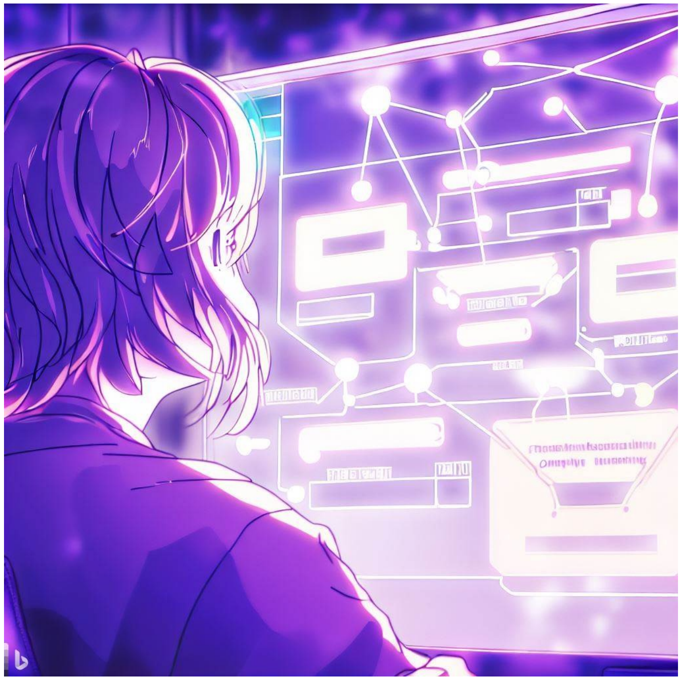


CSE 331

UI Modularity

Kevin Zatloukal

Last Time: Finishing Step 3 for To-Do List

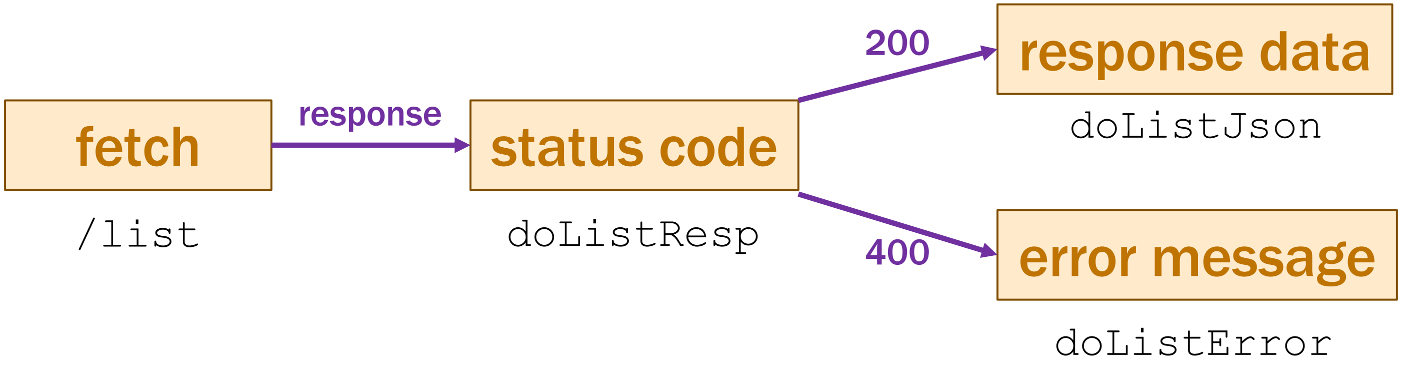
- Rewrote client-side To-Do App into client-server

- Instead of simply updating state:
  - make a request to the server to have it update state
  - once that completes, we update the client’s state
  - this keeps the two copies of the state in sync

Last Time: Fetch Requests Are Complicated

- **Four different methods involved in each fetch:**

  1. method that makes the fetch
  2. handler for fetch Response
  3. handler for fetched JSON
  4. handler for errors



Last Time: Finishing Step 3 for To-Do List

- Rewrote client-side To-Do App into client-server

- Instead of simply updating state:
  - make a request to the server to have it update state
  - once that completes, we update the client's state
  - this keeps the two copies of the state in sync

- App gets the list from the server…
  1. Initially
  2. 5 seconds after an item is completed

New TodoApp – Refresh Timeout

```
// Called to refresh our list of items from the server.
doRefreshTimeout = (): void => {
  fetch("/api/list").then(this.doListResp)
      .catch(() => this.doListError("failed to connect"));
};

// Called with the response from a request to /api/list
doListResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doListJson)
       .catch(() => this.doListError(”200 response is not JSON"));
  } else if (res.status === 400) {
    res.text().then(this.doListError)
       .catch(() => this.doListError(”400 response is not text"));
  } else {
    this.doListError(`bad status code ${res.status}`);
  }
};
```

