React

Teacher

- Background & Experience
- Info Support

Attendees

- Role
- Background & Experience
- What can I teach *you*?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Agenda

- Introduction React
- JSX, Elements & Components
- Props, State, Lifecycle & Events
- Advanced Rendering
- Forms & Validations
- Architecting Applications
- Strategies and Best Practices
- Routing
- Global State & Persistence
- Testing

Rules of engagement

- Training hours
- Lunches & Breaks
- Phones
- Evaluation

> *Have fun!*

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Context

- History of React
- React vs ..
- React Spinoffs
- Hello React

History of React

- 2010: Facebook introduces XHP, open sources it
- 2011: Jordan Walke creates an early prototype of React: FaxJS
  - Search element on Facebook
- 2013: Jordan Walke introduces React at ConfUS; React gets open sourced
- 2014: React Developer Tools becomes an extension of the Chrome Developer Tools
  - Release of React Hot Loader

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

History of React (2)

- 2015: React considered stable
  - Netflix likes React
  - Airbnb uses React
  - Facebook releases first version of React *Native*
- 2016: React gets mainstream
- 2017: The year of further improvements
  - Facebook relicenses React and others (15.6)
  - Fiber open sourced at F8 (16.0)
  - *Fragments* added (16.2)
- 2018: *Contexts* added (16.3)
- 2019: *Hooks* added (16.8)

React Spinoffs

- *React Native*: Android and iOS apps
- *Preact*: Fast 3kB alternative to React
- *React360*: For VR environments
- *React-PDF*: Creating PDF files on the browser, mobile and server
- *Proton Native*: Native desktop applications for Windows, Linux and macOS.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Defining React

> *React is an library for building composable user interfaces using JavaScript and (optionally) XML*

Benefits of React

- Simple
  - Automagically rerenders when state changes
- Scalable
  - Write self-contained, composable components
- Pure JavaScript
  - Write components in plain JavaScript instead of using templates
- Abstraction
  - Abstracts the view, making it possible to render to other platforms (iOS / Android / VR)

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Setup

Getting started

What you need:

- NodeJS and NPM/NPX
- Babel
- React
- React DOM

What you need to know:

- HTML
- JavaScript/ES6

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

How to include React and Babel

You can load them on the fly from a CDN (great for prototyping, bad for performance).

Or...

Use Grunt, Gulp or Webpack to create a development workflow with hot-reloading, unit testing and minification.

Hint: this has been done by others, too!

Hello React

First React Component with JSX

(ad-hoc class syntax)

```
var Banner = React.createClass({
  render: function() {
    const className = this.props.show ? '' : 'hidden';
    return (
      <div className={ className }>{ this.props.message }</div>
    );
  }
});
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Hello React

First React Component with JSX

(ES6 class syntax)

```
class Banner extends React.Component {
  render() {
    const className = this.props.show ? '' : 'hidden';
    return (
      <div className={ className }>{ this.props.message }</div>
    );
  }
}
```

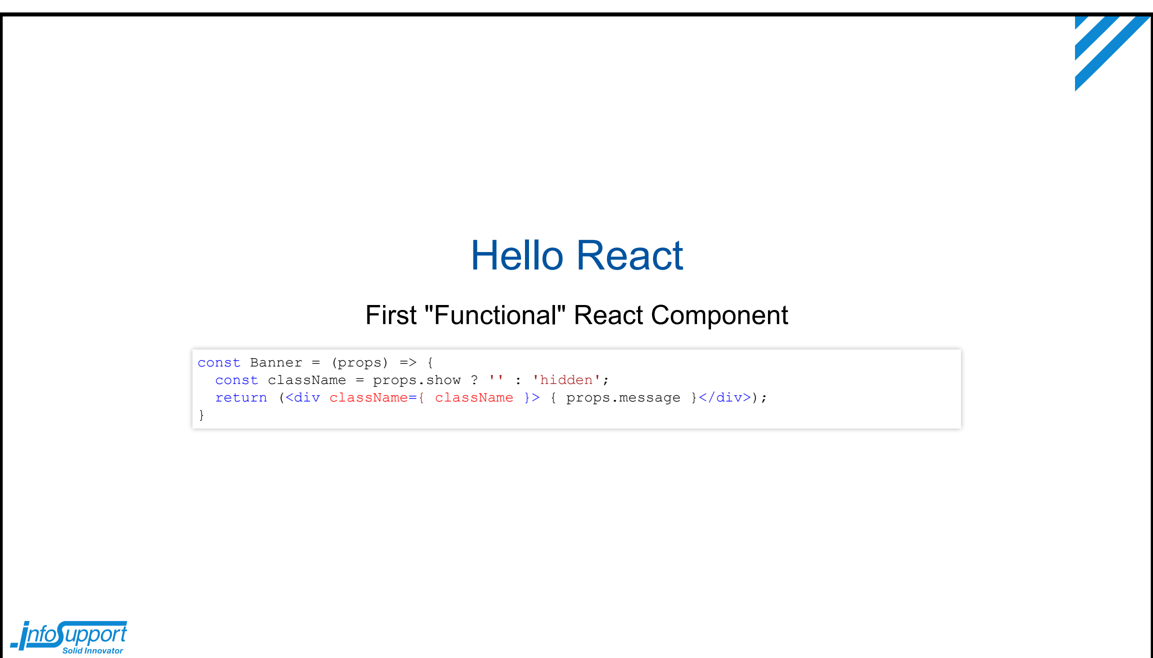
Hello React

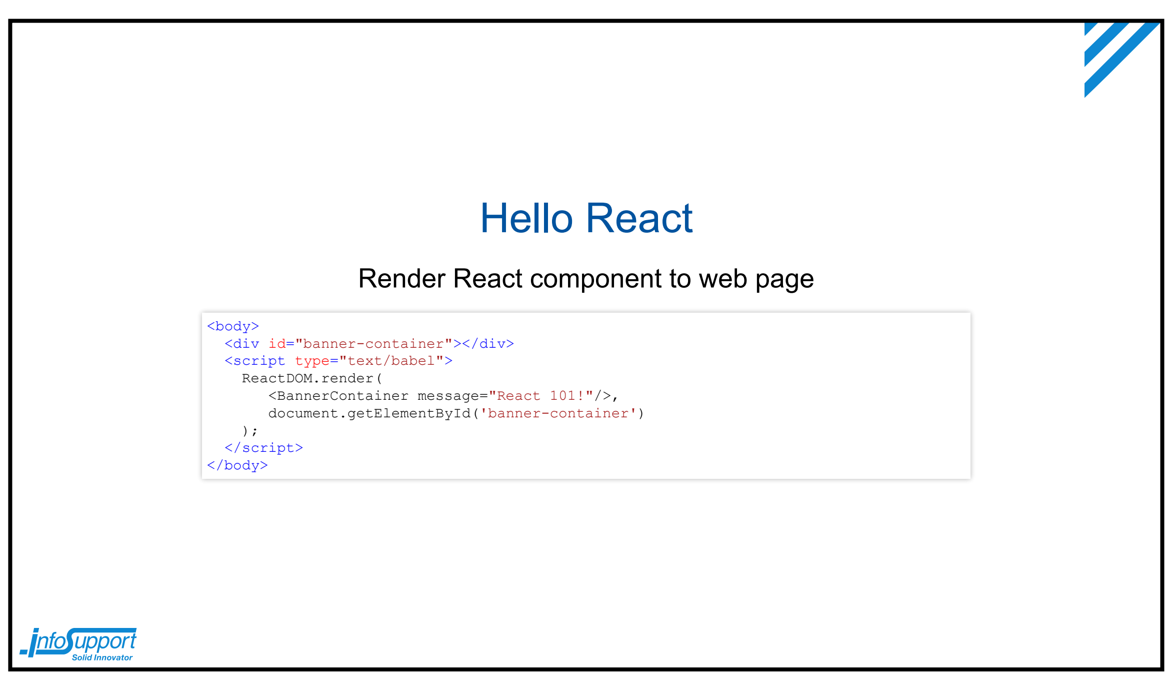
First React Component with JavaScript

(React component transpiled to JavaScript)

```
'use strict';

var Banner = React.createClass({
  displayName: 'Banner',

  render: function render() {
    const className = this.props.show ? '' : 'hidden';
    return React.createElement(
      'div',
      { className: className },
      this.props.message
    );
  }
});
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support





© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

JSX, Elements & Components

What is JSX?

```
const something = <div className="greeting">Hello world!</div>;
```

- A syntax extension to JavaScript
  - real XML, not a string of characters
  - allows embedded expressions
  - supports attributes
- Can be nested
- Automatic XSS prevention
- Needs to be *transpiled* to JavaScript
  - e.g. `React.createElement(...)`

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

JSX vs HTML

Three important differences

1. Tag attributes are camel cased

```
<input type="text" maxLength="10" />
```

2. Elements must be balanced: `<br>` → `<br/>`
3. Attribute names are based on DOM API, not HTML language specs:

```
<div id="box" class="class"></div>
```

```
<div id="box" className="class"></div>
```

