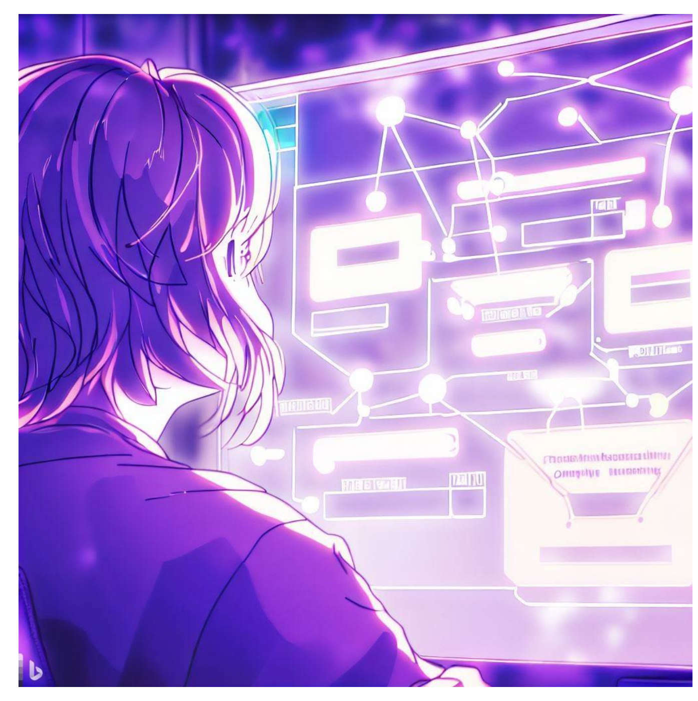


CSE 331

Full Stack Apps

James Wilcox & Kevin Zatloukal

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)
3. Connect the client to the server
   - use fetch to update data on the server before doing same to client

Design on the Client Side

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

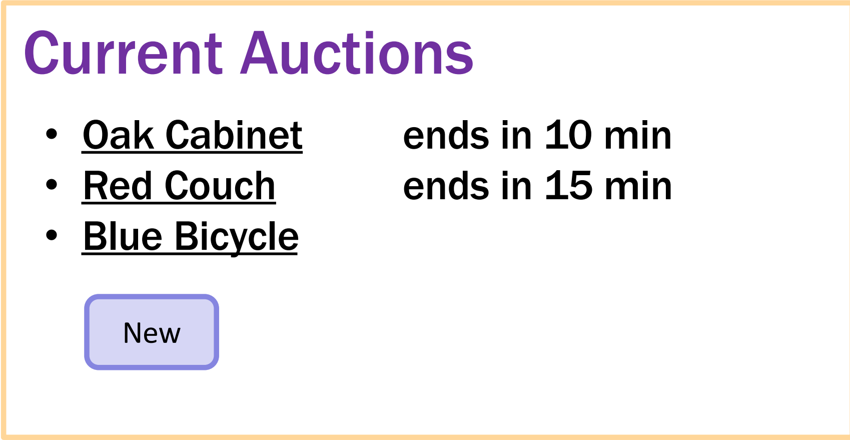
- Design the client by thinking about what you see
  - what components do you need to show that UI

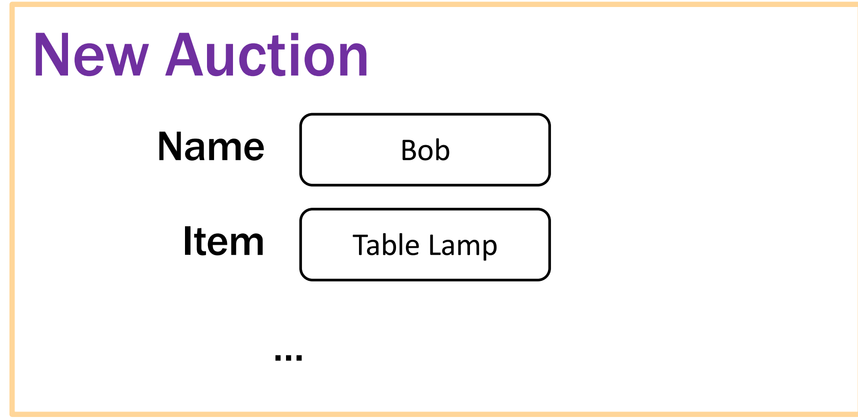
    different "pages" should be different components
  - what information do you need to draw each component

    must be provided in props or stored in state

Last time: Auction pages

- Auction site has three different “pages”





