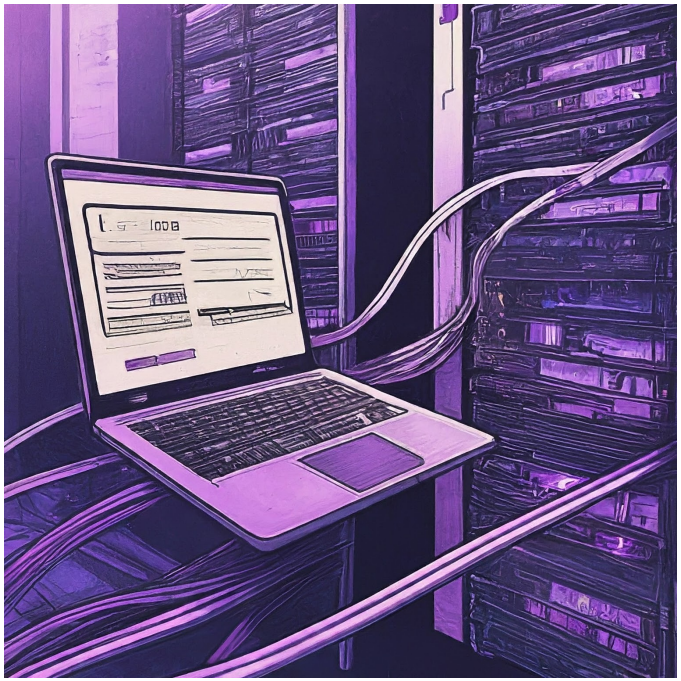


CSE 331

Client-Server Interaction

Kevin Zatloukal

Steps to Writing a Full Stack App

- Data stored only in the client is generally *ephemeral*
  - closing the window means you lose it forever
  - to store it permanently, we need a server

- We recommend writing 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

Steps to Writing a Full Stack App

- **We recommend writing 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., add/remove from the list, mark an item completed

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

    each way of changing the list becomes an **operation**
  - also need a way to get the list initially
  - only provide those operations