New TodoApp – Refresh Timeout

```
// Called with the JSON response from /api/list
doListJson = (data: unknown): void => {
  if (!isRecord(data)) {
    console.error("bad data from /list: not a record", data)
    return;
  }

  const items = parseItems(data.items);
  if (items !== undefined)
    this.setState({items: items});
};

// Called when we fail trying to load the list from the server
doListError = (msg: string): void => {
  console.error(`Error fetching /list: ${msg}`);
};
```

New TodoApp – Refresh Timeout

```
// Called with the JSON response from /api/list
doListJson = (data: unknown): void => {
  if (!isRecord(data)) {
    console.error("bad data from /list: not a record", data)
    return;
  }

  const items = parseItems(data.items);
  if (items !== undefined)
    this.setState({items: items});
};
```

- often useful to move this type checking to helper functions

New TodoApp – parseItems

```
// Ensure that this is an array of items. Returns it with that type
// or undefined if invalid (after logging an error message).
const parseItems = (val: unknown): Item[] | undefined => {
  if (!Array.isArray(val)) {
    console.error(“not an array”, val);
    return undefined;
  }

  const items: Item[] = [];
  for (const item of val) {
    if (!isRecord(item) || typeof item.name !== 'string’ ||
        typeof item.completed !== 'boolean') {
      console.error(“not an item”, item);
      return undefined;
    } else {
      items.push({name: item.name, completed: item.completed});
    }
  }
  return items;
};
```

actual code has
3 separate cases

New TodoApp – Refresh Timeout

```
// Called with the JSON response from /api/list
doListJson = (data: unknown): void => {
  if (!isRecord(data)) {
    console.error("bad data from /list: not a record", data)
    return;
  }

  const items = parseItems(data.items);
  if (items !== undefined)
    this.setState({items: items});
};
```

- often useful to move this type checking to helper functions
- we provide code for this in HW8

  functions `toJson` / `fromJson` convert between `unknown` and `Square`

  (both directions sometimes needed since **not all JavaScript is valid JSON**)

For .. Of

```
for (const item of val)
```

- “for .. of” iterates through array elements *in order*
  - … or the entries of a `Map` or the values of a `Set`

    entries of a `Map` are (key, value) pairs
  - fine to use this now
  - no need to write an invariant for such loops

    do X for each Y is simple enough that we can skip the invariant

    (do not abuse this)

Lifecycle Events

Lifecycle Methods

- 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

- Often use “mount” to get initial data from the server
  - constructor shouldn’t do that sort of thing

```
componentDidMount = (): void => {
  fetch(“/api/list”)
    .then(this.doListResp)
    .catch(() => this.doListError(“connect failed”);
};
```