Oak Cabinet

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

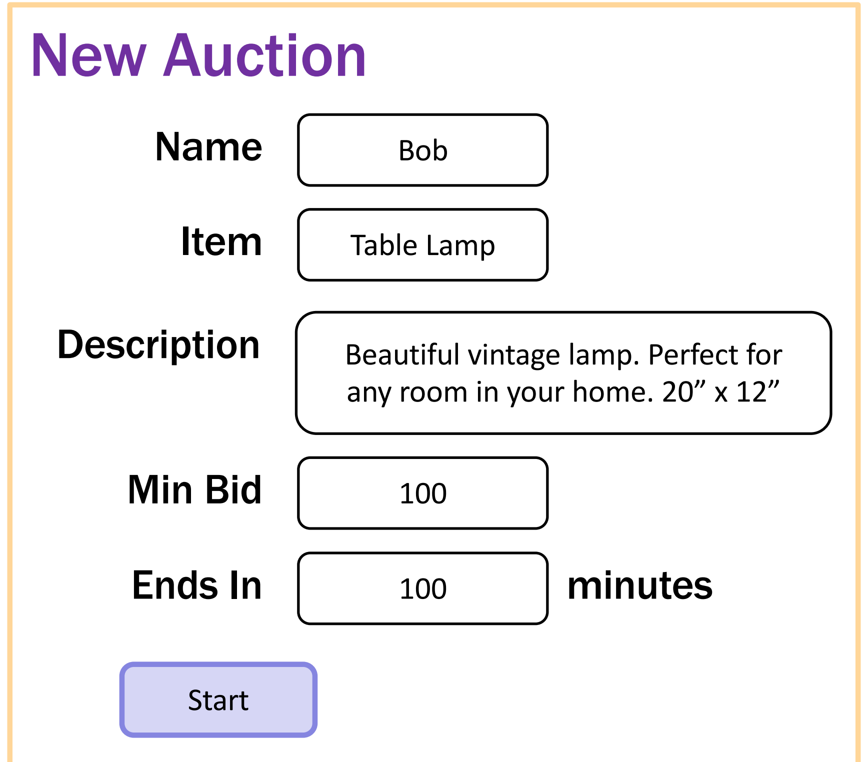
Current Bid: $250

Name [Fred]

Bid [251] [Submit]

Example: Auction Site

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



click Start to start auction

Auction Client: `NewAuction.tsx`

– Figured out the props before. This HTML:

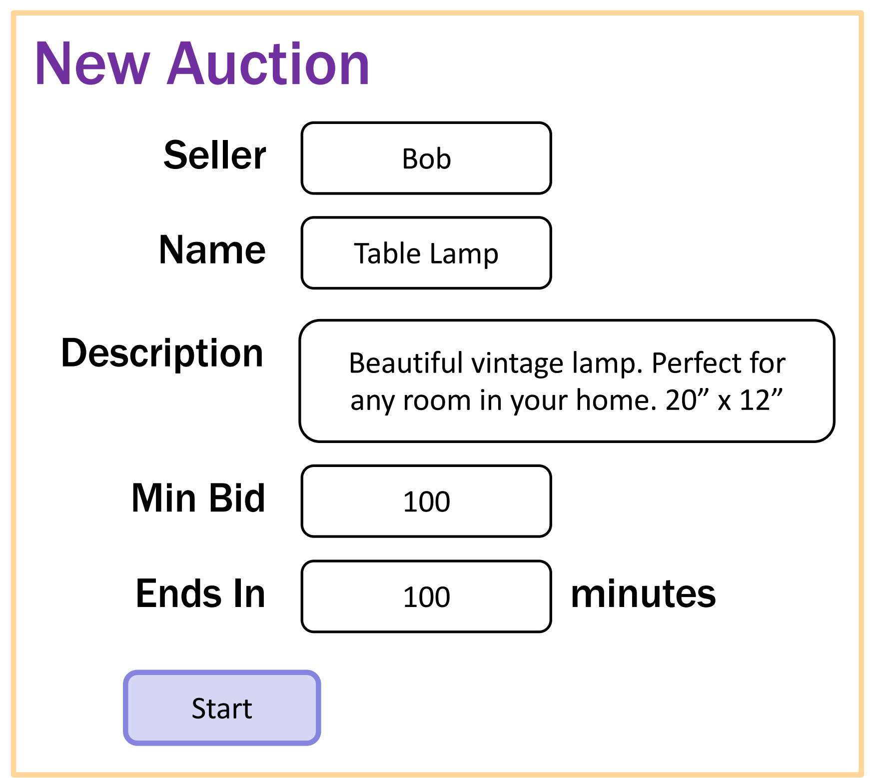
```
return <NewAuction onStartClick={this.doStartClick}
                   onBackClick={this.doBackClick}/>;
```