    can always add more operations later

Example: To-Do List Server

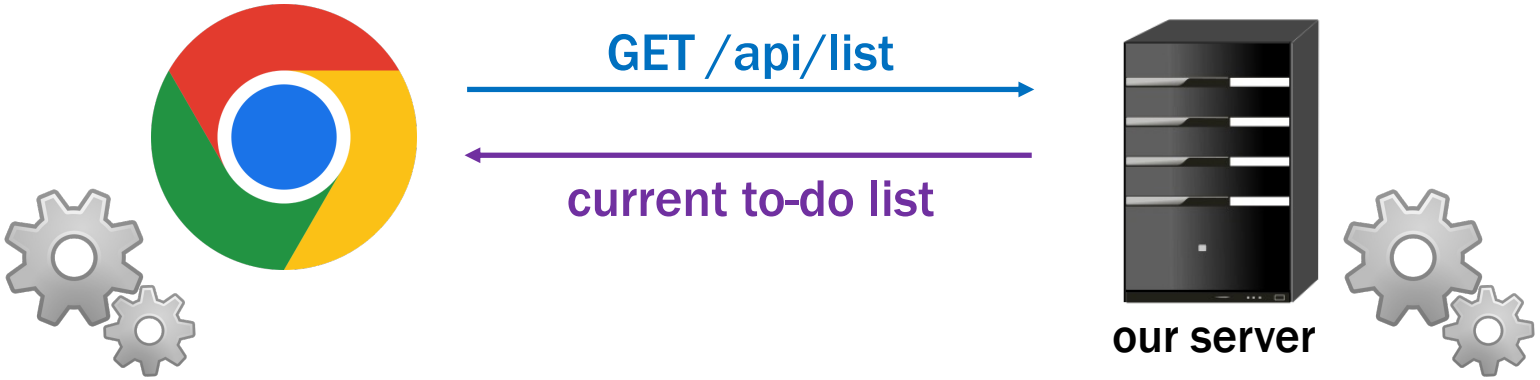
Steps to Writing a Full Stack App

- **We recommend writing 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

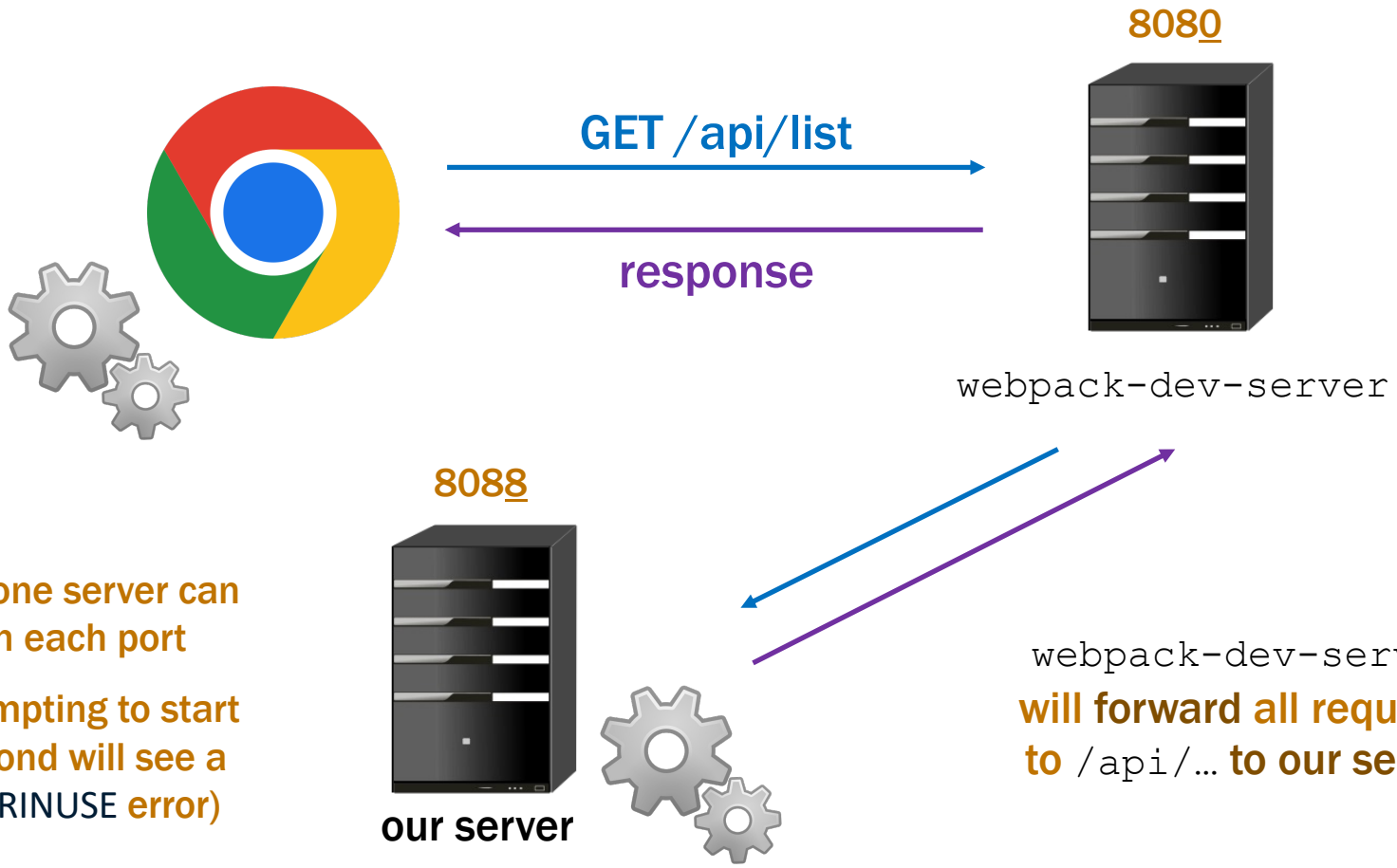
- Clients need to talk to server & update UI in response



Client will make requests to the server to
- get the list
- add, remove, and complete items