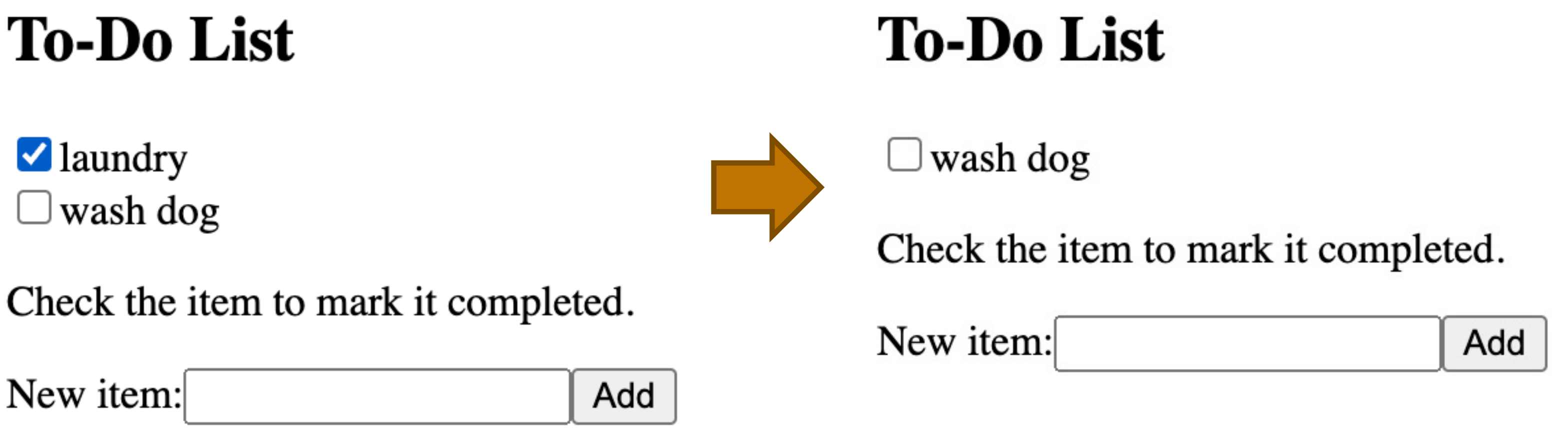
One More Change

- **Don’t have the items initially…**

```
type TodoState = {
  items: Item[] | undefined;  // items or undefined if loading
  newName: string;            // mirrors text in name-to-add field
};

renderItems = (): JSX.Element => {
  if (this.state.items === undefined) {
    return <p>Loading To-Do list...</p>;
  } else {
    const items = [];
    // … old code to fill in array with one DIV per item …
    return <div>{items}</div>;
  }
};
```

New TodoApp — Requests



To-Do List

☑ laundry
□ wash dog

Check the item to mark it completed.

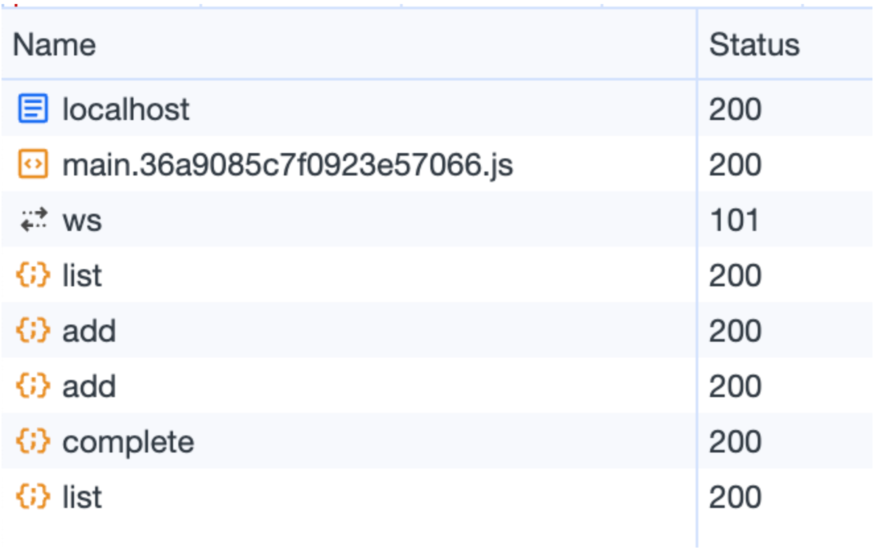
New item: [ ] Add

To-Do List

□ wash dog

Check the item to mark it completed.

New item: [ ] Add



Name	Status
localhost	200
main.36a9085c7f0923e57066.js	200
ws	101
list	200
add	200
add	200
complete	200
list	200

Summary of To-Do List Example

- Built it in the following order:

  1. Wrote the client UI with local data
     - no client/server interaction at the start

  2. Wrote the server
     - official store of the data (client state is ephemeral)
     - only provided the operations needed by the client
       - `/list` to get the list when the page loads
       - `/add` and `/complete` are the updates we make (no remove)

  could swap these

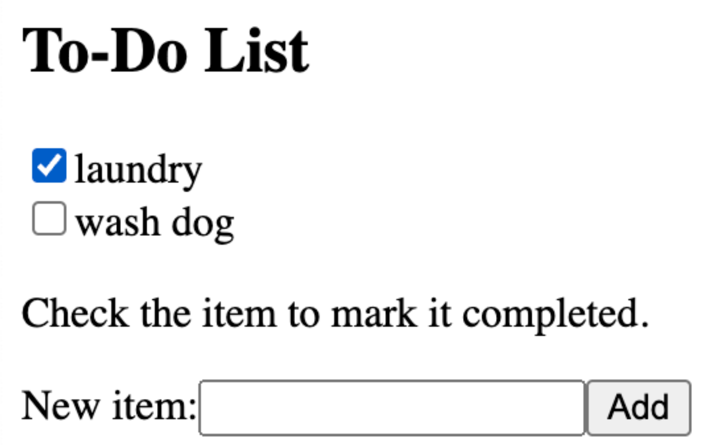
  3. Connected the client to the server
     - used fetch to update data on the server before doing same to client

- These are good steps to write any full-stack app

Another Example

More Complex UI

- To-Do List UI is basic
  - all of it easily fits in a single component (`TodoApp.tsx`)



To-Do List

☑ laundry
□ wash dog

Check the item to mark it completed.

