React in Practice

Spring 2024

React

React lets you build HTML code for your document/page/webapp by using a function `React.createElement()` that takes

- an HTML tag or a React component
- attributes
- sub-elements to put in the body

```
React.createElement('div', {},
   React.createElement('img', {src: 'https://example.com/goofy.jpg'}),
   React.createElement(Comp, {arg: 'something'}))
```

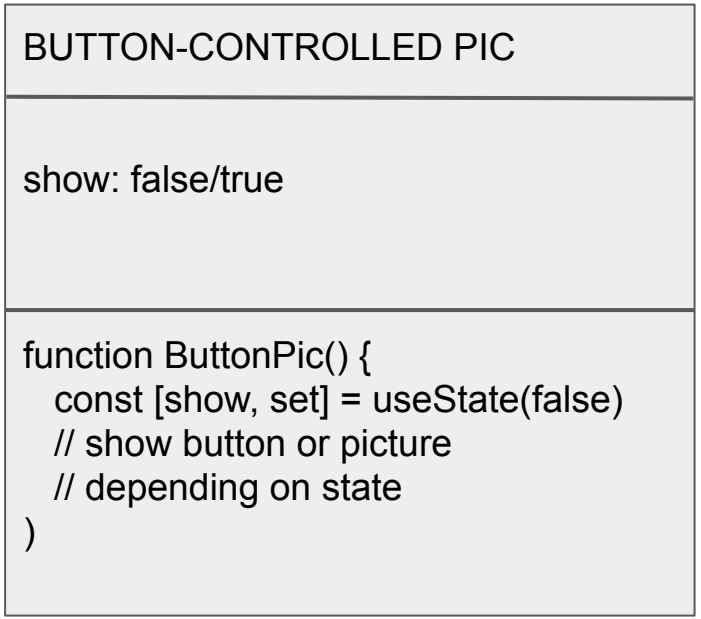
React components

Component = represented by

- a (render) **function** that returns the HTML code of the component
- some **component state**

Every time the component state changes, React re-renders the component by calling the function again to recreate the HTML

Can attach a component anywhere in an HTML document



Demo

- picture viewer split into different files
- App state
- array of thumbnails
- input forms

The Modern Era

Everything we did last time would have felt reasonably familiar to a web developer 15 years ago

It also doesn't scale

React in practice uses three additional features:

- handle your code split into different files
- supports the use of external packages
- provides a more convenient syntax for creating elements (JSX)

Detour: NodeJS

An infrastructure for programming in Javascript like you would in Python

- not tied to browser
- interactive shell, or command-line scripts
- program spread over multiple files (modules)
- can use package managed by an external tool
  - think `pip`

A NodeJS project

If no libraries used, can just create source files

- `main.js`
- `helper.js`

Files can be scripts or modules (like in Python)

NodeJS by default uses **CommonJS** modules

```
const r = require('./something')
```

The file "required" must explicitly export something

```
exports.helper_fn = helper_fn
```

Packages

If you want to use packages, you need a package manager to install and track dependencies

`npm init`

- creates **package.json**

package.json has multiple roles depending on what you do, but it tracks dependencies

`npm install` *`package`*

- adds dependency to package.json
- installs the package in `node_modules/`
- use with `require('`*`package`*`')`

Why the detour?

