View. The same model, representing the same data, may have several different Views.

There are just different ways to see the same data, visualize the same data. And finally, in the MVC programming paradigm, we have the Controller. The Controller can be thought of as all of the complex bits that allow the View to work with the Model, it controls the Model using the View.

When you build your user-interfaces, you'll split up your components to be part of the Model, View, or the Controller. And you'll have them interact with one another using specific APIs such as events. This design pattern or architecture pattern is used to build robust maintainable user interfaces.

Now, where does React fit in? Well, React is a library that you use to build up the View for your application. React focuses entirely on the View. React does not interact directly with the data that you might have stored at the back-end, in a database somewhere.

The React library is focused on building efficient Views, where you can define the interaction between different widgets or components that make up your View. You'll see in the demos that follow in this learning path that when we're working with React, we'll focus solely on how our components displayed to the user look.

Now I've used this term, component, very often. React is used to build component-based user interfaces. What does that mean? Now, every user interface can be thought of as made up of logical units that go together that a user might interact with at the same time.

These logical units are components. The whole idea behind using the React library is that these components that represent logical units in your user interface are reusable. They are developed exactly once, they have their own state, their own life cycle.

And these components can be used across multiple pages, or even multiple React applications. Components are little reusable units, they have their own state, and they know how to interact with other components in well-defined ways.

Simple components in React come together to make up more complex components. So components are not this monolithic UI widget. Instead, you'll break down a very complex user interface into granular bits.

Every granular bit will be a reusable component, which then comes together to make up the complex UI. Thus each reusable component can be maintained separately. React thus enforces this modular architecture for your user interface, and this is referred to as component-driven development.

Where you think in components, where you build components. And then where you have components interact with other components and come together to make up more complex components. This is your component-driven development that sets up a component tree.

**Thinking in React**

[Video description begins] *Topic title: Thinking in React. Your host for this session is Janani Ravi.* [Video description ends]

Let's say, as a web application developer, you're tasked with building a web application that looks like what you see here on the left. At the very top of this user interface, you can see an input text-box that allows you to search for items.

And just below that, you have a check box that says, Only show products in stock. So it seems like this user interface is some kind of search results page on an eCommerce site. And if you look at the actual products displayed here, that should bear you out.

You can see that we have one column that is the Name of the product and another column that is the Price of a product. You can also see subcategories or sections here. We have a Sporting Goods subcategory and an Electronics subcategory.

Within Sporting Goods, we have Football, Baseball, and Basketball. And within Electronics, we have the iPod Touch, iPhone 5 and the Nexus 7. The example that I've chosen here is a standard React example available at this URL, reactjs.org/docs/thinking-in-react.html.

Let's break this user interface down into its component hierarchy. This exercise will allow us to understand how we need to be thinking about our UI when we use the React library.

Breaking down a user interface that is complex into individual simple components is a part of thinking in React and component-driven development. That's what we'll apply here. Let's take this entire user interface.

You can think of this as a top-level component, this is the Filtered Products Table. It's a Filtered Products Table, because it displays all of the products filtered based on the search term that we may have input in the search input box.

This complete user interface here forms the top of our component hierarchy. It's the root of our component hierarchy. It's the highest level component that we'll write. This Filtered Products Table will not be a single monolithic component.

Instead, we'll break this down into nested sub-components. Let's take this Search Bar at the very top here. It includes a search input box and a check box which says, Only show products that are in stock.

This Search Bar you can clearly develop as a smaller, simpler component. A sub-component that goes into a higher-level component. This is a little lower down in your component tree. This search bar here is a component that will accept user input to apply search filters.

This lower part of this user interface can be thought of as another nested component. You can think of this nested component as the product table which displays the products divided into sections which match the search filters that we have specified, the actual display of products available.

Now if you're really applying component-driven development, this product table itself can be further broken down into nested sub-components. You can see that Sporting Goods and Electronics are two separate categories and they can be their own sub-components.

These sub-components specify the category of products available and can serve as category headers. In addition, individual products displayed can also be a sub-component. So each product that is rendered out to screen, the actual products in the rows of this table can be their own component.

Now whether you choose to breakdown your user interface, this granularly is kind of up to you. But such granular breakdown is possible, when you use the React library, when you think in terms of component, when you think in React.

So this very simple user interface that you see here is actually made up of many components nested together to build up a component tree. We have the Filtered Products Table that is the top-level component.

We have the Search Bar, where you specify user input. We have the Product Table, the Category component and the Product component. What exactly is the right breakdown? Is this the right breakdown? That depends on your use case.

All you have to think about is components are considered to be well designed when each component has a single responsibility. This principle of single responsibility is a classic design pattern which generally applies to object-oriented programming and classes.

But is equally important when you're doing component-driven development. Each component has a single responsibility and thus is a granular bit of reusable code. So remember React is NOT a framework, instead, it is a concept and a library built using that concept.

**React Features**

[Video description begins] *Topic title: React Features. Your host for this session is Janani Ravi.* [Video description ends]

So what is it about React that makes it so cool and the library of choice for a web developers? That's exactly what we'll discover in this video. The first feature that makes React intuitive and easy to work with is the fact that it's a Declarative framework.

Once you've broken down a complex user interface into individual components, you can focus your development efforts to a single component. You will then design the view associated with the component based on the state that the component holds.

So each component has its own internal state and you'll design views associated with state. As the state changes, your views will be updated automatically. Your components will be re-rendered with the updated state.

React is also intuitive and easy to work with because of component-driven development. You don't think of the complex user interface as a single entity. You break it down into individual component. This component-based development allows you to work on simple components which come together to make complex components.

Encapsulated components manage their own state. So each component is just responsible for its own state. It doesn't need to worry about the internal state of other components. Each component is responsible for displaying the right view based on its internal state.

React is also extremely Performant. When you're developing web UIs, you have to interact with the document object model of your browser, referred to as the DOM. Now, anyone who has developed a web application knows that DOM updates are slow.

Re-rendering your HTML structure is often the slowest part of your app. Developers of the React library have understood this well. React applications don't deal directly with the DOM of the browser. Instead, React under the hood has a virtual representation of the DOM.

All changes to your component are first applied to this virtual representation of the DOM, which is then reconciled with the actual DOM. And this reconciliation process ensures that only those DOM nodes that have actually changed are re-rendered.

There is no re-rendering that is done wastefully. You'll see in the demos that follow that when you build up your React applications you'll do so within Node.js projects. Which means React applications can be rendered server side as well.

And this helps improve the performance of your apps. Server-side rendering is supported using node and communication within a React application follows a very strict direction of flow of data. Data flows from the top level components to lower level components.

In this learning path, we'll be working with React for web applications that is to be displayed within a browser. But you should know that the React library also supports React for mobile applications using React Native. So the concepts will remain the same, the rendering will be within your mobile app.

**Exploring React Features**

[Video description begins] *Topic title: Exploring React Features. Your host for this session is Janani Ravi. Screen title: Declarative* [Video description ends]

Let's understand what each of these features mean in a little more detail. When you work with React, you'll see that it is a declarative library. You focus your attention on designing your views based on internal component state.

That's all you need to worry about. The declarative nature of React allows you to tell React what state you want the user interface to be in. You'll say, take a look at this data and render this user interface in this way.

It abstracts out attribute manipulation DOM updates, working with HTML tags, etc. You'll just look at the data and render your UI. Your focus should be on user interface design. It's React's job to update and render the right component based on its internal state.

All you need to do is change the state. And it's React's responsibility to re-render the component based on the new state of that component. You don't have to write all of this wiring up code. So code is predictable, robust, and easy to debug.

Your responsibility is to say, this is how my UI should look for this state. All of the wiring up is done by React, making your code easy to maintain.

[Video description begins] *Screen title: Component-based* [Video description ends]

React is a Component-based library, and you have to follow component-driven development to code in React. What does this mean? View components in React include both the logic of the view and the behavior of the view.

Within a React component, you'll write the JavaScript for both logic and behavior. So how a component behaves in different situations is present right there in the same place as the rendering of the component.

This completely eliminates any bridging code that you might have to write where user interface changes have to be relayed to the JavaScript code. This allows you to write the code for your components in a very decluttered and intuitive manner.

The JavaScript code checks the logic right there within your component. And declaratively specifies what this component should now look like, based on the logic that was calculated. And React will take care of updating the DOM. Grouping together how the view is rendered, and the logic and behavior of a component all in one place, makes your code manageable and more intuitive to maintain.

[Video description begins] *Screen title: Virtual DOM* [Video description ends]

Component-driven development and all of the other cool features that React supports is powered using the Virtual DOM. Now, if you have any experience with web applications. And hopefully you do, you know that DOM rendering is what tends to make web applications really slow.

It takes a really long time if you have a complex HTML structure that needs to be rendered out by our browser. And React speeds this up. How? Well, React maintains a mirror copy of every DOM element.

The entire HTML structure of your web page has an equivalent within React. This mirror copy is referred to as the Virtual DOM. Imagine, every DOM node on your web page has a corresponding mirror copy within memory.

This Virtual DOM that React maintains is an in-memory copy of the real DOM that your browser displays. It might seem wasteful to have a mirror copy of the real DOM. Let's see how the Virtual DOM helps.

The use of this Virtual DOM is a fundamental design strategy in React. And we'll talk more about the virtual DOM in the next video as well. The virtual DOM is an abstraction or design which allows us to make UI changes very, very fast. So any changes that you make in your dynamic UI will be re-rendered quickly, thanks to this Virtual DOM.

This is because when updates change the internal state of your component, this change in the internal state does not re-render that component immediately in the actual DOM. All changes are first made to the virtual DOM, the mirror copy of the actual DOM. Changes to the virtual DOM are in-memory changes.