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:</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:

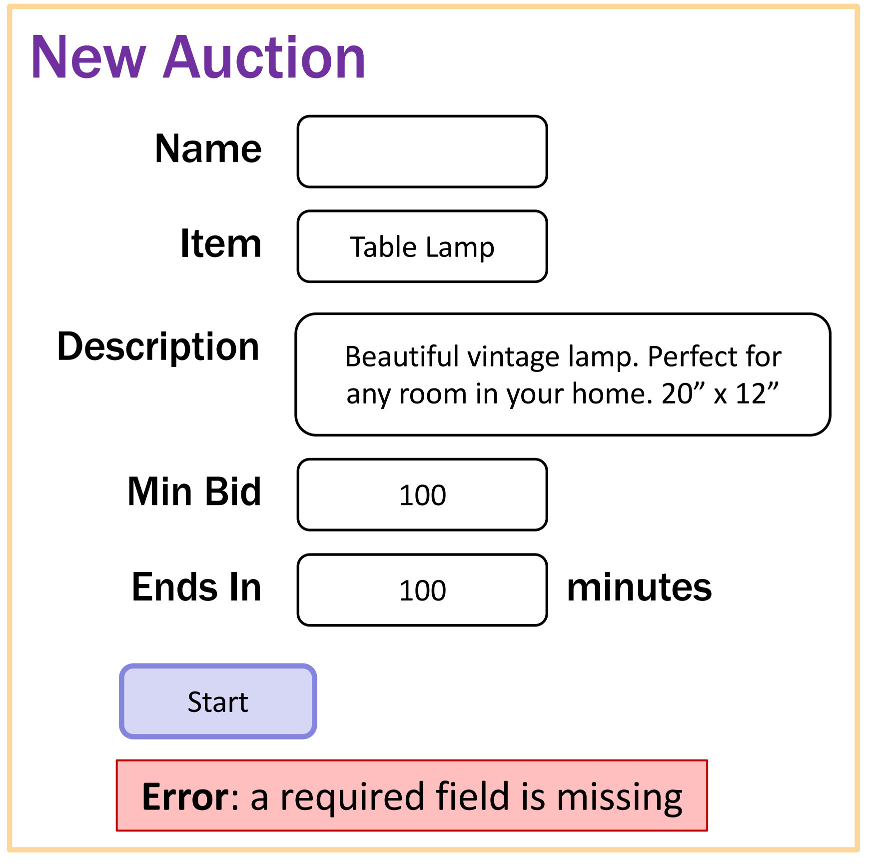
```
render = (): JSX.Element => {
  …
  <label htmlFor="minutes">Minutes:</label>
  <input id="minutes" type="number"
         value={this.state.minutes}
         onChange={this.onMinutesChange}/>
  …
}

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

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

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

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

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”.

Final Bid: $250

Won By: Alice

Sold By: Bob

```
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});
};
```

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

Submit

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.

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

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

New

can click on item name

can click on New

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
};
```

Current Auctions

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

New | Refresh

- Could replace Refresh with a timer

  timer calls refresh every 10 seconds, say

- Nothing else new in `AuctionList.tsx`

Moving Data to the Server

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)

  3. **Connect the client to the server**
     - use fetch to update data on the server before doing same to client

Designing the Server

- **Decide what state you want to be permanent**
  - **e.g., items on the To-Do list**
- **Decide what operations the client needs**
  - **e.g., retrieve the list, add to the list, remove from list**

    look at the **client code** to see how the list changes

    each way of changing the list becomes an **operation**
  - **only provide those operations**

    can always add more operations later

Writing the Server

- Full-stack apps introduce new ways of failing
  - can fail in the client due to a bug in the server
  - can fail in the server due to a bug in the client

- Debugging a full-stack app is much harder
  - requires understanding client, server, & interactions
  - will take more time…

- Make sure you unit test the server thoroughly
  - if client & server are correct, failures are due to interaction

    e.g., misunderstanding about what to data to send or where to send it
  - limits the scope of the search

Example: To-Do List <u>Server</u>

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:

