CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Modern Web UIs

Dynamic Web Content

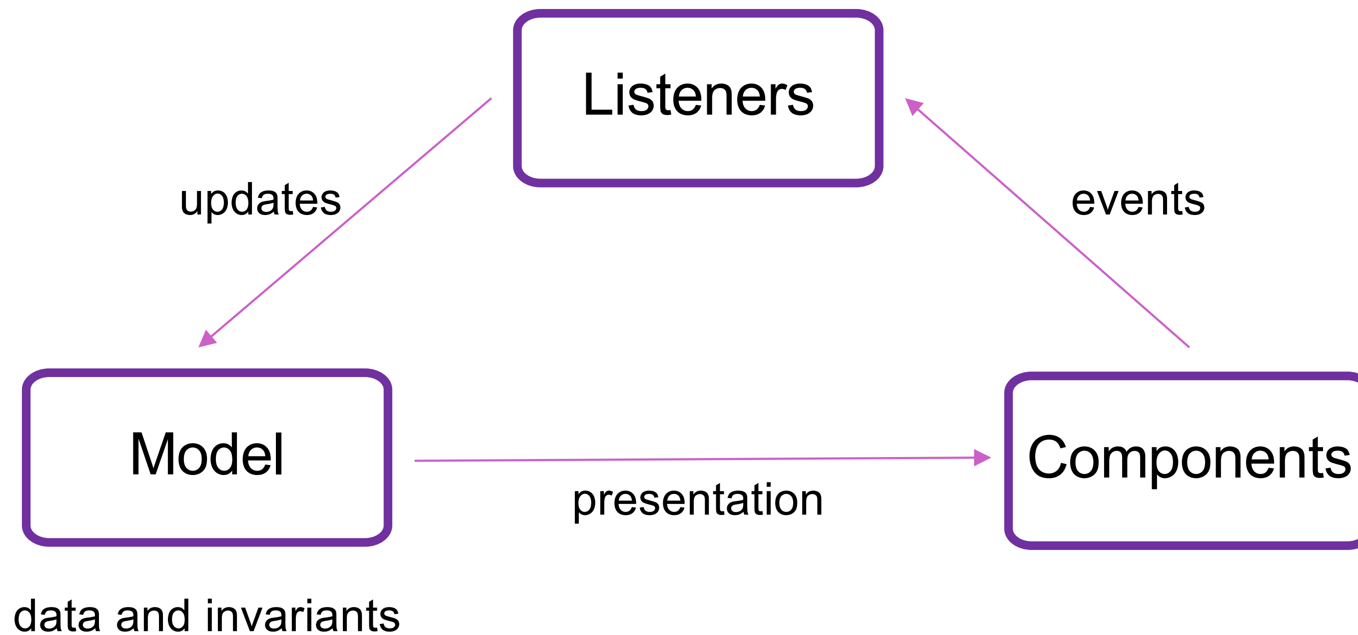
- Earlier example had a **fixed** set of components.
  - same for iPhone / Android apps

- More realistic apps need to change the set of components displayed on the screen dynamically
  - consider Gmail as an example
  - need the components to come from code

ES6 Example 1

register-js/index.js

Structure of a GUI



Problems

These tools can be used to write Gmail

But it has a number of problems…

1. Lack of tool support
   – no checking of types, tags, etc.
2. No support for modularity
   – all the code and UI in a single file
3. (…one more on Friday…)

JS Modules

- EcmaScript6 (ES6) added support for modules.
- Each file is a separate unit (“namespace”)
- Only exported names are visible outside:

```
export function average(x, y) { …
```

- Others can import using:

```
import { average } from ‘./filename’;
```

ES6 Example 2

register-js2/…

JS Classes

- ES6 added new syntax for classes:

```
class Foo {
  constructor(val) {
    this.secretVal = val;
  }

  secretMethod(val) {
    return val + this.secretVal;
  }
}
```

ES6 Example 2

register-js2/…

Problems

These tools can be used to write Gmail

But it has a number of problems…

1. Lack of tool support
   – no checking of types, tags, etc.
2. Limited support for modularity
   – whole UI in a single file
   – need to join strings into one big string

TypeScript

- Adds type constraints to the code:
  - arguments and variables

```
let x : number = 0;
```

  - fields of classes

```
quarter: string;
```

TypeScript Example

register-ts/…

TypeScript Types

- Basics from JavaScript:
  number, string, boolean, string[], Object

- But also
  - specific classes Foo
  - tuples: `[string, int]`
  - enums (as in Java)
  - allows null to be included or excluded (unlike Java)
  - `any` type allows any value
  - …

TypeScript

- Type casts
  - `x as Foo` is an unchecked cast to `Foo`
  - `x!` casts to non-null version of the type (useful!)

