REACT

REACT FUNDAMENTALS

Naveen Pete

Agenda

- Introduction to React
- Create React App
- Rendering UI
- JSX
- Components
- Props
- Functional Components
- Rendering Components
- Composing Components
- Managing State
- PropTypes
- Component Lifecycle Events
- Handling Events
- Conditional Rendering
- Handling Lists
- Controlled Components
- Building SPAs in React
- Q & A

Introduction to React

- A JavaScript library for building UI
- Used to produce HTML that is shown to an user in a Web browser
- Current version – v16.2

Introduction to React

- Composition
  - Combine simple functions to build complex functions
  - A good function should follow the DOT rule
    - Do One Thing
  - React makes use of composition, heavily!
  - React builds up pieces of a UI using components
  - Components
    - Key building block in React
    - Pieces of UI

```
<Page />
<Header />
<Article />
<Footer />
```

    - Combine simple components together, create a complex component

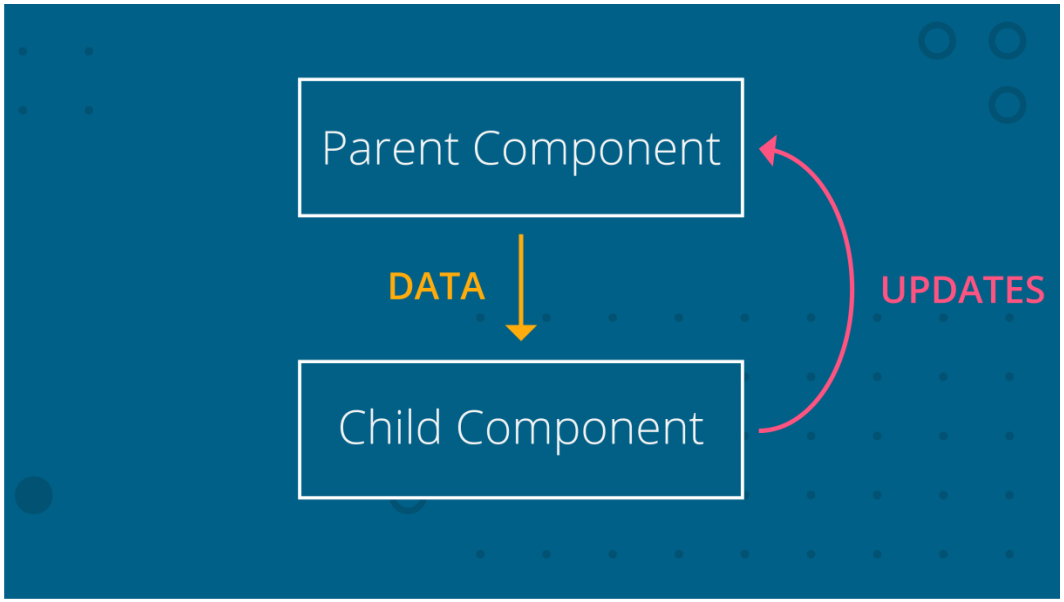
```
<Page>
  <Header />
  <Article />
  <Footer />
</Page>
```

Introduction to React

- React is Declarative
  - Imperative Code
    - Expressing a command; commanding
    - Instructs JavaScript on *HOW* it should perform each step
  - Declarative Code
    - We tell JavaScript *WHAT* we want to be done
    - Let JavaScript take care of performing the steps
- React is just JavaScript
  - React builds on JavaScript
  - No need to learn new way of doing things

Introduction to React

- Unidirectional Data flow
  - Data lives in the parent component
  - Data flows from parent component to child component
  - Data updates are sent to the parent component
  - Parent performs the actual change



Create React App

- Command line tool that scaffolds a React app
- Helps to build React SPAs
- Sets up dev environment
  - Latest JS features
  - Optimizes your app for production
- Uses build tools like Babel and Webpack
- Works with zero configuration

```
npm install -g create-react-app
create-react-app my-app
cd my-app
npm start
```

Rendering UI

- React uses JavaScript objects to build the UI
- React elements
  - Light weight JavaScript objects
  - Used to describe what the page should look like and let React do DOM manipulation

- `React.createElement(<type>, <props>, <content>)`
  - <type> - either a string or a React Component
  - <props> - either null or a JavaScript object, an object of HTML attributes and custom data about the element
  - <content> - null, a string, a React Element, or a React Component

