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

This lesson discusses how the context API has improved the implementation of dependency injection in later versions.

For years the context API was not really recommended by Facebook. They mentioned in the official docs that the API is not stable and may change. And that is exactly what happened. In version 16.3, we got a new one which is more natural and easy to work with.

Let's use the same example with the string that needs to reach a `<Title>` component.

We will start by defining a file that will contain our context initialization:

```
// context.js
import { createContext } from 'react';

const Context = createContext({});

export const Provider = Context.Provider;
export const Consumer = Context.Consumer;
```

`createContext` returns an object that has `.Provider` and `.Consumer` properties. Those are actually valid React classes. The `Provider` accepts our context in the form of a `value` prop. The consumer is used to access the context and basically read data from it. And because they usually live in different files it is a good idea to create a single place for their initialization.

Let's say that our `App` component is the root of our tree. That is where we have to pass the context.

```
import { Provider } from './context';

const context = { title: 'React In Patterns' };

class App extends React.Component {
  render() {
    return (
      <Provider value={ context }>
        <Header />
      </Provider>
    );
  }
};
```

The wrapped components and their children now share the same context. The `<Title>` component is the one that needs the `title` string so that is the place where we use the `<Consumer>`.

```
import { Consumer } from './context';

function Title() {
  return (
    <Consumer>{
      ({ title }) => <h1>Title: { title }</h1>
    }</Consumer>
  );
}
```

Notice that the `Consumer` class uses the function as children (render prop) pattern to deliver the context.

The new API feels easier to understand and eliminates the boilerplate. It is still pretty new but looks promising. It opens a whole new range of possibilities.

Using the module system

If we don't want to use the context there are a couple of other ways to achieve the injection. They are not exactly React specific, but worth mentioning nevertheless. One of them is to use the module system.

As we know, the typical module system in JavaScript has a caching mechanism. It's nicely noted in Node's documentation:

> Modules are cached after the first time they are loaded. This means (among other things) that every call to require('foo') will get exactly the same object returned, if it would resolve to the same file.

> Multiple calls to require('foo') may not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

How does that help in our injection? Well, if we export an object we are actually exporting a singleton and every other module that imports the file will get the same object. This allows us to `register` our dependencies and later `fetch` them in another file.

Let's create a new file called `di.jsx` with the following content:

```
var dependencies = {};

export function register(key, dependency) {
  dependencies[key] = dependency;
}

export function fetch(key) {
  if (dependencies[key]) return dependencies[key];
  throw new Error(`"${ key } is not registered as dependency.`);
}

export function wire(Component, deps, mapper) {
  return class Injector extends React.Component {
    constructor(props) {
      super(props);
      this._resolvedDependencies = mapper(...deps.map(fetch));
    }
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          {...this._resolvedDependencies}
        />
      );
    }
  };
}
```

We'll store the dependencies in `dependencies` global variable (it's global for our module, not for the whole application). We then export two functions `register` and `fetch` that write and read entries. It looks a little bit like implementing setter and getter functions against a simple JavaScript object.

Then we have the `wire` function that accepts our React component and returns a higher-order component. In the constructor of that component we are resolving the dependencies and later, while rendering the original component, we pass them as props. We follow the same pattern where we describe what we need (`deps` argument) and extract the needed props with a `mapper` function.

Having the `di.jsx` helper, we are once again able to register our dependencies at the entry point of our application (`app.jsx`) and inject them wherever (`Title.jsx`) we need.

```
// app.jsx
import Header from './Header.jsx';
import { register } from './di.jsx';

register('my-awesome-title', 'React in patterns');

class App extends React.Component {
  render() {
    return <Header />;
  }
};

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

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

// -----------------------------------
// Title.jsx
import { wire } from './di.jsx';

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

export default wire(
  Title,
  ['my-awesome-title'],
  title => ({ title })
);
```

If we look at the `Title.jsx` *file we'll see that the actual component and the* *wiring may live in different files. That way the component and the mapper function become easily unit testable.*

Final thoughts

Dependency injection is a tough problem. Especially in JavaScript. However, employing proper dependency management is a key process of every development cycle. The JavaScript ecosystem offers different tools that can be used for dependency management and we as developers should pick the ones that fit our needs best. Following is the complete dependency injection code for the example we have been following in this lesson:

```
import React from 'react';
import PropTypes from 'prop-types';
import { wire } from './di.jsx';

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

Title.propTypes = {
  title: PropTypes.string
};

export default wire(Title, ['my-awesome-title'], title => ({ title }));
```

Now that we know how to handle dependencies in React, the next step is to look at how to present elements in React. In the following chapter, we will discuss some miscellaneous topics including styling and integration of third-party libraries.