


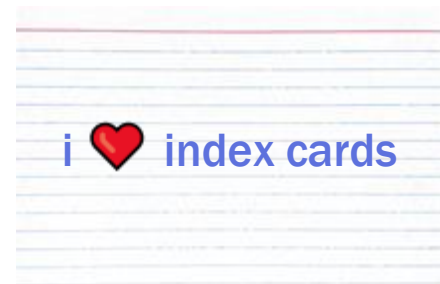
CSE 331 Summer 2025

Client-Server Interaction I

Jaela Field

HW1 Feedback request!

- Any thoughts 
 - how long did it take you?
 - opinions on submitting answers on Gradescope
 - was it good JS & server code practice?
 - any unclear directions?
 - any gripes?
- Anonymous Feedback



Reminders

- **HW 2**
 - **Must identify defect causing specific failure**
include only that line of code, or label clearly
[\(ed post\)](#)
 - **No Ed after 11pm Thursday!**

Client-Server Interaction

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: 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

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

Server Documentation

- Client cannot use the server unless it knows how!
 - What are the required inputs?
 - What are the expected outputs? (including error responses)

```
/**
 * Add the item to the list.
 * - 200 response containing {name: string, added: boolean},
 *   where 'added' indicates if item was added or not (because
 *   it already exists in list)
 * - 400 if required name was not provided
 * @param req the request. Must contain string name of item to
 *           add in body.
 * @param res the response.
 */
export const addItem = (req: SafeRequest, res: SafeResponse)
```

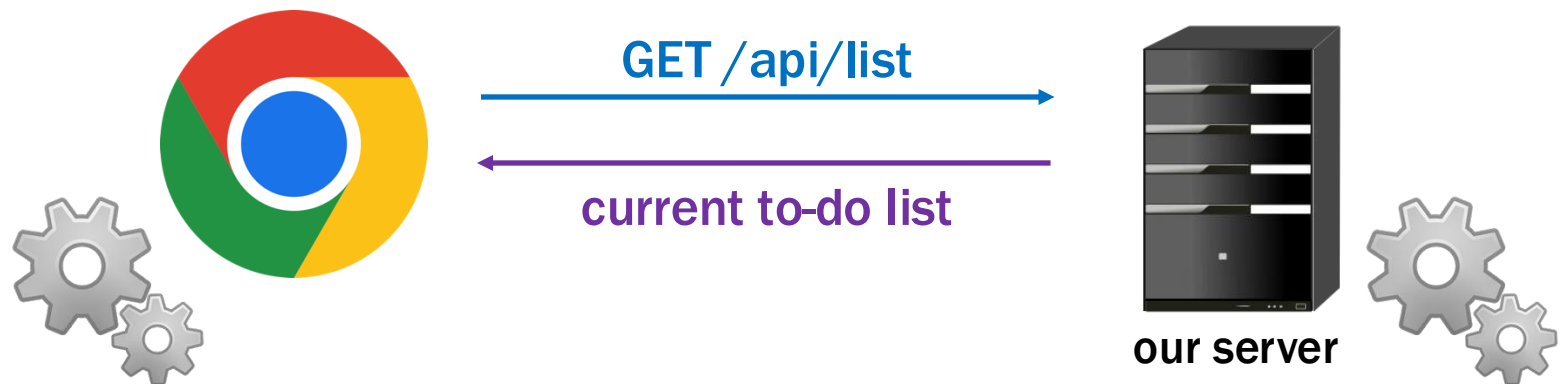
- We'll talk more about JSDocs later on!