That is, they don't actually require re-rendering the user interface, which is why they're fast. Once the in-memory changes are complete, React will check to see whether there are any actual changes to be made.

By comparing the virtual DOM with the actual DOM. This process, where the virtual DOM is compared with the actual DOM to see whether any real changes have to be applied to the actual DOM, is called the reconciliation process.

And this reconciliation process is an important performance optimization in React. During the reconciliation process, React will determine what DOM nodes actually need to be updated. It won't go in and update all DOM nodes blindly.

Only the updated properties are applied to the actual DOM, thus minimizing rendering. And the cool thing about all this is that this is just done for you when you use React. You don't have to do anything special to minimize rendering. The Virtual DOM takes care of everything. You just have to use React components.

[Video description begins] *Screen title: Server-side Rendering* [Video description ends]

A very cool feature in React that developers love is the ability to perform Server-side Rendering. You can render your pages on the server. And then simply display static pages on your site. And the Virtual DOM is what enables this as well.

The fact that a mirror-copy of the entire DOM structure is available in memory means that you can render your user interface without actually having a browser. Just by looking at the attributes, the DOM nodes and the properties in the virtual DOM.

If you're looking to make your application extremely performant, React allows you to create the entire web page on the server. You'll use the virtual DOM to build up the web page. And set up the HTML structure of your page.

And once this has been set up on the server, all you need to do is to send the HTML page down to the client. This server-side rendering can allow your React applications to become extremely fast. The initial page load is just about rendering the HTML. There is no JavaScript load involved.

[Video description begins] *Screen title: One-way Data Flow* [Video description ends]

Another useful feature to understand in React is the way the flow of information works or how data flows. React uses composition to build individual view components. So you'll build granular view components, which are then composed within one another, nested within one another, to build complex user interfaces.

This means that complex views within your web application are not monolithic views. They comprise of components that contain other components within them. Now, the question is, when you interact with these complex views, how do you pass information to them?

And how does data flow between components? Data that you pass to React components flows down the component tree hierarchy from parent to child. So you'll pass in a superset of data to the top-level component.

And the data related to the granular components are then passed down the hierarchy. So data flow is from the top level to the bottom. As we design our React components in the demos that follow in this learning path, you'll see this flow setup over and over again.

You'll pass data into the root component in your component tree. From there, it'll flow down to more granular components down the tree. It's important you understand how your data flow should be set up within your React components.

Because this data flow from the top to the bottom streamlines communications between your component and makes React very intuitive. And finally, let's briefly talk about React Native. This is what you'll use to build mobile applications in JavaScript using React.

You'll see when you work with React that React libraries are separated into component libraries and DOM libraries. When you work with React Native, instead of using the web browser's DOM, you'll use React Native's rendering instead.

The concepts remain the same. Component-based development, how data flows, how state works, how props work. Everything will remain the same, except that you're rendering to a mobile application. React Native allows you to build platform agnostic native components with high-performance across multiple mobile platforms.

**The Virtual DOM**

[Video description begins] *Topic title: The Virtual DOM. Your host for this session is Janani Ravi.* [Video description ends]

It's impossible to discuss React without mentioning the Virtual DOM. We've already been introduced to the Virtual DOM earlier. The Virtual DOM is an abstraction or a design, which allows us to make highly performant user interfaces where user interface components can be rendered very quickly.

In this video, we'll discuss the Virtual DOM in a little more detail, and we'll also understand how it works visually. The Virtual DOM is essentially a hierarchy of components in your component tree that represents your application.

Now you know that every web page can be represented as a hierarchy of HTML elements. All webpages are essentially a tree structure where every node in the tree is an HTML element. And different HTML elements are rendered differently by the web browser.

Here is an example of what the HTML structure of your web page might look like. You have the HTML node, that is typically the root node. The HTML node may have a header node where you specify the library files that you reference from within your application.

The header tag holds additional metadata such as the title of the page, the icon that is displayed within your browser tab, etc. Now all of the visual elements rendered within the content of your page is within a body tag. So within the body tag you can have headers.

They can be h1, h2, h3, h4 headers. You can have paragraph elements, you can have images, divs, spans, and so on. This is the actual DOM hierarchy that represents your page. React, behind the scenes, creates an in-memory representation of this DOM hierarchy that represents the HTML structure of your page.

This in-memory representation is the Virtual DOM. In this Virtual DOM, every HTML element that makes up your actual DOM's hierarchy is a React node. React nodes are also referred to as React elements, and that's how I'll generally reference them in the demos that follow.

You can see that we have React nodes at different levels of this hierarchical tree representation. Essentially, React now has an in-memory mirror copy of the actual DOM structure of your web page.

Now this is in-memory, this is under the hood, you don't have to do anything explicit to have this Virtual DOM be set up. Now, the DOM of your page will be changed in response to user events or server events. This is how typically webpages work.

The user might input some text or click on something, which might cause the page to re-render. These are examples of user events or there might be some data behind the scenes that has changed, maybe a chat has come in.

This will cause your chat box to re-render to show the incoming message. Webpages thus update their document object models in response to user events or server events. What are some examples of User events?

Well, clicks on the webpage, swipes on a mobile application, hovering over a button, all of these are User events. Just like user events, you might have Server events that cause your webpage to be re-rendered or to be updated.

Examples of Server events could be a message has come in, a new email has come in which requires your page to be updated. These Server events are typically sent to the browser in the form of server responses. Responses from the server can indicate either success or failure.

Responses from the server can also be the reaction to an operation that the user has performed. Both of these user and server events can update the webpage that is rendered within the browser.

Essentially, user and server events both change the state of the underlying data model that is represented by your user interface, that is represented by your view. When the model changes, the view has to be updated as well.

Updating the view in response to model changes is the essence of the model view controller paradigm that we use to build robust user interfaces. And here is where the Virtual DOM design that React uses is so useful.

React will first render the changes that need to be made to your UI components on the Virtual DOM. So the changes will be made in memory before the actual changes are displayed to the user. So now you have an updated tree structure for the Virtual DOM.

Once the virtual DOM has been updated, React will then go component by component, node by node, through every node in the virtual DOM to see how the actual DOM differs from the virtual DOM.

This process involves a lot of comparison and is very detail-oriented. React will calculate the difference between the previous virtual DOM and the newly updated virtual DOM, which is the result of either user events or server events.

Since React has all of the information with it, it will be able to perform this comparison at a very granular level. React then does something intelligent, it then only updates what is needed in the new DOM.

Rather than doing a blanket update of the HTML structure of your user interface, React will check to see whether a particular attribute in a particular HTML element requires a re-render. If so, only that element will be re-rendered.

This entire process where user and server events require changes to the virtual DOM, which are then compared with the actual DOM and changes are performed incrementally to the actual DOM is called reconciliation.

Reconciliation makes DOM updates very quickly, because it's not the entire tree structure that is re-rendered, only those HTML elements whose properties have changed will be re-rendered. And this reconciliation process of the virtual DOM with the actual DOM is what allows the user interface to be extremely snappy or extremely fast.

This reconciliation process allows React to minimize the number of DOM elements which should be updated. Let's say you have about 100 elements rendered onto your browser and only three have changed, only those three will be updated.

Remaining elements will not be re-rendered. And the best part about this entire reconciliation process is the fact that you won't even know it's happening. It's completely abstracted away from the user, you just have to write React code using best practices.

The rest of this, all of this rendering and performant updates is taken care of for you. You simply write code as though the DOM were being updated in real time. It's good to know that the Virtual DOM exist and that React updates are performant, but you don't have to do anything special in your code to make this work.

Just write your code in the regular way and wait for React to perform its magic. When you first hear of the Virtual DOM it can be a little confusing, so here is a little bit of clarification. Virtual DOM is an abstraction or a design pattern that React uses, it's not an actual technology.

React is built using the concept of the Virtual DOM. The Virtual DOM is not a technology that React harnesses for more performant rendering of webpages. Developers often confuse the Virtual DOM with another component called the Shadow DOM.

The Shadow DOM is an actual browser technology designed for scoping variables and CSS in web components. The Shadow DOM is completely different, it's a technology and not a design pattern. The Virtual DOM is a design pattern or a concept.

What makes the Virtual DOM and reconciliation possible is something known as React Fiber. React Fiber refers to the implementation of React's core algorithm for blazingly fast rendering and updates using incremental rendering. The idea of reconciliation and incremental rendering of HTML elements is possible only because of React Fiber. All of this works under the hood.

**Creating a Simple Static HTML Page**

[Video description begins] *Topic title: Creating a Simple Static HTML Page. Your host for this session is Janani Ravi.* [Video description ends]

Let's start writing our very first simple HTML page. This is going to be a static HTML page that will be built up using very simple React components. Here is the editor that we'll be using for this course and all of the other courses that follow on the React path.

I'm using Sublime Text, and here I am within my current working directory that is the React directory. You can see the directory here at the top left of my screen. Within the React directory, I'm now going to create a New Folder, and this folder will be a subfolder within the top-level React directory.

[Video description begins] *She right-clicks the "React" directory and selects the "New Folder" option from the context menu that appears.* [Video description ends]

I will call this subfolder StaticHTML, and this will contain our very first demos in React. So you can see we have the React folder, and under that, the StaticHTML folder, and I'm going to create my first HTML file within this StaticHTML folder.

[Video description begins] *She right-clicks the "StaticHTML" folder and selects the "New File" option from the context menu. A tab named "untitled" displays in the working pane and the host pastes some HTML code in it.* [Video description ends]

And here we set up some very simple HTML and some JavaScript, creating our DOM elements that makes up our first React application. This is where we're going to start, and you'll see how we build upon this in future demos.