- `ReactDOM.render(<element>, <container>)`
  - <element> - a React Element or Component
  - <container> - a DOM node

Rendering UI

- Nesting React Elements

```
const element = React.createElement('ol', null,
   React.createElement('li', null, 'Hari'),
   React.createElement('li', null, 'Krishna'),
);
```

  - React.createElement() returns *one* root element

JSX

- Syntax extension to JavaScript
- Lets us write JS code that looks like HTML
- More concise and easier to follow
- Returns *one* root element

  - Example #1

```
const element = <ol>
    <li>Hari</li>
    <li>Krishna</li>
</ol>;
```

  - Example #2

```
const people = [
    { name: 'Hari' },
    { name: 'Krishna' }
];
const element = <ol>
    {people.map(person => (
        <li key={person.name}>{person.name}</li>
    ))}
</ol>;
```

Components

- Key feature of React
- Encapsulate UI elements, data and the behavior of a view
- Allows you to break a complex web page into smaller, manageable & reusable parts
- Help us create our own custom elements
- Group many elements together and use them as if they were one element
- Classes that extend React.Component
  - Only method that is required in a Component is render()
  - Responsible for returning HTML to be rendered onto the page

Components

- Clear responsibilities
  - *Single Responsibility Principle*, just “DOT”
- Well defined interfaces
- Benefits
  - Code is modular and reusable
  - Ability to configure components with different props
  - Configure components independently

Props

- A prop is any input that you pass to a React component
- Added just like an HTML attribute
  - prop name and value are added to the component
- Props are stored on the ‘this.props’ object
- Props are read-only - a component must never modify its own props

```
// passing a prop to a component
<Welcome name=‘Hari’ />
```

```
// access the prop inside the component
...
render() {
   return <h1>Hello, {this.props.name}</h1>
}
...
```

Functional Components

- A function that
  - accepts a single ‘props’ object argument
  - returns description of UI (React element)
  - has no ‘this’ keyword
  - does not keep track of internal state

```
function Email(props) {
  return <div>{props.text}</div>;
}
```

```
class Email extends React.Component {
  render() {
    return <div>{this.props.text}</div>;
  }
}
```

Rendering Components

- Following code renders Welcome component

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Hari" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Composing Components

- Components can refer to other components in their output

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Hari" />
      <Welcome name="Krish" />
      <Welcome name="Shiv" />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Managing State

- Props
  - refer to attributes from parent components
  - represent "read-only" data that are immutable

- State
  - represents mutable data
  - affects what is rendered on the page
  - managed internally by the component itself
  - is meant to change over time, commonly due to user input
  - is a plain JavaScript object that is used to record and react to user events

Managing State

- Every class based component has its own state object
- Whenever a component state is changed, the component immediately re-renders and also forces all of its children to re-render as well

Managing State

```
class User extends React.Component {
  state = {
    username: 'Hari'
  }

  render() {
    return (
      <div>Username: {this.state.username}</div>
    );
  }
}

ReactDOM.render(
  <User />,
  document.getElementById('root')
);
```

Managing State

```
class User extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      username: 'Hari'
    }
  }

  render() {
    return (
      <div>Username: {this.state.username}</div>
    );
  }
}

ReactDOM.render(
  <User />,
  document.getElementById('root')
);
```

Managing State

- A component may update its state using ‘this.setState()’ method
- Whenever setState() is called, React, by default, re-renders the entire app and updates the UI
- Component’s UI is just a function of component state

```
UI = fn(state)
```

Managing State

- There are two ways to use setState()

```
this.setState({
  subject: 'Hello! This is a new subject'
})
```

- When a component's new state depends on the previous state, we can use the functional setState()

```
this.setState((prevState) => ({
  count: prevState.count + 1
}))
```

Managing State

- Important
  - Do not modify the state directly
  - State Updates May Be Asynchronous
  - State Updates are Merged

PropTypes

- PropTypes
  - is a package that lets us define the data type of props for a component
    - npm install --save prop-types
  - can be used to make sure the data you receive is valid

