Building User Interfaces

React 3

Component Lifecycle

Professor Bilge Mutlu

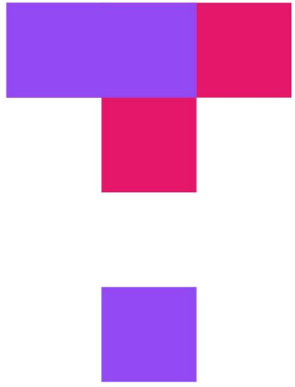
What we will learn today?

» Let's build an app!

» The component lifecycle

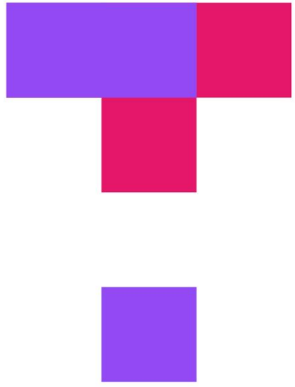
» Assignment Preview

TopHat Attendance



TOP HAT

TopHat Questions



TOP HAT

Let's build an application

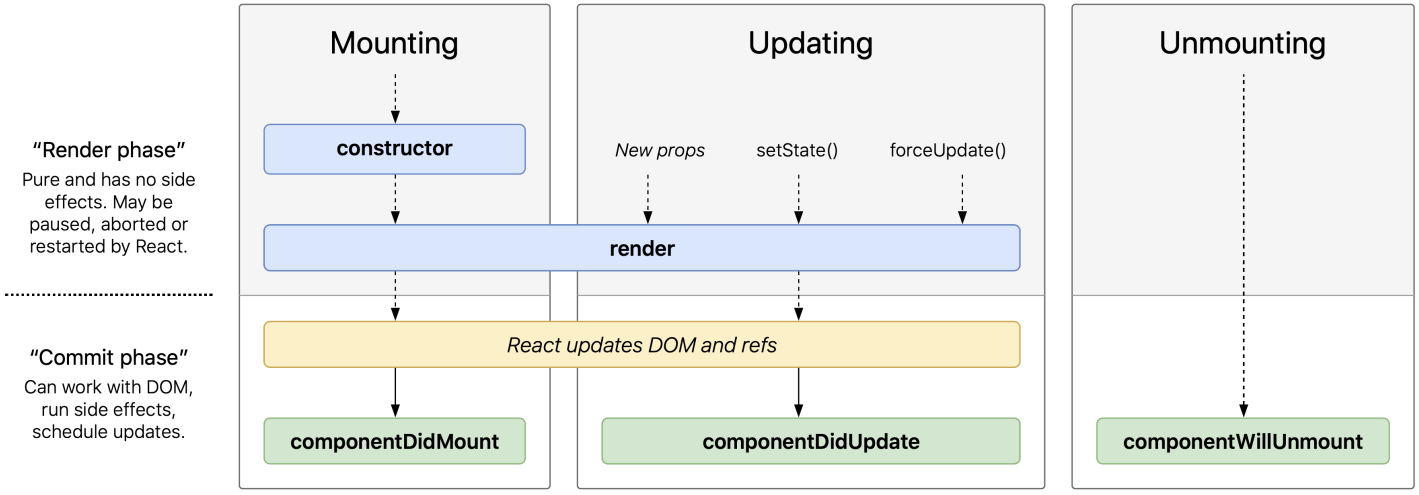
What will we need?

- Node.js, ReactJS, terminal, code environment, browser
- Adobe XD, Zeplin
- Data from APIs:
  - https://randomuser.me/
  - http://www.randomtext.me/api/

Let's build!

The Component Lifecycle

The Component Lifecycle[1]



[1] Wojciech Maj

The Component Lifecycle[2] [3] is made up of three actions:

1. **Mounting** -> Preparation
2. **Updating** -> Re-rendering
3. **Unmounting** -> removing / cleanup

Each action has a number of *lifecycle methods* associated with it and *render* and *commit* phases.

We will use a StackBlitz to illustrate all three actions.[4]

---

[2] ReactJS.org: React.Component

[3] The (new) React lifecycle methods in plain, approachable language

[4] See on StackBlitz

Mounting

Definition: *Mounting* is the process of creating an instance of a component and inserting it into the DOM.