Let's take a look at our head tag within our top-level html tag, and you can see that we have two script tags that reference JS files.

[Video description begins] *She highlights the two .js files which are present in the two script tags, which are in turn present within the head tag.* [Video description ends]

These JS files contain the React libraries that we'll be using. The latest versions of React libraries, whether they're development versions or product versions, are hosted at unpkg. Which is an open-source, fast, global content delivery network for everything that works with NPM, or the Node Package Manager.

As we move on to running React in a proper development as well as a production environment a little bit later on, we'll be using the Node Package Manager, or NPM, extensively. But as we get started with React and understand how it works, we'll simply reference the JavaScript libraries which are hosted on unpkg.

Let's examine and parse the structure of how we reference individual files from the unpkg content delivery network. So we'll first reference unpkg.com. This is an HTTPS link indicating that it's secure. This is a reference to the base URL of the unpkg content delivery network.

[Video description begins] *She highlights the following code displaying in the two scripts tags: https://unpkg.com* [Video description ends]

After this base URL, we have a /, and after that, we have the name of the package that we want to access. React is the first package, react-dom is the second package, and I'll talk about the differences between these two in just a bit.

After we specify the name of the package, we have an @ sign and then the version of the package that we are using. The versions I've chosen here is the latest version of react as well as react-dom at the time of this recording.

We are using react version 16.7.0 and react-dom version 16.7.0. Once we specify the version, we specify the path to the file that we want to include in the StaticHTML, react.development.js and react-dom.development.js.

The React package here allows us to work with React in general. The ReactDOM package offers special APIs that allow you to perform DOM manipulation on your HTML page. When you work with React, you will be using APIs from the React library as well as the ReactDOM library.

React is what you'll use to build components and work with the life cycle of components. ReactDOM is what uses these components and renders them within your web browser. Let's say you're working on mobile applications where you won't be using a web browser.

You'll continue using the React library but you may not use ReactDOM. You'll use React Native instead. Because we're working on web applications for all of the demos, we'll be using the React library as well as the ReactDOM library.

So you'll see that these libraries will be included in all of our demos, either in the form of script tag references or within our NPM project. There is one last detail to observe here in the script tag files that we've included in this HTML file.

Notice that we have react.development.js and react-dom.development.js. React offers JavaScript libraries for development as well as production. Development JavaScript is not minified. That is, you can see the actual variables and the structure of the code.

So you get better error reporting as well, which is great for debugging while you're developing your code. Production JavaScript is minified, which means it occupies less space and is more performant.

If you have done web programming before, you're probably familiar with the fact that the size of the JavaScript that is downloaded from the server onto the browser makes a difference to how fast your page loads.

Smaller the JavaScript, the more performant your page is. That is, your page loads faster. Now that we're familiar with the React libraries that we're going to be using and the unpkg content delivery network, let's turn our attention to the HTML contents of the StaticHTML file.

Observe that we have a body tag, and within that, we have a div element with the id my-react-app.

[Video description begins] *She highlights the following lines of code from line 7 to line 9. Line 7 reads: <body>. Line 8 reads: <div id="my-react-app">. Line 9 reads: </div>.* [Video description ends]

When you're working with React, it's common to have a top-level element within your body tag into which all of your React components will be rendered. The id of this element is typically set to be root, but you can specify any id that makes sense for your application.

My top-level HTML element which will hold the entire contents of my React app has the id my-react-app. And within the body, I have another script tag where I've written some JavaScript.

[Video description begins] *She highlights the following lines of code from line 11 to line 15. Line 11 reads as: <script>. Line 12 reads as: var el=React.createElement("h1",{},"Welcome to the world of React!"); Line 14 reads as: ReactDOM.render(el, document.getElementById('my-react-app')); Line15 reads as: </script>* [Video description ends]

In order to keep things simple, I've inlined the JavaScript that I plan to use to set up my first elements using the React library. We'll set up our first DOM element, which is going to be an h1 tag using React.createElement. createElement is a function available in the React library.

We specify the first argument to this function. That is, the name of the tag, the HTML element that we want to set up, which in this case is h1. This tag has no additional attributes, which is why the curly braces which forms our second input argument is completely empty.

This curly braces is what you'll use to specify other attributes, such as the style or class that you want to apply to this tag. And finally, the last input argument to React.createElement is the contents of this tag.

This is going to be a very simple header which simply has the text content, Welcome to the world of React! Once we've created this element, this is our DOM element, and assigned it to the variable el, we now need to use the ReactDOM library to render this element onto our HTML page.

And for this, we use the ReactDOM.render function. ReactDOM.render needs two bits of information to function. First is the element that you want to render into your HTML page. And the second is the parent element which holds this element that you want to render.

And the parent element I get by referencing the document of my HTML page document.getElementById('my-react-app'). This div that you can see here on-screen with the id my-react-app is going to be the parent of our h1 element that we've created. And that's it. We've used React.createElement to create our first HTML element and then rendered it using ReactDOM.

**Exploring the Static HTML Page**

[Video description begins] *Topic title: Exploring the Static HTML Page. Your host for this session is Janani Ravi. The Sublime text editor interface appears, displaying the HTML file from the previous demonstration.* [Video description ends]

Let's go ahead and save this HTML file and give it a meaningful name. I'm going to call this FirstReact.html. Observe that I save this HTML file under the StaticHTML folder. Which is once again under the top-level React folder.

Go ahead and save this file in this current working directory. And we are now ready to view this HTML page on a web browser. Now, before we actually install an HTTP server that will run our web pages, I'm going to take the easiest way possible to view my HTML content. I’ve right clicked here on this file. And I'm going to copy the absolute path of this file onto my clipboard.

[Video description begins] *She selects the option called "Copy File Path".* [Video description ends]

And then I'm going to reference this file within a browser window using the absolute path. So I'm viewing this file within the browser using a file path, not a server path. This you may already know as a quick and dirty way to see whether the HTML content of your page has been set up right.

And it will work for simple static HTML pages. My browser of choice is Chrome. And here is our very first static HTML page constructed using React libraries. Here is our header, Welcome to the world of React!

A great advantage of working with Chrome is that you can right-click and bring up Chrome Developer Tools by hitting the Inspect option on the right click menu. Inspect will launch Chrome Developer Tools, allowing you to debug your JavaScript and view the HTML structure of your page.

It also offers a bunch of other options. Such as performance monitoring and everything that you need to debug and work with web pages.

[Video description begins] *In the Chrome Developers Tools interface, the HTML structure of the React file displays. Each tag has an arrowhead on the left to expand it. Above are three tabs called "Elements", which is active now, "Console", and "Sources".* [Video description ends]

The HTML structure of our page has a head tag and a body tag. Let's expand the head tag and see what it looks like. And here you can find the script tags that reference React and React DOM development JavaScripts.

Let's expand the body tag and take a look at the actual content of our web page. Here you'll find the top level div that we had set up in our page HTML, that is the div with the id my-react-app.

[Video description begins] *She highlights the following line of code: <div id="my-react-app"> ... </div>.* [Video description ends]

And under this div, you'll find the h1 header. Which says, Welcome to the world of React! I'll leave that to you as an exercise to confirm that whether indeed your h1 tag was rendered under this div tag.

Let's move on and explore Chrome Developer Tools. I'm going to head over to the Console tab. That will allow me to view any log messages that I might have printed out. Our static HTML file contain no log messages. But there was a message from the React library.

[Video description begins] *The following URL also displays with the suggestion: https://fb.me/react-devtools.* [Video description ends]

It makes a suggestion that you might want to download React DevTools for a better development experience. React DevTools allow you to view the components that you set up using React in the form of components. It understands React’s composition structure. We'll explore and work with React development tools later on in this path.

[Video description begins] *She highlights the following message: You might need to use a local HTTP server (instead of file://): https://fb.me/react-devtools-faq.* [Video description ends]

Here it also makes a suggestion that you might want to use a local HTTP server instead of referencing the file directly on your local machine. These are all very reasonable messages. And we'll fix these later on when we're more familiar with React.

I’ll now open up a new tab. And we’ll head over to the unpkg.com site, and take a look at the React development libraries. These are the libraries that we referenced in our script tag.

[Video description begins] *She enters the URL: https://unpkg.com/browse/react@16.12.0/umd/.* [Video description ends]

Here is the path that I've specified, React at 16.12.0. This is the version of React that we’re working with for this path.

[Video description begins] *A webpage titled "UNPKG" displays. It lists three utility files: react.development.js, react.production.min.js, and react.profiling.min.js.* [Video description ends]

You can see that we have a react.development.js, and React production files as well. I'm going to open up this development file. And you can see that this JavaScript file is not minified. You can see that React here has been originally developed by Facebook.

The copyright is Facebook, Inc. And it's licensed under the MIT license. Let's take a look at the JavaScript. And you can see that it's easy to read. And you'll find that the debugging messages that you get for this development environment will also be far more detailed than the production.js file.

In exactly the same way, we can take a look at react-dom.development.js. And I'm going to open up a new tab and do exactly that. Once again, let's head over to unpkg.com. And we go to react-dom 16.12.0, that is the version of the react-dom API that we’ll be using.

You can see that there are a number of utility files here along with the react-dom development.js, which is what we're going to look at. At the bottom here, you’ll find react-dom.development.js. Which is the file that we reference in our script tag to access the react-dom API.

You can scroll down here and view that this is un-minified development JavaScript. These are the two JavaScript files that we referenced in the script tags of the very simple React app that we built, a simple static HTML app.

**Referencing Production React Libraries**

[Video description begins] *Topic title: Referencing Production React Libraries. Your host for this session is Janani Ravi.* [Video description ends]

In an earlier demo, we built a very simple static HTML page in React by referencing development libraries for React as well as ReactDOM.