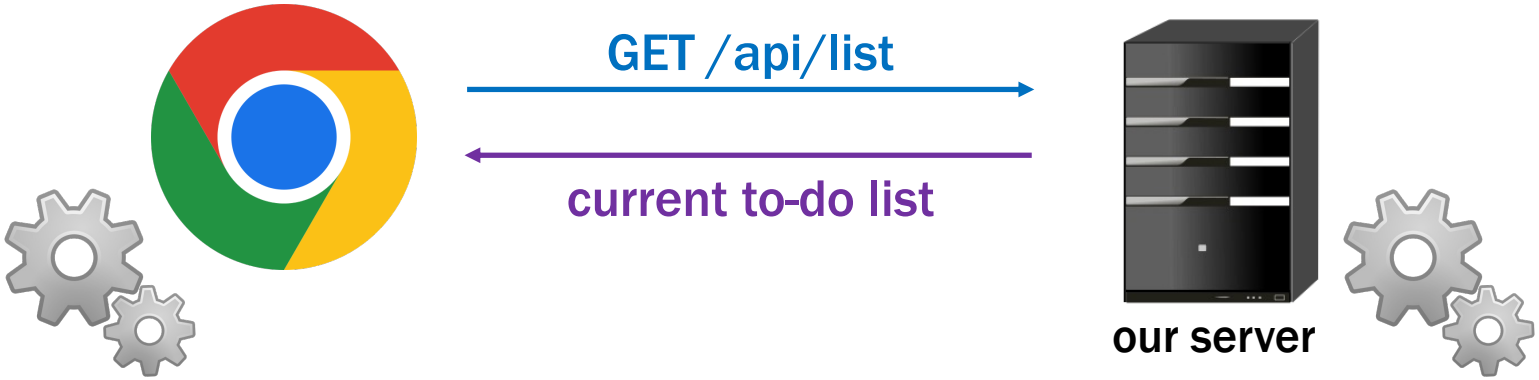
Development Setup

- **Two servers: ours and** `webpack-dev-server`



Recall: Client-Server Interaction

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



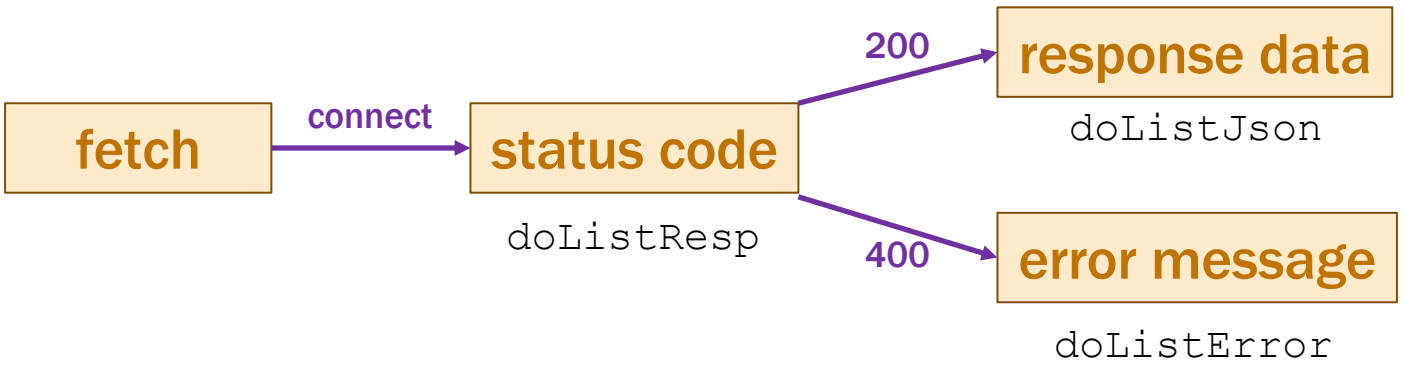
Components give us the **ability** to update the UI when we get new data from the server (an event)

How does the client make requests to the server?

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



Making HTTP Requests

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

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

Making HTTP Requests

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

```
fetch("/api/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`"**

```
const url = "/api/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 **catch every error**
  - **painful** debugging if an error occurs and you don't see it!

