CSE 331

Full Stack II: Put On Your Thinking App

Kevin Zatloukal

Recall: Steps to Writing a Full Stack App

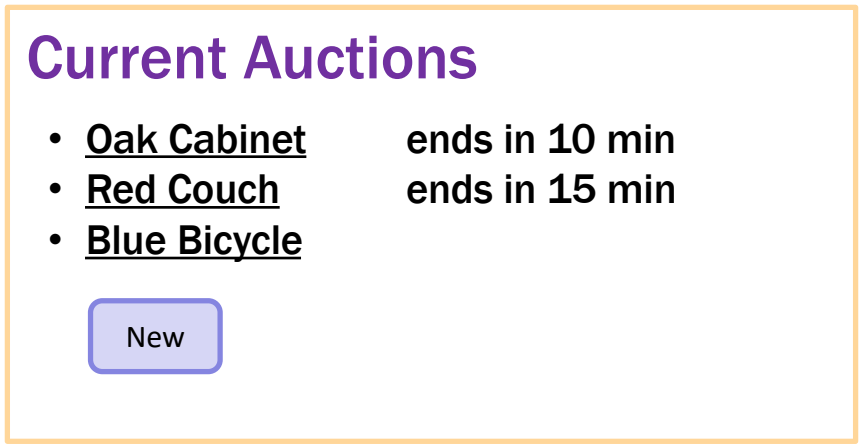
- **Assume we know what the app should look like**
  - **all different interactions are described to us**

