CSE 331

Full Stack 3: To App It All Off

Kevin Zatloukal

Administrivia

- **HW8 due tonight**

  combines HW3-5 (functional UI) with HW7 (servers)

- **Section tomorrow on debugging**
  - **will email instructions on the code setup for class**

- **HW9 released tomorrow night**
  - **9 days to finish it**
  - **start early**

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”

- 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

Design on the Client Side

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

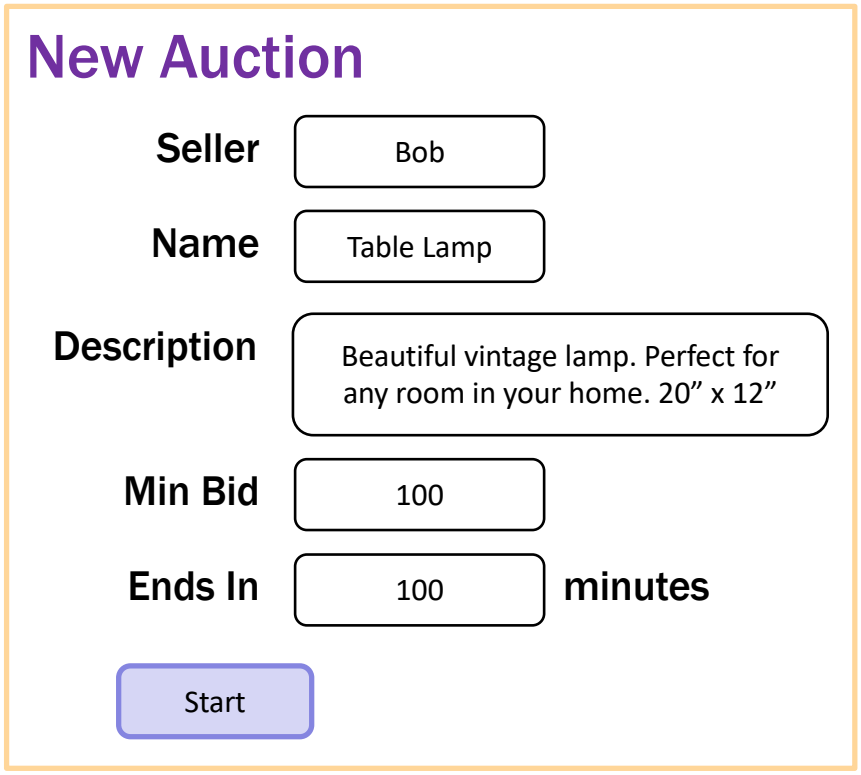
- Design the client by thinking about what you see
  - what components do you need to show that UI

    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
- 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`

- 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`

- 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);
};
```

- This calls `doStartClick` in `App`

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`

– **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: ""});
};
```

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...

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?

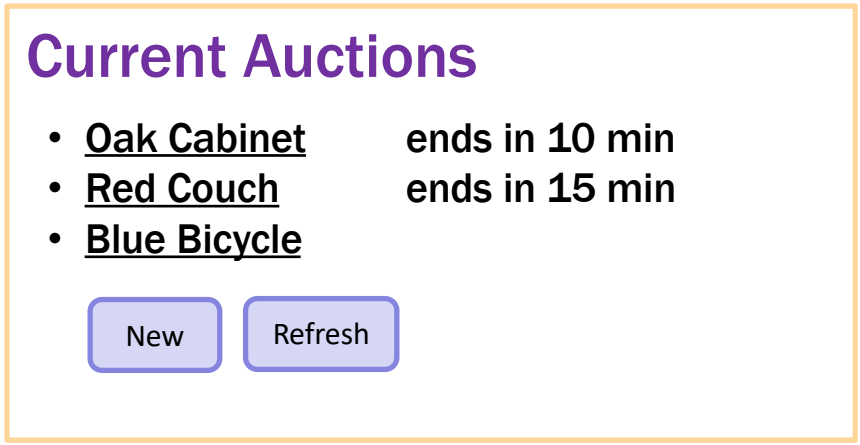
look at the UI

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`**

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 Server

Writing the Server

- First decide what data to store in the server
  - what parts of the UI do we not want to disappear on refresh?