JSX Quirks: single root node

```
return (<h1>Hello world</h1> )
// translates to
return React.createElement("h1", null, "Hello world");
// but
return (<h1>Hello world</h1><p>test</p> )
// translates to
return React.createElement("h1", null, "Hello world")
       React.createElement("p", null, "test");
// which does not compile
```

→ Wrap your elements in a containing element for return

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

JSX Quirks: Conditional clauses

```
return (<h1 className={if (condition) { "title"}}>Hello React</h1>);
// translates to
return React.createElement(
  "h1", {className: if (condition) { "title"}}, "Hello React})
// which obviously does not compile
```

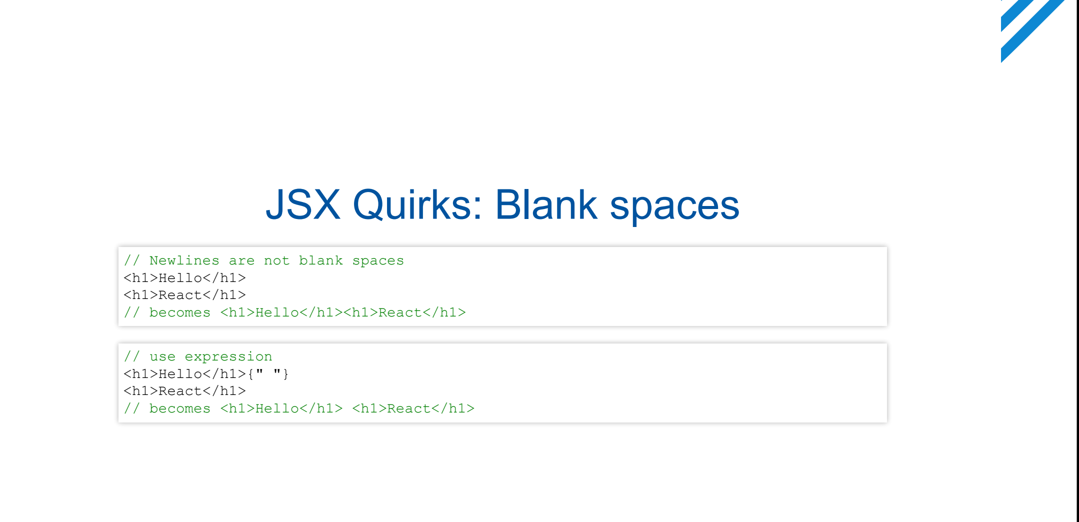
```
return (<h1 className={condition ? "title" : ""}>Hello React</h1>);
// translates to
return React.createElement(
  "h1", {className: condition ? "title" : ""}}, "Hello React"})
// which does compile
```

JSX Quirks: Conditional clauses (2)

```
// move the condition out
let className;
if (condition) { className="title"}
<h1 className={ className }>Hello React</h1>
```

- Use ternary operator
- Move the condition out
- `undefined` is handled automatically, no class attribute is generated

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



JSX Quirks: Blank spaces

```
// Newlines are not blank spaces
<h1>Hello</h1>
<h1>React</h1>
// becomes <h1>Hello</h1><h1>React</h1>
```

```
// use expression
<h1>Hello</h1>{" "}
<h1>React</h1>
// becomes <h1>Hello</h1> <h1>React</h1>
```

JSX Quirks: Comments

```
const content = (
  <Nav>
    { /* child comment, put {} around */ }
    <Person
      /*  multi
          line
          comment */
      name={ window.isLoggedIn ? window.name : '' } // end of line comment
    />
  </Nav>
);
```

→ Can't use HTML comment `<!-- -->`

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Render Dynamic HTML

React has built-in XSS attack protection. It won't allow dynamic HTML tags

```
render() {
  const content = "<h2>Hello world</h2>";
  return (
    // => renders literally
    <h1>{content}</h1>
    // => renders inline HTML
    <h1 dangerouslySetInnerHTML={{ __html: content }}></h1>
  )
}
```

What is an element?

```
const element = <div className="greeting">Hello world!</div>;
ReactDOM.render(element, document.getElementById('root'));
```

- "just" a JavaScript object
  - immutable
  - representing a DOM node
- React will create / update the DOM based on those objects
- Lets you focus on *what* you want to see, not on *how* to achieve it

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

What is a component?

```
const Greeter1 = () => <div className="greeting">Hello, world!</div>;

class Greeter2 extends React.Component {
  render() {
    return <div className="greeting">Hello, world!</div>;
  }
}
```

- A JavaScript function *or* classs
- Implements an independent, reusable piece of user interface
- Return JSX *elements*, which will turn into React elements

Dynamic Values

- Values written in `{ }` are JavaScript expressions

```
import React, {Component} from 'react';
class Greeter3 extends Component {
  render() {
    const place = "World";
    return <div className="greeting">Hello, { place }!</div>;
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Composing Components

- Parent and Child components
- Pass data using properties
  - Configuration properties for passing data from parent to child
  - Immutable, passed and owned by parent component
  - Provided as attributes in JSX

→ Convention: component name starts with an uppercase letter

Example

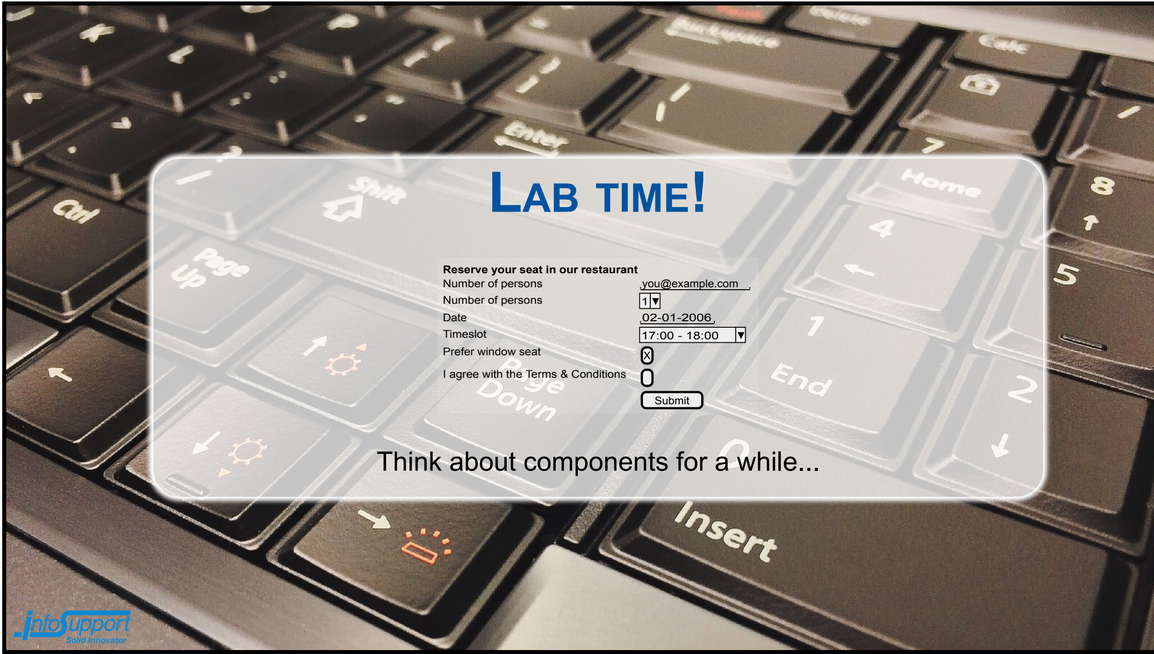
```
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity='1' name='Bread' />
      </ul>
    );
  }
}
class ListItem extends Component {
  render() {
    return (
      <li>{ this.props.quantity }x {this.props.name}</li>
    );
  }
}
```

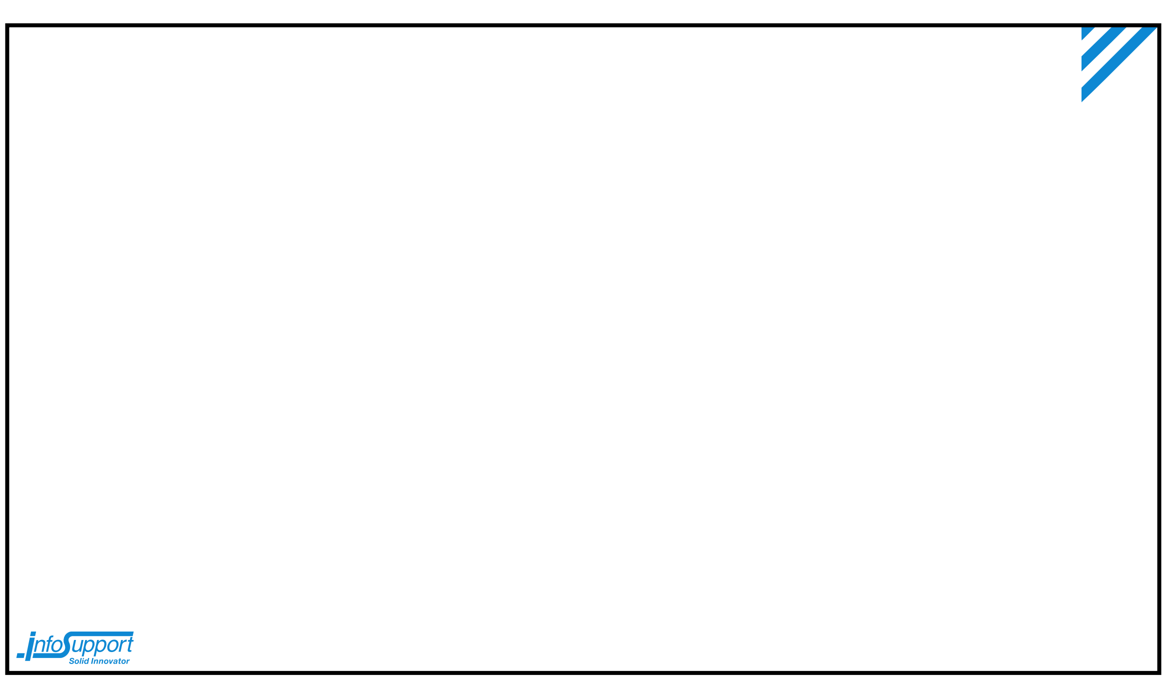
© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Alternative Example

