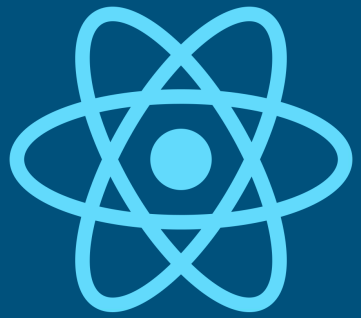
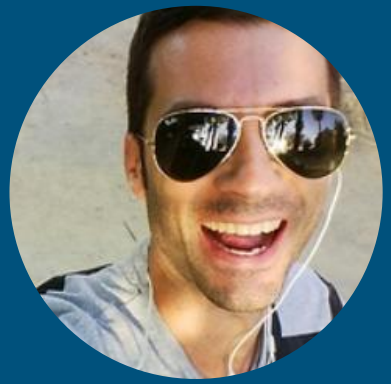
React





About me

Jose Ignacio Paris
@jiparis, http:s//github.com/jiparis
https://es.linkedin.com/in/jiparis

Full Stack dev at bitnami

https://github.com/jiparis/react-meetup

React

Principles:

- **View** (presentation) framework
- **one - way** binding
- Compose UI as **component trees**
- High performance
- **ONLY the VIEW LAYER** ☜

Components

00.helloworld

```
class MyApp extends React.Component {
  render(){
    return (
      <h1>Hello from React!</h1>
    );
  }
}

ReactDOM.render(<MyApp />, document.getElementById('app'));
```

JSX

Components

JSX

- It's NOT HTML
- React has NO TEMPLATES.

How?

Transpilers convert **JSX** to **React.createElement** calls, which internally calls browser's document.createElement

BABEL

Properties and state

Every component has:

- A set of properties (in the form of ~~HTML~~ JSX attributes)

```
this.props
```

- A state (components are stateful by default)

```
this.getState()
```

State changes trigger re-renders

Properties and state

01.propsandstate

```
class MyApp extends React.Component {
  constructor(props){
    super(props);
    // we need to use the React's state to trigger UI updates
    this.state = ({ count: props.initial || 0 });
  }

  render(){
    return (
      <h1>Current count: {this.state.count}</h1>
    );
  }
}
```

Properties and state

01.propsandstate

Properties Validations (only in dev mode)

```
MyApp.defaultProps = {
  initial: 0,
  step: 3
};

MyApp.propTypes = {
  initial: React.PropTypes.number.isRequired,
  step: React.PropTypes.number
};
```

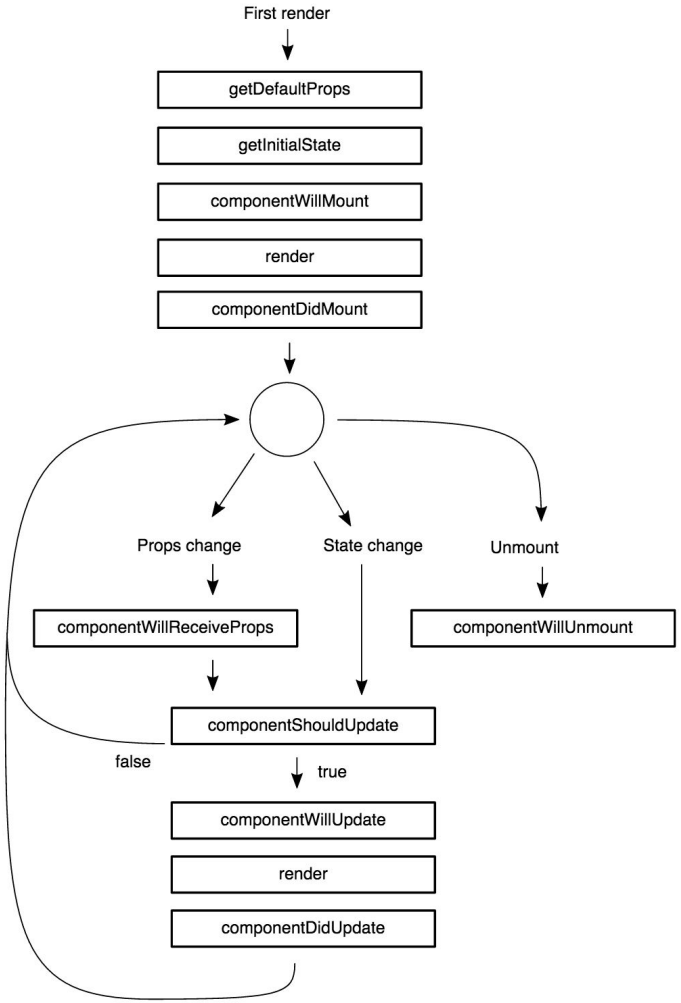
Component Lifecycle

A component is function of:

- Its properties (immutable, passed from its parent)
- Its state (changed with setState())

☞ Data flows in only one direction

☞ There is no two-way binding



Credits: https://gist.github.com/chris-martin/2424924e7a7494d5f98c

Component Lifecycle

Some useful lifecycle hooks:

`componentDidMount(){}.` **Called right after attaching to DOM.**

Good for AJAX calls, event handlers, animations...