Making HTTP Requests

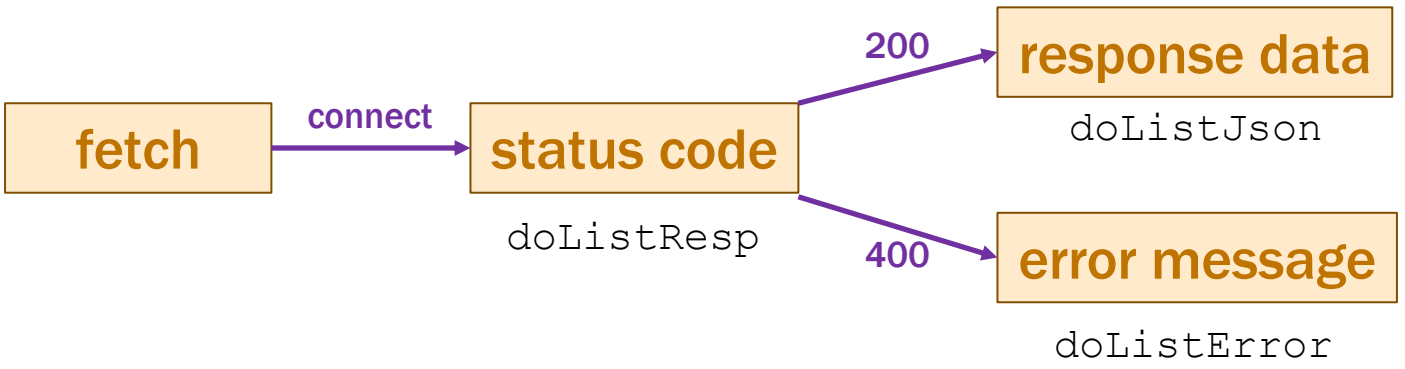
```
doListResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doListJson);
       .catch(() => this.doListError("not JSON");
  } else if (res.status === 400) {
    res.text().then(this.doListError);
       .catch(() => this.doListError("not text");
  } else {
    this.doListError(`bad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

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”);
};
```

Lifecycle Events

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

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

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});
};
```

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);
};
```

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

Another JavaScript Feature: `for … of`

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

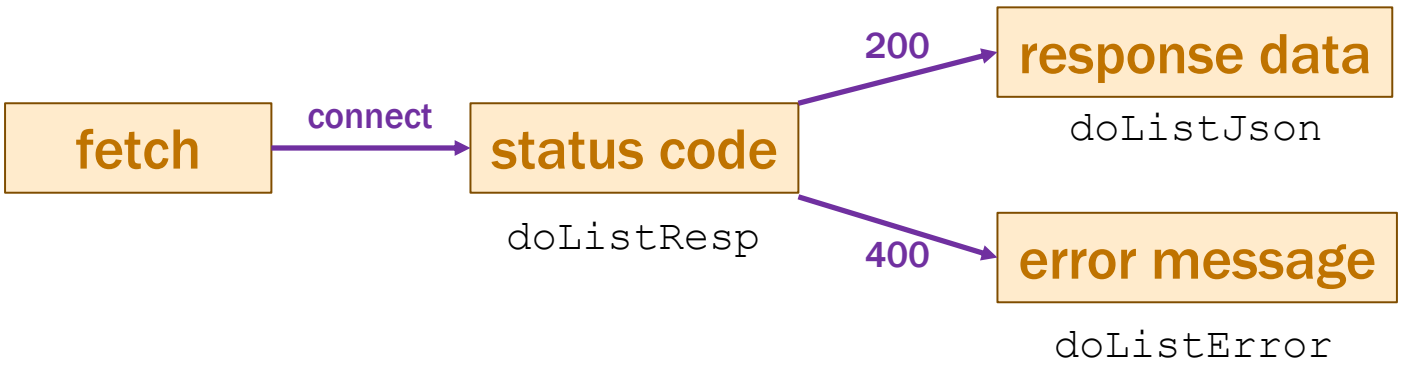
- "for .. of" iterates through array elements *in order*
  - … or the entries of a `Map` or the values of a `Set`

    entries of a `Map` are (key, value) pairs
  - like Java's "`for (… : …)`"
  - fine to use these

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

Recall: Function Literals

- **We used function literals for error handlers**

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

- **Our coding convention:**
  - **one-line functions (no {..}) can be written in place**
    most often used to fill in or add extra arguments in function calls
  - **longer functions need to be declared normally**

Lifecycle Events

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

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

This is used in HW2 in `Editor.tsx`:

- **changes to marker cause an update to name and color state**

Debugging Client-Server

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…

“Engineers are paid to think and understand.”

