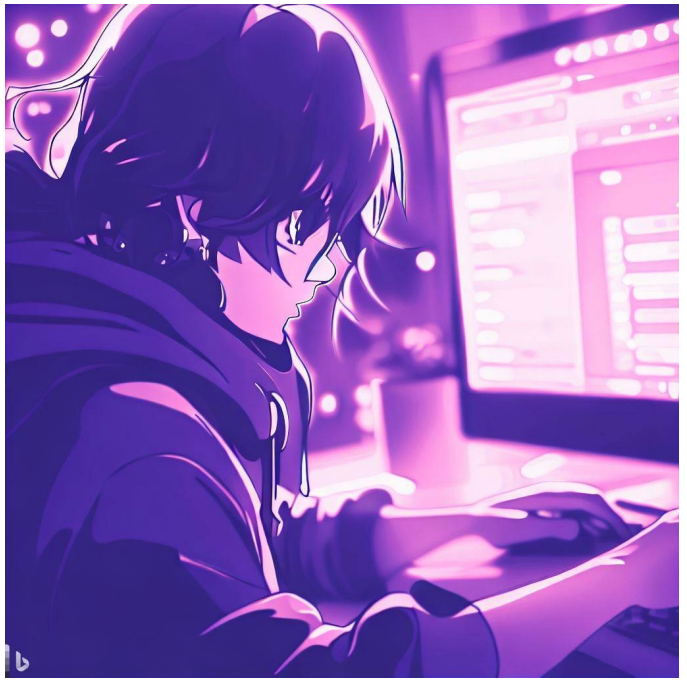


CSE 331

Stateful UI in React

Kevin Zatloukal

Administrivia

- **Midterm on Friday (in class)**
  - **topics that are fair game include:**

    levels, testing, functional code, induction

    imperative code, correctness of loops, ADTs
  - **review in section Thursday**

13 Lecture *Stateful UI in React*	14	15 Lecture *Full-Stack Apps I* 23:00 HW7 due	16 Section *Midterm Review*	17 10:30-11:20 Midterm exam B 14:30-15:20 Midterm exam A
20 Lecture *Full-Stack Apps II*	21	22 Lecture *App Design*	23 Thanksgiving	24 Native American Heritage Day
27 Lecture *Aliasing*	28	29 Lecture *Mutable ADTs* 23:00 HW8 due	30 Section *HW9 prep*	01 Lecture *Subtypes*

Administrivia

- **HW8 out next Monday**
  - **have 10 calendar days due to Thanksgiving**
  - **puts the pieces together**

    work on both the **client** and **server** (so-called “full stack”)

- **Sec8 will be available over the weekend**
  - **not to be submitted, but helpful as usual**