Commonly used mounting lifecycle methods:

1. `render()`
2. `constructor()`
3. `componentDidMount()`

Mounting: `render()`

`render()` is the only required method within a class component, reading `this.props` and `this.state` and returning:

- **React elements**, adding a single element to the container
- **Arrays & fragments**, rendering multiple elements
- **Portals**, adding children to a DOM subtree
- **String & numbers**, rendering text nodes in the container
- **Booleans | null**, rendering nothing

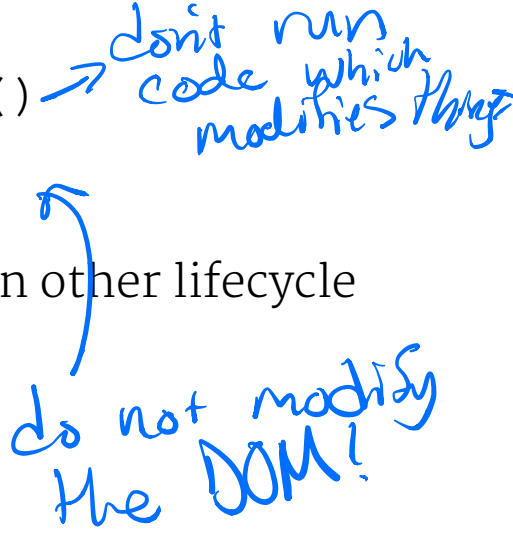
Mounting: `render()`, *continued*

`render()` has to remain *pure*, executing exactly the same way every time:

» no `state` updates are allowed within `render()`

» `render()` does not interact with the browser

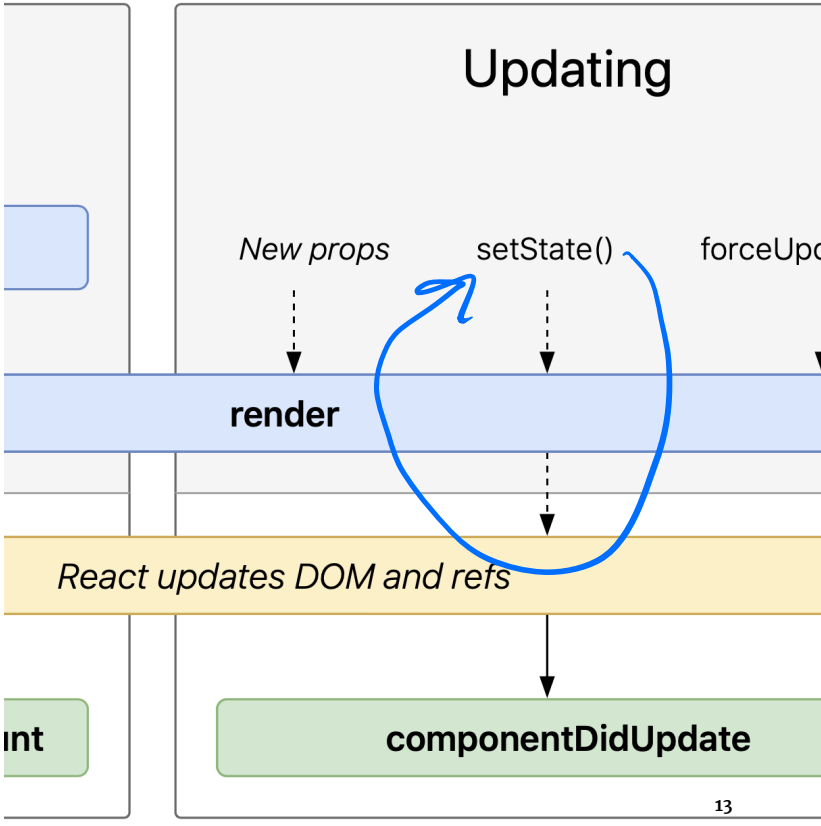
Interactions with the browser should happen in other lifecycle methods.



Mounting: `render()`, *continued*

What will happen if `setState()` is called in `render()`?

