Dependency Injection - Using React's context (prior v. 16.3)

In this lesson, we will be learning to use dependency injection as a means to manage dependencies in React.

Dependency Injection

Many of the modules/components that we write have dependencies. Proper management of these dependencies is critical for the success of the project. There is a technique (most people consider it a *pattern*) called *dependency injection* that helps to solve the problem.

In React the need for dependency injector is easily visible. Let’s consider the following application tree:

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}

// Header.jsx
import Title from './Title.jsx';

export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}

// App.jsx
import Header from './Header.jsx';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { title: 'React in patterns' };
  }
  render() {
    return <Header />;
  }
};
```

The string “React in patterns” should somehow reach the `Title` component. The direct way of doing this is to pass it from `App` to `Header` and then `Header` pass it down to `Title`. However, this may work for these three components but what happens if there are multiple properties and deeper nesting. Lots of components will act as a proxy, passing properties to their children.

We already saw how the higher-order component may be used to inject data. Let’s use the same technique to inject the `title` variable:

```
// inject.jsx
const title = 'React in patterns';

export default function inject(Component) {
  return class Injector extends React.Component {
    render() {
      return (
        <Component
          {...this.props}
          title={ title }
        />
      )
    }
  };
}

// -----------------------------------
// Header.jsx
import inject from './inject.jsx';
import Title from './Title.jsx';

var EnhancedTitle = inject(Title);
export default function Header() {
  return (
    <header>
      <EnhancedTitle />
    </header>
  );
}
```

The `title` is hidden in a middle layer (higher-order component) where we pass it as a prop to the original `Title` component. That’s all nice but it solves only half of the problem. Now we don’t have to pass the `title` down the tree but how this data reaches the `inject.jsx` helper.

Using React’s context (prior v. 16.3)

In v16.3 React’s team introduced a new version of the context API and if you are going to use that version or above you’d probably skip this section.

React has the concept of *context*. The *context* is something that every React component has access to. It’s like an event bus but for data. A single *store* which we access from everywhere.

```
// a place where we will define the context
var context = { title: 'React in patterns' };

class App extends React.Component {
  getChildContext() {
    return context;
  }
  ...
};
App.childContextTypes = {
  title: React.PropTypes.string
};

// a place where we use the context
class Inject extends React.Component {
  render() {
    var title = this.context.title;
    ...
  }
}
Inject.contextTypes = {
  title: React.PropTypes.string
};
```

Notice that we have to specify the exact signature of the context object with `childContextTypes` and `contextTypes`. If those are not specified then the `context` object will be empty. That can be a little bit frustrating because we may have lots of stuff to put there. That is why it is a good practice that our `context` is not just a plain object but it has an interface that allows us to store and retrieve data. For example:

```
// dependencies.js
export default {
  data: {},
  get(key) {
    return this.data[key];
  },
  register(key, value) {
    this.data[key] = value;
  }
}
```

Then, if we go back to our example, the `App` component may look like this:

```
import dependencies from './dependencies';

dependencies.register('title', 'React in patterns');

class App extends React.Component {
  getChildContext() {
    return dependencies;
  }
  render() {
    return <Header />;
  }
};
App.childContextTypes = {
  data: React.PropTypes.object,
  get: React.PropTypes.func,
  register: React.PropTypes.func
};
```

And our `Title` component gets it's data through the context:

```
// Title.jsx
export default class Title extends React.Component {
  render() {
    return <h1>{ this.context.get('title') }</h1>
  }
}
Title.contextTypes = {
  data: React.PropTypes.object,
  get: React.PropTypes.func,
  register: React.PropTypes.func
};
```

Ideally we don't want to specify the `contextTypes` every time we need access to the context. This detail may be wrapped again in a higher-order component. And even better, we may write a utility function that is more descriptive and helps us declare the exact wiring. I.e instead of accessing the context directly with `this.context.get('title')` we ask the higher-order component to get what we need and pass it as props to our component. For example:

```
// Title.jsx
import wire from './wire';

function Title(props) {
  return <h1>{ props.title }</h1>;
}

export default wire(Title, ['title'], function resolve(title) {

  return { title };
});
```

The `wire` function accepts a React component, then an array with all the needed dependencies (which are `register`ed already) and then a function which you may call `mapper`. It receives what is stored in the context as a raw data and returns an object which is later used as props for our component (`Title`). In this example we just pass what we get - a `title` string variable. However, in a real app, this could be a collection of data stores, configuration or something else.

Here is what the `wire` function looks like:

```
export default function wire(Component, dependencies, mapper) {
  class Inject extends React.Component {
    render() {
      var resolved = dependencies.map(
        this.context.get.bind(this.context)
      );
      var props = mapper(...resolved);

      return React.createElement(Component, props);
    }
  }
  Inject.contextTypes = {
    data: React.PropTypes.object,
    get: React.PropTypes.func,
    register: React.PropTypes.func
  };
  return Inject;
};
```

`Inject` is a higher-order component that gets access to the context and retrieves all the items listed under `dependencies` array. The `mapper` is a function that receives the `context` data and transforms it into props for our component.

Below is the full code that injects dependencies into the Title component through context:

```
export default {
  name: 'React in patterns'
};
```

In the next section, we will discuss how dependency injections work in later versions of React.