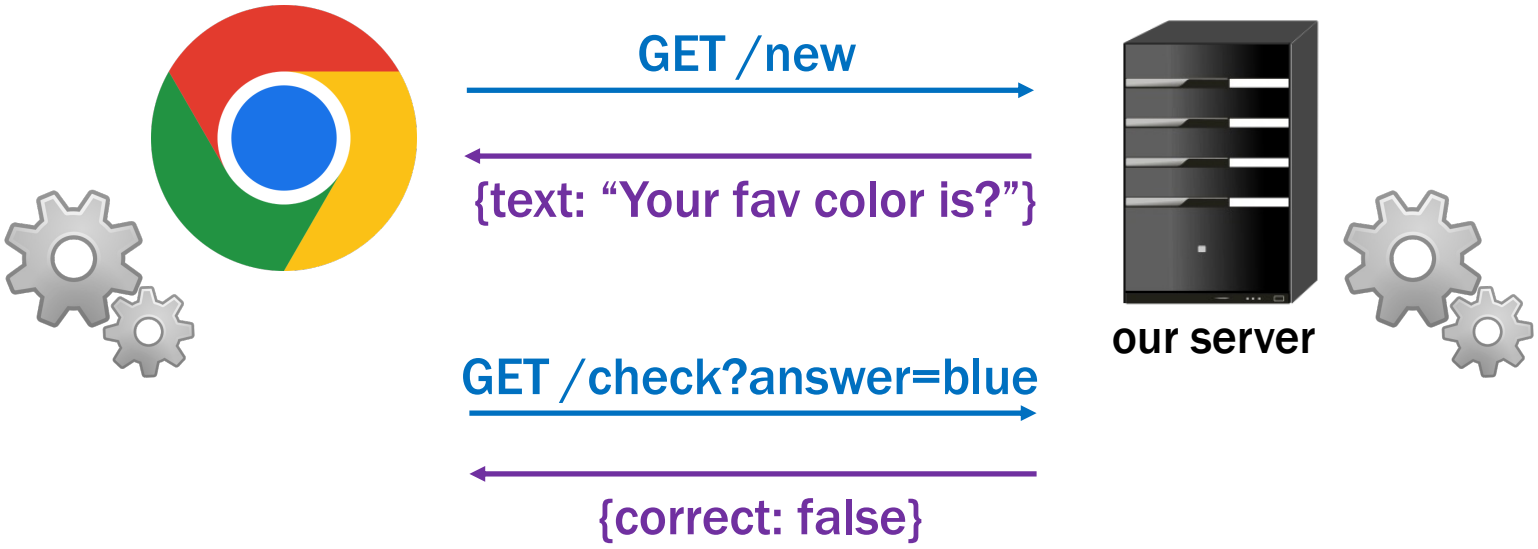
13 Lecture *Stateful UI in React*	14	15 Lecture *Full-Stack Apps I* 23:00 HW7 due	16 Section *Midterm Review*	17 10:30-11:20 Midterm exam B 14:30-15:20 Midterm exam A
20 Lecture *Full-Stack Apps II*	21	22 Lecture *App Design*	23 Thanksgiving	24 Native American Heritage Day
27 Lecture *Aliasing*	28	29 Lecture *Mutable ADTs* 23:00 HW8 due	30 Section *HW9 prep*	01 Lecture *Subtypes*

Stateful UI in React
(React Components)

Client-Server Interaction

- Client needs to update the UI after getting response
  - don't want to reload the whole page to redraw

    reloading is slow and can lose user data (e.g., contents of text fields)
  - need a way to update the UI without a reload



UI in HW1-4

- **UI so far was static**
  - `index.tsx` **calls** `render` **to show a fixed UI**

    UI was different based on query params

    but never changed once rendered

- **Made the UI change by reloading the page**
  - **change the query params, so it renders something different**

UI in HW1-4

- Made the UI change by reloading the page
  - change the query params, so it renders something different

http://localhost:8080/

Word: wooow
Algorithm: Crazy Caps
encode decode
Submit

http://localhost:8080/?word=wooow&...

WoOoW

```
const word = params.get("word");
if (word === null) {
  root.render(<MakeForm/>);
} else {
  root.render(<ShowResults word={word} ../>);
}
```

React Functions

- React let us create custom tags
  - e.g., from HW3

```
root.render(<SquareElem square={sq}/>);
```

  - acts like the call

```
root.render(SquareElem({square: sq}));
```

  - where SquareElem is function taking a record argument

```
const SquareElem = (props: {square: Square}): JSX.Element => {..};
```

- HTML returned by the function is displayed
  - "`SquareElem`" tag is in the HTML
  - `render` spots it, calls the function, and replaces the tag

React Components

- **Use a class via** `<HiElem name={"Fred"}/>`
  - **acts like the call**

```
root.render(new HiElem({name: "Fred"}).render());
```

  - **React instantiates the class and calls its** `render` **method**

- **Can do the same with a class (a React Component):**

```
type HiProps = {name: string};

class HiElem extends Component<HiProps, {}> {
  constructor(props: HiProps) {..}
  render = (): JSX.Element => {..}
}
```