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

Component Lifecycle Events

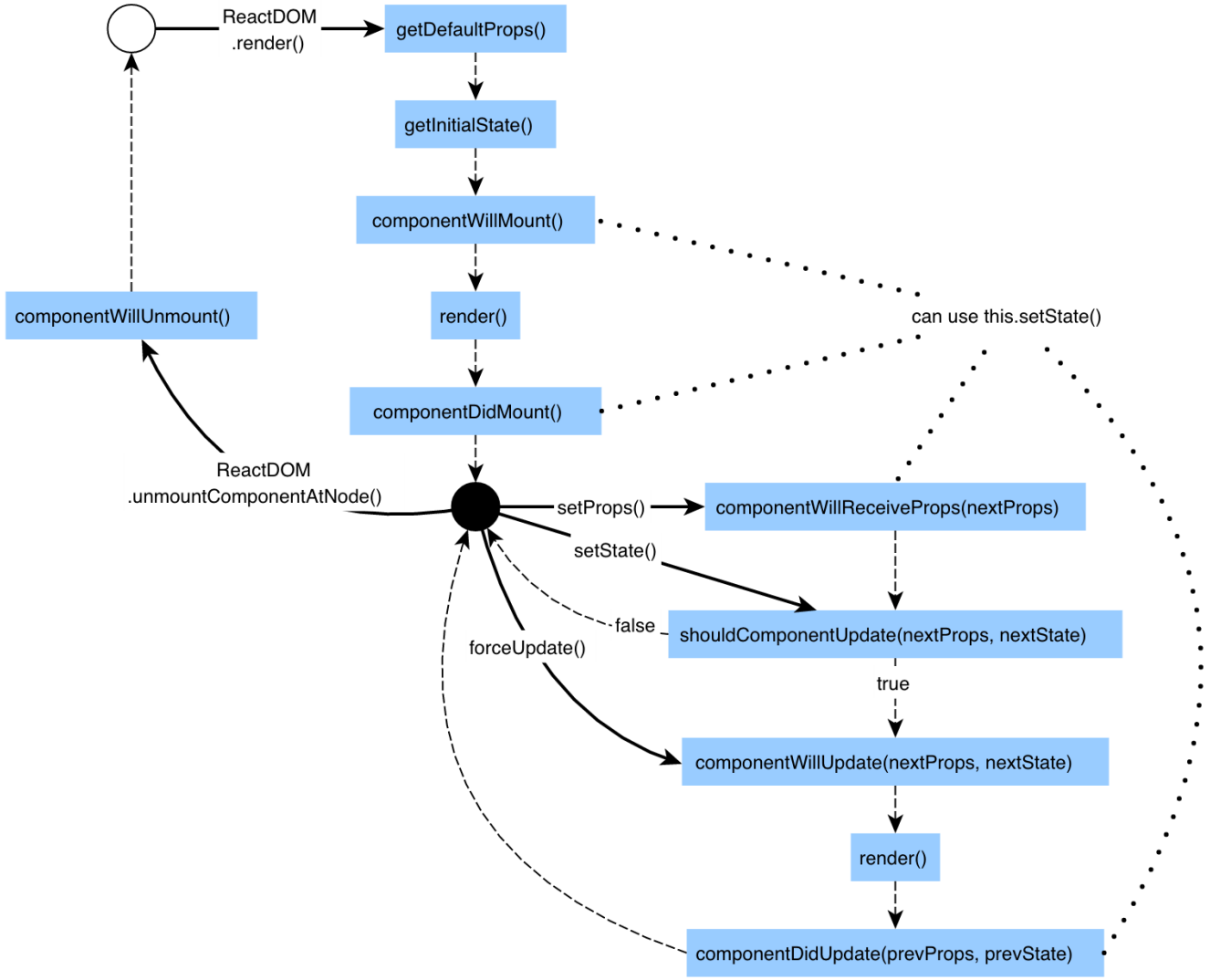
- render() method should
  - only be used for displaying the content
  - not make any AJAX (HTTP) requests
  - not alter the DOM
- We put the code that should handle things like AJAX requests in lifecycle events

- Lifecycle Events
  - specially named methods in a component
  - automatically bound to component instance
  - React will call these methods at certain times during the life of a component

Component Lifecycle Events

- componentWillMount()
  - invoked immediately before the component is inserted into the DOM
- componentDidMount()
  - invoked immediately after the component is inserted into the DOM
- componentWillUnmount()
  - invoked immediately before a component is removed from the DOM
- componentWillReceiveProps()
  - invoked whenever the component is about to receive brand new props

Component Lifecycle Events



Handling Events

- React events are named using camelCase
  - E.g. onClick, onChange, onSubmit
- With JSX you pass a function as the event handler, rather than a string.

```
<form onSubmit={this.handleSubmit}>
...
</form>


<input type="text" className="form-control"
   id="productName" name="name" value={product.name}
   onChange={this.handleChange} />
```