- **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

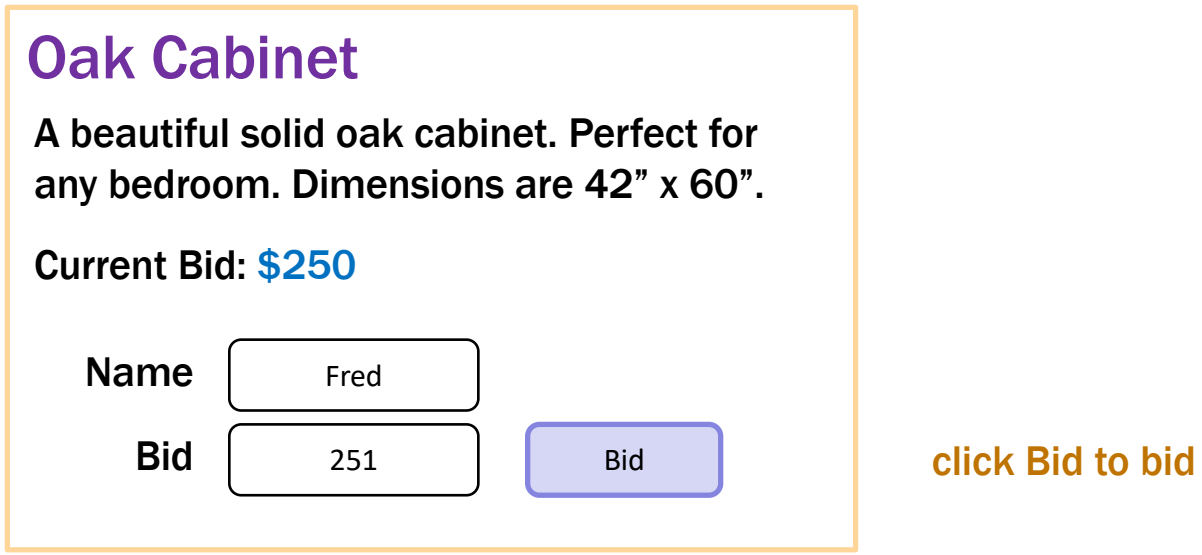


can click on item name

can click on New

Example: Auction Site

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

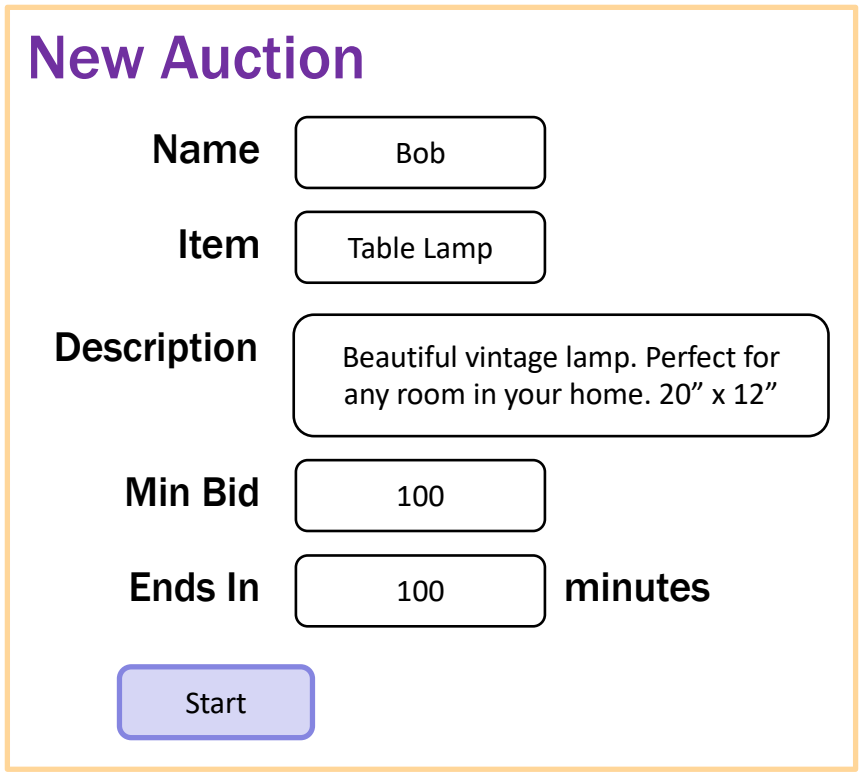
Sold By: Bob

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



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

New

New Auction

Name: Bob

Item: Table Lamp

...

Oak Cabinet

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

Current Bid: $250

Name: Fred

Bid: 251 Bid

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> { … }
```

- storing the list of auctions here

  easiest option, easy to pass it to any page

Auction Client: `App.tsx`

- render shows the appropriate UI

```
render = (): JSX.Element => {
  if (this.state.page === "list") {
    return <AuctionList auctions={this.state.auctions} …/>;

  } else if (this.state.page === "new") {
    return <NewAuction …/>;