React Components

- Can do the same with a class (a React Component):

```
type HiProps = {name: string};

class HiElem extends Component<HiProps, {}> {
  constructor(props: HiProps) {
    super(props);
  }
  render = (): JSX.Element => {
    return <p>Hi, {this.props.name}</p>;
  };
}
```

- React calls render to get the HTML to display
  - constructor stores argument in a field called "`props`"

React Components

- Can do the same with a class (a React Component):

```
type HiProps = {name: string};

class HiElem extends Component<HiProps, {}> {
  constructor(props: HiProps) {
    super(props);
  }
  render = (): JSX.Element => {
    return <p>Hi, {this.props.name}</p>;
  };
}
```

No sensible reason to make Components without state

- Component is a generic type
  - first type parameter is the type of "`props`"
  - second type parameter is for "`state`"...

React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: "Hi"};
  }
```

- **Component is a generic type**
  - **first component is type of `this.props` (readonly)**
  - **second component is type of `this.state`**

    initial value set in the constructor

    never *directly* modified after that

React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {

  render = (): JSX.Element {
    return <p>{this.state.greeting},
              {this.props.name}!</p>;
  };
```

- `render` **can use both** `this.props` **and** `this.state`
  - **difference 1: caller give us props, but we set our state**
  - **difference 2: we can *change* our state**
  - **React will automatically re-render when state changes**

    re-render happens shortly after the state change

React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  …
  setGreeting = (newGreeting: string): void => {
    this.setState({greeting: newGreeting});
  };
}
```

- **Must call `setState` to change the state**
  - **directly modifying `this.state` is a (painful) bug**

    our linter will prevent this, thankfully

- **React will automatically re-render when state changes**
  - **this is the (only) reason to use a Component**

React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  …
  setGreeting = (newGreeting: string): void => {
    this.setState({greeting: newGreeting});
  };
}
```

- **Must call `setState` to change the state**
  - **directly modifying `this.state` is a (painful) bug**

    our linter will prevent this, thankfully

- **Only need to supply the fields that have changed**
  - **all the other fields will stay as they were before**

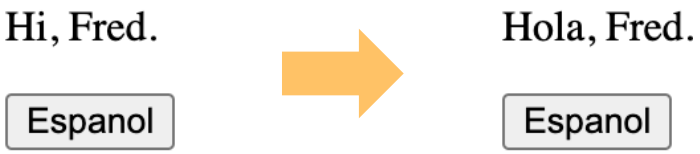
React Components

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: "Hi"};
  }

  render = (): JSX.Element {
    return <p>{this.state.greeting},
              {this.props.name}!</p>;
  };

  setGreeting = (newGreeting: string): void => {
    this.setState({greeting: newGreeting});
  };
}
```

React Components

```
type HiProps = {name: string};
type HiState = {curName: string};

class HiElem extends Component<HiProps, HiState> {
  …
  setGreeting = (newGreeting : string): void => {
    this.setState({greeting: newGreeting});
  };
}
```

- **How could `setGreeting` be called?**
  - **typically happens in a handler for an HTML event**



React Component with an Event Handler

- **Pass method to be called as argument (a "callback")**
  - **value of `onClick` attribute is our `makeSpanish` method**

```
render = (): JSX.Element {
   return (<div>
        <p>{this.state.greeting}, {this.props.name}!</p>
        <button onClick={this.doEspClick}>Espanol</button>
     </div>);
 };
```

- **Browser will invoke that method when button is clicked**

```
doEspClick = (evt: MouseEvent<HTMLButtonElement>) => {
   this.setState({greeting: "Hola"});
 };
```

  - **Call to `setState` causes a re-render (in a bit)**

React Component with an Event Handler