[Video description begins] *The Sublime text editor interface displays. It displays file called "FirstReact\_Production.html" in the working pane.* [Video description ends]

In this HTML file called FirstReact\_Production.html, we'll construct static HTML page. But this time we'll reference the productionized JS files, the minified JS files. Observe that this static HTML page is laid out very much like the previous one. The main difference is within the top header tags and within that we have the script tags where we reference the React JavaScript libraries.

[Video description begins] *She highlights the code in lines 4 and 5. Line 4 reads as: <script src="https://unpkg.com/react@16.7.0/umd/react.production.min.js"></script>. Line 5 reads as: <script src="https://unpkg.com/react-dom@16.7.0/umd/react-dom.production.min.js"></script>.* [Video description ends]

Now, the reference is, once again, to the unpkg.com content delivery network. Observe that we are using the React 16.7 production version. The version of React that I'm referencing here, 16.7.0 for React as well as ReactDOM, is not the latest version at the time of this recording.

I wanted to put this in here to demonstrate to you that it's possible for you to reference older React libraries just by referencing the right package on unpkg.com. If you're working in an organization that's on an older version of React, here is how you access those JS files for you to work with.

The main thing to observe here is that we're using react.production.min.js and react-dom.production.min.js. These are the minified versions of the JavaScript files where all of the variables have been renamed to be extremely short.

All of the white spaces have been removed. This is so that the JavaScript size that needs to be downloaded onto your browser is smaller and your overall page load is more performant. When you run your own React application in production, the React environment will allow you to compress your JavaScript as well, the JavaScript that you've written.

The remaining contents of this file should be very familiar to you. We use the React.createElement function in order to create a new HTML element. Once again, it's going to be an h1 element.

[Video description begins] *She highlights the following code in line 12: var el=React.createElement("h1",{},"Welcome to the world of React!");* [Video description ends]

We specified no additional attributes, as shown by the empty curly braces. And the contents of this h1 header is, Welcome to the world of React! We then render this element into the top-level my-react-app div. The div HTML element with the id my-react-app is something that I've set up within the body of this HTML page.

[Video description begins] *She highlights the code in lines 8 and 9. Line 8 reads as: <divid="my-react-app">. Line 9 reads as: </div> .* [Video description ends]

Once again, this is a static HTML page. Now that we have saved this HTML file, I'm going to right-click and copy the file path over so that I can view this file within my Chrome browser window. I'm going to paste it, once again, in the form of an absolute file path. I have not used an HTTP server yet.

[Video description begins] *The URL is /Users/loony/Project/React/StaticHTML/FirstReact\_Production.html.* [Video description ends]

And there you see it. Welcome to the world of React! I'm going to right-click and hit Inspect. And this will bring up Chrome Developer Tools, allowing me to view the HTML structure of my page.

[Video description begins] *The HTML structure comprises of the main tags, html, head, and body. An expand arrowhead displays next to each tag. Above the structure are three tabs, namely "Elements", "Console", and "Sources".* [Video description ends]

The static HTML page here has a header tag, which includes our script tag references. And a body tag, which includes the HTML content that we've just rendered into the browser. Here are the script tag references to production JavaScript files. And if you open up the body tag, you'll find the top level div with id my-react-app.

And under this you'll find the h1 header has been rendered. Let's head over to the Console window to see whether there are any console messages that you can view here. And you can see that there are no messages in the production JavaScript.

Production JavaScript is for live deployments. If you're curious like I am about what the productionized JavaScript looks for React and ReactDOM, I'm going to head over to unpkg.com.

[Video description begins] *A webpage titled "UNPKG-react" displays. It lists three utility files, react.development.js, react.production.min.js, and react.profiling.min.js.* [Video description ends]

And here I am. I'll click through to react.production.min.js and take a look at this JavaScript here.

[Video description begins] *The min.js file displays with the file path "react/umd/react.production.min.js".* [Video description ends]

You can see that all of the variable names that we saw in the original development version of the file have been renamed to be extremely tiny so that they don't occupy much space. You can also see that all of the spaces have been removed from this JavaScript file.

This JavaScript file is hard to read, hard to debug, but really it doesn't occupy very much space at all. The file size of this JavaScript will be tiny and it'll greatly improve the rate at which your initial page load occurs when this JavaScript is used.

This is the production version of the React library. There is a production version for the ReactDOM APIs as well. So open up a new tab, and let's head over to unpkg.com react-dom version 16.12.0. This is not the version that we referenced, but you can see that this is the latest version.

[Video description begins] *A page with path "react-dom/umd displays, consisting many ".js" files lists underneath.* [Video description ends]

If you browse down to the bottom you'll find the production version here, react-dom.production.min.js. You can explore this minified production file but this is an auto-generated file that you'll generate using minifying tools. It's very hard to read and understand what's going on here.

[Video description begins] *A webpage titled "UNPKG-react-dom" displays.* [Video description ends]

Which is why during development, you'll prefer better debugging messages and also a file that you can step into and debug. You'll never work with production when you're developing your React app. Let's now see all of the versions of React that are available at the time of this recording.

Simply go to reactjs.org, that is the main React site, / versions. All of the versions of React that are currently available will be listed here. You can see that the latest version at this point in time is 16.12.0.

[Video description begins] *A webpage titled "React Versions" displays.* [Video description ends]

If you need to choose any other version and you want to see its contents, you can always take a look at the Changelog links here in addition to the documentation.

**Creating Nested Elements - Part 1**

[Video description begins] *Topic title: Creating Nested Elements-Part 1. Your host for this session is Janani Ravi.* [Video description ends]

Now that we know how to reference React libraries in development and production, let's move on.

[Video description begins] *The Sublime text editor interface displays.* [Video description ends]

Let's set up a more complex HTML page which includes nested elements. Here we are in a new HTML file called NestedElements.html. For each file, you'll be able to view the folder that this file is present in off to the left of your screen.

Under the top-level React directory, I have a CreatingElements folder, and within that, I have this NestedElements.html file. Once again, this is going to be a static web page. And I'm going to paste in the entire contents, as you can see here on screen.

Here on in, we'll continue using the development JavaScript files, react.development.js and react-dom.development.js.

[Video description begins] *She highlights the code in lines 3 and 4. Line 3 reads as: <script src="https://unpkg.com/react@16.7.0/umd/react.development.js"></script>. Line 4 reads as: <script src="https://unpkg.com/react-dom@16.7.0/umd/react-dom.development.js"></script>.* [Video description ends]

In this HTML page here, I've also included a little bit of CSS to style the page. This is present within style tags. So I have a CSS, which references an id, that is, #para-id, which will be displayed in the green color, and

[Video description begins] *She highlights the following CSS code: #para-id { color:green; }.* [Video description ends]

I have CSS that applies to an HTML tag, that is, h2.

[Video description begins] *She highlights the following code: h2 { color:red; font-style: italic; } </style>.* [Video description ends]

Once again, I have a top-level div within my body tag into which we'll render the React elements that we create. This div has the id my-react-app.

[Video description begins] *She highlights the code in lines 17 to 19. Line 17 reads as: < div id="my-react-app">. Line 19 reads as: </div>.* [Video description ends]

If you scroll down below, you'll be able to see the code that we use to create nested elements. Once again, I've used a React.createElement function in order to create an HTML element. The element here is the div element with the id, inner-div-id. Observe how I have specified the attributes for this element within curly braces.

[Video description begins] *She highlights the code in lines 22 and 23. Line 22 reads as: var el=React.createElement("div",. Line 23 reads as: {id:"inner-div-id:}, .* [Video description ends]

Within this div element with the id, inner-div-id, I've created another element using React.createElement once again. This is a h2 header element with the id, header-id, and the contents of this h2 header says, Welcome to the world of React!

[Video description begins] *Line 24 reads as: React.createElement("h2",{id:"header-id"}, . Line 25 reads as: "Welcome to the world of React!"));.* [Video description ends]

This div with the h2 header within it, I've assigned to the variable called el. I've then created another element using React.createElement, which is a paragraph element denoted using p. This paragraph element has the id, para-id, and it's contents are, React seems rather tedious to work with.

[Video description begins] *Line 27 reads as: var another\_el=React.createElement(. Line 28 reads as: "p",{id:"para-id"},. Line 29 reads as: "React seems rather tedious to work with. ");* [Video description ends]

And indeed, that might seem true to you when you look at all of these createElement calls, but don't worry, we'll improve upon this in just a bit. I've now set up a JavaScript list called element\_list, which includes el and another\_el.

[Video description begins] *Line 31 reads as: var element\_list=[el,another\_el];* [Video description ends]

That is our div with id, inner-div-id, and another\_el that is our paragraph HTML element. I'll now create yet another element using the React.createElement function. This is a div, once again, with the id, outer-div-id. And nested within this outer-div-id is the list of elements that we had created earlier.

[Video description begins] *Line 33 reads as: var outer\_div\_el=React.createElement("div",{id:"outer-div-id"},element\_list);* [Video description ends]

Once we have this nested structure of elements, I'll use ReactDOM.render in order to render the outer\_div. Once you render the outer\_div, all of the elements nested within this outer\_div will also be rendered. And this outer\_div, with all of its nested elements, will be rendered under the my-react-app top-level div.

[Video description begins] *She highlights the following code on line 35: ReactDOM.render(outer\_div\_el,document.getElementById("my-react-app"));* [Video description ends]

Once you've understood how this page is structured, make sure you save this HTML file. Right-click, select the option to copy the file path so that we can view this file within our Chrome browser window. Paste this path in, NestedElements.html.