New item:

- More complex UI can be too much code for one file
  - necessary to split it into multiple components

Recall: Other Properties of High-Quality Code

- Professionals are expected to write high-quality code

- Correctness is the most important part of quality
  - users **hate** products that do not work properly

- Also includes the following:
  - easy to understand
  - easy to change
  - modular

via abstraction

Component Modularity

- Poor design to put all the app in one Component
  - it works, but is lacks properties of high-quality code
  - better to break it into smaller pieces (modular)

- Two ways to the UI into separate components:

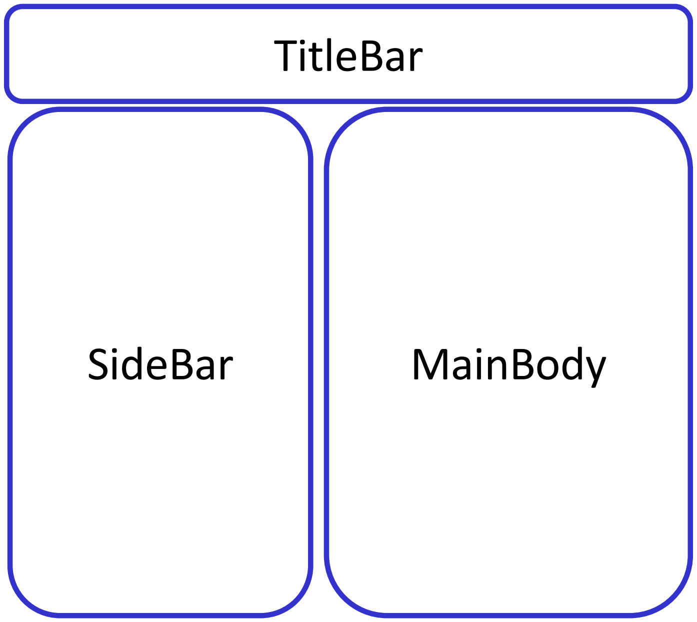
  1. Separate parts that are next to each other on screen

  2. Separate parts on the screen at different times

Component Modularity

- **Separate parts that are next to each other**

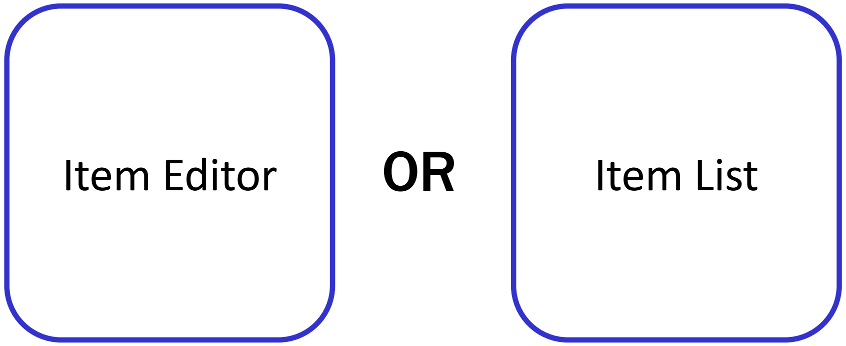
```
class App extends Component<..> {
  render = (): JSX.Element {
    return (<div>
        <TitleBar title={"My App"}/>
        <SideBar/>
        <MainBody/>
      </div>);
  };
}
```



Component Modularity

- Separate parts on the screen at different times

- App is always on the screen
  - App chooses which child component to display



  - sometimes it has an Editor child and sometimes not

Component Modularity

- **Separate parts on the screen at different times**

```
type AppState = {editing: boolean};

class App extends Component<{}, AppState> {
  …
  render = (): JSX.Element {
    if (this.state.editing) {
      return <ItemEditor item={this.state.item}/>;
    } else {
      return <ItemList/>;
    }
  };
  …
}
```

Example: Auctions

Recall: Steps to Writing a Full Stack App

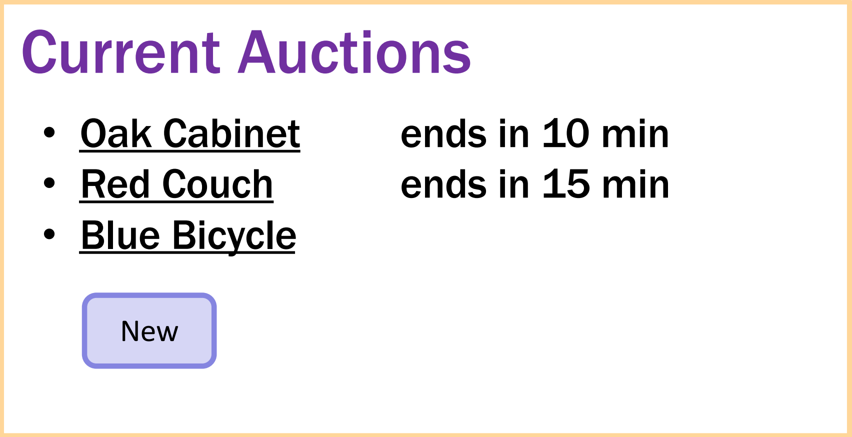
- Assume we know what the app should look like
  - all different interactions are described to us