— Class slogan #1

Client-Server Debugging

client

New item: laundry Add

server

doAddClick
- fetch `/api/add`

express
- find route

addItem
- check parameters
- send `{added: true}`

doAddJson
- check response
- update state

To-Do List

☐ laundry

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
  - (will practice this in section next week!)

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))
  throw new Error(`not a record: ${typeof data}`);
```

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

    still need to check the type of **each** array element!

- Would be reasonable to log an error instead
  - using `console.error` is probably easier for debugging

Checking Types of Requests & Response

- Fields of the record can have any types

```
if (typeof data.name !== 'string') {
  throw new Error(
      `name is not a string: ${typeof data.name}`);
}

if (typeof data.amount !== 'number') {
 throw new Error(
     `amount is not a number: ${typeof data.amount}`);
}
```

TodoApp: processing `/api/list` JSON

```
// Called with the JSON response from /api/list
doListJson = (data: unknown): void => {
  const items = parseListResponse(data);
  this.setState({items: items});
};
```

- often useful to move this type checking to helper functions
  - we will may provide these for you in future assignments
- not part of the UI logic, so doesn't belong it that file

TodoApp: `parseListResponse`

```
// Retrieve the items sent back by /api/list
const parseListResponse = (data: unknown): Item[] => {
  if (!isRecord(data))
    throw new Error(`not a record: ${typeof data}`);

  return parseItems(data.items);
};
```

- **can only write "`data.items`" after we know it's a record**
  - type checker will object otherwise
  - retrieving a field on `undefined` or `null` would crash

TodoApp: `parseItems`

```
const parseItems = (data: unknown): Item[] => {
  if (!Array.isArray(data))
    throw new Error(`not an array: ${typeof data}`);

  const items: Item[] = [];
  for (const item of data) {
    items.push(parseItem(item));
  }
  return items;
};
```

TodoApp: `parseItems`

```
const parseItem = (data: unknown): Item[] => {
  if (!isRecord(data))
    throw new Error(`not an record: ${typeof data}`);

  if (typeof data.name !== "string")
    throw new Error(`name is not a string: ${typeof data.name}`);

  if (typeof data.completed !== "boolean")
    throw new Error(`not a boolean: ${typeof data.completed}`);

  return {name: data.name, completed: data.completed};
};
```

Use Type Checking to Avoid Debugging

- 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

- Be careful of turning off type checking:

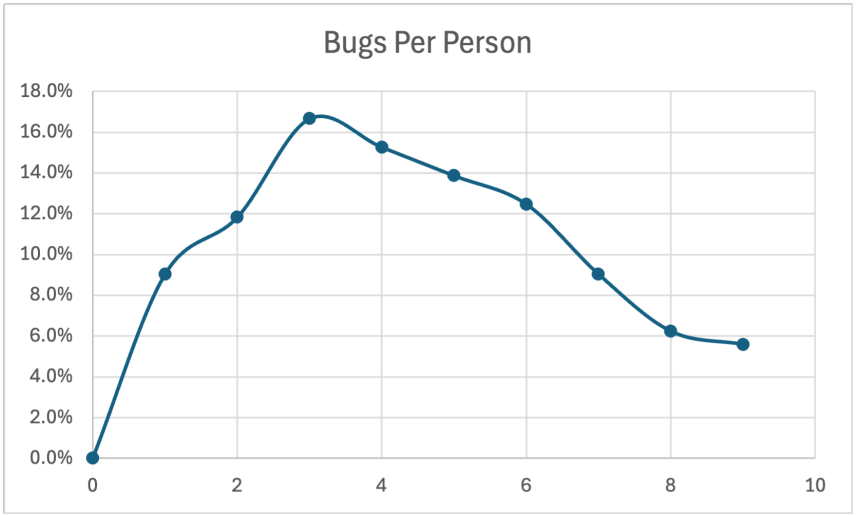
```
    resp.json().then(this.doAddJson)
    …

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

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

imagine this debugging
when you make a mistake

Summary of HW2

- **Number of bugs logged:**
  - average of 5.0 (median of 5)