Steps to Writing a Full Stack App: Connect

- **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

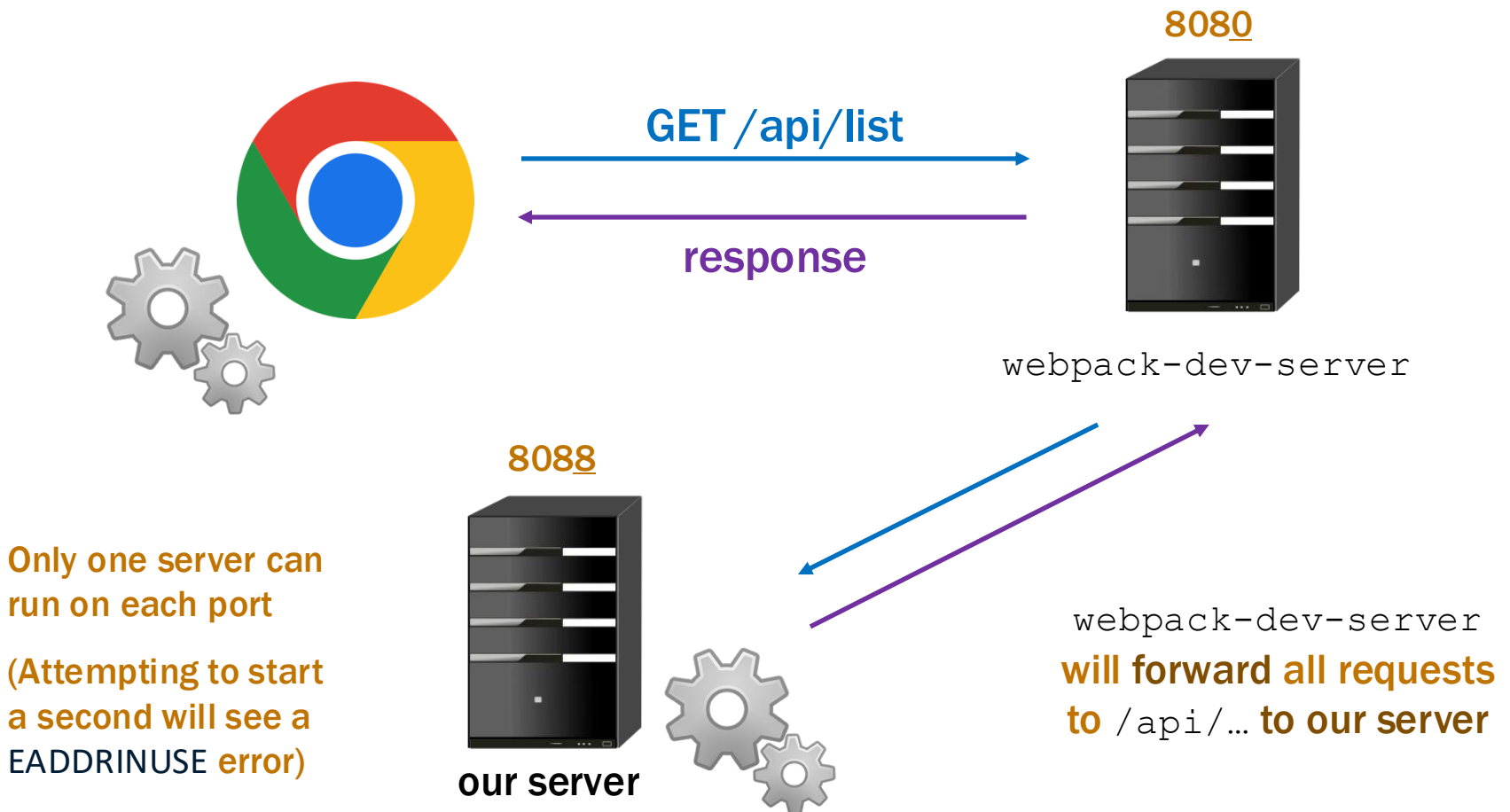


Client will make requests to the server to

- get the list
- add, remove, and complete items

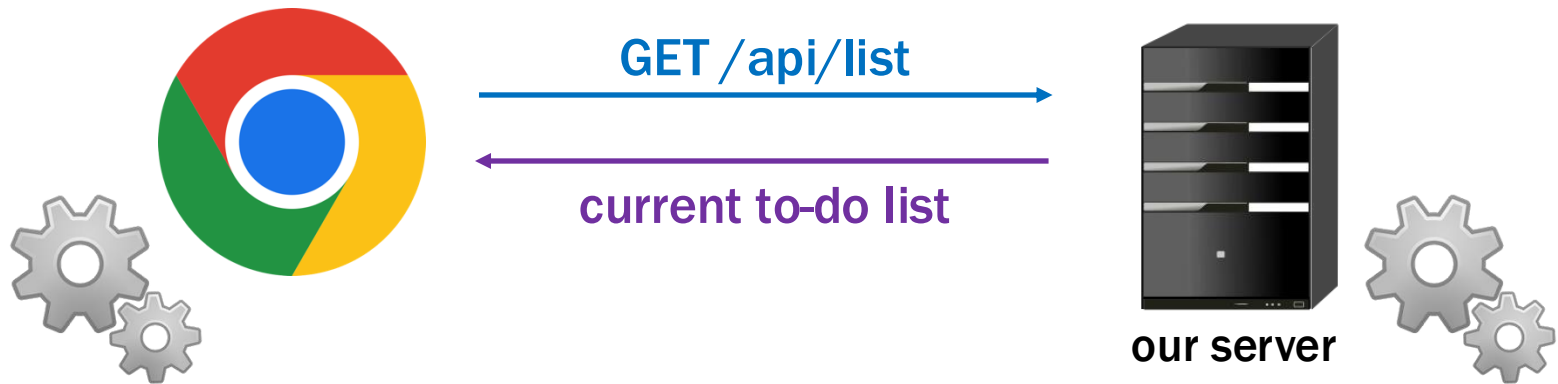
Development Setup

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



Client-Server Interaction: Making Requests?