- For the auction app:
  - need to keep the auctions: `Auction[]`
  - don't need to keep other parts

    which page we are on

    text in any of the text boxes

Writing the Server

- Next decide what read operations we need
  - these will become GET requests
- Simplest case is when the client can store all data
  - just let the client retrieve all of it
  - with lots of data, client would need to query a subset
- For the auctions app:
  - `/api/list` returns all the auctions

Auction Server: routes.ts

```
// List of all auctions, in order by creation time (only pushed)
const auctions: Auction[] = [];

/**
 * Returns a list of all the auctions, sorted so that the
 * ongoing auctions come first and the completed ones after. …
 */
export const listAuctions =
    (_req: SafeRequest, res: SafeResponse): void => {
  res.send({auctions: auctions});
};
```

Writing the Server

- Next decide what update operations we need
  - these will become POST requests
  - what updates do we make to that data in the client?

- For the auctions app:
  - look in `App.tsx` to see how we change auctions

    no other component is allowed to modify the auctions array
  - we change it in two ways:
    1. add a new auction
    2. change an auction to have a new highest bidder

Writing the Server

- Next decide what update operations we need
  - these will become POST requests
  - what updates do we make to that data in the client?

- For the auctions app:
  - `/api/add` adds an auction
  - `/api/bid` updates to a new, higher bid

    better to have a more specific update vs general “change” operation

    can do more error checking with more specific updates

Auction Server: routes.ts

```
export const addAuction =
    (req: SafeRequest, res: SafeResponse): void => {

  const name = req.body.name;
  if (typeof name !== 'string') {
    res.status(400).send("missing 'name' parameter");
    return;
  }
  // check the others (including minutes & minBid are valid ints)
  …
  const endTime = Date.now() + minutes * 60 * 1000;  // in ms
  const auction: Auction = { id: auctions.length,
    name: name, description: description, seller: seller,
    endTime: endTime, maxBid: minBid - 1, maxBidder: seller };
  auctions.push(auction);          // add this to the list
  res.send({auction: auction});  // send this to the client
};
```

Testing the Server

- Write unit tests for each route
  - test creates fake request and response objects
  - some tests may need to apply multiple operations

    need to perform a few `/api/add` and then `/api/list`

- Test the server thoroughly before continuing
  - debugging later will be painful, so make sure it's right!

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

Connecting Client & Server

Recall: Finishing Step 3 for To-Do List

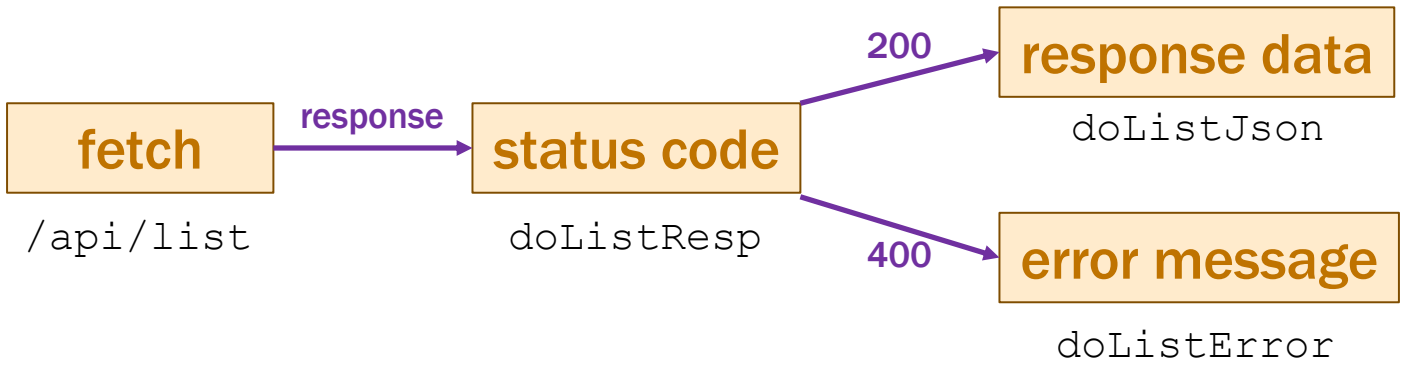
- Rewrote client-side To-Do App into client-server