```
class GroceryList extends Component {
  render() {
    return (
      <ul>
        <ListItem quantity='1'>Bread</ListItem>
      </ul>
    );
  }
}
class ListItem extends Component {
  render() {
    return (
      <li>{ this.props.quantity }x {this.props.children}</li>
    );
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support





© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Styling Components

Inline styles

Components are styled inline as JavaScript object

- Camel cased names
- No need to specify pixels

Example

```
render() {
  const style = {
    backgroundColor: 'red',
    color: this.props.color,
    width: 100
  };
  return <div style={ style }>Hello World</div>
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Props, State, Lifecycle & Events

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Recap

A component is either

→ a **function** that returns element(s)

or

→ a **class** whose `render` method returns element(s).

Both accept *one* parameter, commonly called `props`.

Props

- An object with arbitrary inputs for a component.
- Method argument for function components

```
const Greeter = (props) => <div>Hello, { props.name }!</div>;
```

- Instance variable for class components

```
class Greeter extends React.Component {
  render() {
    return <div className='greeting'>Hello, { this.props.name }!
  }
}
```

- In use, props look like XML attributes

```
const App1 = (props) => <Greeter name={ world } />;
const App2 = (props) => <Greeter name='world' />;
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Typing & structure

- JavaScript is in itself weakly typed
- So how do we know what the `props` will contain?
- *Object destructuring* can help you:

```
const Greeter = ({ name }) => <div>Hello, { name }!</div>;
```

- Note how `props.name` turns into `name`.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Re-rendering

As soon as you change the `props` passed to a component, that component will re-render.

1. Invoking the component function.
2. Invoking the class' `render()` method.

This is called one-way databinding.

Stateful Components

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

State

- Mutable object that holds state for the component
- Initialised in constructor
- Needs to be set using `setState` function
- `setState` triggers rerender of the component

State: example

```
class Greeter4 extends Component {
  constructor(props) {
    super(props);
    this.state = { toggle: false }
  }
  render() {
    const details = this.state.toggle ? <div>More...</div> : undefined;
    return (
      <div>
        <div onClick={ () => this.setState({ toggle: !this.state.toggle }) }>hit me</di
        <div>{ details }</div>
      </div>
    );
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

More State

- Components may need to keep state that is private.
- A *pure* function component can never keep state, because *pure* functions
  - do not attempt to change their inputs.
  - always return the same result for the same inputs.
- Keeping state means you need a class!
  - Unless you use hooks, to be discussed later.

Something odd?

```
class Checkbox extends React.Component {
  constructor(props) {
    super(props);
    this.state = { checked: false };
  }

  render() {
    return <input checked={ this.state.checked } />;
  }
}
```

→ In this class, `state` is a constant value.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Three rules for mutating state

1. *Never* modify state directly!
   - Bad: `this.state.checked = true`
   - **Good**: use `setState({checked: true});`
2. Mutating state is *async*
   - State may not be updated immediately after invoking `setState()`.
3. State is *merged*:
   - No need to copy previous state when invoking `setState()`.

Checking the checkbox

```
class Checkbox extends React.Component {
  constructor(props) {
    super(props);
    this.state = { checked: false };
  }

  render() {
    return <input checked={ this.state.checked } />;
  }
}
```

- How to toggle the internal state of the checkbox?

```
toggle() {
  this.setState({ checked: !this.state.checked });
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Toggle, toggle, toggle

```
toggle() {
  this.setState({ checked: !this.state.checked });
}
```

This snippet contains a subtle bug. Can you spot it?

Toggle, toggle, toggle

```
toggle() {
  this.setState({ checked: !this.state.checked });
}
```

A call to `setState` is async.

We can not rely on the value of `this.state` that we observe when we invoke it.

We should pass a `mutator`: a function that will be invoked when the actual mutation takes place.

```
toggle() {
  this.setState((prev, props) => ({ checked: !prev.checked }));
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Events

Events

HTML has an API for event handling using attributes

```
<div onclick="javascript:..."></div>
```

Downsides:

- pollutes the global namespace
- hard to track in big HTML file
- can be slow and lead to memory leaks

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

JSX Event Listeners

Support for similar API

Benefits:

- Callback functions scoped to component
- Auto unmounting
- Event delegation

Difference:

- properties are camel-cased
  - `onClick` instead of `onclick`

Recap

- Mutating state is often a reply to some (user) event.
  - Mouse clicked, key pressed, input changed, form submitted
- Basic principle is just like DOM or jQuery events, except

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Recap

	Event naming	Event handling
DOM	lowercase	pass function *name*
jQuery	lowercase	pass function (often inline)
React	camelCase	pass function (by reference)

```
<button onClick={ someFunction }>Click me</button>
```

Synthetic event

- Event handling functions will receive an `event` parameter.
- This is not the browser's native event!
- It's called a *synthetic event* and it wraps the browser’s native event.
- Same interface, and works the same across all browsers.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

How to prevent `this`?

1. Manually bind the callback to the instance of the class in the constructor:

```
this.handleClick = this.handleClick.bind(this);
```

2. Use public class fields syntax as callbacks:

```
handleClick = (e) => { ... }
```

3. ~~Use arrow functions as callback.~~

```
onClick={ (e) => this.handleClick(e) }
```

→ Why not?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

No bind or arrow functions in JSX Props

```
(e) => this.handleClick(e)    // Arrow function
this.handleClick.bind(this)   // Bind
```

- Both will create a new function *every single time*.
- Remember: as you change the props passed to a component, that component will re-render.
- Imagine rendering a (large) list of items, each of them declaring their own `onClick` callback this way.
  - Each item in the list is fully re-rendered!

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Component Lifecycle

Component LifeCycle

- React components have a lifecycle.
- Each lifecycle phase provides hook methods.
- Automatically called by React

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Lifecycle Phases and Methods

In the life of a component, these are the phases

- Mounting (attaching to DOM)
- Unmounting (releasing from DOM)
- Props change
- State change

In each of these, methods on your component are called. Some of them are scheduled for removal in React 17.

Mounting

Method/Callback	Description
(Class Constructor)	
~~componentWillMount~~	Once, before initial rendering (setting state does not trigger re-render)
(render)	
componentDidMount	Once, after initial rendering

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Unmounting

Method/Callback	Description
componentWillUnmount	Immediately before a component is unmounted from the DOM (clean up)

Props changes

Method/Callback	Description
~~componentWillReceiveProps~~	When a component is receiving new props
shouldComponentUpdate	Before rendering, to optionally skip rendering
~~componentWillUpdate~~	Immediately before rendering
(render)	
componentDidUpdate	Immediately after rendering

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

State changes (almost same as Props changes)

Method/Callback	Description
shouldComponentUpdate	Before render, to optionally skip rendering
componentWillUpdate	Immediately before rendering
(render)	
componentDidUpdate	Immediately after rendering

→ No `componentWillReceiveState` because an updated prop may cause state change, but never the other way around

Summary

1. When a component is rendered to the DOM, `componentDidMount()` will run.
2. When a component is removed from the DOM, `componentWillUnmount()` will run.

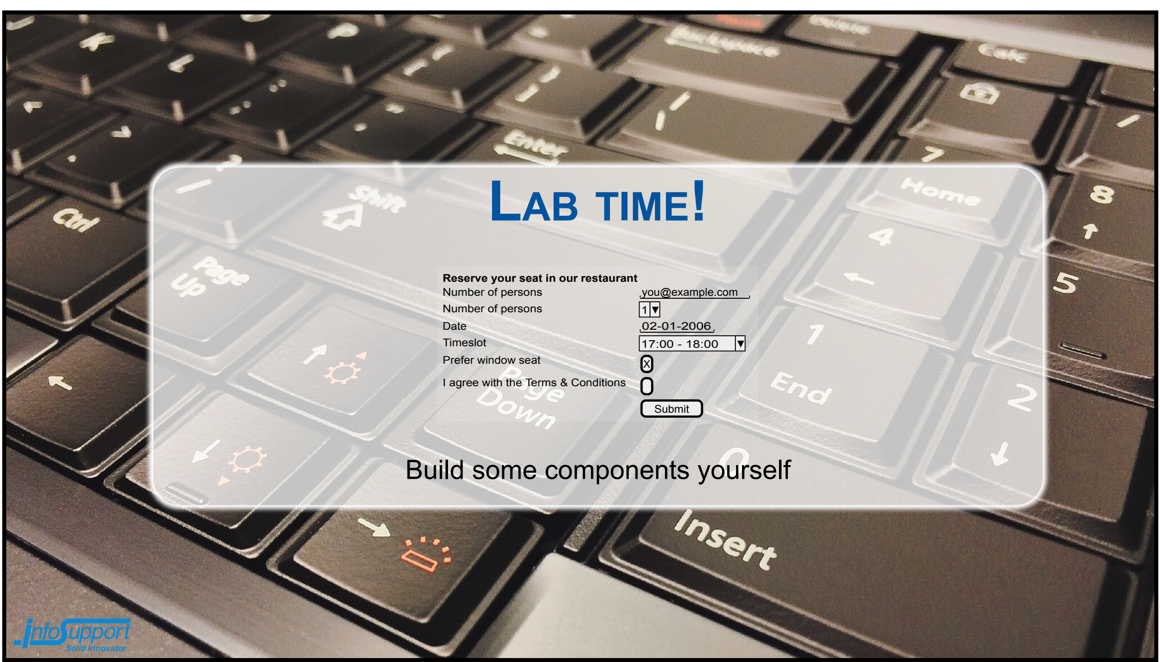
→ Can you think of a use case for both of them?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

`componentDidMount` vs. constructor

- Constructor can/should
  - initialise state to *sane* defaults
  - bring component in a state that can be rendered
  - *not* have side effects
- `componentDidMount` can/should
  - update state with new data (e.g. fetched / loaded)
  - keep component in a state that can be rendered

> *Forgetting `this.state = { ... }` in the constructor can easily lead to errors inside `render()`.*



© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

References to DOM

In occasions you want to bypass the virtual DOM

- Usually bad practice
- But needed for uncontrolled components!