```
type HiProps = {name: string};
type HiState = {greeting: string};

class HiElem extends Component<HiProps, HiState> {
  constructor(props: HiProps) {
    super(props);
    this.state = {greeting: "Hi"};
  }

  render = (): JSX.Element {
    return (<div>
        <p>{this.state.greeting}, {this.props.name}!</p>
        <button onClick={this.doEspClick}>Espanol</button>
      </div>);
  };

  doEspClick = (evt: MouseEvent<HTMLButtonElement>) => {
    this.setState({greeting: "Hola"});
  };
```

React Component with an Event Handler

- **Pass method to be called as argument (a "callback")**
  - **value of `onClick` attribute is our `makeSpanish` method**

```
render = (): JSX.Element {
   return (<div>
        <p>{this.state.greeting}, {this.props.name}!</p>
        <button onClick={this.doEspClick()}>Espanol</button>
      </div>);
 };
```

- **Including parentheses here is a (painful) bug!**
  - **that would call the method inside render**

    passing its return value as the value of the `onClick` attribute
  - **we want to pass the method to the button, and have it called when the click occurs**

React Components are Like ADTs

```
type HiProps = {name: string};
type HiState = {greeting: string};
```

- "Props" are part of the specification (arguments)
  - public interface, used by clients

```
root.render(<Hi name={"Fred"}/>);  // pass in name
```

- "State" is like the concrete representation
  - private choice of data structures, hidden from clients

```
constructor(props: HiProps) {
  super(props);
  this.state = {greeting: "Hi"};   // initial state
}
```

React Components are Like ADTs

- Can have RIs on state as well
  - write down any necessary facts not included in the types

```
// RI: 0 <= index < options.length
type OptionState = {
  options: string[],
  index: number
};
```

- Good idea to write code to double check this
  - a `checkRep` method is good defensive programming

    (see also `CheckInv1` in HW7 for complex loops)

React Components are Like ADTs

- HTML on the screen is a (hidden) part of the state
  - components work with React to manage this state

- `render` method is like an AF
  - function applied to the state to make something important
  - defines what it looks like, rather than what it means

- Components have an extra invariant like an RI

HTML on screen = render(this.state)

React Components are Like ADTs

HTML on screen = render(this.state)

	Component	React
t = 10	this.state = s_1	doc = $HTML_1$ = render(s_1)
t = 20	this.setState(s_2)	
t = 30	this.state = s_2	doc $HTML_2$ = render(s_2)

React updates this.state **to** s_2 **and** doc **to** $HTML_2$ ***simultaneously***

React Components are Like ADTs

- **Components have an extra invariant like an RI**

  HTML on screen = render(this.state)

  – **don't want to be in a state where that is not true**

  unless you like **painful** debugging!

  1. **Do not mutate `this.state` (call `setState`)**

     React will update `this.state` and HTML on screen at the same time

  2. **Make sure no data on screen would disappear on re-render**

     More on this later…

React Components are Level 3

- Like ADTs, methods are sharing state
  - change in one method is read in other methods
- Error in one method (writing) fails in another (reading)
  - debugging will be harder!
- Error in the server fails in the client (or vice versa)
  - debugging will be harder!
- HW8-9 are the debugging assignments
  - necessary to understand all the parts of the code

React Components are Level 3

- Hard debugging makes correctness more important
- Move complex parts into separate functions
  - test and reason carefully through those functions
  - class is ideally just be rendering and event handlers

    move everything complex into helper functions

    e.g., calculation of new state can be a helper function
  - harder to reason about and test Level 3, so keep it simple
- Write code to check your invariants
  - ensure the new state is valid before calling `setState`
  - practice defensive programming

Example: To-Do List

TodoApp – State

```
// Represents one item in the todo list.
type Item = {
  name: string;
  completed: boolean;
};

// State of the app is the list of items and the text that the
// the user is typing into the new item field.
type TodoState = {
  items: Item[];     // existing items
  newName: string;   // mirrors text in the field to add a new name
                     //   (need this for two reasons…)
};
```

TodoApp – Class