- Then we can write it in the following order:

  1. Write the client UI with local data
     - no client/server interaction at the start
  2. Write the server
     - official store of the data (client state is ephemeral)
     - only provide the operations needed by the client
  3. Connect the client to the server
     - use fetch to update data on the server before doing same to client

Example: Auction Site

- Initial page shows user a list of auctions
  - can also add their own



Current Auctions

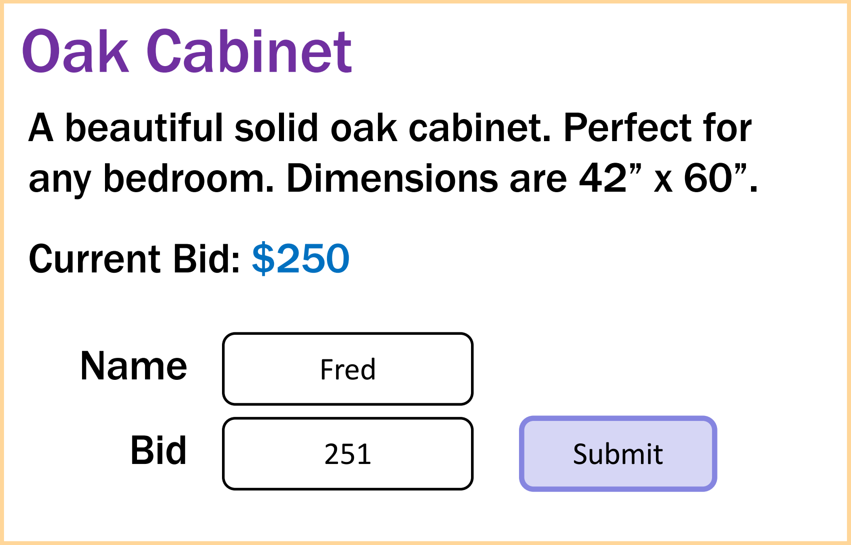
- Oak Cabinet ends in 10 min
- Red Couch ends in 15 min
- Blue Bicycle

can click on item name

can click on New

Example: Auction Site

- Clicking on an item shows the full details
  - allows user to bid



Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

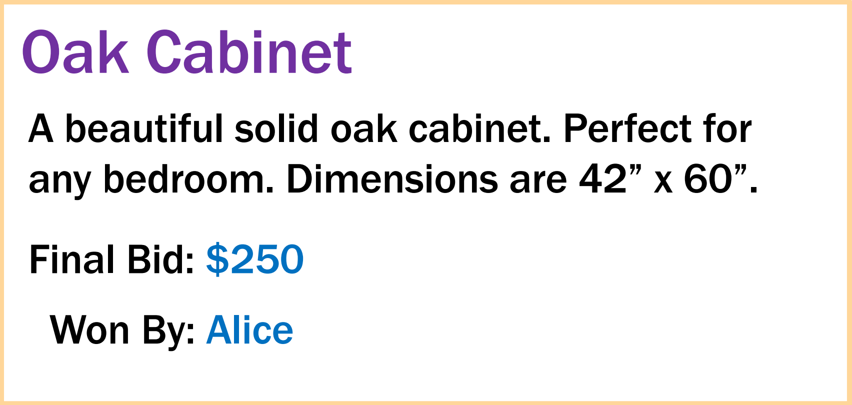
Name	Fred
Bid	251

click Submit to bid

Show an error if the user:
- does not enter a name
- enters a non-number bid
- enters a bid smaller than the current bid