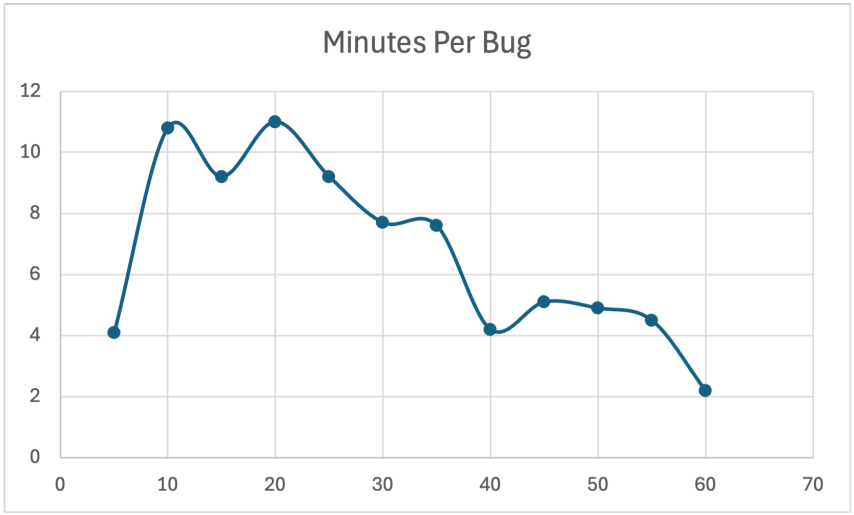


- **Average solution was 90 lines of code** (prob an over-estimate)
  - **1 bug every 18 lines of code**

    some of those "lines" are plain HTML… 1 bug every 16 is probably closer
  - **1 bug per 20–70 is normal even for professionals**

Summary of HW2

- Time spent per bug:
  - average of 40 minutes per bug
  - 20% more than 1 hour



- Long tail is making itself visible…

Summary of HW2

- Was the bug due to a disallowed mutation?
  - students reported 'yes' for 10% of bugs
  - such bugs took 22% longer to debug on average

- More than 5% had a mutation bug they didn't catch!
  - those are just the ones I found
  - that means it gets sent out to users

    not just reputation damage… also **painful** debugging

Summary of HW2

- **User reports the following bug:**

  "Sometimes, I can't click on one of the markers.
  Usually, it it works fine. But occasionally, you can't click on it."

- **First step is to figure out how to reproduce it**
  - **can't debug otherwise**

    wouldn't know that you've fixed the bug
  - **key reason why event-driven debugging is harder**

    command-line failure is instantly reproducible
  - **debugging a crash is easier than a non-crash!**

    crash comes with a stack trace (line of code with a failure)

Summary of HW2

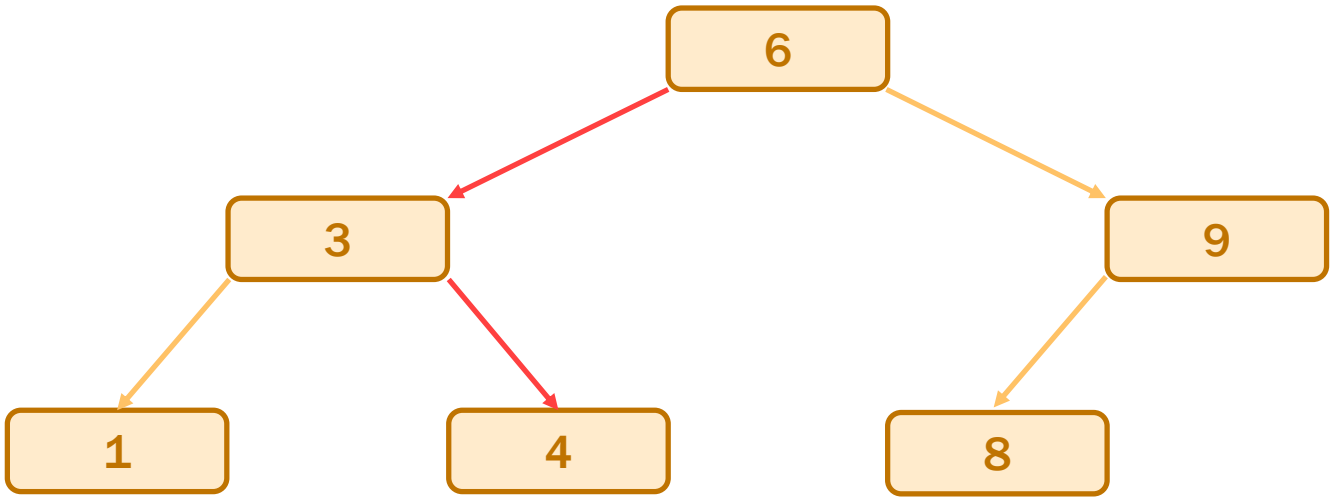
- Eventually, you find a way to reproduce it
  - no longer clickable after you move it very far away