Handling Events

- To prevent default behavior, you must call preventDefault() function explicitly

```
<form onSubmit={this.handleSubmit}>
...
</form>

handleSubmit(event) {
   event.preventDefault();

   // Continue with form submission logic
}
```

- In the above example, ‘event’ is a SyntheticEvent instance
  - a cross-browser wrapper around the browser’s native event

Handling Events

- To prevent default behavior, you must call preventDefault() function explicitly

```
<form onSubmit={this.handleSubmit}>
...
</form>

handleSubmit(event) {
   event.preventDefault();

   // Continue with form submission logic
}
```

- In the above example, ‘event’ is a SyntheticEvent instance
  - a cross-browser wrapper around the browser’s native event

Handling Events

- An event handler is a method on the component class
  - In JavaScript, class methods are not bound by default
  - If you refer to a method without () after it, you should bind that method in the constructor

```
class ProductForm extends Component {
  constructor(props) {
    super(props);

    this.handleSubmit = this.handleSubmit.bind(this);
  }

  render() {
    return(
      ...
      <form onSubmit={this.handleSubmit}>
      ...
    );
  }
}
```

Handling Events

- You can also use an arrow function in the callback

```
class ProductForm extends Component {
  constructor(props) {
    super(props);
  }

  handleSubmit(e) { ... }

  render() {
    return(
      ...
      <form onSubmit={e => this.handleSubmit(e)}>
      ...
    );
  }
}
```

- **Note:** Binding in the constructor is generally recommended as arrow function method has a performance problem

Conditional Rendering

- Works the same way conditions work in JavaScript
- Use JavaScript operators like ‘if’ or the ‘conditional’ (ternary) operator

```
function UserGreeting(props) { ... }

function GuestGreeting(props) { ... }

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

Conditional Rendering

- You can use variables to store elements

```
render() {
  const isLoggedIn = this.state.isLoggedIn;

  let button = null;
  if (isLoggedIn) {
    button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={this.handleLoginClick} />;
  }

  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {button}
    </div>
  );
}
```

Conditional Rendering

- You may embed any expressions in JSX by wrapping them in curly braces

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

Conditional Rendering

- The JavaScript conditional operator can also be used to conditionally render elements or components

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Handling Lists

- Basic List Component
  - Following component accepts an array of numbers and outputs an unordered list of elements using JavaScript map() function

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

Handling Lists

- Keys
  - Help React identify which items have changed, are added, or are removed
  - Should be given to the elements inside the array to give the elements a stable identity
  - Use a string that uniquely identifies a list item among its siblings

  - **Note:** Using indexes for keys can negatively impact performance and may cause issues with component state

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Handling Lists

- Embedding map() in JSX
  - JSX allows embedding expressions in curly braces so we could inline the map() result