Example: Auction Site

- Clicking on an item shows the full details
  - allows user to bid



Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Final Bid: $250

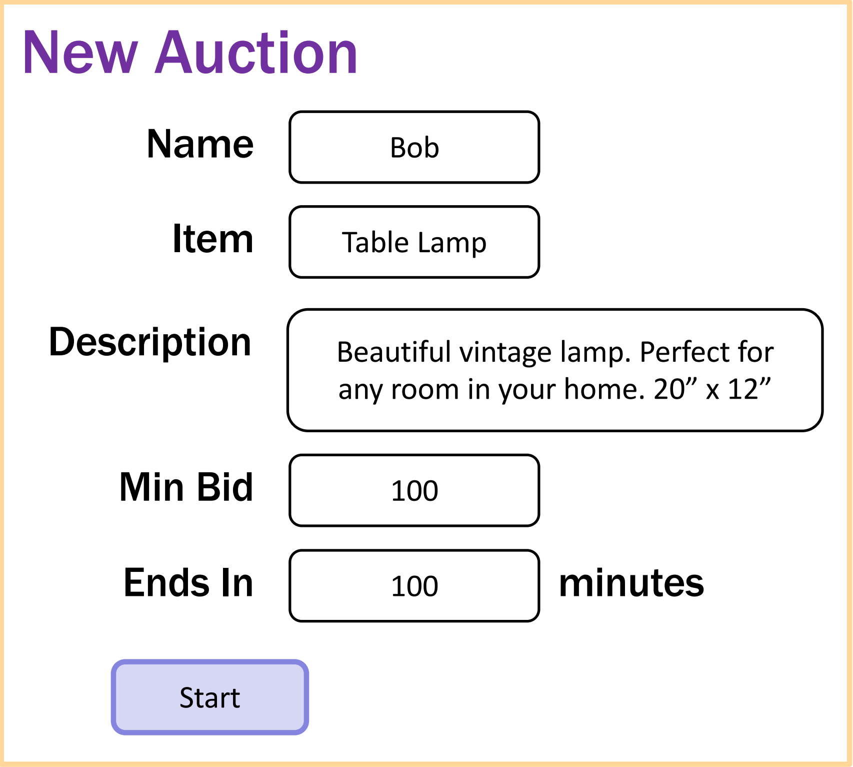
Won By: Alice

Don’t let users bid if the auction is over.

Instead, show who won the auction.

Example: Auction Site

- Clicking on New allows the user to start a new auction
  - user provides the full details of the item to auction



New Auction

Name	Bob
Item	Table Lamp
Description	Beautiful vintage lamp. Perfect for any room in your home. 20” x 12”
Min Bid	100
Ends In	100 minutes

click Start to start auction

Steps to Writing a Full Stack App

- Assume we know what the app should look like
  - all different interactions are described to us

- Then we can write it in the following order:

  1. Write the client UI with local data
     - no client/server interaction at the start
  2. Write the server
     - official store of the data (client state is ephemeral)
     - only provide the operations needed by the client
  3. Connect the client to the server
     - use fetch to update data on the server before doing same to client

Writing the Client

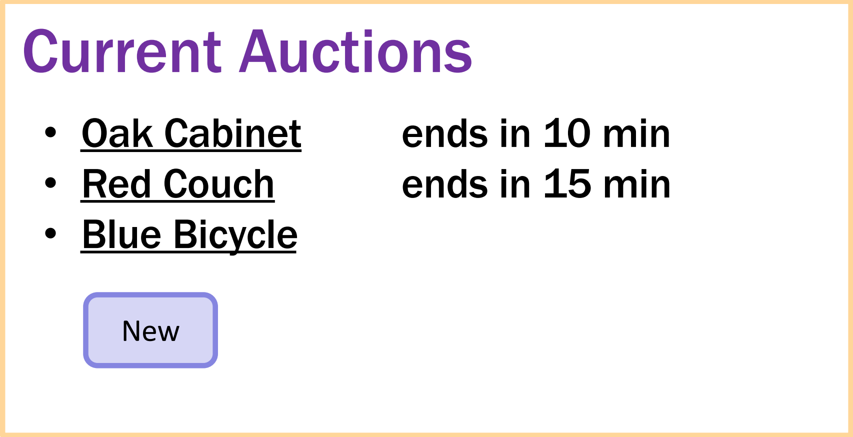
Design on the Client Side

- Component state is **tightly coupled** with UI on screen
  - must store state to render exactly what you see

- Design the client by thinking about what you see
  - what components do you need to show that UI
    - different “pages” should be different components
  - what information do you need to draw each component
    - must be provided in props or stored in state