- 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

Recall: 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



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

```
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`

- change start to make a request to the server

```
doStartClick = (name: string, seller: string, …): void => {
  const args = {name, seller, description, minutes, minBid);
  fetch("/api/add", {
        method: "POST", body: JSON.stringify(args),
        headers: {"Content-Type": "application/json"} })
   .then(this.doAddResp)
   .catch(() => this.doAddError("failed to connect to server"));
};
```

Auction Client: `App.tsx`

- change start to make a request to the server

```
doAddResp = (resp: Response): void => {
  if (resp.status === 200) {
    resp.json().then(this.doAddJson)
        .catch(() => this.doAddError("not JSON"));
  } else if (resp.status === 400) {
    resp.text().then(this.doAddError)
        .catch(() => this.doAddError("not text"));
  } else {
    this.doAddError(`bad status code: ${resp.status}`);
  }
};

doAddError = (msg: string): void => {
  console.error(`Error fetching /api/add: ${msg}`);
};
```

Auction Client: `App.tsx`

- change start to make a request to the server

```
doAddJson = (data: unknown): void => {
  if (!isRecord(data)) {
    console.error("bad data from /api/add: not a record", data);
    return;
  }

  const auction = parseAuction(data.auction);
  if (auction !== undefined) {
    const auctions = this.state.auctions.concat([auction]);
    this.setState({page: "list", auctions: auctions});
  } else {
    console.error("not an auction", data.auction);
  }
};
```

One More Feature

- Another user can bid on the item we are viewing
  - no way to find out about it without talking to the server
  - need a way to update the page without bidding
- Simple option: add a “Refresh” button
  - requires `/api/get` on the server also
  - “get” same as “bid” but we don’t change the auction
- Same fix to `componentDidUpdate` needed here
  - the App is redrawing with different props
  - need to update `this.state.now`
  - **NOTE**: same now applies to `AuctionList`!

Improving the App

Summary

- Client / Server version more complicated
  - extra invariant: client & server copies of `Auction[]` match
  - will be a bug if these ever get out of sync!
- Positives of this approach
  - fairly mechanical way to turn client-only into client-server
  - works well for single-user apps
- Negatives of this approach
  - some logic is duplicated in client & server
  - indexes are brittle

    could not sort the auctions list without potentially breaking the client

Alternative Approach

- Do more work only in the server
  - eliminate duplicate work
  - eliminate the extra invariant (no copy in client)

- Will need to make more server requests
  - client no longer has all the data
  - every component will talk to the server

- Server is free to:
  - use more complex data structures

    we will switch to a `Map` (as in HW7-8)
  - implement new algorithms

    we will sort the auctions into completed and not completed

Writing the Client

Auction Client: `App.tsx`

- state needs to indicate which page to be showing

```
type Page = "list" | "new" |
            {kind: "details", name: string};

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

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

- identify an auction by the item name
- no longer storing the list of auctions here

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"
    return <AuctionDetails name={this.state.page.name}
                           onBidClick={this.doBidClick}
                           onBackClick={this.doBackClick}/>;
  }
};
```

- the `App` gets much simpler! (only 3 event handlers, no requests)

Auction Client: `AuctionList.tsx`

- **List of Auctions now in state, not props**

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

type ListState = {
  now: number,
  auctions: Auction[] | undefined
};
```

- **Fetch the list in `componentDidMount`**

  code moves from `App.tsx` to `AuctionList.tsx`

- **No longer matters what order the list returned is in**

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 = …;
  …
  // Ask the server to add this auction…
  this.props.onStartClick(this.state.name, this.state.seller,
      this.state.description, minutes, minBid);
  fetch("/api/add", {..})
    .then(this.doAddResp)
    .catch(() => this.doAddError("failed to connect"));
};
```

- Code moves from `App.tsx` to `AuctionList.tsx`

Auction Client: `NewAuction.tsx`

- **Navigate to `AuctionList` once this completes:**