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

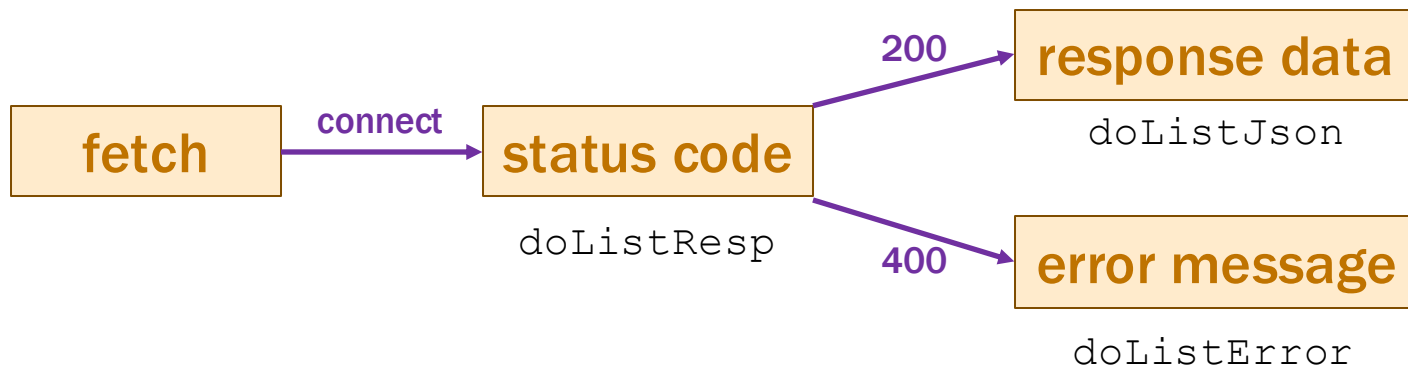


Components give us the **ability** to update the UI when we get new data from the server (an event)

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: Using Fetch

- 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: After Fetch

- **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: Query Parameters

- 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: Status Codes

- 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: Error Handling

```
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 and make it visible
 - **painful** debugging if an error occurs and you don't see it!

Making HTTP Requests: More Error Handling

```
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

Recall: Function Literals

- **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**

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 Gotcha: Unmounting

- **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:

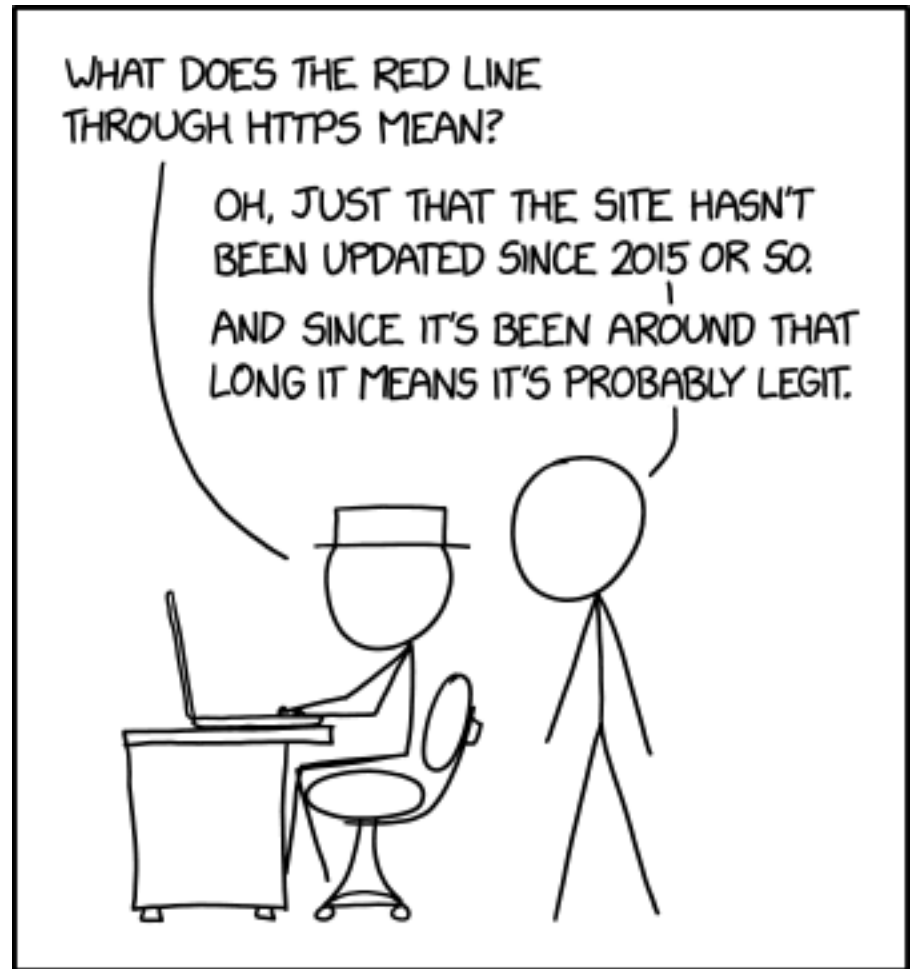
- changes to markers cause an update to name and color state