  } else {  // kind: "details"
    const index = this.state.page.index;
    const auction = this.state.auctions[index];
    return <AuctionDetails auction={auction} …/>;
  }
};
```

Example: Auction UI

Each page will need **callbacks** to change what page is showing

Current Auctions

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

New

Oak Cabinet

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

Current Bid: $250

Name: Fred

Bid: 251

Bid | Back

New Auction

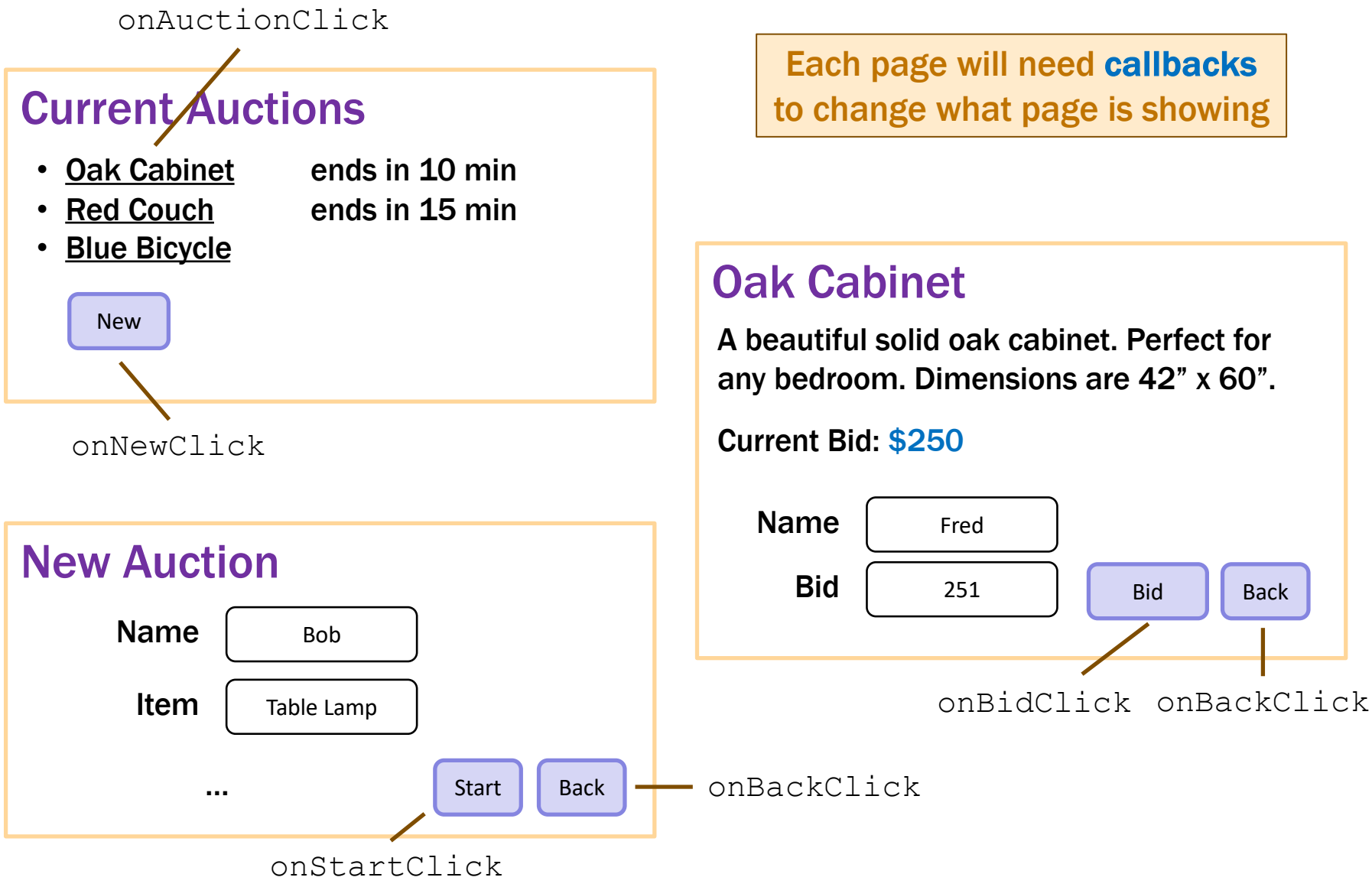
Name: Bob

Item: Table Lamp

...

Start | Back

Example: Auction UI



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 index = this.state.page.index;
    const auction = this.state.auctions[index];
    return <AuctionDetails auction={auction}
                           onBidClick={this.doBidClick}
                           onBackClick={this.doBackClick}/>;
  }
};
```

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, …};  // the new auction
  const auctions = this.state.auctions.concat([auction]);
  this.setState({page: "list", auctions: auctions});
};
```

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 index = this.state.page.index;
    const auction = this.state.auctions[index];
    return <AuctionDetails auction={auction}
              onBidClick={(n, a) => this.doBidClick(index, n, a)}
              onBackClick={this.doBackClick}/>;
  }
};
```

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

```
doBidClick =
     (index: number, bidder: string, amount: number) => {
  const oldVal = this.state.auctions[index];
  const newVal = { …  // oldVal except for:
     maxBid: amount, maxBidder: bidder};
  const auctions = this.state.auctions.slice(0, index)
      .concat([newVal])
      .concat(this.state.auctions.slice(index+1));
  this.setState({auctions: auctions});
};
```

Note: there is subtle issue here we will discuss later...