» Infinite loop > Stack overflow



Mounting: `constructor()`

only override if you need to set state

`constructor()` is only needed to inherit `props`, to initialize `state`, and to bind event-handling functions.

`super(props)` should be called before any other statement, and all other statements should come after it.

`constructor()` is the only place where we should directly assign state using `this.state = { key: value }` and `this.setState()` method should not be used here.

do this elsewhere!

you can only do this here!

Mounting: `constructor()`, *continued*

```
constructor(props) {
    // inherit props
    super(props);
    // set states
    this.state = { key: 'value' };
    // bind event-handling functions
    this.handleClick = this.handleClick.bind(this);
}
```

set initial state

any binding etc.

Mounting: `componentDidMount()`

`componentDidMount()` is automatically called as soon as the component is mounted following `render()`.

This give us an opportunity to do anything we did not want to do in `render()`, e.g., to initiate API calls, request data, etc, before the browser is updated.

Pro Tip: Unlike in `render()`, `setState()` method can be used in `componentDidMount()`. `setState()` will trigger a re-render before the browser reflects the update. State updates here should be used sparingly (e.g., to determine where a tooltip should be rendered) to maintain performance.

Demo!

Updating

→ suppose there is an update to state or props

Definition: *Updating* involves re-rendering a component following changes to `props` or `state`.

Commonly used updating lifecycle methods:

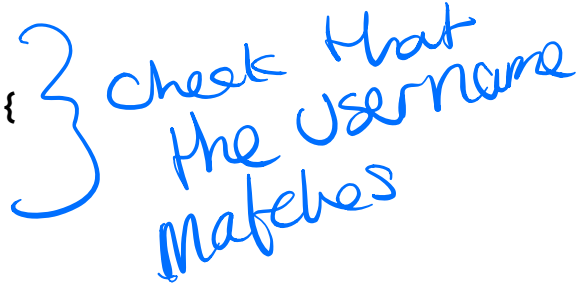
1. `render()`
2. `componentDidUpdate()`

Updating: `componentDidUpdate()`

`componentDidUpdate(prevProps, prevState, snapshot)` is invoked as soon as there is an update.

Again, this is an opportunity to do anything we do not want to do within `render()`, e.g., placing network requests.

```
componentDidUpdate(prevProps) {
 if (this.props.userName !== prevProps.userName) {
   this.fetchData(this.props.userName);
 }
}
```



Unmounting

Definition: *Unmounting* involves removing a component from the DOM.

Unmounting lifecycle method:

1. `componentWillUnmount()`

stop any timers, api calls, etc.
if you don't, you may keep slowing down the page!

Unmounting: `componentWillUnmount()`

`componentWillUnmount()` is invoked as soon as a component is unmounted — an opportunity to perform any necessary cleanup, e.g., resetting counters, invalidating timers, canceling network requests.

`setState()` method should not be called within `componentWillUnmount()` as it will never be rendered.

Key considerations in using `state`

Why is `state` so important?

Remember that a `state` update is how React knows that a component needs to be re-rendered. Once an application is loaded, all React does is to monitor changes to `state` and re-render components based on the changes.

React bases its changes on State and Props

How `state` should be updated

State should not be modified directly. The following will not re-render the component:

```
this.state.TAName = 'Andy';
```

We must use `setState()`:

```
this.setState({TAName: 'Andy'});
```

Considering asynchronous updates

Because React may batch-process state updates to improve performance, `state` and `props` may be updated asynchronously. Former may not update the counter, but the latter will.

Because updates may be asynchronous, subsequent attempts to access the `state` may not provide updated information.

```
this.setState({ counter: this.state.counter + 1 });
console.log(this.state.counter);
```

async

may not be changed!