CSE 331

Summer 2025

Client-Server Interaction II

Jaela Field



xkcd #1537 (and Matt, ty)

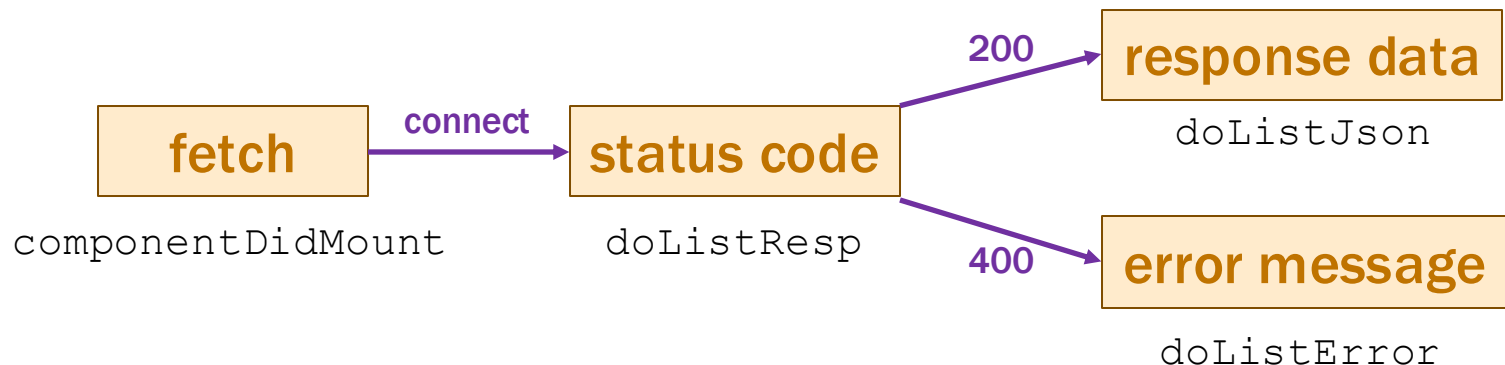
Feedback from last time

- **Lectures: too fast!**
 - Please ask me to chill out
- **HW1**
 - **Wide range of amount of time to complete**
come visit us in OH! Ask questions on Ed!
 - **Unclear expectations for Gradescope questions and Scope of debugging**
Will add examples again for HW3! + see HW1 feedback



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: 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
 - can use `.entries()` function on array, to iterate over [index, value] tuples for each entry
 - like Java's "`for (... : ...)`"
 - fine to use these

To-Do List: wrap up requests

One More Change

- Don't have the items initially...

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

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








New TodoApp — Requests

To-Do List

- ☒ laundry
- ☐ wash dog

Check the item to mark it completed. Click delete to remove it.

New item:

Name	Status
 localhost	200
 main.92b8bcac2eee343ace7...	200
 ws	101
 list	200
 add	200
 add	200
 toggle	200











To-Do List

- ☐ wash dog

Check the item to mark it completed. Click delete to remove it.

New item:

Name	Status
 localhost	200
 main.92b8bcac2eee343ace7...	200
 ws	101
 list	200
 add	200
 add	200
 toggle	200
 remove	200

Dynamic Type Checking

New TodoApp – Add Json and Types

```
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
- Type Narrowing**
- only turn off type checking if you love **painful** debugging!
 - otherwise, check types at runtime

Checking Types of Requests & Response (1/2)

- 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 (2/2)

- 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 in 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 – Type Checking the Array

```
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 – Type Checking Items

```
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 (1/2)

- 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 (2/2)

- 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

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 Communication Complexity



client

New item:



server

doAddClick

- **fetch** `/api/add`



express

- **find route**

doAddJson

- **check response**
- **update state**



addItem

- **check parameters**
- **send** {
 name: "laundry",
 added: true
}

To-Do List

☐ laundry

Client-Server Debugging

- 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 tomorrow!)

Client-Server Debugging Tips (1/2)

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 Debugging Tips (2/2)

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