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

Auction Client: `NewAuction.tsx`

- Figured out the props before. This HTML:

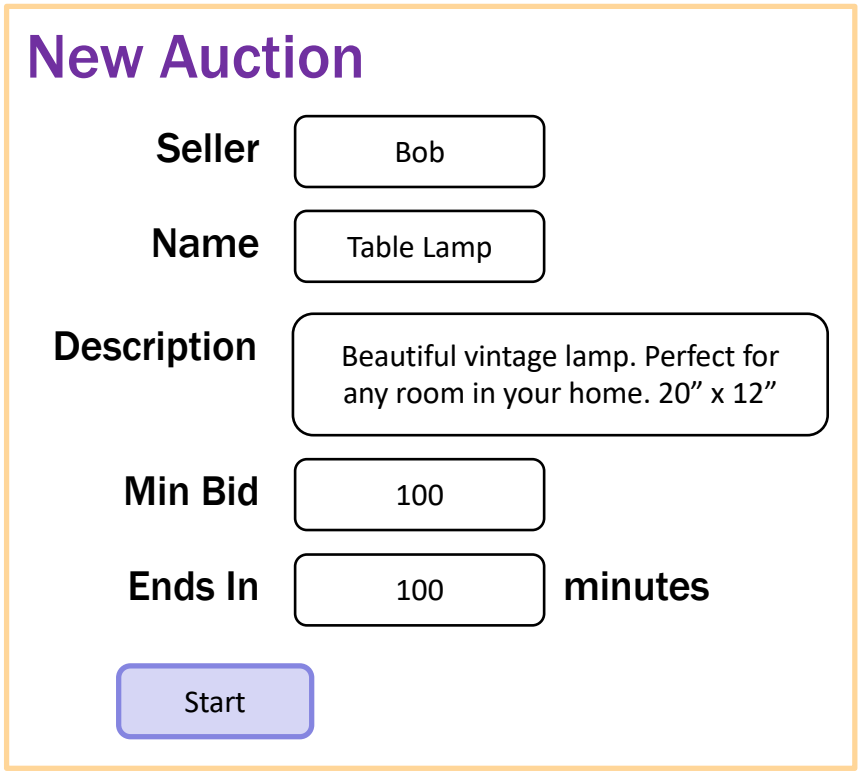
```
return <NewAuction onStartClick={this.doStartClick}
                   onBackClick={this.doBackClick}/>;
```

means these props:

```
type NewAuctionProps = {
  onBackClick: () => void,    // when user clicks "Back"
  onStartClick: (name: string, seller: string, …) => void
};
```

Auction Client: `NewAuction.tsx`

- figured out the props before
- what state should we store?




```
type NewAuctionState = {
  seller: string,
  name: string,
  description: string,
  minBid: string,
  minutes: string
};
```

Note that user input is a string!
(We will need to check validity.)

Auction Client: `NewAuction.tsx`

– **state must mirror input on the screen:**

```
render = (): JSX.Element => {
  …
  <label htmlFor="seller">Seller Name:</label>
  <input id="seller" type="text" value={this.state.seller}
         onChange={this.onSellerChange}/>
  …
}

onSellerChange = (evt: ChangeEvent<HTMLInputElement>) => {
  this.setState({seller: evt.target.value});
};
```

```
type NewAuctionState = {
  seller: string,
  name: string,
  description: string,
  minutes: string,
  minBid: string
};
```

Auction Client: `NewAuction.tsx`

– **state must mirror input on the screen:**

```
render = (): JSX.Element => {
  …
  <label htmlFor="minutes">Minutes:</label>
  <input id="minutes" type="number"
         value={this.state.minutes}
         onChange={this.onMinutesChange}/>
  …
}

onMinutesChange = (evt: ChangeEvent<HTMLInputElement>) => {
  this.setState({minutes: evt.target.value});
};
```

type="number" prevents text that isn't a number but "" is still allowed

```
type NewAuctionState = {
  seller: string,
  name: string,
  description: string,
  minutes: string,
  minBid: string
};
```

Auction Client: `NewAuction.tsx`

- need to validate the input before creating an auction
- show an error message


```
type NewAuctionState = {
  seller: string,
  name: string,
  description: string,
  minutes: string,
  minBid: string,
  error: string
};
```

Auction Client: `NewAuction.tsx`

– state records whether an error is showing

```
render = (): JSX.Element => {
  …
  {this.renderError()}
  …
}

renderError = (): JSX.Element => {
  if (this.state.error === "") {
    return <div></div>;  // show nothing
  } else {
    return <div><b>Error</b>: {this.state.error}</div>;
  }
};
```