```
class FocusText extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    <input type="text" ref={ this.textInput } />
  }
}
```

Use case for Refs

Suppose we want to set the focus on an input *after* button click

```
class FocusText extends Component {
  handleClick = (e) => {
    this.textInput.current.focus();
  }
  render(){
    return (
      <div>
        <input type="text" ref={ this.textInput } />
        <input type="button" onClick={ this.handleClick } />
      </div>
    );
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Aside: expressions in JSX

Inside JSX we can embed *any* valid JavaScript expression.

```
const Ex1 = () => <div>{ 40 + 2 }</div>;

const answerToQuestionOfLife = 42;
const Ex2 = () => <div>{ answerToQuestionOfLife }</div>;

const askQuestionOfLife = () => answerToQuestionOfLife;
const Ex3 = () => <div>{ askQuestionOfLife() }</div>;
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Aside: expressions in JSX (2)

JSX itself is an expression, too. So you can embed control statements inside JSX:

```
class Mood extends React.Component {
  render() {
    if (this.props.isHappy) {
      return <ClapHands />;
    } else {
      return <DryTears />;
    }
  }
}
```

Aside: expressions in JSX (3)

JSX itself is an expression, too. So you can embed control statements inside JSX:

```
const Ticker = (props) => <div>Ticker for <strong>{ props.symbol }</strong></div>;

class TickerList extends React.Component {
  render() {
    return this.props.symbols.map(
      (symbol) => <Ticker symbol={ symbol } />
    );
  }
}

<TickerList symbols={ ["ASML", "PHIA"] } />
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Under the hood: Virtual DOM

- So far, we've *declared* components & elements
- Which are *transpiled* into `React.createElement(...)` calls:

```
<Greeter name={ 'world' } />

/** transpiles into */

React.createElement(Greeter, { name: 'world' }, null)
```

- React "automagically" (re-) builds the DOM using its **virtual DOM**.
  - Lives in-memory.
  - Shadows the **actual DOM**.

Reconciliation (1)

- It's Reacts job to keep the **virtual** DOM and the **actual** DOM in sync.
- This *reconciliation* makes two important assumptions:
  1. If two elements are of different type, the (sub) tree will be different.
  2. The `key` prop identifies child elements over re-renders.

→ Do these assumptions make sense to you?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Reconciliation (2)

If two elements are of different type, the (sub) tree will be different.

```
const FrenchGreeter = ({ name }) => <div>Salut, { name }</div>;
const GermanGreeter = ({ name }) => <div>Hallo, { name }</div>;
const EnglishGreeter = ({ name }) => <div>Hello, { name }</div>;

const App = ({ language, name }) => {
  switch(language) {
    case 'fr': return <FrenchGreeter name={ name } />
    case 'de': return <GermanGreeter name={ name } />
    case 'en':
    default  : return <EnglishGreeter name={ name } />
  }
};
```

The `key` prop identifies child elements over re-renders.

```
const TickerList = ({ symbols }) => symbols.map(
  (symbol) => <Ticker symbol={ symbol } />
);

const symbols = ["ASML", "PHIA"];
<TickerList symbols={ symbols } />
```

- A new symbol is added to the array. How does React know which elements in the **virtual** DOM are new compared to the **actual** DOM?

```
const TickerList = ({ symbols }) => symbols.map(
  (symbol) => <Ticker key={ symbol } symbol={ symbol } />
);
```

- The `key` can be anything, as long as it's **stable**.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Forms and Validation

Forms in React

Two ways of handling forms

- controlled components
  - The preferred and React way
  - Immutable
- uncontrolled components
  - The HTML way

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Uncontrolled Components

- Form Component without
  - `value` or `checked` property
  - `onChange` event handler
  - **You** are responsible for bringing DOM state in sync with app state

```
constructor() {
  this.searchTermRef = React.createRef();
}

handleSubmit = (e) => {
  this.setState({ searchTerm: this.searchTermRef.current.value });
}

render() {
  return (
    <form onSubmit={ this.handleSubmit }>Search Term:
      <input
        ref={ this.searchTermRef }
        type='search'
      />
    </div>
  );
}
```

Controlled Components

- Form Component with
  - `value` or `checked` property
  - `onChange` event handler
  - **React** keeps DOM state in sync with app state

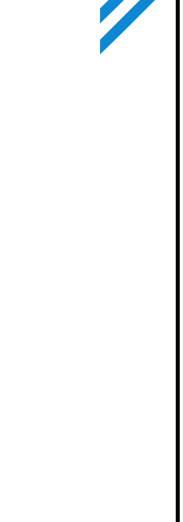
```
constructor(props) {
  super(props);
  this.state = { searchTerm: 'React' };
}

handleChange = (event) => {
  this.setState({ searchTerm: event.target.value });
}

render() {
  return (
    <div>Search Term:
      <input
        type="search"
        onChange={ this.handleChange }
        value={ this.state.searchTerm }
      />
    </div>
  );
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Advantages of controlled components

- Stays true to React, state is kept out of interface
- Easy input validation and sanitisation

```
this.setState({
  searchTerm: event.target.value.subtr(0, 50)
});
```



Special cases: Text Area

- Normally textarea uses inner content

```
<textarea>text here</textarea>
```

- In React

```
<textarea value='text here'></textarea>
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Special Cases: Select

- Normally uses the `selected` attribute

```
<select>
  <option selected>a</option>
</select>
```

- In React

```
<select value='B'>
  <option value='B'>Mobile</option>
</select>
```

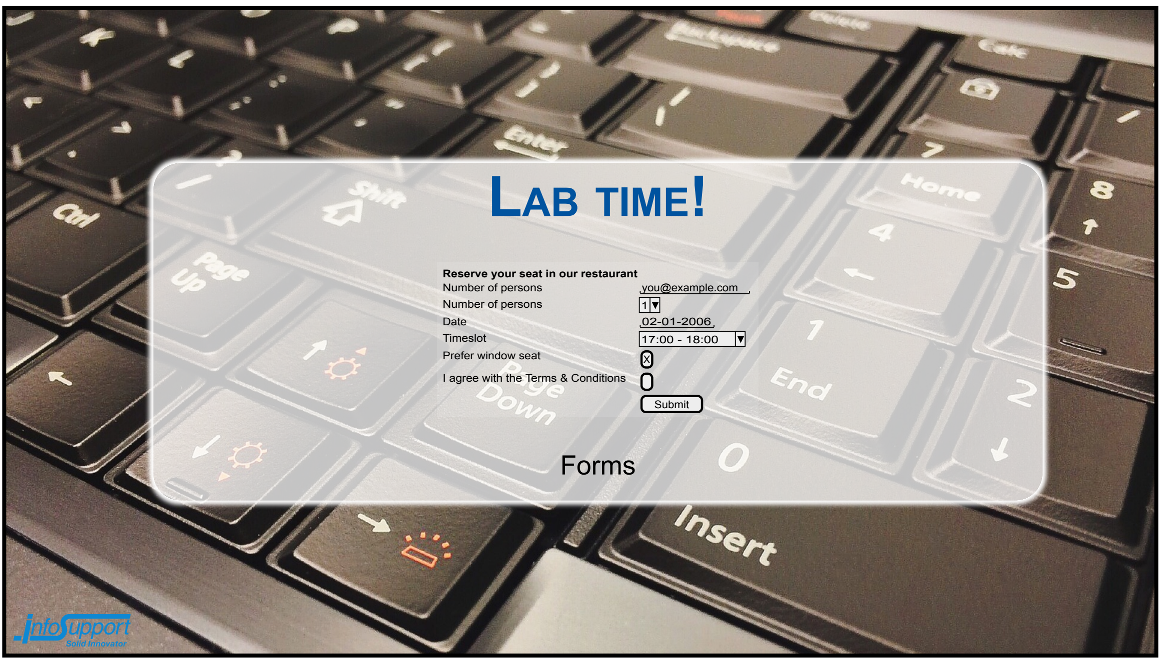
Uncontrolled Components

- Often used for larger forms where everything is processed when the user is done
- Any input that does not supply a `value` property is uncontrolled