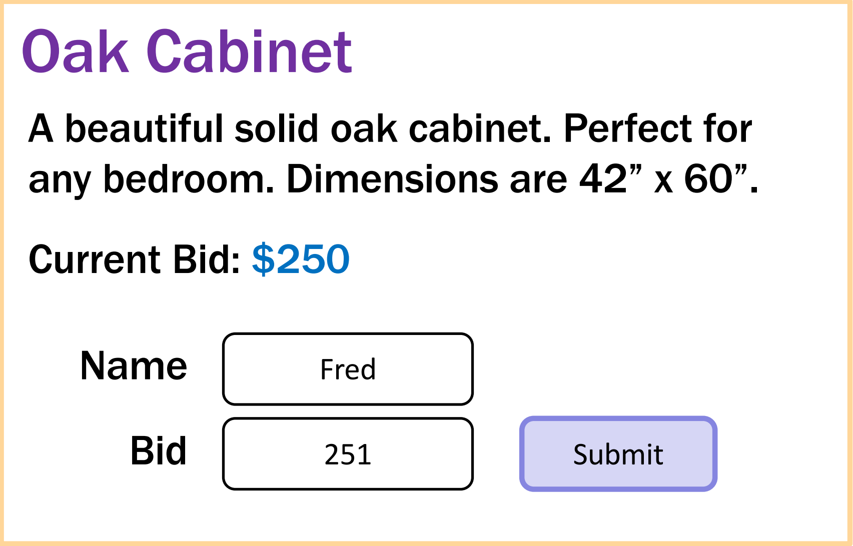
Example: Auction UI

- Auction site has three different "pages"



Current Auctions

Oak Cabinet	ends in 10 min
Red Couch	ends in 15 min
Blue Bicycle	



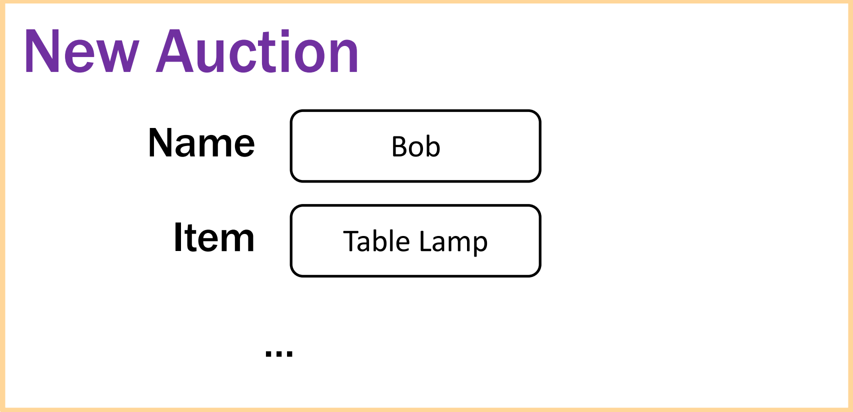
Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42" x 60".

Current Bid: $250

Name: Fred

Bid: 251



New Auction

Name: Bob

Item: Table Lamp

...

Example: Auction UI

- Auction site has three different “pages”
- Need four different components:
  - Auction List: shows all the auctions (and Add button)
  - Auction Details: shows details on the auction (w Bid button)
  - New Auction: lets the user describe a new auction
  - **App**: decides which of these pages to show

Auction Client: `App.tsx`

– state needs to indicate which page to be showing

```
type Page = “list” | “new” |
            {kind: “details”, index: number};

type AppState = {page: Page, auctions: Auction[]};

class App extends Component<{}, AppState> { … }
```

– What is `Page` an example of?

it is an **inductive data type** (of the “enum” variety)

$$\textbf{type}\ \mathrm{Page} := \mathrm{list}\ \mid\ \mathrm{new}\ \mid\ \mathrm{details}(n : \mathbb{N})$$