Auction Client: `NewAuction.tsx`

- update the state to show an error

```
doStartClick = (): void => {
  if (this.state.seller.trim().length === 0) {
    // re-render with an error message
    this.setState({error: "seller name is missing"});
    return;
  }
  …
}

onSellerChange = (evt: ChangeEvent<HTMLInputElement>) => {
  this.setState({seller: evt.target.value,
                 error: ""});  // remove error message
};
```

Auction Client: `NewAuction.tsx`

- update the state to show an error

```
doStartClick = (): void => {
  // Check that all fields were provided.
  …
  // Check that minutes is a positive integer.
  const minutes = parseFloat(this.state.minutes);
  if (isNaN(minutes) || minutes < 1 ||
      Math.floor(minutes) !== minutes) {
    this.setState(
        {error: "minutes is not a positive integer"});
    return;
  }
  …
};
```

Auction Client: `NewAuction.tsx`

- If all checks pass, we can create the auction

```
doStartClick = (): void => {
  // Check that all fields were provided.
  …
  // Check that minutes & minBid are a positive integers.
  const minutes: number = …;
  …
  // Can now use callback to start the auction...
  this.props.onStartClick(this.state.name, this.state.seller,
      this.state.description, minutes, minBid);
};
```

- What data goes in the auction?

  State of `NewAuction` is for what **it** needs to draw.

  Auction created is for `AuctionDetails` and `AuctionList` to draw.

Auction Client: `NewAuction.tsx`

– Look at other UI to see what data Auction needs


```
type Auction = {
  seller: string,
  name: string,
  description: string,
  endTime: number,        // need to know when auction ends
  maxBid: number,         // need to know current max bid
  maxBidder: string,      // need to know who is winning
};
```

Auction Client: App.tsx

```
doStartClick = (name: string, seller: string, desc: string,
                minutes: number, minBid: number): void => {
  // Ends this many minutes from now (convert to ms)
  const endTime = Date.now() + minutes * 60 * 1000;

  // Seller keeps it if no one bids min or higher
  const maxBid = minBid - 1;
  const maxBidder = this.state.seller;

  const auction = {
     seller: this.state.seller,
     name: this.state.name,
     description: this.state.description,
     endTime, maxBid, maxBidder };

  const auctions = this.state.auctions.concat([auction])
  this.setState({page: "list", auctions: auctions});
};
```

Auction Client: `AuctionDetails.tsx`

- Figured out the props before. This HTML:

```
return <AuctionDetails auction={auction}
                       onBidClick={this.doBidClick}
                       onBackClick={this.doBackClick}/>;
```

means these props:

```
type DetailsProps = {
  auction: Auction,
  // update the highest bid to this
  onBidClick: (bidder: string, amount: number) => void,
  onBackClick: () => void
};
```

- How do we figure out the state?

look at the UI

Auction Client: `AuctionDetails.tsx`

– **Needs to know the current time**

if it is past auction end time, show left; otherwise, show right

```
type DetailsState = {
  now: number,
  bidder: string,
  amount: string,
  error: string
};
```

Oak Cabinet

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

Final Bid: $250

Won By: Alice

Oak Cabinet

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

Current Bid: $250

Name Fred

Bid 251 Bid Back

Auction Client: `AuctionDetails.tsx`

- use the current time to decide how to draw

```
render = (): JSX.Element => {
  const auction = this.props.auction;
  if (auction.endTime <= this.state.now) {
    return this.renderCompleted();
  } else {
    return this.renderOngoing();
  }
};
```

- add a “Refresh” button to update UI to current time

```
// User clicked the Refresh button.
doRefreshClick = (_evt: MouseEvent<HTMLButtonElement>) => {
  this.setState({now: Date.now(), error: ""});
};
```

Recall: 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

```
doBidClick =
     (index: number, bidder: string, amount: number) => {
  const oldVal = this.state.auctions[index];
  const newVal = { …  // oldVal except for:
     maxBid: amount, maxBidder: bidder};
  const auctions = this.state.auctions.slice(0, index)
      .concat([newVal])
      .concat(this.state.auctions.slice(index+1));
  this.setState({auctions: auctions});
};
```

Note: there is subtle issue here we will discuss later...