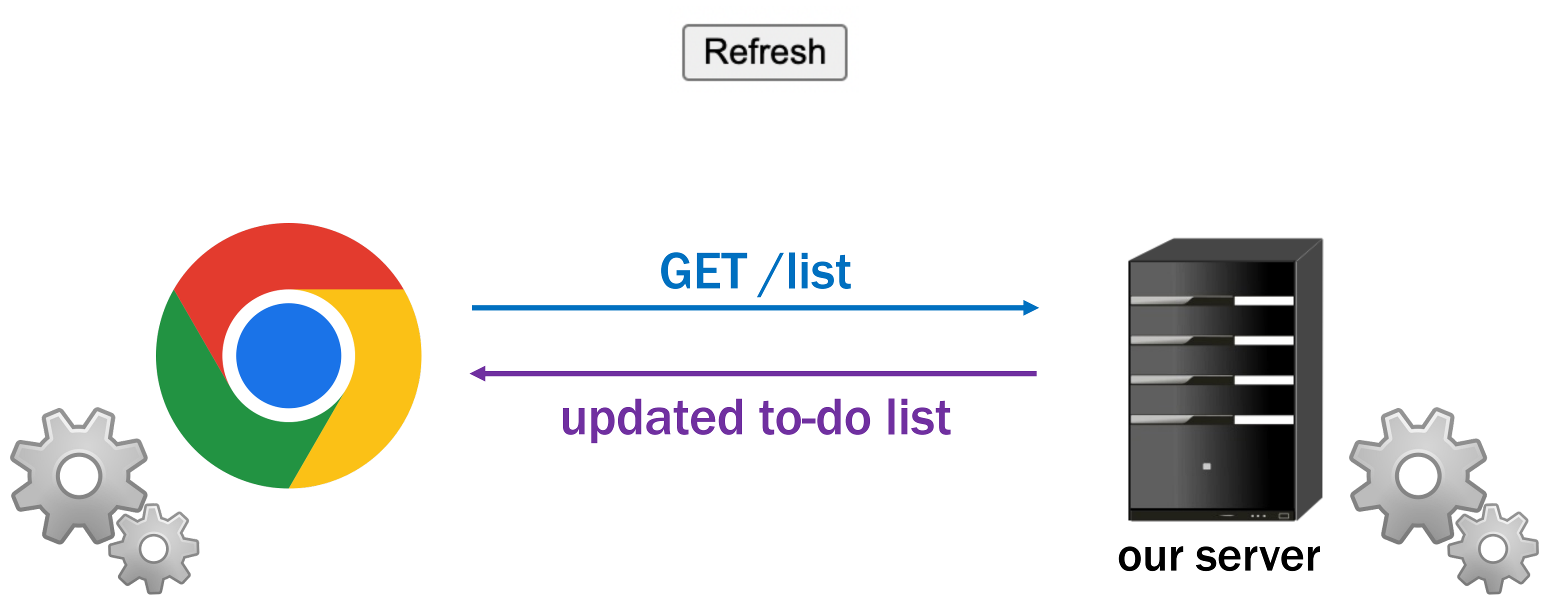
  1. Write the client UI with local data
     - no client/server interaction at the start

  2. Write the server
     - official store of the data (client state is ephemeral)

  3. Connect the client to the server
     - use fetch to update data on the server before doing same to client

Recall: Client-Server Interaction

- Clients need to talk to server & update UI in response



Components give us the **ability** to update the UI when we get new data from the server (an event)

How do we get new data from the server?

Making HTTP Requests

- **Send & receive data from the server with “`fetch`”**

```
fetch(“/list”)
  .then(this.doListResp)
  .catch(() => this.doListError(“failed to connect”))
```

  - **then handler is called if the request can be made**
  - **catch handler is called if it cannot be**

    only if it could not connect to the server at all

    status 400 still calls then handler
  - **catch is also called if then handler throws an exception**

Making HTTP Requests

- Send & receive data from the server with "`fetch`"

```
fetch("/list")
  .then(this.doListResp)
  .catch(() => this.doListError("failed to connect"))
```

- Fetch returns a "promise" object
  - has `.then` & `.catch` methods
  - both methods return the object again
  - above is equivalent to:

```
const p = fetch("/list");
p.then(this.doListResp);
p.catch(() => this.doListError("failed to connect"));
```

Making HTTP Requests

- **Send & receive data from the server with "`fetch`"**

```
const url = "/list?" +
    "category=" + encodeURIComponent(category);
fetch(url)
  .then(this.doListResp)
  .catch(() => this.doListError("failed to connect"))
```

- **All query parameter values are strings**
- **Some characters are not allowed in URLs**
  - **the `encodeURIComponent` function converts to legal chars**
  - **server will automatically decode these (in `req.query`)**

    in example above, `req.query.name` will be "laundry"