May not increment:

```
this.setState({
    counter: this.state.counter + this.props.increment
});
```

Will ensure increment:

anonymous function

```
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Complex state manipulations[5] [6]

- Adding to and removing from arrays
- State updates from children

[5] See in solutions CodePen

[6] A good article on managing `state` with arrays

Assignment Preview

Some Announcements

- We have made changes to the upcoming build assignments
  - React 3 will be the final deliverable; due in 2.5 weeks
  - React 4 will be extra credit; due at the same time

React 3

Deliverable options for React 3:

1. Course **recommender** application
2. Course **planner** application

React 3: Recommender

Problem 1

Load in a json file of previous courses located at https://mysqlcs639.cs.wisc.edu/classes. The user should be able to view the contents at this url as courses that the user has previously taken.

https://mysqlcs639.cs.wisc.edu/students/5022025924/classes/completed

Problem 1: Suggested Workflow

1. Fetch data from server
2. Create a new component to represent previously taken courses. This component will look somewhat like the Course component, but it will be simpler and won't have options to add the course to the cart.
3. Create a new component to hold the previously taken courses. Make this component accessible from the app (maybe another tab on the top or a tab within the cart page).

Problem 2

Create a rating system for previously taken courses. The user should be able to rate some or all of the previously taken courses loaded from the `json` file.

Problem 2: Suggested Workflow

1. Create a component for rating a specific course.
2. Create a component for holding all of the rating components.
3. Make the holder accessible from the search tab.
4. Save the data from the holder in the state of the lowest ancestor of any component that will need it

Problem 3

Create a way for the user to select areas of interest that you define. These areas can be general or specific. Some examples might be `computer science`, `artificial intelligence`, *or* `science`.

Problem 3: Suggested Workflow

1. Generate a list of interest areas based on the course data.
2. Create a component for the user to select interest areas as defined in step 1.
3. Make this component accessible from the search tab.

Problem 4

Recommend courses to the user based off of the rated previously taken courses and the user's specified interest areas.

Problem 4: Suggested Workflow

1. Create the recommender algorithm that takes in the rated courses and interest areas, searches through subjects and keywords, and returns courses most similar to the highly rated courses and the courses that match the most interest areas.
2. Display the recommended courses to the user.

React 3: Planner

Problem 1

Based on the courses that the user has added to the cart, allow the user to select any subset of courses, sections, and subsections to later generate all possible schedules for. The user should be able to select 3 slight variations of course information for planning:
1. A course with ***all sections and subsections***
2. A course with ***one specific section*** *of a course with* ***all subsections***
3. A course with ***one specific section*** *that contains* ***one specific subsection***

Problem 1: Suggested Workflow

1. Create a new planner tab
2. Create a component that displays all of the courses, sections, and subsections in the cart with check boxes next to each one
3. Store the data of what is checked to make the check boxes related (if a course is checked, all of its sections and subsections will also be checked)

Problem 2

Based on the selections from problem 1, generate all possible schedules for the courses.

Problem 2: Suggested Workflow

1. Create a function to generate all of the possible schedules based on the data from PlannerSidebar

Problem 3

Create a way for the user to go through and view all of the possible schedules generated.

Problem 3: Suggested Workflow

1. With the provided components, create a Schedule component that displays a generated schedule
2. Create two buttons to switch between schedules by changing the data that is sent to the Schedule component

Implementations will be evaluated for:

» **Build**: efficiency, elegance, clean, and readable

» **Design**: usability, visual design, navigation model

My Simple Recommender

My simple recommender, *constraints*

Requirements:

» Users are given a set of elements to evaluate

» Evaluations are standardized into a ranking scheme

» The ranking scheme is used to look up matches

» Top match is returned as a recommendation

My simple recommender, *design decisions*

Design decisions:

» Give the user a *randomly generated* set of swatches

» Allow users to provide *like/dislike* ratings

» *Average out* the colors of liked swatches

» Give the user a recommend swatch *with the average*

Italics indicate the simplest possible implementation.

My simple recommender, *component structure*

```
<App />
    <Swatch />
    ...
    <RecommendedSwatch />
```

Plus, possibly a function component for ranking.

Planner visualization starter component[7]

The started code will give the base code to visualize courses on a given day, which you can extend to build the planner.

[7] See on StackBlitz

A few pieces of advice

- Start early
- Google (or Bing, DuckDuckGo, etc.) is your friend
  - E.g., even if we cover correct syntax in class, slides are not useful for debugging
- Use debugging tools
  - Compiler errors, React Development Tools, `console.log()`
- Come to office hours (early)

What did we learn today?

» Let's build an app!

» The component lifecycle

» Assignment Preview