And here is what the resulting HTML page looks like. We have a header which says, Welcome to the world of React! And then a paragraph which says, React seems rather tedious to work with.

Now, the way we had set up this web page, we had many nested elements, right-click and hit Inspect, and we'll use Chrome developer tools to observe what the HTML structure of this page looks like.

[Video description begins] *The HTML structure displays in the Chrome Developer Tools interface in a tab called Elements.* [Video description ends]

Go ahead and expand the header of this document. You can see we have the script tags here and our style elements.

[Video description begins] *The code in lines 3 and 4 display when she expands the head tag.* [Video description ends]

Open up the body tag and let's take a look at the nested structure of this page. We have the top-level div, called my-react-app, and if you open that up, you'll see we have the outer-div-id.

[Video description begins] *She highlights the following code: <divid="my-react-app">. The code in the next line reads as: <divid="outer-div-id"> ...</div>.* [Video description ends]

This outer-div is what we rendered into our top-level route div called my-react-app. Now expand this outer-div-id, and within that you should see the nested elements.

[Video description begins] *She highlights the following lines of code: <divid="inner-div-id">...</div> <p id="para-id">React seems rather tedious to work with.</p>.* [Video description ends]

Here we have two elements that we had set up in the form of a list, these are sibling elements. This is the div with the id, inner-div-id, and the paragraph with the id, para-id. If you now expand this inner-div-id, you'll find nested within it our h2 header, which says, Welcome to the world of React!

[Video description begins] *She highlights the following lines of code. <h2 id="header-id"> Welcome to the world of React! </h2>.* [Video description ends]

And this is the h2 to header with the id, header-id. This is the same nested structure that we had set up using the React.createElement function. And this script that contains the React JavaScript code that set up the structure is available here within this script tag that I've expanded and you can now view.

Let's head over to the Console, because it seems like there is an error on the Console. And an error is something that we ought to be worried about. If you look at this error message, you'll see that it's a warning from React. It says, each child in an array or iterator should have a unique key prop.

This is not something that we've seen before. And this is because we have sibling elements in our structure. This warning here refers to the two elements that we had set up in the element\_list. The div with id, inner-div-id, and the paragraph with id, para-id.

These two elements are sibling elements in our HTML tree structure, that is, they're at the same level at the tree. And they have the same parent. In order for React to optimize the rendering of elements that are siblings or at the same level of the tree, you need to specify unique key attributes associated with these elements.

The key attribute in React is something that you specify on HTML tags, and this allows the React library to uniquely identify elements to track which elements have changed, which have been removed or deleted, which have been added.

And React will ensure that only those elements are re-rendered in the browser. This is an optimization that React performs in order to ensure that your apps remain interactive by only updating those elements where updates are needed.

We'll discuss Keys in a lot more detail as we dive deeper into React. For now, let's see what we can do to fix this problem and have this warning go away. I'm now going to switch back to our HTML page, and I'm going to add in an additional key attribute to the two sibling elements in our HTML page structure.

[Video description begins] *She adds the following code in line 23: key: "uniq-react-key-1"* [Video description ends]

So we have the inner-div-id, which is one of the sibling elements. The unique key that I've assigned here is the uniq-react-key-1. The other element that requires this key attribute is our paragraph HTML element. And I'm going to assign it the key uniq-react-key-2.

[Video description begins] *She adds the following code in line 28: key: "uniq-react-key-2"}.* [Video description ends]

Both of these key attributes have special meaning for React libraries. They have no meaning as far as the HTML for the browser is concerned. The browser knows nothing about these attributes, but React uses these attributes to uniquely identify sibling elements to make updates more performant.

Let's switch back to our browser once again, where we had observed the error on the console log messages. I'm going to refresh this page. And this time, you'll observe that the error has disappeared. That's because we specified unique keys for sibling elements.

**Creating Nested Elements - Part 2**

[Video description begins] *Topic title: Creating Nested Elements-Part 2 . Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll continue working with nested elements in React. And we'll see how we can set up list elements using the React.create element function.

[Video description begins] *The Sublime text editor interface displays. It displays a blank "Lists.html" file in the working pane.* [Video description ends]

The folder that we are working under in the top level React directory is the CreatingElements folder. And the file that we are writing our code in is the Lists.html file. I am going to paste in the HTML code here, which includes the JavaScript containing the React functions. Once again, we'll work with react.development.js. I have a few CSS styles within the style tag.

[Video description begins] *She highlights the code in lines 7 to 9. Line 7 reads as: .my-fruits {. Line 8 reads as: color:blue;. Line 9 reads as: }.* [Video description ends]

And I have the top level div called my-react-app.

[Video description begins] *She highlights the following line of code on line 13:<divid="my-react-app">.* [Video description ends]

We'll now take a look at the JavaScript code which uses React in order to set up the HTML structure of this page. I use React.createElement in order to create a div tag.

[Video description begins] *Line 17 reads as: var el=React.createElement("div",{key:"el-key"},"Some of my favorite fruits");* [Video description ends]

I've specified a unique key for this div tag within curly braces. The key is called el-key. And the content of this div tag is simply text content which says, Some of my favorite fruits. Once I have this outer div element, I'm going to go ahead and create three list elements using the li tag.

There are three fruits that I include in my list of favorite fruits. I use React.createElement, create three li tags. Observe that each of these li tags have a unique key value, O-key, A-key, and B-key. The three li tags contain the fruits, Orange, Apple, and Banana.

[Video description begins] *She highlights the code from line 20 to 22. Line 20 reads as: var fr\_el1=React.createElement("li",{key:"O-key"},"Orange");. Line 21 reads as: var fr\_el2=React.createElement("li",{key:"A-key"},"Apple");. Line 22 reads as: var fr\_el3=React.createElement("li",{key:"B-key"},"Banana");* [Video description ends]

I now have a JavaScript list called element\_list containing the list of the four elements that I've created, the div element and the three list elements.

[Video description begins] *She highlights the following code on line 24: var element\_list=[el, fr\_el1, fr\_el2, fr\_el3];* [Video description ends]

I'll now create a list element using React.createElement. And I use the ul tag indicating that it's an unordered list.

[Video description begins] *She highlights the following line of code on line 26: var list\_el=React.createElement("ul",{className:"my-fruits"},element\_list)* [Video description ends]

I want this list to have a specific style so I use the className attribute to apply a CSS class to this ul list. Observe how this attribute is className rather than class. Class is a reserved keyword in JavaScript, which is why we need to specify this attribute as className.

And React will correctly assign the class HTML attribute to this tag. All of the four elements that we had created earlier become nested elements within this outer list element. And I use ReactDOM.render to render this list element to my HTML DOM.

[Video description begins] *She highlights the following line of code on line 28: ReactDOM.render(list\_el, document.getElementById("my-react-app"));* [Video description ends]

Make sure you save this HTML file and go ahead right-click and Copy File Path. We'll now paste this into a Chrome browser window in order to view what this page looks like. Remember, this contains an unordered list. And here you can see some of my favorite fruits, Orange, Apple, and Banana.

[Video description begins] *The browser displays a page with the URL /Users/loony/Project/CreatingElements/List.html* [Video description ends]

This is an unordered list, which is why my favorite fruits have bullets in front of them rather than numbers. If you want this to be an ordered list rather than an unordered list, we simply have to change the element that we create to hold our list. Instead of ul, we'll change this to be ol.

This will give us an ordered list. Now, if we go back to our HTML page and hit refresh, this unordered list will become an ordered list.

[Video description begins] *Numbers 1, 2, and 3 display against the three list items.* [Video description ends]

You know the drill already. We have Chrome Developer Tools open. Let's head over to the Elements tab here and let's expand the body tag. And under the body tag the div with id=my-react-app.

[Video description begins] *She highlights the following lines of code in the Chrome Developer tools window: < div id="my-react-app">.* [Video description ends]

Here is our ordered list with the class my-fruits applied. And if we expand this ordered list you'll see the three list items that we had set up, Orange, Apple, and Banana. Under the ol tag, we also have the div which says Some of my favorite fruits.

An important thing that you can observe here is the fact that under this ordered list the div tags and the li tags all have unique key attributes. The key attributes are not reflected in the HTML.

They are for the React library. React uses these keys behind the scenes to uniquely identify components to update. The keys don't become part of your mock-up.

[Video description begins] *The key elements display in lines 17, 20, 21, and 22 of Sublime editor.* [Video description ends]

**Introducing JSX**

[Video description begins] *Topic title: Introducing JSX. Your host for this session is Janani Ravi. Screen title: JSX* [Video description ends]

All of the demos that you've performed with React so far have used React.createElement(). And you must admit that it has been kind of tedious to create HTML elements by invoking this function each time. Specifying the attributes for each element is also difficult.

So in summary, React.createElement() is tedious to work with, and if this is what you had to continue to use, you would run away from React and never use it again. There are actually several disadvantages to working with React.createElement().

It's intuitive to understand because you know you're creating an HTML element at each step. But it's code that is hard to read, because you can't really see the HTML structure of the code. React.createElement() obscures how the HTML structure is set up.

Code like this is a nightmare to maintain. If you had to maintain all of those nested React.createElement() calls that we saw earlier. You'll keep running into very thorny issues that are hard to debug.

Errors will creep in, you will have no idea where they are. And you won't understand why your elements work the way they do. This is why React.createElement() is not really used when you write React code.

When you're working with React in the real world, you'll use an HTML like syntax to define the HTML content of your page. This HTML or XML like syntax is called JSX. The whole idea behind how HTML is set up and structured is that you should be able to see the hierarchy of elements that make up your DOM. JSX helps with exactly that.

JSX stands for JavaScript Syntax Extension. You can think of JSX as syntactic sugar, allowing you to define the HTML structure of your web page in a way that is easy to read and easy to maintain. This is a concise and familiar syntax to define the tree structure and the attributes of every node in your tree.