Making HTTP Requests

- **Still need to check for a 200 status code**

```
doListResp = (res: Response): void => {
  if (res.status === 200) {
    console.log("it worked!");
  } else {
    this.doListError(`bad status ${res.status}`);
  }
};

doListError = (msg: string) => {
  console.log("fetch of /list failed: ${msg}");
};
```

  - **(often need to tell users about errors with some UI…)**

Handling HTTP Responses

- Response has methods to *ask for* response data
  - our `doListResp` called once browser has **status code**
  - may be a while before it has all response data (could be GBs)

- With our conventions, status code indicates data type:
  - with 200 status code, use `res.json()` to get record

    we always send records for normal responses
  - with 400 status code, use `res.text()` to get error message

    we always send strings for error responses

- These methods return a **promise** of response data
  - use `.then(..)` to add a handler that is called with the data
  - handler `.catch(..)` called if it fails to parse

Making HTTP Requests

```
doListResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doListJson);
        .catch(() => this.doListError("not JSON");
  } …
  …
};
```

- **Second promise can also fail**
  - e.g., fails to parse as valid JSON, fails to download
- Important to **<u>catch every error</u>**
  - **painful** debugging if an error occurs and you don't see it!

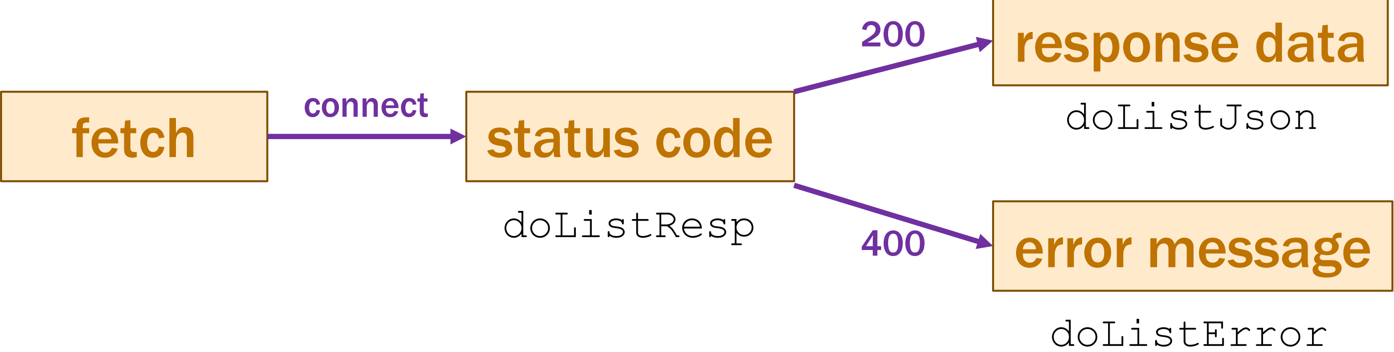
Making HTTP Requests

```
doListResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doListJson);
        .catch(() => this.doListError("not JSON");
  } else if (res.status === 400) {
    res.text().then(this.doListError);
        .catch(() => this.doListError("not text");
  } else {
    this.doListError(`bad status: ${res.status}`);
  }
};
```

- **We know 400 response comes with an error message**
  - **could also be large, so** `res.text()` **also returns a promise**

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



Fetch Requests Are Complicated

- **Four** different methods involved in each fetch:
  1. method that makes the fetch
  2. handler for fetch Response — e.g., `doListResp`
  3. handler for fetched JSON — e.g., `doListJson`
  4. handler for errors — e.g., `doListError`

- Three different events involved:
  - getting status code, parsing JSON, parsing text
  - any of those can fail!

    important to make all error cases **visible**

Recall: HTTP GET vs POST

- When you type in a URL, browser makes "GET" request
  - request to read something from the server
- Clients often want to write to the server also
  - this is typically done with a "POST" request
    - ensure writes don't happen just by normal browsing
- POST requests also send data to the server in body
  - GET only sends data via query parameters
  - limited to a few kilobytes of data
  - POST requests can send arbitrary amounts of data

Making HTTP POST Requests

- Extra parameter to fetch for additional options:

```
fetch("/add", {method: "POST"})
```

- Arguments then passed in body as JSON

```
const args = {name: "laundry"};
fetch("/add", {method: "POST",
    body: JSON.stringify(args),
    headers: {"Content-Type": "application/json"}})
  .then(this.doAddResp)
  .catch(() => this.doAddError("failed to connect"))
```

  – add as many fields as you want in `args`
  – Content-Type tells the server we sent data in JSON format