- To debug, you must learn how `App.tsx` works
  - markers are stored in some kind of tree
  - searches the tree to find markers near the click

- To debug, you must learn how `marker_tree.ts` works
  - internal tree nodes split into NW, NE, SE, SW regions
  - marker was inserted into the correct region
  - when you search for it, it's no longer in the right region

Recall: Binary Search Trees

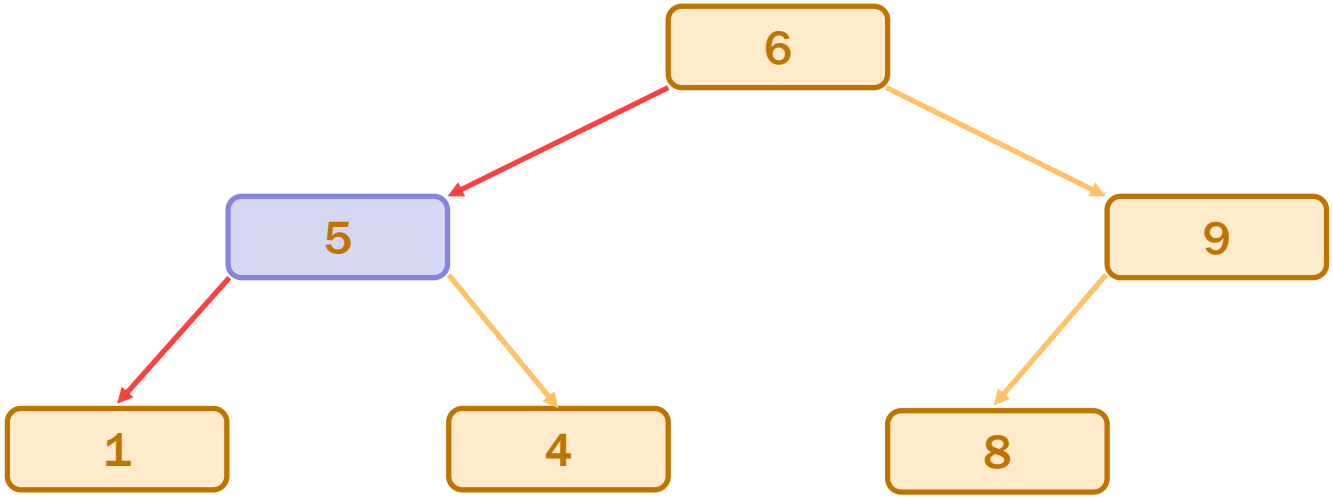
- Consider the following tree
  - searching for "4" proceeds as follows:



- Suppose someone changed "3" into "5"...

Recall: Binary Search Trees

- Suppose someone changed "3" into "5"...
  - now this happens when we search for "4":



  - It can no longer be found!

    Doesn't crash. It's just not found.
  - Problem doesn't occur on the line with the change

One "Solution" to HW2

```
type EditorState = {
  newMarker: Marker;
  …
};

doNameChange = (evt: ChangeEvent<…>): void => {
  this.state.newMarker.name = evt.target.value;
  this.setState({newName: evt.target.value});
};

doSaveClick = (evt: MouseEvent<…): void => {
  this.props.onSaveClick(newMarker.name, …);
};
```