- Full description of the language at `typescriptlang.org`

Problems

This is better, but it still has problems…

1. Still no checking of HTML (opaque strings)

2. Limited support for modularity
   – whole UI in a single file
   – need to join strings into one big string

JSX

- Fix the first problem by adding HTML as a JS type
- This is supported in `.jsx` files:

```
let x = <p>Hi, {name}.</p>;
```

- Compiler can now check that this is valid HTML
- {…} replaced with string value of expression

Problems

This is even better, but it still has problems…

1. Limited support for modularity
   – whole UI in a single file
   – need to join strings into one big string

React

- Regain modularity by allowing custom tags

```
let app = (
    <div>
      <TitleBar name="My App"/>
      <EditPane rows="80" />
    </div>);
```

- `TitleBar` and `EditPane` can be separate modules
  - their HTML gets substituted in these positions

React

- Custom tags implemented using classes

```
class TitleBar extends React.Component {
```

- Attributes (`name="My App"`) passed in `props` arg
- Method `render` produces the HTML for component
- Framework joins all the HTML into one blob
  - can update in a single call to `innerHTML = …`

React Example

register-react/…

JSX Gotchas

- Put (..) around HTML if it spans multiple lines
- Cannot use `class="btn"` in your HTML
  - `class`, `for`, etc. are reserved words in JS
  - use `className`, `htmlFor`, etc.
- Must have a single top-level tag:
  - not: `return <p>one</p><p>two</p>;`
  - usually fixed by wrapping those parts in a `div`

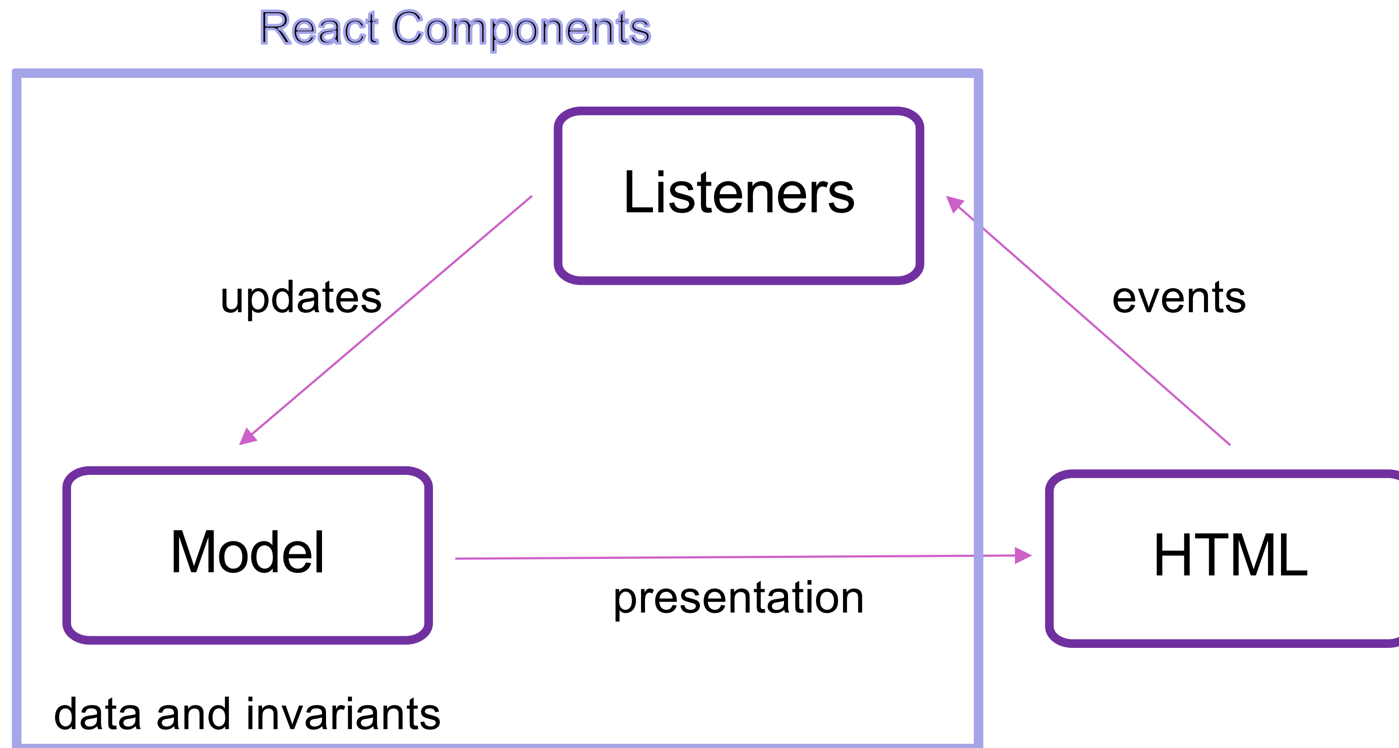
React State

- Last example was not dynamic!
  - there was no model
- Components become dynamic by maintaining state
  - stored in fields of `this.state`
  - call `this.setState({field: value})` to update
- React will respond by calling `render` again
  - will automatically update the HTML to match the HTML produced by this call

Example 5

register-react2/…

Structure of a React Application



React Gotchas

- Model must store all data necessary to generate the exact UI on the screen
  - react may call `render` at any time
  - must produce identical UI

- Any state in the HTML components must be mirrored in the model
  - e.g., every text field's `value` must be part of some React component's state
  - render produces

```
<input type="text" value={…}>
```

React Gotchas

- `render` should not have side-effects
  - only *read* `this.state` in render
- Never modify `this.state`
  - use `this.setState` instead
- Never modify `this.props`
  - read-only information about parent's state
- Not following these rules may introduce bugs that will be hard to catch!

React Gotchas

- `setState` does not update state instantly:

```
// this.state.x is 2
this.setState({x: 3});
console.log(this.state.x); // still 2!
```

- Update occurs after the event finishes processing
  - `setState` adds a new event to the queue
  - work is performed when that event is processed
- React can batch together multiple updates