Recall: 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 index = this.state.page.index;
    const auction = this.state.auctions[index];
    return <AuctionDetails auction={auction}  // newVal replaced oldVal
              onBidClick={(n, a) => this.doBidClick(index, n, a)}
              onBackClick={this.doBackClick}/>;
  }
};
```

Re-rendering `AuctionDetails` with different auction

Lifecycle Events

- **Warning: React doesn't unmount when props change**
  - **instead, it re-renders and calls `componentDidUpdate`**

    just as state can change, props can change
  - **you can detect a props change there**

```
componentDidUpdate = (prevProps: HiProps): void => {
  if (this.props.field !== prevProps.field) {
    …  // our props were changed!
  }
};
```

  - **better to avoid this if possible**

    good setup for **painful** debugging

Auction Client: `AuctionDetails.tsx`

- Often arises when props used to set initial state values
- Here, we initialize bid amount to be valid

```
constructor(props: DetailsProps) {
  super(props);

  const amount = this.props.auction.maxBid + 1;
  this.state = {now: Date.now(),
      bidder: "", amount: '' + amount, error: ""};
}
```

- When auction changes, want to update state to match

  happens each time we call `onBidClick` to update the auction!

  in that case, old bid amount is no longer valid

Auction Client: `AuctionDetails.tsx`

- When auction changes, update state to match:

```
componentDidUpdate = (prevProps: DetailsProps): void => {
  if (prevProps.auction !== this.props.auction) {
    const amount = parseFloat(this.state.amount);
    const minBid = this.props.auction.maxBid + 1;
    if (!isNaN(amount) && amount < minBid) {
      this.setState({amount: '' + minBid});
    }
  }
};
```

- Fixes a stale amount to be a legal value again

  (must be careful changing text the user typed, but this case is okay.)

- (Note: code also updates "`now`" and "`error`" here.)

Auction Client: `AuctionList.tsx`

- Figured out the props before. This HTML:

```
return <AuctionList auctions={this.state.auctions}
                    onNewClick={this.doNewClick}
                    onAuctionClick={this.doAuctionClick}/>;
```

means these props:

```
type ListProps = {
  auctions: ReadonlyArray<Auction>,
  onNewClick: () => void,
  onAuctionClick: (index: number) => void  // clicked on one
};
```

- How do we figure out the state?

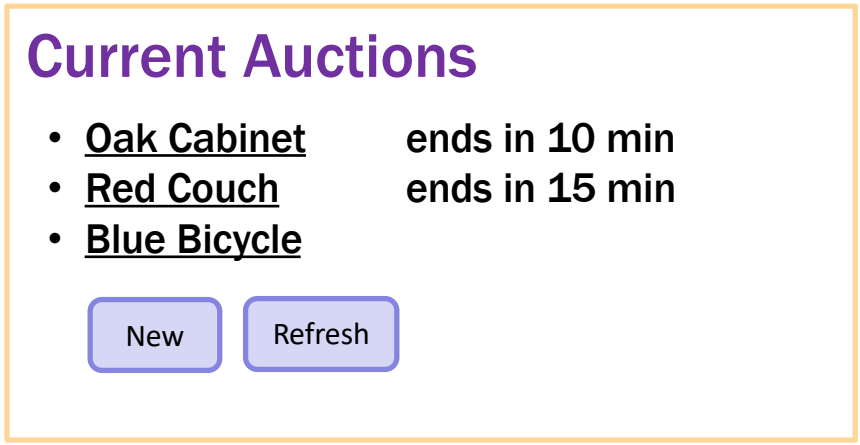
look at the UI

Auction Client: `AuctionList.tsx`

- **Needs to know the current time for text on right**

  if it is past auction end time, show left; otherwise, show right

```
type ListState = {
  now: number
};
```



- **Could replace Refresh with a timer**

  timer calls refresh every 10 seconds, say

- **Nothing else new in `AuctionList.tsx`**