Example: To-Do List 2.0

Recall: (Old) 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
};
```

New 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;

  // Ask the server to add the new item.
  const args = {name: name};
  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"));
};
```

New TodoApp – Add Response & Error

```
// Called when the server confirms that the item was added.
doAddResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doAddJson)
      .catch(() => this.doAddError(”200 response is not JSON"));
  } else if (res.status === 400) {
    res.text().then(this.doAddError)
      .catch(() => this.doAddError(”400 response is not text"));
  } else {
    this.doAddError(`bad status code ${res.status}`);
  }
};

// Called when we fail trying to add an item
doAddError = (msg: string): void => {
  console.error(`Error fetching /add: ${msg}`);
};
```

New TodoApp – Add Json

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

  if (typeof data.name !== 'string') {
    console.error("bad data from /add: name missing / wrong", data);
    return;
  }

  // Now that we know it was added, we can update the UI.
  const items = this.state.items.concat(
      [ {name: data.name, completed: false} ]);
  this.setState({items: items, newName: ""});  // clear input box
};
```

Recall: (Old) TodoApp – Item Clicked

```
// Called when the user checks the box next to an uncompleted item.
// The second parameter is the index of that item in the list.
doItemClick =
    (_: ChangeEvent<HTMLInputElement>, index: number): void => {
  const item = this.state.items[index];

  // Note: we cannot mutate the list. We must create a new one.
  const items = this.state.items.slice(0, index)     // 0 .. index-1
      .concat([{name: item.name, completed: true}])
      .concat(this.state.items.slice(index + 1));   // index+1 ..
  this.setState({items: items});

  // Remove the item in 5 seconds…
  setTimeout(() => this.doItemTimeout(index), 5000);
};
```

New TodoApp – Item Clicked

```
// Called when the user checks the box next to an uncompleted item.
// The second parameter is the index of that item in the list.
doItemClick =
    (_: ChangeEvent<HTMLInputElement>, index: number): void => {
  const item = this.state.items[index];

  const args = {name: item.name};
  fetch("/api/complete", {
      method: "POST", body: JSON.stringify(args),
      headers: {"Content-Type": "application/json"} })
    .then((res) => this.doCompleteResp(res, index))
    .catch(() => this.doCompleteError("failed to connect"))
};
```

- passing `index` as an extra argument
- we'll need it later…

New TodoApp – Item Clicked

```
// Called when the server confirms that the item was completed.
doCompleteResp = (res: Response, index: number): void => {
  if (res.status === 200) {
    res.json().then((data) => this.doCompleteJson(data, index))
      .catch(() => this.doCompleteError(”200 response is not JSON"));
  } else if (res.status === 400) {
    res.text().then(this.doCompleteError)
      .catch(() => this.doCompleteError(”400 response is not text"));
  } else {
    this.doCompleteError(`bad status code ${res.status}`);
  }
};
```

– passing `index` as an extra argument

New TodoApp – Item Clicked

```
// Called with the JSON response from /api/complete
doCompleteJson = (data: unknown, index: number): void => {
  if (!isRecord(data)) {
    console.error("bad data from /complete: not a record", data)
    return;
  }
  // Nothing useful in the response itself…

  // Note: we cannot mutate the list. We must create a new one.
  const item = this.state.items[index];
  const items = this.state.items.slice(0, index)     // 0 .. index-1
      .concat([{name: item.name, completed: true}])
      .concat(this.state.items.slice(index + 1));   // index+1 ..
  this.setState({items: items});

  // Refresh our list after this item has been removed.
  setTimeout(this.doRefreshTimeout, 5100);
};
```

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}`);
  }
};
```