already suspicious...
mutating `this.state` directly

One "Solution" to HW2

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

  this.state = {newMarker: this.props.marker, …};
}

doMoveToChange = (evt: ChangeEvent<…>): void => {
  const bldg = findBuildingByName(evt.target.value);
  newMarker.location = bldg.location;
  this.setState({moveTo: evt.target.value});
};
```

- Starting to get nervous…
  - are we allowed to mutate that marker?
  - no! that location is a key in a tree

Scary Bugs

- Do not fear crashes
  - often no debugging at all

    get a stack trace that tells you exactly where it went wrong

- Do fear unexpected mutation
  - failure will give you no clue what went wrong

    will take a long time to realize the BST invariant was violated by mutation
  - bug could be almost anywhere in the code

    anyone who mutates a `Location` could have caused it
  - could take *weeks* to track it down

Aliasing

Heap State

- “Heap state” = still used after the call stack finishes
  - after current function and those calling it all return
  - state could be arrays or records
- Extra references to the objects are called "aliases"
- No different from before when *immutable*
  - we don’t care who reads the data
- Vastly more complex when **mutable**…
  - within an event-driven application
  - creates the potential for failures far from bugs

Coupling

- High-quality code needs to be "modular"
  - split into pieces that can be understood *individually*
- When not possible, pieces are "coupled"
  - must understand both parts to understand each one
- Mutable heap state creates coupling
  - all pieces must know who else has aliases
  - all pieces must know who is allowed to mutate
- Coupling creates potential for **painful** debugging
  - bugs in one piece can cause failures in another

Mutable Heap State

- "With great power, comes great responsibility"
  - from Uncle Ben

- With aliases to mutable heap state:
  - gain efficiency in some cases
  - must keep track of every alias that could mutate that state

    any alias, anywhere in the *entire* program could cause a bug

“Programmers overestimate the importance of **efficiency** and underestimate the difficulty of **correctness**.”

— Class slogan #2

Easy Ways to Stay Safe

1. Do not mutate heap state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases…
   - create the state in your constructor and **don't share** it

```
class MyClass {
  vals: Array<string>;

  constructor() {
    this.vals = new Array(0);   // only alias
  }
  …
```

Easy Ways to Stay Safe

2. Do not allow aliases

(a) do not hand out aliases yourself

- return copies instead

```
class MyClass {
  // RI: vals is sorted
  vals: Array<string>;

  …

  values: (): Array<string> => {
    return this.vals;              // unsafe!
    return this.vals.slice(0);     // make a copy
  };

  …
```

Easy Ways to Stay Safe

2. Do not allow aliases

(b) make a copy of anything you want to keep

– does not matter if the caller mutates the original

```
class MyClass {
  // RI: vals is sorted
  vals: Array<string>;

  …

  // @requires A is sorted
  constructor(A: Array<string>) {
    this.vals = A;              // unsafe!
    this.vals = A.slice(0);     // make a copy
  };

  …
```

Easy Ways to Stay Safe

1. Do not use mutable state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases to *mutable* state
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

ensures only one reference to the object (no aliases)

- For 331, mutable aliasing across files is a bug!
  - gives other parts the ability to break your code
  - we will stick to these simple strategies for avoiding it

Rules of Thumb

Client Side

1. Data is small
   - anything on screen is O(1)
2. Aliasing is common
   - UI design forces modules
   - data is widely shared

Rule: avoid mutation

- create new values instead
- performance will be fine

Server Side

1. Data is large
   - efficiency maters
2. Aliasing is avoidable
   - you decide on modules
   - data is not widely shared

Rule: avoid aliases

- do not allow aliases to your data
- hand out copies not aliases
- (good enough for us in 331)

Language Features & Aliasing

- Most recent languages have some answer to this…
- Java chose to make `String` immutable
  - most keys in maps are strings
  - hugely controversial at the time, but great decision
- Python chose to only allow immutable keys in maps
  - only numbers, strings, and tuples allowed
  - surprisingly, not that inconvenient
- Rust has built-in support for tracking ownership
  - ownership can be “borrowed” and returned
  - type system ensures there is only one usable alias

Readonly in TypeScript

- TypeScript can ensure values aren't modified
  - extremely useful!

- Readonly tuples:

```
type IntPair = readonly [bigint, bigint];
```

- Readonly fields of records:

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

Readonly in TypeScript

- **Readonly fields of records:**

```
type IntPoint = {readonly x: bigint,
                 readonly y: bigint};
```

- **Readonly records:**

```
type IntPoint = Readonly<{x: bigint, y: bigint}>;
```

  - `this.props` **is** `Readonly<MyPropsType>`

- **More readonly...**

```
ReadonlyArray<bigint>
ReadonlyMap<string, bigint>
ReadonlySet<string>
```