example:

```
handleSubmit = (event) => {
  console.log(`Submitted: ${event.target.name.value}`);
  event.preventDefault();
}
render() {
  return (
    <form onSubmit={ this.handleSubmit }>
      <div>
        Name: <input type="text" name="name" />
        <button type="submit">Submit</button>
      </div>
    </form>
  );
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Prop Validation

- Applications exist of a composition of components
- Components need to be consistent and documented
- PropTypes
  - Documents type and required properties for components
  - React throws descriptive errors when requirements are not met

```
class Greeter extends Component {
  render() {
    return (
      <h1>{ this.props.greeting }</h1>
    )
  }
}
Greeter.propTypes = {
  greeting: PropTypes.string.isRequired
}
```

- If the requirements are not respected, a warning is logged to the console
- Optional props go without `isRequired`

Default Prop values

To provide default values

```
Greeter.propTypes = {
  greeting: PropTypes.string
}

Greeter.defaultProps = {
  greeting: "Hello World"
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

PropType validation

Validator	Description
PropTypes.array	Props must be an array
PropTypes.bool	Props must be a Boolean value
PropTypes.func	Props must be a function
PropTypes.number	Props must be a number
PropTypes.object	Props must be an object
PropTypes.string	Props must be a string
PropTypes.oneOfType	Props could be one of many types

PropType validation

Validator	Description
PropTypes.arrayOf	Props must be an array of certain type
PropTypes.objectOf	Props must be an object with property values of certain type
PropTypes.shape	Props must conform to certain shape
PropTypes.node	Props can be any value that can be rendered
PropTypes.element	Props must be a React element
PropTypes.instanceOf	Props must be an instance of a given class
PropTypes.oneOf	Props must be specific value (enum)

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Examples of PropType validation

```
// a multi shaped property
PropTypes.oneOfType([PropTypes.string, PropTypes.number]);

// an array property with typed members
PropTypes.arrayOf(PropTypes.number);

// a shape object property type
PropTypes.shape({ color:PropTypes.string, fontSize: PropTypes.number});

// a object type validator
PropTypes.instanceOf(Message);

// an enum validator
PropTypes.oneOf(['News', 'Photos']);
```

Custom PropType validators

You can create your own!

```
const titlePropType = (props, propName, componentName) => {
  if (props[propName]) {
    const value = props[propName];
    if (typeof value !== 'string' || value.length > 80) {
      return new Error(`${propName} in ${componentName} is too long!`);
    }
  }
}

Greeter.propTypes = {
  title: titlePropType,
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Stateful vs Pure Components

Components can have data as props and/or state:

- Props are component's configuration
- State often contains mutable data
  - is optional

Components *without* state are called *pure components*.

Components *with* state are called *stateful components*. They usually live higher in the component hierarchy.

Benefits of pure components

- Easier to understand
- Easier to test
- Easier to reuse

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Which Components are Stateful

1. Identify a piece of state.
2. Identify every component that depends on that state, i.e. renders based on its content
3. Find a common owner component, this should *own* the data
4. If there is no common owner component, create one

Quiz

```
- ContactsApp
  - SearchBar
  - ContactList
      - ContactItem
```

Given this component tree and the requirement to filter the contactlist based on a *search* argument.

→ which components are stateless, which ones are stateful?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Example Answer to Quiz

```
- ContactsApp           # Stateful (holds current search criteria)
  - SearchBar           # Stateful/Stateless (could hold search criteria if complex)
  - ContactList         # Stateful (holds contacts)
      - ContactItem     # Stateless
```

The common owner is `ContactsApp`.

→ How does the `SearchBar` communicate changes back to the `ContactsApp`?

Using callbacks to update a parent component

A two-step approach

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

In the parent component:

```
render() {
  return <SearchBar updateSearchTerm={ this.updateSearchTerm } />;
}

updateSearchTerm = (searchTerm) => {
  this.setState((prev, props) => {
    return { searchTerm };
    }
  });
}
```

In the child component:

```
render() {
  return <input onChange={ this.onChange } />;
}

onChange = (event) => {
  const input = event.target.value;
  this.setState({ value: input });
  if (validate(input)) {
    this.props.updateSearchTerm(input);
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Global State & Persistence

Global State and Persistence

1. React Context API
2. Flux Architectures

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Exploring the problem: "prop drilling"

```
const Green = (props) => (
  <div className='green'>{props.number}</div>
);
```

```
const Blue = (props) => (
  <div className='blue'>
    <Green number={props.number} />
  </div>
);
```

```
class Red extends Component {
  state = { number : 42 }
  render() {
    return (
      <div className='red'>
        { this.state.number }
        <Blue number={ this.state.number } />
      </div>
    );
  }
}
```

→ Imagine what would happen if we have more of those parent-child relations...

State management with the Context API

- allows to define data stores and access them where needed
- no need to have to pass down data through properties
- something like an *application global state*

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

How to use the Context API

```
const NumberContext = React.createContext(); // 1. Setup a context provider
```

```
class Red extends Component {              // 2. Build a context provider component
  state = { number: 42 };
  render() {
    return (
      <NumberContext.Provider value={ this.state }>
        <div className='red'>
          { this.state.number }
          <Blue />
        </div>
      </NumberContext.Provider>
    );
  }
}
```

```
const Green = () => (                      // 3. Consume the context
  <NumberContext.Consumer>{ (value) => value.number }</NumberContext.Consumer>
);
```

Modifying data in the Context

Expose supported modifications via Context.

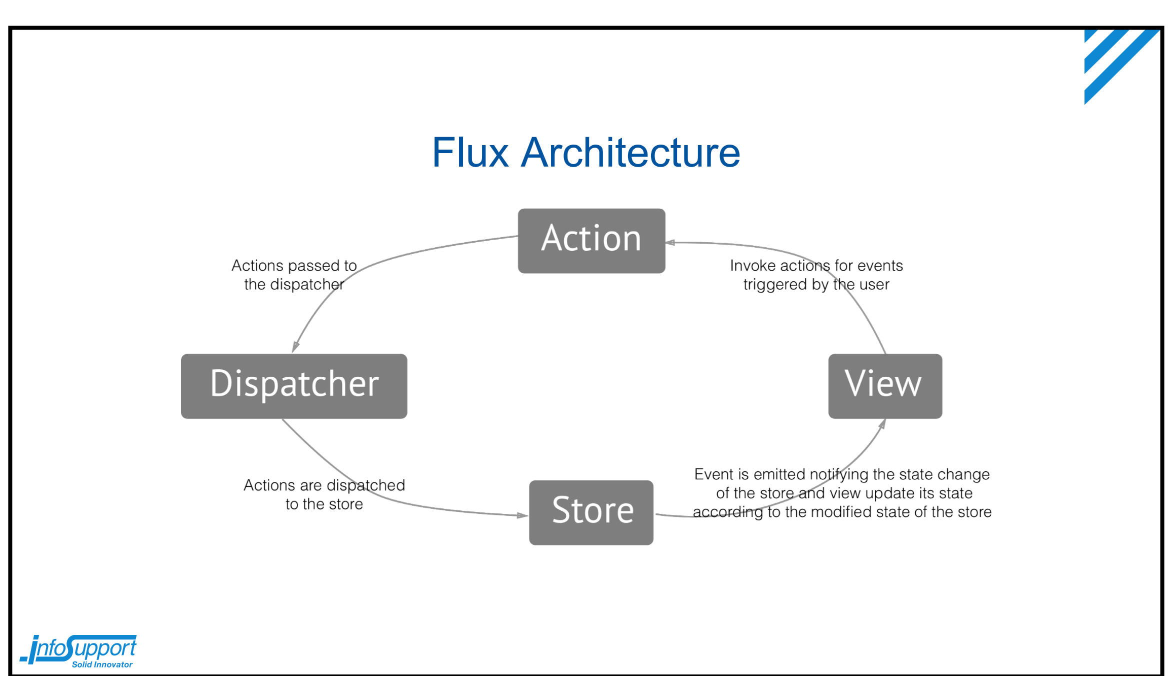
```
class AppProvider extends Component {
  state = {
    number : 10
    increment: () => {
      this.setState((prev, props) => {
        return { number: prev.number++ };
      );
    }
  }
}
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Flux Architectures

- **Action**
  - things that happen in your app
  - a user clicking on the button is the event
  - showing a pop-up message is the action
- **Dispatcher**
  - dispatches the action to all registered stores
- **Store**
  - receives actions and updates state accordingly
  - only the store itself can mutate its data
- **View**
  - the user-visible layer (React Components)



© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Redux

There's tons of Flux implementations (including Flux), but the most popular one seems Redux:

> *A predictable state container for JavaScript apps.*

Getting started with Redux

You'll need

- a constants file (uniquely identifiable names for action types)
- an *reducer*
- a few action creators (convenience methods)
- a store, which will keep track of the data

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Core ideas of Redux

1. State is single source of truth
   → no copies
2. State is read-only
   → state is mutated by emitting **actions**
3. Changes are made with pure functions
   → **reducers** take current state and action (+ params)
   → **always** return **new** objects

BTW, pure functions are easy to test :)

A Scenario

Suppose we have a bank application

- the bank manages your account
- the bank performs withdrawals and deposits
- the bank holds the balance

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Action Creators