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

- [x] 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

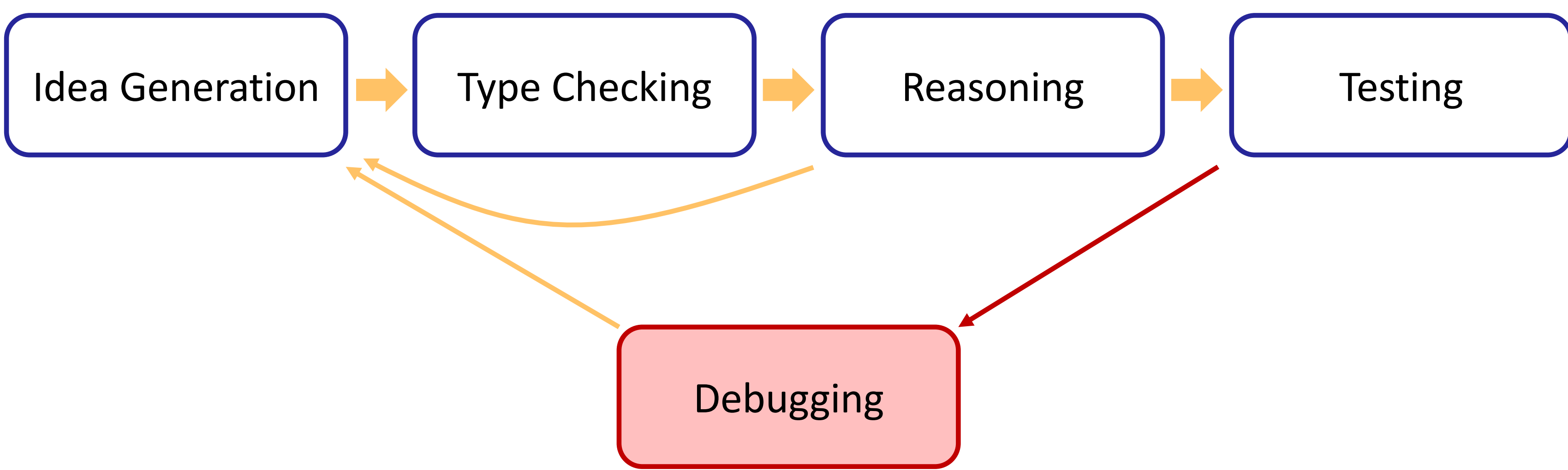
Debugging Client-Server

“Engineers are paid to think and understand.”

— Class slogan #1

Recall: Software Development Process

Given: a problem description (in English)



- As cost of debugging increases, must work <u>harder</u> to avoid it
  - correctness gets more important as code gets more complex

Client-Server Communication

- Client-server communication can fail in many ways
  - almost always requires debugging
- Include all required `.catch` handlers
  - *at least* log an error message
- Here are steps you can use when
  - the client should have made a request
  - but you don't see the expected result afterward
  - (practice this in section tomorrow!)

Client-Server Communication

1. **Do you see the request in the Network tab?**
   - the client didn't make the request
2. **Does the request show a 404 status code?**
   - the URL is wrong (doesn't match any `app.get` / `app.post`) **or** the query parameters were not encoded properly
3. **Does the request show a 400 status code?**
   - *your* server rejected the request as invalid
   - look at the body of the response for the error message **or** add `console.log`'s in the server to see what happened
   - the request itself is shown in the Network tab

Client-Server Communication

4. Does the request show a 500 status code?

- the server crashed!
- look in the terminal where you started the server for a stack trace

5. Does the request say “pending” forever?

- your server forgot to call `res.send` to deliver a response

6. Look for an error message in browser Console

- if 1-5 don’t apply, then the client got back a response
- client should print an error message if it doesn’t like the response
- client crashing will show a stack trace

Dynamic Type Checking

New TodoApp – Add Json

```
doAddJson = (data: unknown): void => {
  … // how do we use data?
};
```

- type of returned data is `unknown`
- to be safe, we should write code to check that it looks right
  - check that the expected fields are present
  - check that the field values have the right types
- only turn off type checking if you love painful debugging!
  - otherwise, check types at runtime

Checking Types of Requests & Response

- All our 200 responses are records, so start here

```
if (!isRecord(data)) {
  console.error("not a record", data);
  return;  // fail fast and friendly!
}
```

  - the `isRecord` function is provided for you
  - like built-in `Array.isArray` function

- Would be reasonable to throw an Error instead
  - but `console.error` is probably easier for debugging
  - second argument prints out the value of "`data`"

Checking Types of Requests & Response

- **Fields of the record can have any types**

```
if (typeof data.name !== ’string’) {
  console.error(“name is missing or invalid”, data);
  return;
}

if (typeof data.amount !== ‘number’) {
  console.error(“amount is missing or invalid”, data);
  return;
}
```

  - **should check each element of an array before you use it!**

    call `Array.isArray` and then loop through the elements to check `typeof`

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 will do this (and provide) tree `toJson` / `fromJson` in **HW8**

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

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)

Use Type Checking to Avoid Debugging

- Work harder to avoid debugging when it's painful
  - mistakes now help you learn where to be extra careful

- Resist the temptation to skip checking types in JSON
  - "easy is the path that leads to debugging"

- Query parameters also require checking:

```
const url = "/list?" +
    "category=" + encodeURIComponent(category);
```