```
// Top-level application that lets the user pick a quarter and
// then pick classes within that quarter.
export class TodoApp extends Component<{}, TodoState> {

  constructor(props: {}) {
    super(props);
    this.state = {items: [], newName: ""};
  }

  …
```

TodoApp – Render

```
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
               value={this.state.newName}
               onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
                onClick={this.doAddClick}>Add</button>
      </p>
    </div>);
}
```

TodoApp – Render Items (abbreviated)

```
renderItems = (): JSX.Element[] => {
  const items: JSX.Element[] = [];
  for (let i = 0; i < this.state.items.length; i++) {
    if (!this.state.items[i].completed) {
      items.push(
        <div className="form-check" key={i}>
          <input className="form-check-input" type="checkbox"
                 id={"check" + i} checked={false}
                 onChange={evt => this.doItemClick(evt, i)} />
          <label className="form-check-label" htmlFor={"check"+i}>
            {this.state.items[i].name}
          </label>
        </div>);
    } else { … /* read-only once completed */ }
  }
  return items;
};
```

TodoApp – Render

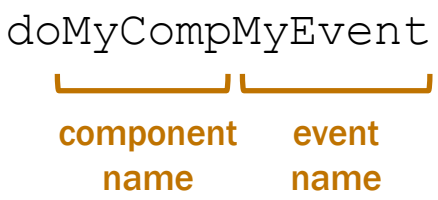
```
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
               value={this.state.newName}
               onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
                onClick={this.doAddClick}>Add</button>
      </p>
    </div>);
}
```

TodoApp – Add Click

```
// Called when the user clicks on the button to add the new item.
doAddClick = (_: MouseEvent<HTMLButtonElement>): void => {
  // Ignore the request if the user hasn't entered a name.
  const name = this.state.newName.trim();
  if (name.length == 0)
    return;

  // Cannot mutate this.state.items! Must make a new array.
  const items = this.state.items.concat(
      [ {name: name, completed: false} ]);
  this.setState({items: items, newName: ""});  // clear input box
};
```

Event Handler Conventions

- We will use this convention for event handlers



  - e.g., `doAddClick`, `doNewNameChange`

- Reduces the need to explain these methods
  - method name is enough to understand what it is for
  - method name is the only thing you know they read

- Components should be just rendering & event handlers

TodoApp – Render

```
// Return a UI with all the items and elements that allow them to
// add a new item with a name of their choice.
render = (): JSX.Element => {
  return (
    <div>
      <h2>To-Do List</h2>
      {this.renderItems()}
      <p className="instructions">Check an item to mark it…</p>
      <p className="more-instructions">New item:
        <input type="text" className="new-item"
               value={this.state.newName}
               onChange={this.doNewNameChange} />
        <button type="button" className="btn btn-link"
                onClick={this.doAddClick}>Add</button>
      </p>
    </div>);
}
```

TodoApp – New Name Change

```
// Called each time the text in the new item name field is changed.
doNewNameChange = (evt: ChangeEvent<HTMLInputElement>): void => {
  this.setState({newName: evt.target.value});
}
```

- Most event handlers are passed an event object
  - field "`evt.target`" stores the object that fired the event
  - hence, "`evt.target.value`" is the text in that input box

- Make sure no data on screen would disappear on re-render
  - must record the text the user typed into the field
    goes into the value={..} attribute of the input box
  - otherwise, render would produce an input box with no text

Other Events

- Components should be just rendering & event handlers
  - our linter will enforce this
- Timers have events that fire after a given time
  - call to `setTimeout` invokes callback after a delay
- React also includes events about its "life cycle"
  - `componentDidMount`: UI is now on the screen
  - `componentDidUpdate`: UI was just changed to match render
  - `componentWillUnmount`: UI is about to go away
  - will see a reason to need `componentDidMount` next time...