The whole thing about JSX is that it's an XML based specification. So, essentially, it looks very much like HTML, but there is an important difference. With JSX, you can specify user-defined tags as well, user-defined components.

JSX allows you to write HTML-like syntax within your JavaScript code, so that the HTML representation of your components can coexist with your JavaScript code. So while you're writing JavaScript, you don't have to switch over to another file to specify the HTML structure of your user interface.

JSX is seamlessly interleaved with your JavaScript code. Thus allowing the definition of your UI to coexist with the logic and behavior of your component, thus making your React components easier to manage overall.

[Video description begins] *Screen title: JavaScript Syntax Extension* [Video description ends]

When you're working with React, you'll use JSX to specify the HTML structure of your page. That is the default natural option. JavaScript Syntax Extension is Concise and familiar. Large trees representing your Document Object Model are intuitively represented with an XML like syntax.

JSX allows you to define not just HTML tags but also user-defined tags that represent your components. When you use JSX with React it makes your code easier to read and maintain.

In addition, JSX also maintains the semantics of JavaScript which means you can interleave JSX within JavaScript and everything will work just fine. Your code will be easier to follow.

**The Babel Compiler**

[Video description begins] *Topic title: The Babel Compiler. Your host for this session is Janani Ravi.* [Video description ends]

JavaScript Syntax Extension, or JSX, is very similar to HTML.

[Video description begins] *Screen title: What does JSX look like?* [Video description ends]

So if you quickly look at it, you might think it's raw HTML, but it isn't. It's actually written within script tags and interpreted as JavaScript. The way you'll typically work with JSX is that you'll be writing your code in JavaScript, and embedded within that will be some JSX representing your user interface.

Let's take a look at some example JSX here, embedded within our JavaScript. You can see that we have a some underscore list variable that is a JavaScript variable.

[Video description begins] *The first line of code reads as: var some\_list= (* [Video description ends]

This JavaScript variable is assigned some XML-like code. This XML-like code is JSX. In order to ensure that your syntax for JSX is correct, it's good practice to enclose your JSX within parenthesis followed by a semicolon.

[Video description begins] *The following code displays within parentheses: <ul id="nav-id"> <li>Red</li> <li>Green</li> </ul>* [Video description ends]

Here, you can see that our JSX represents an unordered list with id equal to nav-id. Within the unordered list, we have two li tags representing two list items. The first list item is the color Red. The second list item is the color Green.

This is JSX, not HTML. Observe how it's embedded within your JavaScript. Now, the browser does not understand JSX, which means any JSX you embed within your JavaScript has to be compiled or translated to a form that the browser understands.

So notice that the HTML elements are specified within JavaScript. That is, JSX is essentially a snippet of JavaScript. You write your logic in JavaScript, put in your if conditions, put in your for loops, and anything else that you need, and you'll simply include XML tags where you want some HTML to be represented.

When you write code in JSX, the browser has no idea what that code represents. The browser does not understand JSX! The browser knows how to display your HTML tree structure, knows how to apply CSS styles, the browser is able to understand JavaScript, that's all. There is nothing new that the browser needs to know.

This means that this bit of JSX code cannot run directly on the browser. All the browser expects is JavaScript within script tags or within your JS files. When it sees the XML representing JSX, it'll just barf, it won't know what to do.

In order to get the browser to understand the JSX that you've written, the JSX has to be transformed to JavaScript before the browser can understand and render the element that you represent.

So we need a transformation step to convert the JSX to a form that the browser can interpret, understand, and render. There is a special pre-compiler for exactly that. But before we talk about that, here is the equivalent JavaScript into which this JSX will be converted.

This conversion will be performed by a pre-compiler behind the scenes. So by the time the browser sees the code, it's raw JavaScript. So the variable some\_list is essentially assigned a React element using React.createElement().

[Video description begins] *The first line reads as: var some\_list=React.createElement(* [Video description ends]

The React.createElement() will create the unordered list tag, assigned the corresponding id attribute id is equal to nav-id, and there are nested React.createElements to generate the nested list items within the outer unordered list.

[Video description begins] *He refers to the following code: "ul", { id: "nav-id"}, React.createElement( "li" , null, "Red" ), React.createElement ( "li" , null, "Green" ) );* [Video description ends]

Any JSX that you write within your JavaScript code is actually compiled to React.createElement() calls under the hood. You won't actually see this compilation, but it has to occur before you can have your JavaScript run within your browser.

This transformation of JSX to function calls in JavaScript that a browser can understand is done by a special JavaScript compiler called Babel. This Babel compiler has to be included within your React application in the form of a library.

If you have any JavaScript code that contains embedded JSX, that code has to pass through a Babel compiler so that, that code can be converted to regular JavaScript that is understood by the browser.

The input that you pass into a Babel compiler is JSX. And the output of the Babel compiler is pure JavaScript, JavaScript functions and other definitions that a browser understands. Now, transforming JSX to pure JavaScript is, of course, very useful, but there actually exist another reason to use the Babel compiler with React.

Now, this goes back to what React supports. React is compatible with new features of ECMAScript6 or ES6. This is the latest version of JavaScript available at the time of this recording. All of the demos that we work on use the ES6 syntax for React.

Now, unfortunately, there exist older browsers out there in the real world which do not yet support ES6. You want to use new features of React, you want to use ES6. But if there are browsers out there which don't support ES6, that can be a major pain, and this is where Babel is so useful.

Babel can be used to convert ES6 code to ES5 and older versions so that, that code is compatible with all browsers. To round off our discussion on Babel, you should know that the Babel compiler comes in two flavors, it's available for development as well as for production.

[Video description begins] *Screen title: Babel Transforms JSX to JavaScript in Two Ways* [Video description ends]

So Babel Transforms JSX to JavaScript in Two Ways. One transformation is On the fly within the browser. This On the fly compilation allows for rapid prototyping of your JavaScript code, and this is what you would prefer during development.

This involves simply including the Babel compiler as a script tag reference in your HTML. Just before your JavaScript is read by the browser, the Babel compiler will run, convert JSX to JavaScript so that the browser can understand JavaScript.

So it's done on the fly during runtime. While this is fine during development where the loading time of your app does not matter and performance does not really matter, it's not really great for production environments.

Which is why the Babel transformer transforms JSX at build time as well. This is what you'd use for production environments. In production environments, when you minify and productionize your JavaScript and CSS to improve the load time performance of your app, the Babel compiler will run at build time to convert JSX to JavaScript. The conversion will not be done on the fly.

[Video description begins] *Screen title: Babel for Development and Production* [Video description ends]

So Babel for Development and Production. Let's quickly summarize what we just discussed. Babel will run on the JavaScript just before the browser renders the page. This is a pre-processing step.

Babel will check for JSX elements within your JavaScript and convert that JSX to pure JavaScript, JavaScript that the browser can understand. Now, doing this on the fly while your application runs is inefficient and a major performance hit.

This on the fly transformation of your JSX code to JavaScript is only done in development environment for rapid prototyping. It's never used in production because of what a great impact it has on performance.

When you want something working quickly, well, on the fly transformation works well. When you want to use Babel in a production environment, you should run it as a pre-compiler. Babel will pre-process and compile the JSX to JavaScript during built-time and not at run time.

**Working with JSX and the Babel Compiler**

[Video description begins] *Topic title: Working with JSX and the Babel Compiler. Your host for this session is Janani Ravi.* [Video description ends]

If it crossed your mind that all of the React code that we've written so far has been rather tedious, you won't be wrong. That's because we haven't harnessed the additional powerful functionality of React. We haven't even started with components yet. But before we get to components, let's work with JSX or JavaScript extensions.

[Video description begins] *The Sublime text editor interface displays. It displays a file called "SimpleJSX.html" in the working pane.* [Video description ends]

JSX is simply a React extension that allows us to write JavaScript that looks like HTML. Now a good question to ask right now would be, why do we even need this? If you remember, we had a bunch of clunky code, react.createElements in order to set up the HTML structure of our page.

Instead of using React.createElement(), it's far better to specify HTML directly within our JavaScript. And that's exactly what JSX helps us do. With JSX you avoid the very hard to read React.createElement() calls that we saw earlier.

In addition, you specify the structure of your HTML page within your JavaScript file. So it's much more easy and intuitive to work with. Let's take a look at a very simple webpage here where we specify the HTML structure for the webpage using JSX rather than React.createElement() calls.

If you're working with JSX, it's useful for you to go ahead and install additional extensions to your Sublime text editor so that you get syntax highlighting for your JSX code.

In order to do this I'm going to hit cmd+shft+p, it's Ctrl+Shift+P, if you're working on a Windows machine, in order to bring up the additional packages that can be installed with Sublime Text. cmd+shift+p will bring up a little pop up here and you can type within that pop up what you are looking for is the Install Package Control.

Select Install Package Control and this will bring up an additional pop up. Click on OK, Package Control was successfully installed. And this will bring up an additional search box where you can search for the specific packages that you want to install to get additional syntax highlighting.

Head over to Package Control: Install Package. And once you've selected that, let's go ahead and install the Babel Snippets package. Select Babel Snippets, bring up the search box once again by hitting cmd+shift+p, and install the Babel package as well.

These two packages for Sublime text, Babel and Babel Snippets, are required for syntax highlighting all of your JSX code. This is the HTML code embedded within your JavaScript. I'll discuss what Babel is in just a bit. I haven't explained what this code does yet but syntax highlighting hasn't yet been enabled. For that, we need to perform an additional action.