  - converting from a string back to JS data is also *parsing*
  - can be a bug in encoding or parsing

Use Type Checking to Avoid Debugging

- Work harder to avoid debugging when it's painful
  - mistakes now help you learn where to be extra careful

- Be careful of turning off type checking:

```
  resp.json().then(this.doAddJson)
  …

doAddJson = (data: TodoItem): void => {
  this.setState(
      {items: this.state.items.concat([data])});
};
```

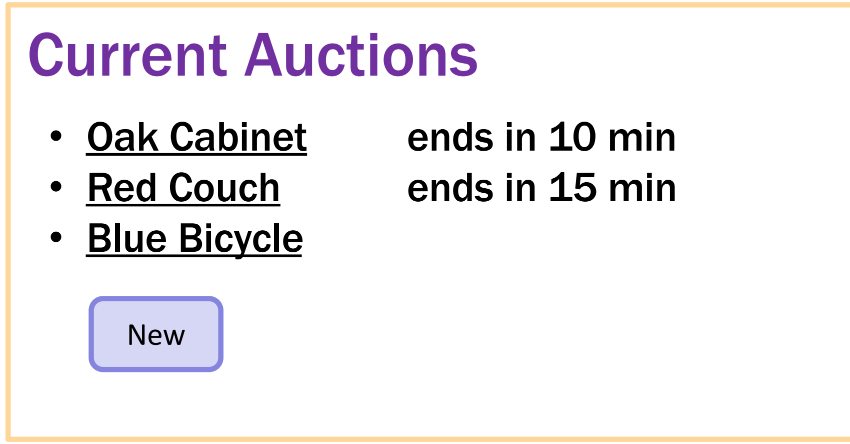
imagine this debugging
when you make a mistake

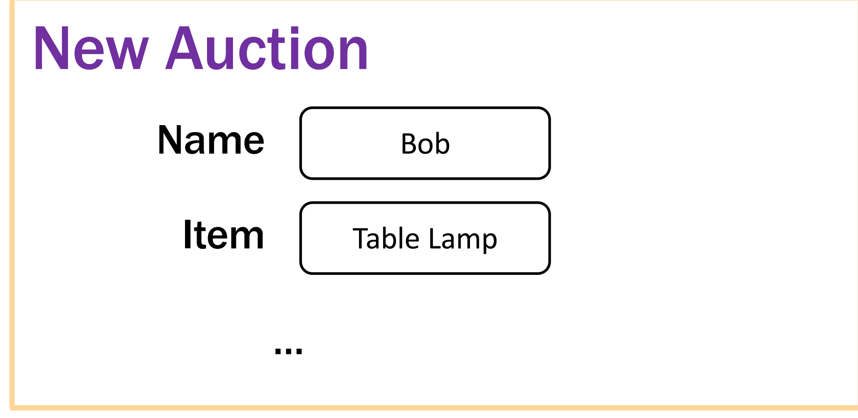
  - promises use "`any`" instead of "`unknown`", so TypeScript let you do this

Example: Auctions <u>Server</u>

Recall: Auction UI

- Auction site has three different "pages"





Oak Cabinet

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

Current Bid: $250

Name: Fred

Bid: 251

Submit

Recall: 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

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)

  3. Connect the client to the server
     - use fetch to update data on the server before doing same to client

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!

Example: Auctions 2.0

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)
  3. Connect the client to the server
     - use fetch to update data on the server before doing same to client

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: Auction Client: `App.tsx`

- **Permanent state is the list of auctions**
  - **the `App` component stores the auction list**
    passed down to subcomponents in their props
  - **subcomponents do not mutate the auction list**
    they invoke **callbacks** to have the `App` update the auction list
  - **can update the App to do client-server communication**

Recall: Auction Client: `App.tsx`

- **Permanent state is the list of auctions**
  - **the `App` component stores the auction list**

    passed down to subcomponents in their props
  - **subcomponents do not mutate the auction list**

    they 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);
  }
};
```

More Auction Features

- Our "Bid" button cannot mutate the auction
  - must call back to the App and have it change the auction
  - App will call `setState` with a new array of auctions
  - App will later render `AuctionDetails` with a new auction

- This runs into a problem:
  - RI relates props and state (bid is above min bid)
  - RI is established in the constructor
  - constructor *will not run* in this case!

Lifecycle Events

- Warning: React doesn't unmount when props change
  - instead, it re-renders and calls `componentDidUpdate`
  - you can detect a props change there

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

  - would need to fix the RI in this method (ugh)

More Auction Features

- 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`!

What's Still Missing?

- Saw everything needed 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