Auction Client: `App.tsx`

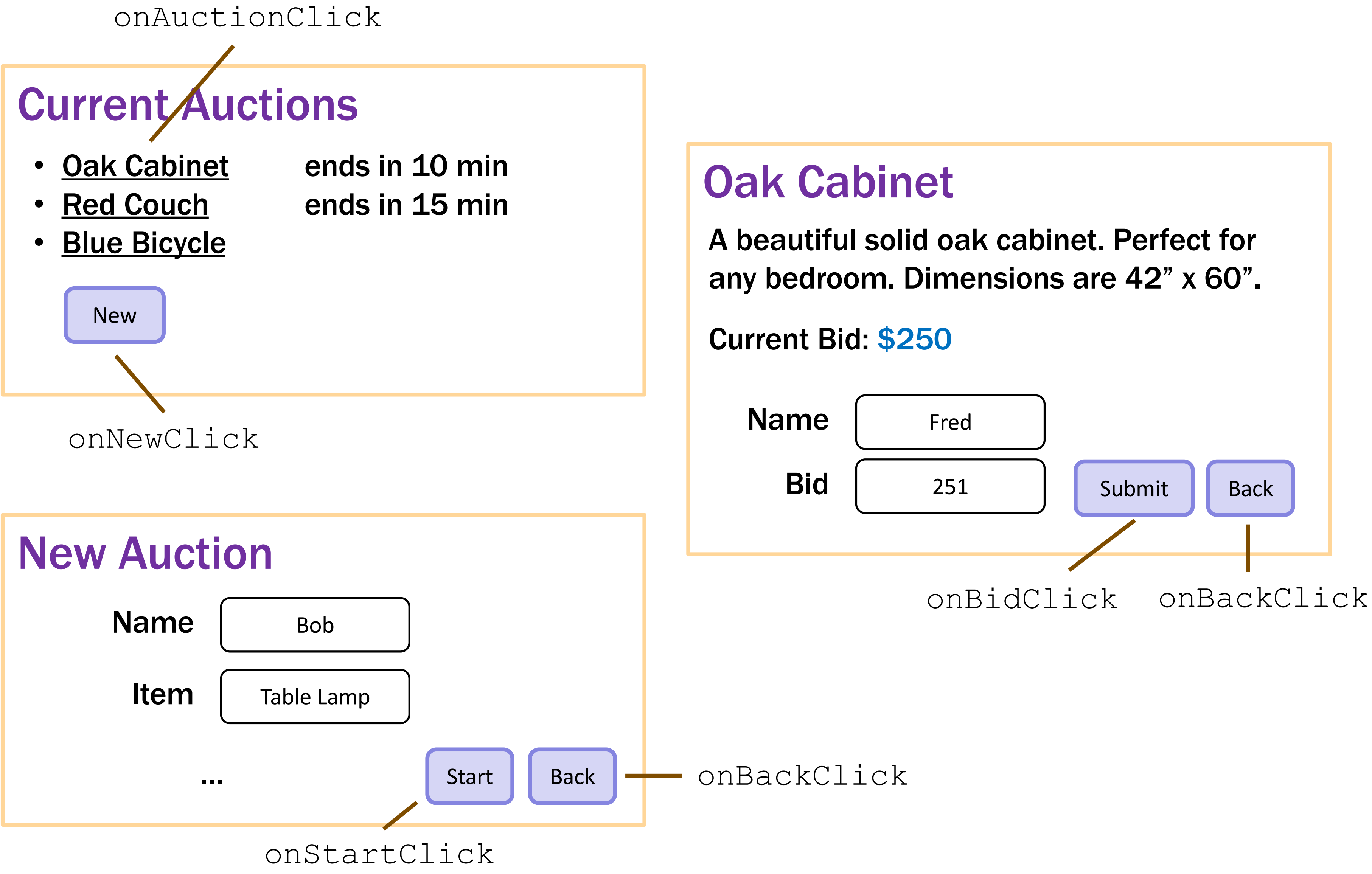
- render shows the appropriate UI

```
render = (): JSX.Element => {
  if (this.state.page === “list”) {
    return <AuctionList auctions={this.state.auctions}
                   onNewClick={this.doNewClick}
                   onAuctionClick={this.doAuctionClick}/>;

  } else if (this.state.page === “new”) {
    return <NewAuction onStartClick={this.doStartClick}
                       onBackClick={this.doBackClick}/>;

  } else {  // kind: “details”
    const auction = this.state.page.auction;
    return <AuctionDetails auction={auction}
                   onBidClick={this.doBidClick}
                    onBackClick={this.doBackClick}/>;
  }
};
```

Example: Auction UI



onAuctionClick

Current Auctions

- Oak Cabinet ends in 10 min
- Red Couch ends in 15 min
- Blue Bicycle

New

onNewClick

Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

Name Fred

Bid 251

Submit Back

onBidClick onBackClick

New Auction

Name Bob

Item Table Lamp

...

Start Back

onBackClick

onStartClick

Auction Client: `App.tsx`

– event handlers change what is shown

```
doNewClick = (): void => {
  this.setState({page: “new”});  // show new auction page
};

doBackClick = (): void => {
  this.setState({page: “list”}); // show auction list page
};

doAuctionClick = (index: number): void => {
  // show details list page for the given auction
  this.setState({page: {kind: “details”, index: index}});
};
```

Auction Client: `App.tsx`

- **the `App` component stores the auction list**

  easy to pass it down to subcomponents in their props

- **subcomponents cannot mutate the auction list!**

  they must invoke **callbacks** to have the `App` update the auction list

```
doStartClick = (name: string, seller: string, …): void => {
  const auction = {name, seller, …};
  const auctions = this.state.auctions.concat([auction]);
  this.setState({page: “list”, auctions: auctions});
};

doBidClick = (index: number, bidder: string, amount: number) => {
  const newVal = …;  // update the auction to have a new high bidder
  const auctions = this.state.auctions.slice(0, index)
        .concat([newVal])
        .concat(this.state.auctions.slice(index+1));
  this.setState({auctions: auctions,
                 page: {kind: ”details”, index: index});
};
```