```
doAddJson = (data: unknown): void => {
   if (!isRecord(data)) {
     console.error("bad data: not a record", data);
     return;
   }

   this.props.onBackClick();  // show the updated list
 };

 doAddError = (msg: string): void => {
   this.setState({error: msg})
 };
```

- **Request can fail due to duplicate auction name**

  show this error message to the user, so they can fix it

Auction Client: `AuctionDetails.tsx`

- **The Auction is now in state, not props**

```
type DetailsProps = {
  name: string,
  onBidClick: (bidder: string, amount: number) => void,
  onBackClick: () => void
};

type DetailsState = {
  now: number,
  auction: Auction | undefined,
  bidder: string,
  amount: string,
  error: string
};
```

- **Fetch the list in `componentDidMount`**

  code moves from `App.tsx` to `AuctionDetails.tsx`

Auction Client: `AuctionDetails.tsx`

- The Auction is now in state, not props

```
type DetailsProps = {
  name: string,
  onBidClick: (bidder: string, amount: number) => void,
  onBackClick: () => void
};

type DetailsState = {
  now: number,
  auction: Auction | undefined,
  bidder: string,
  amount: string,
  error: string
};
```

- Note: no longer need `componentDidUpdate`!!

Auction Client: `AuctionDetails.tsx`

- Handle the bidding within this component

```
doBidClick = (): void => {
  // Check that bidder was provided.
  // Check that amount is a valid bid.
  const amount: number = …;
  …
  // Ask the server to update the bid for this auction…
  this.props.onBidClick(this.state.bidder, amount);
  fetch("/api/bid", {..})
    .then(this.doBidResp)
    .catch(() => this.doBidError("failed to connect"));
};
```

- Code moves from `App.tsx` to `AuctionList.tsx`

Auction Client: `AuctionDetails.tsx`

- **Navigate to `AuctionList` once this completes:**

```
doBidJson = (data: unknown): void => {
   if (!isRecord(data)) {
     console.error("bad data: not a record", data);
     return;
   }

   // Update state to show auction in data.auction
   …
};

doBidError = (msg: string): void => {
   this.setState({error: msg})
};
```

- **Request can fail if someone else outbid since refresh**

  this approach works better with multiple users

Writing the Server

Auction Server: routes.ts

```
// Map from name to auction information
const auctions: Map<string, Auction> = new Map();

// Put ongoing auctions before completed ones, and
// those about to complete before those completing later.
const compareAuctions = (a: Auction, b: Auction): number => …

/**
 * Returns a list of all the auctions, sorted so that the
 * ongoing auctions come first and the completed ones after. …
 */
export const listAuctions =
    (_req: SafeRequest, res: SafeResponse): void => {
  const vals = Array.from(auctions.values());
  vals.sort(compareAuctions);
  res.send({auctions: vals});
};
```

Auction Server: routes.ts

```
export const getAuction =
    (req: SafeRequest, res: SafeResponse): void => {

  const name = req.body.name;
  if (typeof name !== "string") {
    res.status(400).send("missing or invalid 'name' parameter");
    return;
  }

  const auction = auctions.get(name);
  if (auction === undefined) {
    res.status(400).send(`no auction with name '${name}'`);
    return;
  }

  res.send({auction: auction});  // send current auction state
};
```

What's Still Missing?

- See everything we need for proof-of-concept apps
  - can test these with real users
- For non-demo, can't store user data on one machine
  - machines break, hard drives fail, etc.
- Sharing state between servers is complex
  - requires even more sophisticated **invariants**
  - see 452 for more on this

What's Still Missing?

- Most apps use dedicate storage servers
  - see 344 for sophisticated storage services
- Especially easy to do this with Map
  - many options for extremely scalable Map services
  - easy to swap out an in-memory Map for a service
- Our server becomes a client ("front-end server")
  - read/write from the map service is like a fetch

    event handlers in the server now
  - server can now be functional!

    easier to get everything right