At the bottom right, you'll see a message that says HTML. This basically indicates that Sublime text considers this file to be an HTML file. We need to change this and indicate that this is a JSX file. Click on HTML and then choose the option which says Open all with current extension as.

And in the submenu that pops up, select Babel & then select JavaScript(Babel). All HTML files will now be considered to be JavaScript(Babel) files. And you can see that we immediately get syntax highlighting.

Now that we have the syntax highlighting setup we can take a look at what exactly is going on in the code here where we use JSX. We'll first take a look at what JSX looks like before we move on to the other features that are interesting here.

Look at the ReactDOM.render() function that we've invoked. Instead of passing in a JavaScript element, we have specified something that looks very much like HTML. We have an h1 tag and the contents of this h1 tag says Welcome to the world of React! It might seem strange to you to have HTML-like code embedded within your JavaScript, but this is JSX.

You can specify your HTML elements in a very HTML-like form, rather than using React.createElements(). This h1 element will be rendered within your my-react-app top level div.

[Video description begins] *She highlights the following line of code on line 8: <div id="my-react-app">.* [Video description ends]

The question to ask yourself here now is, how does the browser recognize that this is JSX and how does it know that this JSX is actually just JavaScript under the hood? It's not the browser which recognizes JSX, there is a special compiler that you need to use.

And you know the compiler will run when you specify a special attribute on the script tag that contains JSX.

[Video description begins] *She highlights the following line of code on line 11: <script type="text/babel">.* [Video description ends]

Observe that we have specified a type attribute for this script tag indicating that the type here is text/babel. This is needed, indicating that it's not just pure JavaScript that's going to be written within this script tag, we're going to be writing JSX.

And JSX requires to be compiled using the Babel compiler. Babel is a JavaScript compiler that compiles JSX into regular JavaScript. This attribute type is equal to text/babel, tells the Babel compiler that there is some JSX code within the script tag which needs to be compiled. And where will we get the Babel compiler from?

Take a look at our script tag references in the header of our HTML page. Observe we now have a script tag reference to unpackage.com/babel-standalone@6.26.0/babel.js.

[Video description begins] *She highlights the following line of code on line 6: <script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>* [Video description ends]

This is a standalone version of the Babel compiler that will compile our JSX into regular JavaScript. This Babel compiler will look for all script tags with type text/babel, find the JSX within the JavaScript, and convert that JSX to just regular JavaScript.

[Video description begins] *She highlights the following code: <script type="text/babel">ReactDOM.render( <h1>Welcome to the world of React!</h1>, document.getElementById("my-react-app") ); </script>* [Video description ends]

In summary, JSX allows you to write HTML-like code within your JavaScript. JSX needs to be compiled to regular JavaScript before it's understood by the browser, and this compilation is done using Babel.

And you specify to Babel that you have JSX code by specifying the type attribute on your script tag, type = "text/babel". This will invoke the Babel compiler that we've added as a script tag reference to the unpackage content delivery network, and convert this JSX to JavaScript.

I'm going to right-click, copy the file path over, and paste it into the browser window in order to see what this page looks like. Go to SimpleJSX.html. And here we have it, our simply header here, Welcome to the world of React!

As we've done in other demos, I'm going to open up Chrome Developer Tools and expand the HTML structure of my page so that I can verify for myself that the page is set up using JSX correctly. I'm going to expand the body tag here.

Here is my top-level div called my-react-app. I'm going to expand this div, and under there you'll see the h1 header that we had set up using JSX. If you expand the script tag element here, you'll see our JavaScript with Babel. ReactDOM.render(), takes in an XML-like specification for this h1 header that was rendered.

Now, I don't know how many of you noticed, there seems to be a warning in our console messages. You can see one next to the warning sign at the top-right of your screen. Warnings are never a good thing. Let's head over to the Console tab and see what this warning is about.

And you can see that this warning tells us that we're using the in-browser Babel transformer. What we're doing right now is on the fly, compiling our JSX into JavaScript using the in-browser Babel transformer.

When you're running in production, doing this on the fly adds a performance overhead because on the fly you're converting JSX to JavaScript. Your JSX should actually be pre-compiled into JavaScript before you run your code.

When you use the correct create React app environment in production, this happens automatically under the hood. Your JSX is compiled to JavaScript in the pre-compile phase. This transformation is not done on the fly in production.

**More Practice with JSX**

[Video description begins] *Topic title: More Practice with JSX. Your host for this session is Janani Ravi.* [Video description ends]

For this demo, we move on to a new HTML file called ListsJSX.html under the folder called JSXBabel. We've seen in an earlier demo that JSX is a React extension that allows us to write HTML code within our JavaScript code. And JSX is compiled to JavaScript using the Babel compiler.

This is what makes working with React so simple, along with other things. Let's work with JSX further and build up more complex structures in JSX. In order to use JSX, we'll continue to use the in-browser Babel compiler. Make sure you have a script tag that references babel.js hosted at the unpkg content delivery network.

[Video description begins] *She highlights the following line of code on line 7: <script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>.* [Video description ends]

If you scroll down and look at this JSX embedded within our JavaScript here, you can see that the syntax highlighting isn't present, because by default, Sublime Text assumes that this is an HTML file.

So I'm going to select HTML here at the bottom right of my screen. And then choose the option to open all with current extension as Babel, JavaScript (Babel). Now that you're familiar with this, you'll know to do this for your future HTML files as well.

Sublime Text should open up HTML files by default as Babel files, though sometimes it doesn't work and you may need to specify this explicitly. Now that we have the syntax highlighting, everything looks much nicer.

And let's take a look at the JSX that defines the HTML structure for this page. Within ReactDOM.render(), we have an outer div. And within that, we have an inner div with the key el-key.

[Video description begins] *She highlights the following code: ReactDOM.render( <div><div key="el-key" className="my-fruit">Some of my favorite fruits </div>.* [Video description ends]

We've applied a CSS class to this inner div, where the class name is my-fruit. Observe that the attribute is className and not class, because class is a reserved keyword. The contents of this div says Some of my favorite fruits. And then we have another sibling element to this div, which is a ul HTML tag. This is an unordered list.

[Video description begins] *She highlights the following code: <ul> <li key="O-key">Orange</li> <li key="A-key">Apple</li> <li key="B-key">Banana</li> </ul> .* [Video description ends]

And this ul tag contains three list items. The three list items contain my favorite fruit, Orange, Apple, and Banana. Each list item is associated with the unique key that React needs O-key, A-key, and B-key. And this entire HTML that we have specified in JSX is rendered into the top-level div with id my-react-app.

[Video description begins] *She highlights the following line of code on line 31: document.getElementById("my-react-app").* [Video description ends]

You can immediately observe how much easier it is to work with JSX, rather than with React.createElement() calls. React.createElement() obscured the structure of our HTML, whereas JSX makes everything very clear.

An important thing to note when we use JSX along with ReactDOM.render(), you can have just one top-level HTML tag. And then all of your other HTML should be nested within this tag. We have a top-level div, which has no other attribute, which encompasses all the rest of our HTML.

This is important. Make sure you save this HTML file, right-click, copy the file path over, and paste it into your browser window. This will now use the in-browser Babel compiler to render the right HTML onto screen.

Some of my favorite fruits are these, present in the form of an unordered list. You can always use Chrome Developer Tools to verify that the HTML structure of the webpage is what you expect, based on the HTML that you specified using JSX.

The head element contains our script tag references to React libraries as well as to the Babel compiler. And within the body element, we have all of the elements that are rendered within our browser. We have our top-level div called my-react-app.

And if you expand this div, you'll find all of the nested structure that we set up using JSX. We have an outer div, and if you expand this outer div, you'll view the div which have the class name applied, my-fruit, and you have the unordered list.

[Video description begins] *She highlights the following code: <div class= "my-fruit">Some of my favorite fruits </div> <ul>...</ul>* [Video description ends]

Everything's exactly what we expect. We don't get a warning about the keys for our elements, because all sibling elements have unique keys to identify them. The keys are not rendered within the HTML structure of the browser, though. The keys, though, are used by React under the hood. And here are the key specifications within our JavaScript.

**Simple Expressions with JSX**

[Video description begins] *Topic title: Simple Expressions with JSX. Your host for this session is Janani Ravi.* [Video description ends]

So far, we've only scratched the surface of what is possible to do with JSX. In this demo, we'll go a little further and we'll see how we can use JSX with simple expressions.

[Video description begins] *The Sublime text editor interface displays. It displays a file called "SimpleExpressionsJSX.html" in the working pane.* [Video description ends]

Here we are in a file called SimpleExpressionsJSX.html. Under the folder JSXBabel. I'm going to paste in the code for this particular demo. We'll then examine at leisure what exactly is going on. Make sure you reference babel.js so we can use the Babel compiler and JSX.

[Video description begins] *She highlights the following line of code on line 6: <script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>.* [Video description ends]

All of our JavaScript code is written within the script tag where the type is text/babel. This is important to let Babel know which script tag needs to be compiled using Babel. So far the JSX that we had written contained html which was static in nature, it did not include any expressions.

That is what we are about to change. I have declared three string constants here within JavaScript, holding the names of my three favorite fruits, Orange, Apple and Banana.

[Video description begins] *She highlights the following code: const orange\_str="Orange";const apple\_str="Apple"; const banana\_str="Banana";* [Video description ends]

I now have another constant called fruit\_list which contain the reference to my JSX code.

[Video description begins] *She highlights the following section of code on line 23: const fruit\_list.* [Video description ends]

So I'm not writing JSX directly within the ReactDOM.render() function. I'm going to assign this JSX to a JavaScript variable, or rather a JavaScript constant in this case. fruit\_list is the name of my constant, and observe that I've enclosed all of my JSX code within parentheses.