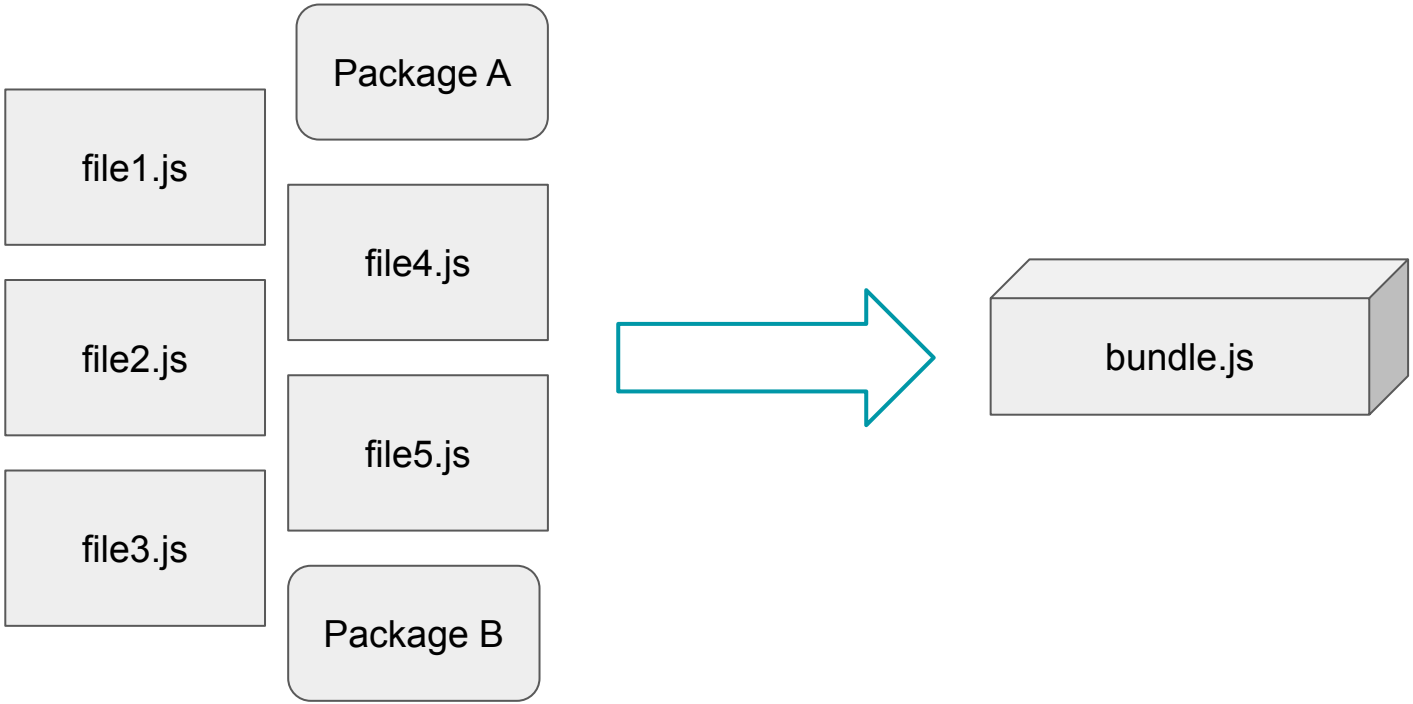
Modern front-end development tools tend to reuse the architecture set down by NodeJS (npm, package.json)

- can spread frontend code over multiple files (modules)
- can use packages installed via npm
- React itself is such a set of packages

But:

- frontend code needs to be ultimately run in the browser,
- browser don't know (much) about modules and packages
- need to bundle all source into a "single" .js file for deployment
    - bundlers: browserify, webpack, rollup, parcel, ...

Bundling


Package A
file1.js
file4.js
file2.js
file5.js
file3.js
Package B
bundle.js

Tools: create-react-app

It all gets super complicated super fast — and it changes every 6 months

People have developed tools to help manage this complexity

E.g., **create-react-app**:

NodeJS app to create a `package.json`, install the basic react packages, adds a bundler, and a way to run the bundler to create a distributable frontend

JSX

Writing components using React.createElement is clunky

JSX was introduced as a notation for writing an expression in Javascript that looks like HTML

A "translator" that's part of the bundling step translates those JSX expressions into calls to `React.createElement()`

JSX lets you combine HTML with Javascript

JSX

Compare

```
const url = 'https://example.com/goofy.jpg'
return React.createElement('div', {},
         React.createElement('img', {src: url}),
         React.createElement(Comp, {arg: 'something'}))
```

versus

```
const url = 'https://example.com/goofy.jpg'
return ( <div>
           <img src={url} >
           <Comp arg="something">
         </div> )
```

Demo

- create-react-app
- picture viewer as a JSX application