```
const ProductList = ({ products }) => {
  return (
    <div>
      <div className="list-group">
        {products.map(product => (
          <Link
            key={product.id}
            to={`/products/${product.id}`}
            className="list-group-item"
          >
            {product.name}
          </Link>
        ))}
      </div>
    </div>
  );
};
```

Controlled Components

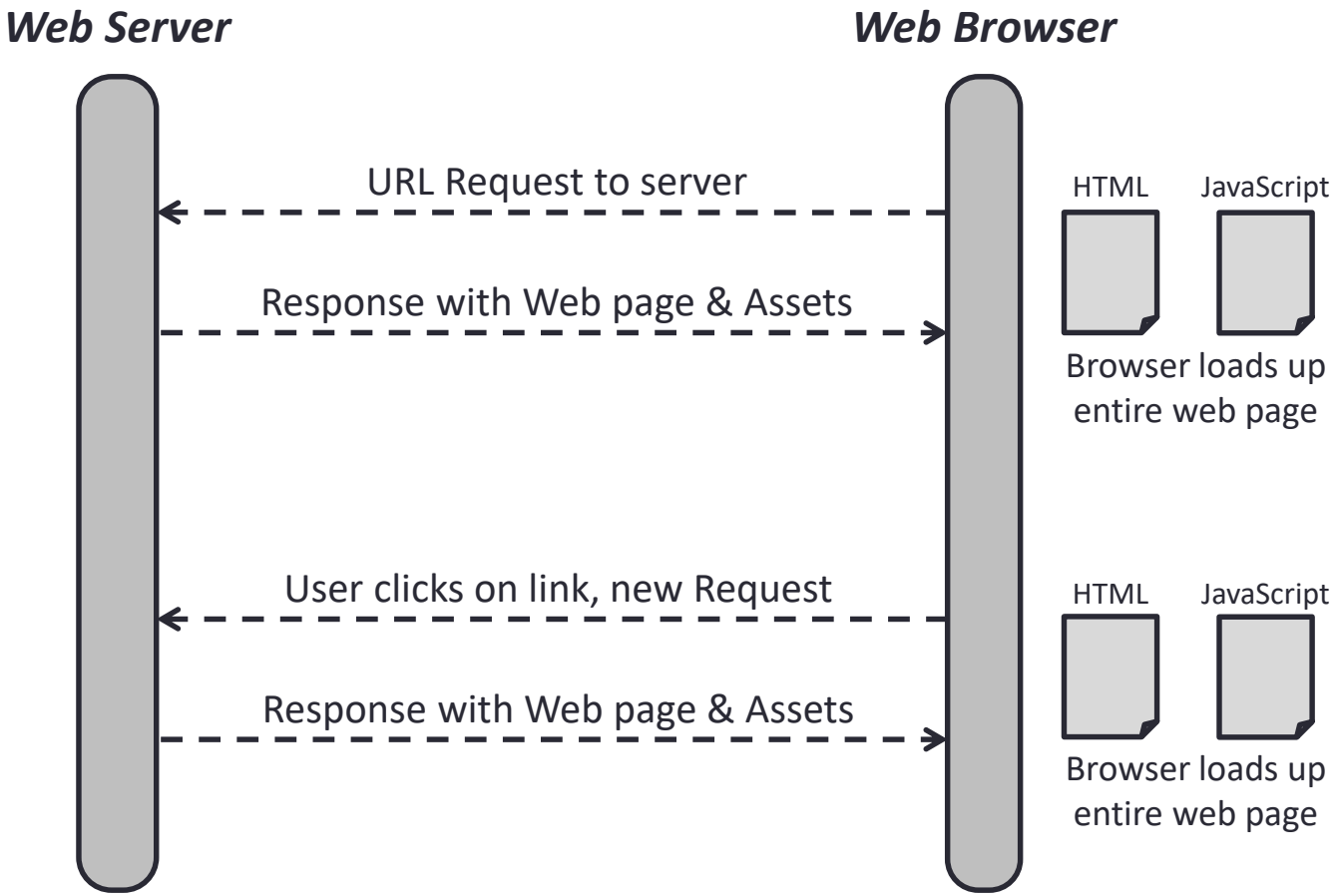
- Refer to components that render a form, but the “source of truth” for that form state lives inside of the component state rather than inside of the DOM

- An input form element whose value is controlled by component’s state
- The value attribute of form element is set by state
- Its value only ever changes when the state changes
- Allows us to dynamically update the UI based on component’s state

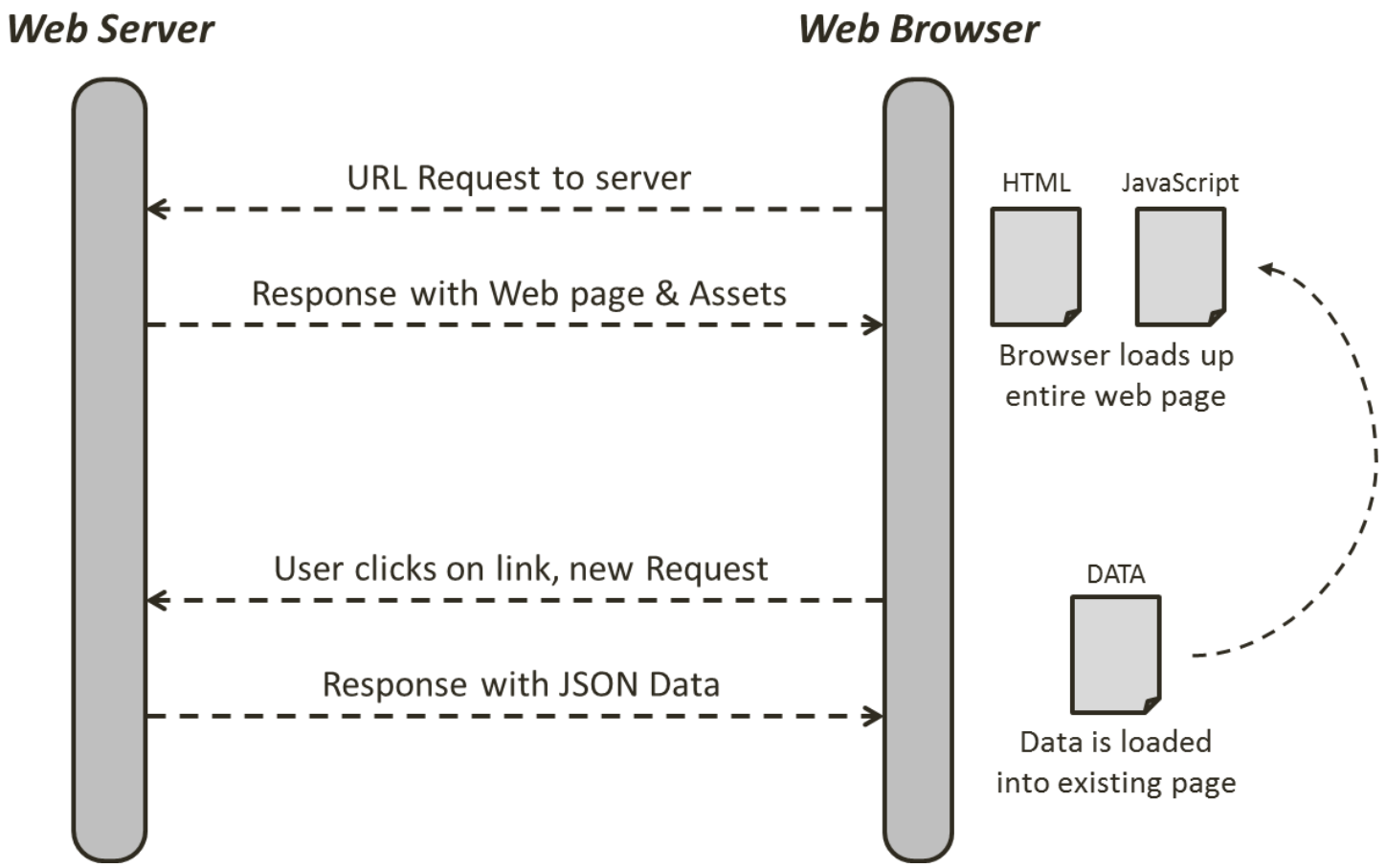
- Benefits
  - instant input validation
  - conditionally disable / enable buttons
  - enforce input formats

Traditional Web App

Request & Response



Single Page App
Request & Response



Building SPAs in React

- Turns React projects into SPAs
- A collection of navigational components that compose declaratively with your application
- Provides a number of specialized components that
  - manage the creation of links
  - manage the app's URL
  - provide transitions when navigating between different URL locations

- Install React Router
  - npm install --save react-router-dom

Building SPAs in React

- BrowserRouter