This is good form. This ensures that you don't make errors with your JSX. You'll see that when we write code in React, it's good practice to always enclose our JSX within parantheses. So all of the JSX is clearly structured in one place.

The parentheses are not really needed but they help improve the readability and maintainability of such code where we have XML embedded within our JavaScript. All right, let's take a look at our JSX, we have the outer div. This is the same outer div that we had in an earlier demo.

[Video description begins] *She highlights the following line of code on line 25: <div key="el-key" className="my-fruit">Some of my favorite fruits</div>.* [Video description ends]

The structure is exactly the same, we then have a div with the key el-key, then we have an un-ordered list, that is the ul tag. And within that we have three list elements.

[Video description begins] *She highlights the following code: <li key="O-key">{orange\_str}</li> <li key="A-key">{apple\_str}</li> <li key="B-key">{banana\_str}</li>.* [Video description ends]

Each list element has its own unique key, O-key, A-key and B-key. All of this is stuff that we already know, what's different here is how we have specified the contents of each of these li elements.

Observe the first li element, we've used curly braces here and within curly braces, we've referenced our JavaScript constant called orange\_str. So if you want to specify JavaScript expressions within JSX, you'll need to specify the curly braces and then add in your JavaScript in there.

The curly braces indicates to the Babel compiler that this is a JavaScript expression and should be evaluated as such. Each li element contains a JavaScript expression which references the constants that we set up earlier, apple\_str and banana\_str. Referencing a JavaScript variable or constant is the simplest possible expression that we can embed within our JSX. We then use ReactDOM.render() and render out this fruit\_list.

[Video description begins] *She highlights the following line of code on line 34: ReactDOM.render(fruit\_list,document.getElementById("my-react-app"));* [Video description ends]

And exactly like we've done in earlier demo, let's view this HTML file within a browser window. Observe that we have our three favorite fruits listed out within an unordered list. JSX correctly knew how to evaluate the expression within our li tag, and we can inspect this using Chrome browser tools.

[Video description begins] *She right-clicks on the browser interface and selects an option called "Inspect" from the context menu.* [Video description ends]

Exactly as we've done before, let's explore and expand the HTML structure of this page. I'm going to expand this body tag, within that we have the my-react-app top-level div.

[Video description begins] *She highlights the following line of code: <divid="my-react-app">...</div>.* [Video description ends]

I'm going to further expand this, we then have the div that we specified in JSX. I'm going to further expand this and here is our list.

[Video description begins] *She highlights the following line of code: <ul>...</ul>.* [Video description ends]

If you expand this unordered list and take a look at the li elements within this list, you can see that the expression within JSX has been evaluated correctly. And you get the values held in the constants that we had specified within the expression.

The value of the constant, orange\_str is Orange, apple\_str is Apple and banana\_str is Banana. Just like you can include or embed JSX within JavaScript, you can evaluate JavaScript expressions within JSX.

**More Expressions with JSX**

[Video description begins] *Topic title: More Expressions with JSX. Your host for this session is Janani Ravi.* [Video description ends]

In the last demo, we were introduced to simple expressions within JSX, which allowed JSX to reference JavaScript code. In this demo, we'll get a little more complex.

[Video description begins] *The Sublime text editor interface displays.* [Video description ends]

We'll explore more expressions with JSX. We'll be working in this HTML file called MoreExpressionsJSX.html. In this demo, we'll build a slightly fancier app in that it'll have an image embedded. Within the JSX Babel directory I have an images subfolder and within this images subfolder I have a fruit.jpg file.

This is the file that we'll reference in our JavaScript code. So let's take a look at what this image looks like. It just has a bunch of fruits and goes along with the fruit motif of our HTML file. Now that we know where this image is located and we have viewed the file, we can move on to writing our JavaScript code JSX with more complex expressions.

Head over to MoreExpressionsJSX.html. And I'm first going to paste in the code and we'll then examine this code in detail. The HTML for this page will once again display a list of fruits but with a few tweaks.

Observe that I have specified 3 JavaScript constants called orange\_str, apple\_str and banana\_str assigned to Orange, Apple and Banana at the very top. This is no different from a previous example.

[Video description begins] *She highlights the following code: const orange\_str="Orange"; const apple\_str="Apple"; const banana\_str="Banana";* [Video description ends]

But I have an additional JavaScript function here.

[Video description begins] *She highlights the following line of code on line 24: function pick\_fruit(index).* [Video description ends]

This function called pick\_fruit() takes as an input argument an index and based on the value of this index will return the fruit that you're interested in.

[Video description begins] *She highlights the following lines of code from line 25 to line 27: if(index==1){ Line 26 reads: return orange\_str; }.* [Video description ends]

If index is equal to 1 it'll return orange\_str, if index is equal to 2 it will return apple\_str. And if index is equal to 3, it will return the banana string.

[Video description begins] *She highlights the following lines of code from line 28 to line 30: if(index==2){ return apple\_str; }. Then, she highlights the following lines of code from line 31 to line 33: if(index==3){ Line 32 reads: return banana\_str; }.* [Video description ends]

If the value of the index input argument is something other than any of these, that is 1, 2 or 3, we'll return a bold tag saying Invalid fruit! Observe how we can return JSX from within a function as well.

[Video description begins] *She highlights the following line of code on line 35: return (<b>Invalid fruit!</b>).* [Video description ends]

It's good practice to enclose all of your JSX within parentheses, so there is no ambiguity about where the JSX ends. So we'll say Invalid fruit! when the index is something other than 1, 2, and 3. This is our pick\_fruit() function. I've then set up another constant called fruit\_img, which references the fruit.jpg file within the images subfolder under the current working directory.

[Video description begins] *She highlights the following line of code on line 38: const fruit\_img='images/fruit.jpg' .* [Video description ends]

I'm now going to set up a JSX element referencing this fruit image. So I have an image\_el constant which I assign img src is equal to and within curly braces, I have specified an expression. This expression references the constant fruit\_img.

[Video description begins] *She highlights the following line of code on line 41: const image\_el=<img src={fruit\_img}/>.* [Video description ends]

The source attribute of the img tag here references the fruit.jpg file under the images sub folder. I'll now scroll down a little further and let's take a look at this constant called fruit\_list. This contains the bulk of my JSX.

If this, JSX looks very similar to the JSX that we've already worked with. What is different though, is how the individual li elements have been specified. Observe the li tags, there are four of them. The contents of each of these li tags specified in JSX is evaluated using an expression.

[Video description begins] *She highlights the following lines of code from line 47 to line 50. Line 47 reads as: <li key="O-key">{pick\_fruits(1)}</li> Line 48 reads as: <li key="A-key">{pick\_fruits(2)}</li>Line 49 reads as: <li key="B-key">{pick\_fruits(3)}</li> Line 50 reads as: <li key="N-key">{pick\_fruits(4)}</li> .* [Video description ends]

 We know it's an expression because we've specified it within curly brackets. And the expression invokes the JavaScript function that we had defined earlier. So the first li tag invokes the pick\_fruit() function and passes in as an input argument the index 1.

This will pick the first fruit, that is Orange. The second li tag invokes the pick\_fruit function() and passes in as an index 2. The third li tag invokes pick\_fruit() and passes in as an index 3. And finally, the fourth li tag invokes pick\_fruit() and passes in as an index 4.

Remember, that's an invalid index. Expressions need not just reference pure JavaScript. Expressions can reference JSX as well. Just under the closing ul tag, you can see that we have an expression within curly braces which references image\_el. And image\_el you can see here on screen is our image tag.

[Video description begins] *She highlights the following line of code on line 52: {image\_el}.* [Video description ends]

And finally we have the ReactDOM.render function, which renders this fruit\_list to our DOM.

[Video description begins] *She highlights the following line of code on line 56: ReactDOM.render(fruit\_list,document.getElementById("my-react-app"));* [Video description ends]

As usual, make sure you save the contents of this HTML file, right-click Copy File Path and paste it into your favorite browser. For me, it's Chrome. And you can see our nice HTML page rendered here to screen.

You can see that the first three bullets are Orange, Apple and Banana and the fourth bullet says Invalid fruit!. If you remember we had invoked this using an index of four which is invalid. Once again in order to get a grasp of how JSX has set up the HTML structure of this page.

[Video description begins] *She right-clicks on the browser window and a context menu appears from where she selects the "Inspect" option.* [Video description ends]

I'm going to head over to Chrome developer tools and expand the body which contains the HTML structure. Expand the ul tag here and let's take a look at our list contents. Every element here except the last element has a valid fruit.

Observe that there is no reference to the pick\_fruit() function anymore. That has been compiled away by the Babel compiler and replaced with the result of invoking pick\_fruit(). The very last li element here, which contains Invalid fruit! is the result of invoking pick\_fruit() with the index four.

Observe that Invalid fruit! is rendered within a bold tag, exactly like we had specified within the pick\_fruit() function. And here is the img tag that we had also rendered using an expression.

**Course Summary**

[Video description begins] *Topic title: Course Summary* [Video description ends]

In this course, we introduced the React library for component-driven development. We explored basic principles involved in creating React applications and explained important React concepts and features, such as the Virtual DOM and JSX.

We discussed how the Virtual DOM helps React improve the performance of rendering DOM elements by updating only those properties that have actually changed. The Virtual DOM is the foundational design concept on which React is built.

We implemented a simple web page using React APIs, and then recognized the basic components of the web page built using React. We also understood the functionality of the Babel Compiler when used with JSX. We created elements using JSX and the Babel compiler.

And performed various operations including rendering a DOM hierarchy, evaluating expressions, and referring to JavaScript variables from within JSX. Now that we know the basics of React, we can now move on to installing React libraries on our local machines using NPM and running React applications on online playgrounds.