```
const Constants = {
    CREATEACCOUNT: 'CreateAccount',
    WITHDRAW: 'Withdraw',
    DEPOSIT: 'Deposit'
};
```

```
const actions = {
    createAccount() {
        return { type: Constants.CREATEACCOUNT };
    },
    deposit(amount) {
        return { type: Constants.DEPOSIT, amount: amount };
    },
    withdraw(amount) {
        return { type: Constants.WITHDRAW, amount: amount };
    }
};
```

Reducer

```
const reducer = (state = {}, action) => {
    switch (action.type) {
        case Constants.CREATEACCOUNT:
            return { balance : 0 };
        case Constants.DEPOSIT:
            return { balance: state.balance + parseInt(action.amount) };
        case Constants.WITHDRAW:
            return { balance: state.balance - parseInt(action.amount) };
    }
};
```

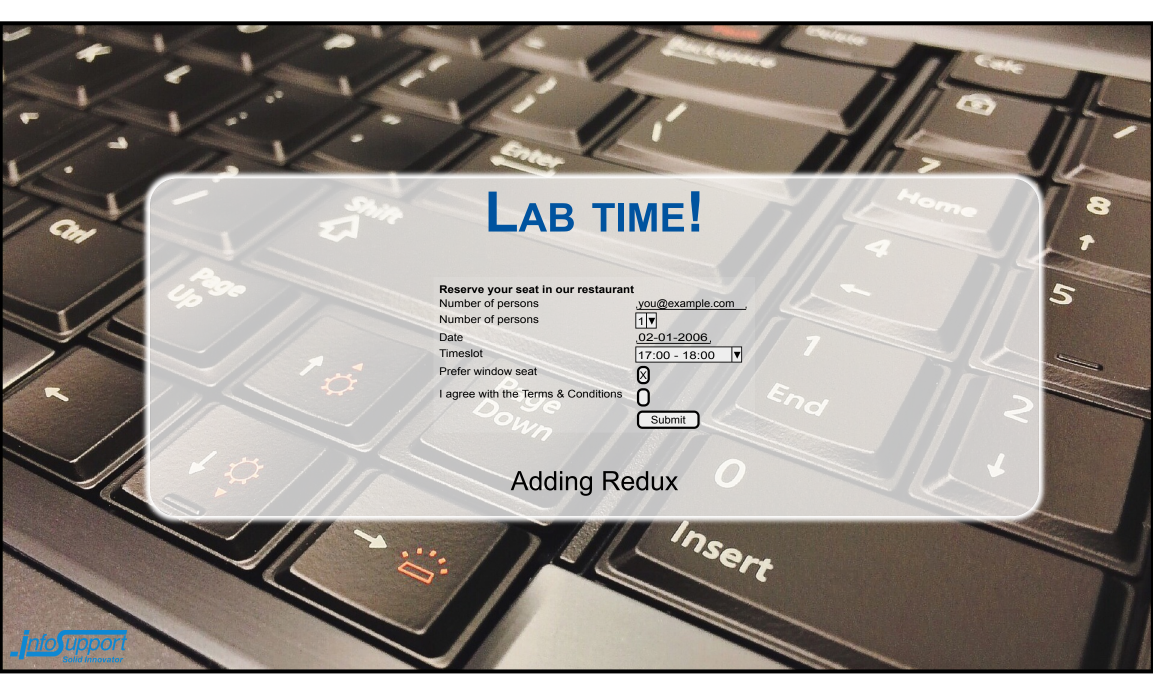
© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Glueing it into React

Provide Redux' store to your app using *react-redux*.

- A little more extensive than Context API.
- Uses *higher order components*.
- Requires functional programming paradigm.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Routing

Routing

- Store state in the url and history of the browser
- Provide deep linking to app pages
- Makes the forward and back buttons of the browser work intuitively
- React makes routing based on urls easy

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Simple Routing

Render different component based on the current url

step 1: listen for url changes

```
class MyApp extends Component {
  componentDidMount(){
    window.addEventListener("hashchange", () => {
      this.setState({
        route: window.location.hash.substr(1);
      })
    });
  }

  // continued...
```

Simple Routing

Render different component based on the current url

step 2. render component that matches route

```
render() {
  var Child;
  switch(this.state.route) {
    case '/about': Child = About; break;
    case '/repos': Child = Repos; break;
    default: Child = Home;
  }

  // continued...
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Simple Routing

Render different component based on the current url

step 3. return correct component and render links

```
// continued...
return (
  <div>
      <header>MyApp</header>
      <menu>
        <ul>
          <li><a href="#/about">About</a></li>
          <li><a href="#/repos">Repos</a></li>
        </ul>
      </menu>
      <Child />
  </div>
)
}
```

Routing

That'll work. But...

- the code is very URL centered
- URL parsing for more complex routing can be difficult

→ Enter the React Router libraries
https://reactrouter.com/web/guides/philosophy

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

React Router

React Router is an external library that contains the core

- Most popular router for React
- Mounts and unmounts automatically when URL changes
- Differentiates between deeplinking and programmatic state changes
- Needs to be completed with a platform:
  - react-router-dom for browser routing
  - react-router-native for mobile apps

React Router Setup

```
npm install --save react-router react-router-dom
```

Main components:

- `Router` and `Route` for declaratively defining routes
- `Link` creates a link to a route
- `Router` to execute the routing logic

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

React Router example

```
ReactDOM.render((
  <BrowserRouter>
    <MyApp />
  </BrowserRouter>
), document.querySelector("#root"));
```

React Router example

The updated component now looks like this

```
class Home extends Component {
  render() {
    return (
      <div>
        <header>My Application</header>
        <menu>
          <ul>
            <li><Link to="/home">Home</Link></li>
            <li><Link to="/about">About</Link></li>
          </ul>
        </menu>
        <Switch>
            <Route exact path='/' component={ Home } />
            <Route path='/about' render={ () => <About number={ 1 } /> } />
            <Route path='/home' component={ Home } />
        </Switch>
      </div>
    );
  }
}
```

- `component` can be `render` function instead - useful when injecting extra properties

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

More Routing

There is more to routing

- child routes aka *nested* routes
  - example `/about/details` and `/about/overview`
- Parameterised routes
  - example `/blog/:id`

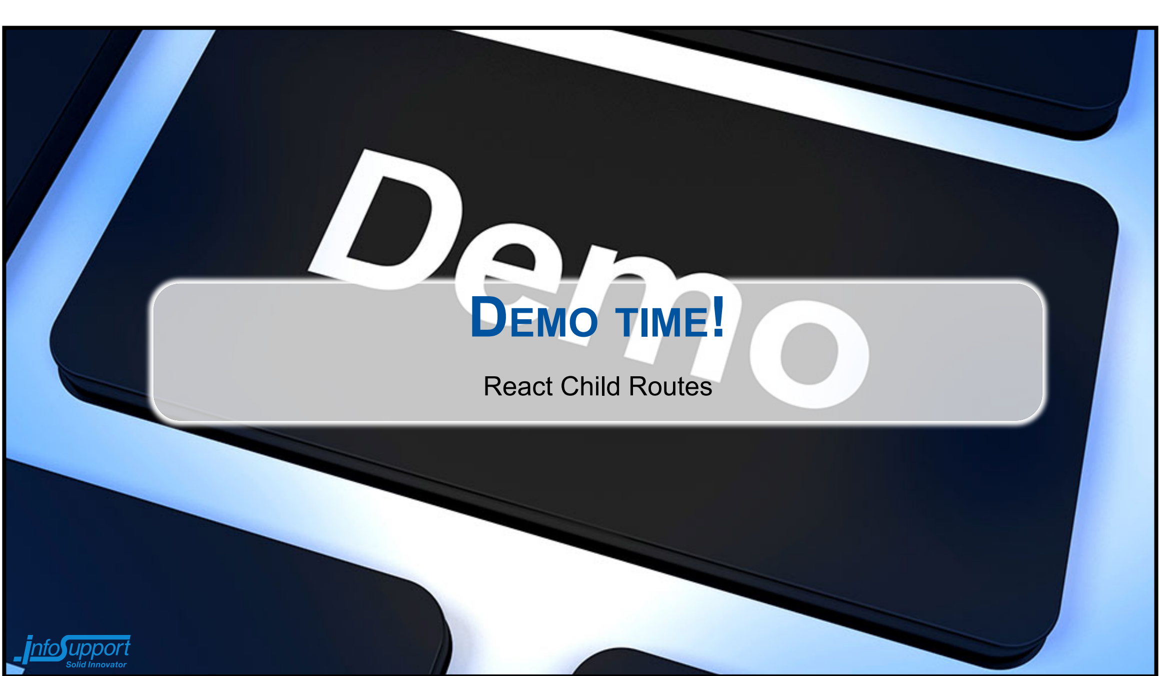
© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Child Routes

To implement child/nested routes, add routing to your child component

```
class About extends Component {
  render() {
    return (
      <Switch>
        <Route exact path='*/details' component={ AboutDetails } />
        <Route path='/about/overview' component={ AboutOverview } />
      </Switch>
    );
  }
}
```

- Note the two different options here..



© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Parameterised Routes

Suppose we want to extract and use parameters from a route

`/blog/:id`

where `:id` is a placeholder for the blog id in a database or something.

How to implement this?

Parameterised Routes

Create a route to it first

```
<Route path='/blog/:id' component={ Blog } />
```

then use it in your component

```
// As an example, but don't ever do this in render()
const id = parseInt(props.match.params.id, 10);
const blog = BlogAPI.get(id);
if (!blog) {
  return <div>Sorry, but the blog was not found</div>
}
return (
  <div>
    <h1>{blog.title}</h1>
    <h2>{blog.content}</h2>
  </div>
);
```

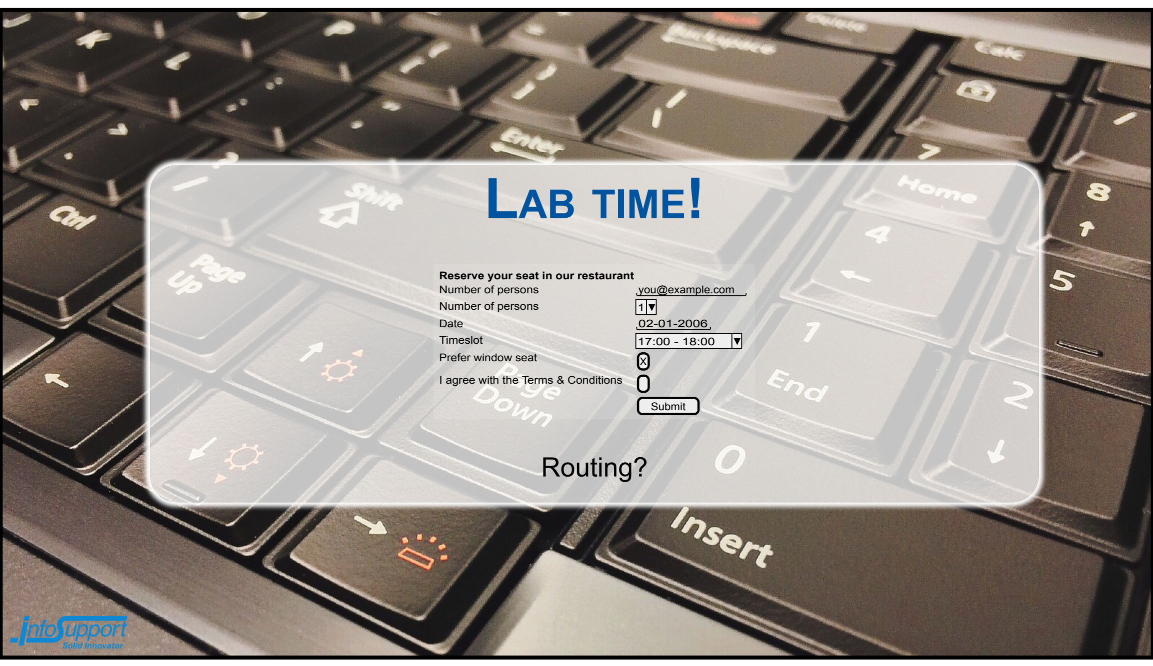
© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Programatically change route

To change to a route programmatically, you can inject Router properties to your class

```
class MyButton extends React.Component {
  navigate = (event) => {
    this.props.history.push('/blog/10');
  }
  render() {
    return (
      <button onClick={ this.navigate }>Go to specific blog</button>
    );
  }
})

// Inject props.history for the new class
const RouteButton = ReactRouterDOM.withRouter(MyButton);
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



Strategies and Best Practices

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Data Flow and Component Communication

- React only allows downward data flow
- Upward communications can be done using callbacks

```
class SearchBar extends Component {
  handleChange = (event) => {
    this.props.onUserInput(event.target.value);
  }

  render() {
    return (
      <input type='search' value={ this.props.filterText }
             onChange={ this.handleChange } />
    );
}
```

```
class ContactsApp extends Component {
  render() {
    return (
      <div>
        <SearchBar filterText={ this.state.filterText }
                   onUserInput={ this.handleInput } />
        <ContactList ... />
      </div>
    );
  }
}
```

Immutability

- Objects and arrays are passed as references
- Replace objects instead of changing them leads to better performance
- Be careful here, for instance
  - pushing into array does not replace the array but changes it!
  - this *may* lead to poor performance
- Use non-destructive methods like `concat`, `map` and `filter`

```
// instead of
const contacts = this.state.contacts;
contacts.push(
  { name: 'John Doe', email: 'john.doe@infosupport.com' }
);
this.setState({ contacts:contacts });

// use this code
const contacts = this.state.contacts.concat(
  { name: 'John Doe', email: 'john.doe@infosupport.com' }
);
this.setState({ contacts:contacts });
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Object.assign

Alternative for generating new objects with mutations

```
const updatedContact = Object.assign({}, this.state.contact, { name:'John' });
this.setState({ contact: updatedContact });
```

Object destructuring

Instead of

```
const updated = Object.assign({}, this.state.contact, { name:'John' });
```

you can also write

```
const updated = { ...this.state.contact, name:'John' };
```

which creates a *new* object with all properties copied from `this.state.contact`, and then overwriting `{ name:'John' }`.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Array destructuring

Instead of

```
const updated = this.state.contacts.slice(0);
updated.push({ name:'John' });
```

you can also write

```
const updated = [ ...this.state.contacts, { name:'John' } ];
```

which creates a *new* array with all elements copied from `this.state.contacts`, and then adding `{ name:'John' }`.

immutability-helper

For more complex scenarios, you could use immutability-helper.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Hooks

- Introduced in React 16.8 (2019)
- Let you "hook into" the component lifecycle
- Defined as methods on `React`

```
import React, { useState } from 'react';
useState(...)

// equivalent

import React from 'react';
React.useState(...)
```

`useState`

- Similar to using a JavaScript class with `this.state`.
- Without classes, or callbacks not being bound to `this`.
- Works for primitive values as well as object structures.
- Returns the state itself and a updater function in an array.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

useState

```
const Counter = () => {
    const [ counter, setCounter ] = React.useState(0);
    const increase = (e) => setCounter(counter + 1);
    const decrease = (e) => setCounter(counter - 1);

    return <div>Counter: { counter }<br />
                <button onClick={ increase }>   +   </button>  
                <button onClick={ decrease }>   -   </button>
           </div>
}
```

useEffect

- Similar to using `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` (at once)
- Applies a *side effect* to the component being rendered
- Can optionally be cleaned up afterwards (e.g. `clearTimeout`);

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

useEffect

```
const Clock = () => {
    const [ time, setTime ] = React.useState(undefined);
    React.useEffect(() => {
        // effect
        const id = setInterval(() => {
            setTime(new Date());
        }, 1000);

        return () => { // clean-up
            clearInterval(id);
        }
    }, []); // empty array

    return <div>Current time: { time }</div>
}
```

useContext

- Similar to *consuming* a Context
- More concise

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

useContext

```
const UserContext = React.createContext();

const UserDisplay = () => {
    const user = React.useContext(UserContext);
    return <div>Welcome, { user ? user.name : 'guest' }</div>;
};

const InBetween2 = () => <UserDisplay />
const InBetween1 = () => <InBetween2 />

const Producer = ({ name }) => {
    const user = name ? { name } : undefined;
    return (<UserContext.Provider value={ user }>
              <InBetween1 />
            </UserContext.Provider>);
}
```

Data Fetching over HTTP

Prerequisites:

1. A JavaScript function to fetch some data
2. A component to show the fetched data

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

1. A data retrieval API

```
const checkStatus = (response) => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response);
  } else {
    return Promise.reject(`HTTP error ${response.status}: ${response.statusText}`);
  }
};

const parseJson = (response) => response.json();

const url = '/api/v1/me';
const getProfile = () => {
    return fetch(url)
        .then(checkStatus)
        .then(parseJson)
        .then(response => response.profile);
};
```