`componentWillUnMount(){}.` **Called just before deattaching from DOM**

Good for cleaning things up

`componentShouldUpdate(){}.` **Tweak the default render path**

Component Lifecycle

02.lifecycle

```
class MyApp extends React.Component {
  ...
  componentDidMount(){
    this.handle = setInterval(() => {
      this.setState({ count: this.state.count + this.props.step});
      console.log('Updating state');
    }, 1000);
  }

  componentWillUnmount(){
    clearInterval(this.handle);
  }
 ...
}
```

Toolchain

03.toolchain

Dependency management: npm

Packager: webpack

Transpiler: babel

Editor: Atom + plugins

Chrome and React Developer Tools

JavaScript or ES2015??

Composition

04.composition

Pattern: containers and components

Presentational Components (Only UI).

Can be written as a function of its properties

```
const Label = ({title, header}) => (
  <h1>{title}</h1>
  <h4>{header}</h4>
)

ReactDOM.render(
  <Label title="Hello world" header="React component!" />,
  document.getElementById("root")
)
```

Composition

04.composition

Pattern: containers and components

Containers

Usually maintain state, have business logic, and are parents of other components

```
toggleTodo(id, checked){
  const item = todoList.find(el => el.id === id);
  item.done = checked;
  this.setState({ todos: todoList });
}

render(){
  return (
    <div>
      <h1>Mimimalist Todo List App</h1>
      <TodoList todos={this.state.todos} onTodoClick={this.toggleTodo}/>
    </div>
  );
}
```

DOM reconciliation

- DIFF algorithm, based on what you see (DOM), not the component tree.
- Extremely fast (That's why one way binding and re-render is enough)

Optimize DOM operations

```
renderA: <div id="before" />
renderB: <div id="after" />
=> [replaceAttribute id "after"]
```

```
renderA: <div style={{color: 'red'}} />
renderB: <div style={{fontWeight: 'bold'}} />
=> [removeStyle color], [addStyle font-weight 'bold']
```

https://facebook.github.io/react/docs/reconciliation.html

DOM reconciliation

04.composition

Special case with lists

```
renderA: <div><span>first</span></div>
renderB: <div><span>second</span><span>first</span></div>
=> [replaceAttribute textContent 'second'], [insertNode <span>first</span>]
```

Solution: keys

```
<div className="todo-list">
{
  this.props.todos.map (todo =>
    <TodoItem key={todo.id}
              onTodoClick={this.props.onTodoClick}
              {...todo} />
  )
}
</div>
```

https://facebook.github.io/react/docs/reconciliation.html

Async (ajax)

05.async

Use whatever you like:

`jQuery.ajax()`, plain `XMLHTTPRequest`...

Or use modern **github/fetch (+ promises)**

https://github.com/github/fetch

```
npm install whatwg-fetch babel-polifill --save
```

Async (ajax)

05.async

Example with reddit (r/frontend):

(Remember to use **componentDidMount**)

```
componentDidMount(){
  fetch('http://www.reddit.com/r/frontend.json')
    .then(response => response.json())
    .then(json => this.setState(json.data.children));
}
```

Redux

06.redux

A framework for state management

Principles of redux (state cycle):

- ACTIONS (events)
- REDUCERS (state **generators**)
- STORE (where the state lives)

States are not reused nor updated. We must ALWAYS generate a new state

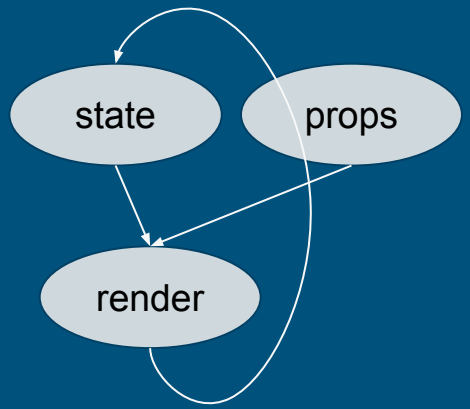
With redux, TDD is a first class methodology — USE CASE oriented