  - Listens to changes in URL
  - Makes sure that the correct screen (component) shows up

  - For React Router to work properly, you need to wrap your whole app in a BrowserRouter component.

```
ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), el)
```

Building SPAs in React

- Link
  - Provides declarative, accessible navigation around your application
  - Used instead of anchor tags (<a>)

```
<Link to="/about">About</Link>
```

- Route
  - Renders some UI when a location matches the route's path

```
<Route exact path="/" component={Home}/>

<Route path="/news" component={NewsFeed}/>

<Route path="/home" render={() => <div>Home</div>}/>
```

Building SPAs in React

- NavLink
  - A special version of the <Link> that will add styling attributes to the rendered element when it matches the current URL

```
<NavLink to="/faq" activeClassName="active">FAQs</NavLink>
```

- Switch
  - Renders the first child <Route> or <Redirect> that matches the location
  - <Switch> is unique in that it renders a route *exclusively*
  - In contrast, every <Route> that matches the location renders *inclusively*

Building SPAs in React

- Switch

```
<Switch>
  <Route path="/products/new" component={ProductForm} />
  <Route exact path="/products/:id"
         component={ProductDetail} />
  <Route exact path="/products/:id/edit"
         component={ProductForm} />
  <Route render={() => (
     <AppAlert type="info"
         message="Select a product from the list." />
     )} />
</Switch>
```

Q & A

- Thank you!