2. A component to show the fetched data

```
const MyProfile = () => {
    const [ { profile, loading }, setState ] = React.useState({ loading: true });
    const fetchProfile = async () => {
        const profile = await getProfile();
        setState({ loading: false, profile });
    }
    React.useEffect(() => {
        fetchProfile()
    }, [ ]);

    if (loading) return <div>Loading...</div>

    return <div>{ profile }</div>;
};
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Testing

Testing

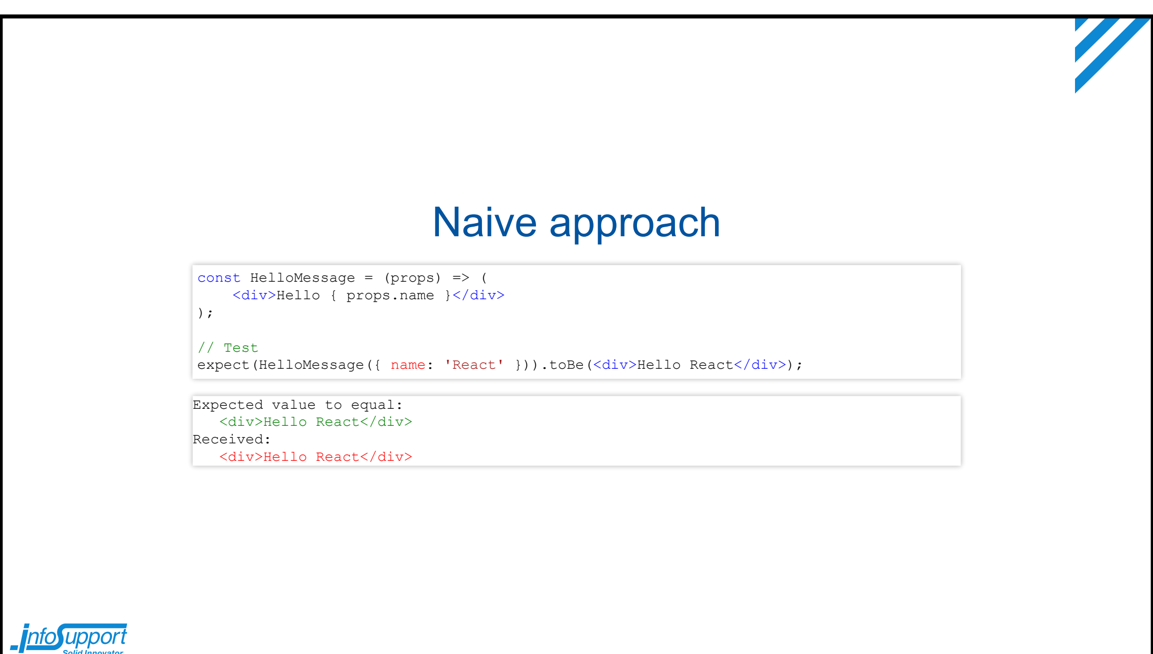
Do your test your code?

Why would you actually test your code?

How would you test *front end* code?

How do you assess the quality of your tests?

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



Why doesn't that work?

The `render()` method returns a JSX-element which gets translated to

```
return React.createElement(...)
```

Invoking that twice will *always* return a new React element of the given type.

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Unit Testing in React

Unit testing React Component is done with

- Jest (testing platform & library)
- the react-testing-library
- Enzyme, a JavaScript testing utility for React
  - provides a way to render a React component in a unit test
  - make assertions about its output and behaviour

Setup

Install jest and enzyme

```
npm install --save-dev jest enzyme enzyme-adapter-react-16
```

Configure the Enzyme adapter in **src/setupTests.js**:

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Shallow Rendering

Tests a component as a unit. Do not depend on behaviour of child components.

```
import { shallow } from 'enzyme';

describe("<HelloMessage />", () => {
    it("should render text", () => {
        const wrapper = shallow(<HelloMessage name="React" />);
        expect(wrapper.text()).toBe("Hello React");
    });
});
```

Full rendering

Tests a component by mounting it in the DOM. You can inspect both the subtree and lifecycle. Requires jsdom or a browser environment.

```
import { mount } from 'enzyme';
// create a mock/stub of some API using framework of choice
const dummy = ...

describe("<HelloMessage />", () => {
    it("should call API during mount", () => {
        const wrapper = mount(<HelloMessage name="React" />);
        expect(dummy).toHaveBeenCalled();
    });
});
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Testing events

How to test event handling?

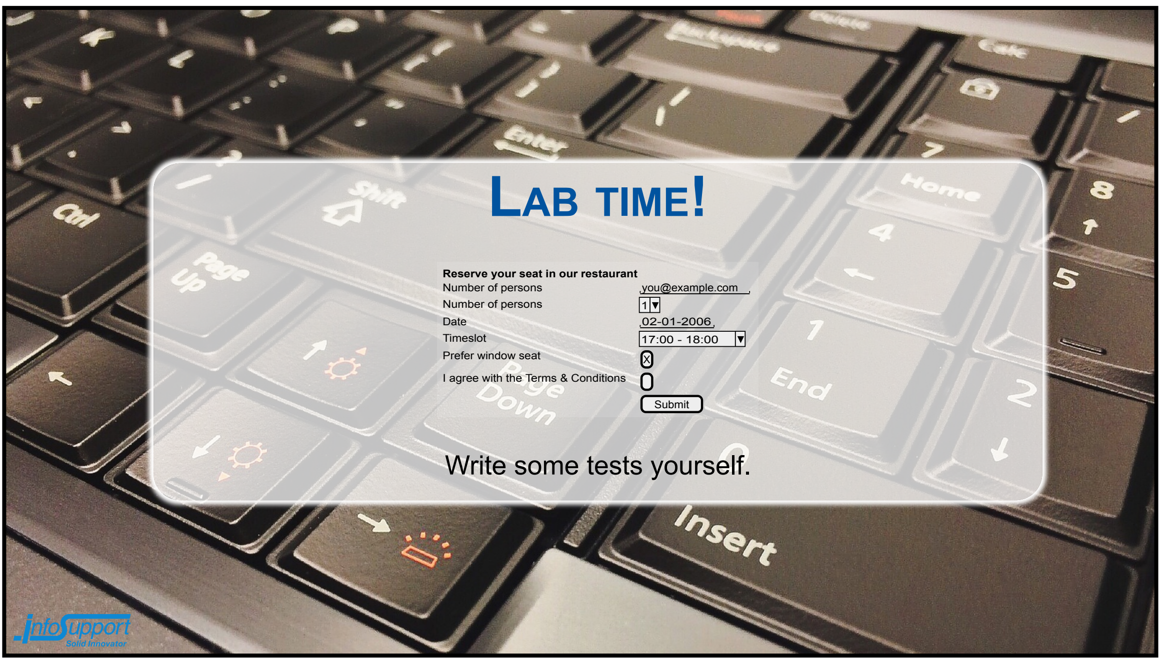
```
import { mount } from 'enzyme';

describe("<AwesomeButton />", () => {
    it("should invoke action on click", () => {
        const dummy = ... // create a mock/stub of some API using framework of choice
        const wrapper = mount(<AwesomeButton action={ dummy } />);
        wrapper.find('a').simulate('click');
        expect(dummy).toHaveBeenCalled();
    });
});
```

Testing events

How to test event handling?

```
import { mount } from 'enzyme';

describe("<AwesomeButton />", () => {
    it("should invoke action on click", (done) => {
        const dummy = ... // create a mock/stub of some API using framework of choice
        const wrapper = mount(<AwesomeButton action={ dummy } />);
        wrapper.find('a').simulate('click');
        setTimeout(() => {
            expect(dummy).toHaveBeenCalled();
            done();
        }, 50);
    });
});
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support



Performance and Tooling

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Performance and Tooling

React tries to optimise performance by

- minimizing DOM operations.
- batching state updates.

But: **you** need to know how the reconciliation process works.

Also, React provides tools to

- inspect and finetune your application.

But: **you** need to know/find bottlenecks in your application code

Reconciliation process

Reconciliation process happens when state causes a component to re-render. When `setState` is called

- component is marked as 'dirty'
- component is re-rendered, but
  - changes do not effect immediately
- changes are batched and effected in next loop cycle
- DOM therefore only updates once per event loop cycle

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Subtree Rendering

Components marked 'dirty' cause subtree re-rendering

- All children of the 'dirty' component are re-rendered
- React renders an in-memory virtual DOM

You can prevent a child from re-rendering when necessary

- `shouldComponentUpdate` is called before re-rendering
- gives you an option to performance tune rendering
- occasionally needed; prevent premature optimisation because
  - code gets more complicated
  - it can lead to hard to track bugs

Memoization

Alternatively, apply *memoization*:

- when props are equal, the outcome of the previous render is re-used
- useful when rendering (calculating the tree) is expensive
- *all-or-nothing* - you cannot exclude certain props

```
const Something = (props) => (<div>something</div>);

export default React.memo(Something);
```

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Tooling

There is tooling to inspect your app

- Install the *React Developer Tools* in your browser (Firefox or Chrome)
- Developer Toolkit gets an extra tab: "React"
  - Inspect (shows component tree with props and state)
  - Performance (visualises rendering)

Bonus Topics

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

Menu

- Higher Order Components
- Custom Hooks
- TypeScript

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support

© Info Support · All Rights Reserved · This data may not be copied or distributed without the prior approval of Info Support