Redux

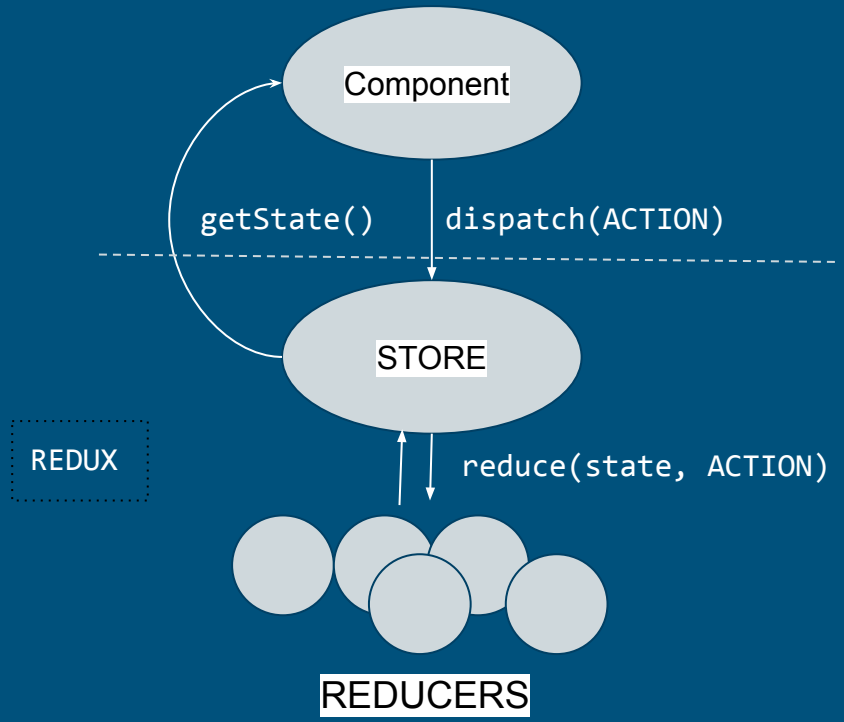
06.redux

So our app becomes:

FROM



TO



Redux and React

07.redux-async

PROBLEM: Dispatching actions: require passing STORE everywhere

We can use a global `store` object

Or more React-friendly: use `context`

Use `<Provider>` to pass Store in context to any "`connected`" component

Redux and React

07.redux-async

`<Provider/>`: make store available in `this.context`

```
const store = createStore(
  todoReducer
);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Redux and React

07.redux-async

Connect

Generates containers

Link container to stores and auto subscribe to them

`mapStateToProps` and `mapDispatchToProps` 💩

```
// Map REDUX STATE to component properties
const mapStateToProps = (state) => {
  return {
    todos: state.todos
  }
};

// Used to hook Redux into our container component
export default connect(
  mapStateToProps
)(App);
```

Redux and React

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Redux (async)

07.redux-async

Where do we make server requests?

- **Reducers**? No. 💩 reducers only can return a new state
- **Actions**? Yes ... but ... weren't actions simple flag objects? 💩

  Yes, they were: say welcome to Thunks:

```
npm install redux-thunk --save
```

* see also redux-saga library

Redux (async)

07.redux-async

Thunk action creators:

return a `function (dispatch, getState) => { ... }` instead of a hash

Can dispatch other actions

```
...
export function getTodos(){
  return (dispatch) => {
    fetch('/todos')
      .then(response => response.json())
      .then((todos) => {
        dispatch(todosLoaded(todos))
      });
  }
}
...
```

```
...
const store = createStore(
  todoReducer,
  applyMiddleware(
    thunkMiddleware
  )
);
...
```

Patterns

UNDO: using an enhanced reducer

```
npm install redux-undo --save
```

```
// REDUCER
import undoable, { distinctState } from 'redux-undo'

const todos = (state = [], action) => {
  /* OUR REDUCER */
}

const undoableTodos = undoable(todos, {
  filter: distinctState()
});

export default undoableTodos
```

```
// UndoRedo component
import { ActionCreators as UndoActionCreators } from 'redux-undo'

...
let UndoRedo = ({ canUndo, canRedo, onUndo, onRedo }) => (
  <p>
    <button onClick={onUndo} disabled={!canUndo}>
      Undo
    </button>
    <button onClick={onRedo} disabled={!canRedo}>
      Redo
    </button>
  </p>
)
...
const mapDispatchToProps = (dispatch) => {
  return {
    onUndo: () => dispatch(UndoActionCreators.undo()),
    onRedo: () => dispatch(UndoActionCreators.redo())
  }
}
```

Patterns

Project organization: component type vs feature

Small projects

```
.

├── actions
├── components
├── containers
└── reducers
```

Large projects

```
.

├── Feature0
    ├── actions
    ├── components
    ├── containers
    └── reducers
├── Feature1
    ├── actions
    ├── components
    ├── containers
    └── reducers
...
```

Conclusions

React (only) is simple to understand

But it's just the UI. We need everything else

There are no conventions:
- (Flux, Redux, plain OOP ...)
- But also: AngularJS, Backbone ...

Check your requirements before starting with React!

Thanks !