CSE 331
Software Design & Implementation

Winter 2025

Section 3 – Full Stack Apps

Clone the section code!

```
git clone
https://gitlab.cs.washington.edu
/cse331-25wi/materials/sec03.git
```

Administrivia

- HW 3 released later today, due wednesday (1/29) at 11pm
    - Try to get it done on time because the next homework is released the next day

Client-Side vs Server-Side – Review

- **Client-Side JavaScript**
  - Code so far has run inside the browser
    - webpack-dev-server handles HTTP requests
    - Sends back our code to the browser
  - In the browser, executes code of index.tsx

- **Server-Side JavaScript**
  - Can run code in the server as well
    - Returns different data for each request (HTML, JSON, etc.)
  - Can have code in *both* browser and server

Client-Side vs Server-Side – Review

Client-Side

HTTP GET

index.html
index.tsx etc.

webpack-dev-server

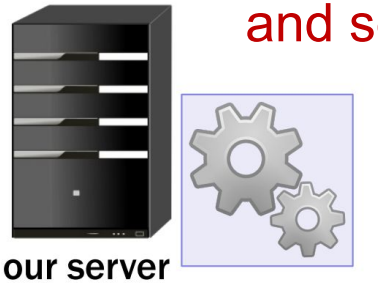
Code only on browser

VS

Server-Side

HTTP GET

response data

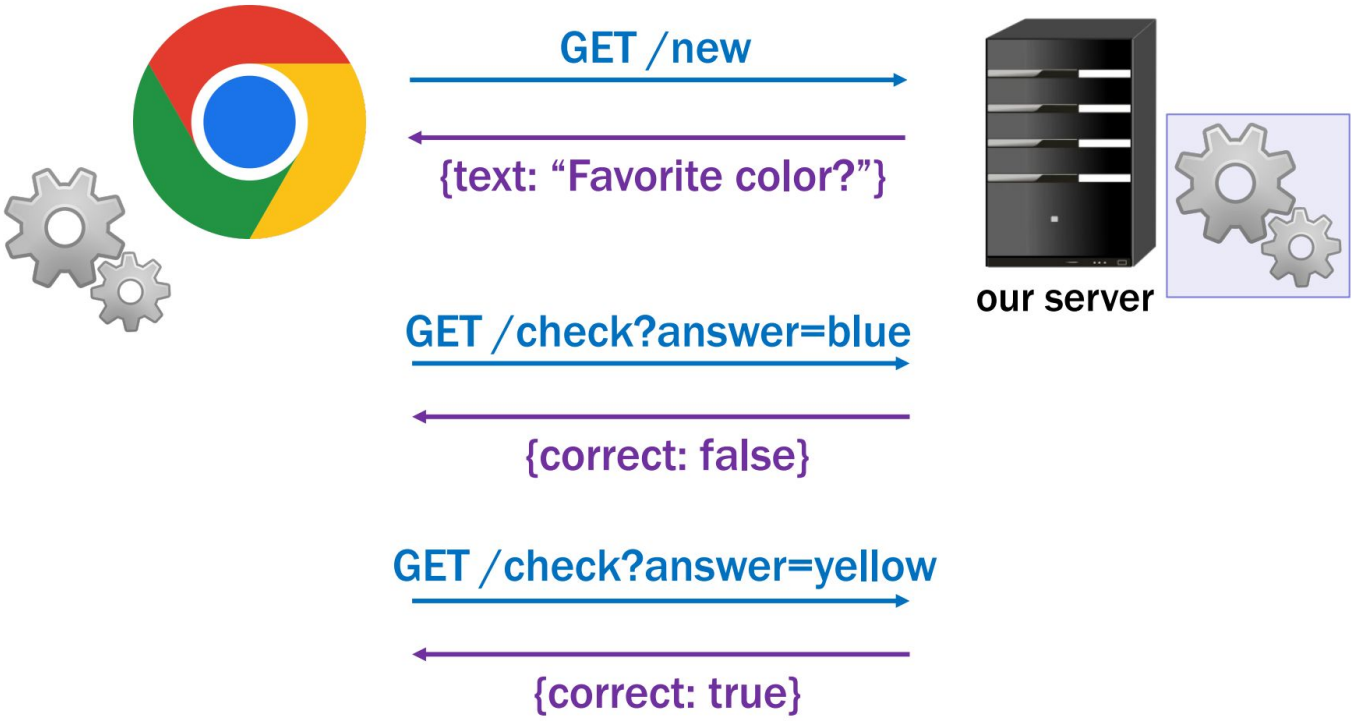


our server

Code on browser and server

Custom Server

- In a custom server, we can define useful routes
- Interacting with app will result in a series of requests and responses



Aliases

(a) Class that maintains an array in a specific order

```
class MyClass {
    // RI: vals is sorted
    vals: Array<string>;
    …
    values: (): Array<string> => {
        return this.vals;              // unsafe!
        return this.vals.slice(0); // make a copy

    };
    …
```

- **Do not hand out access to your own array**

Aliases

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

```
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
};
…
```

- **Do not make your own fields be something someone else has access to.**

Aliases

- Objects in “Heap State” means that its still being used after the call stack finishes.

- Extra references to these objects are called “aliases”

- When having aliases to mutable heap state:
  - We can gain efficiency in some cases.
  - We must keep track of all aliases that can mutate that state.

- For 331, mutable aliasing across files is a BUG!
  - Allows other portions of your code to break you code
  - “Copy in, copy out” to avoid aliases

Steps to Writing Full Stack App (Review)

- **Data stored only in the client is *ephemeral***

  – closing the window means you lose it forever

- **Write apps in this order:**

  1. Write the client UI with local data

     – no client/server interaction at the start

  2. Write the server

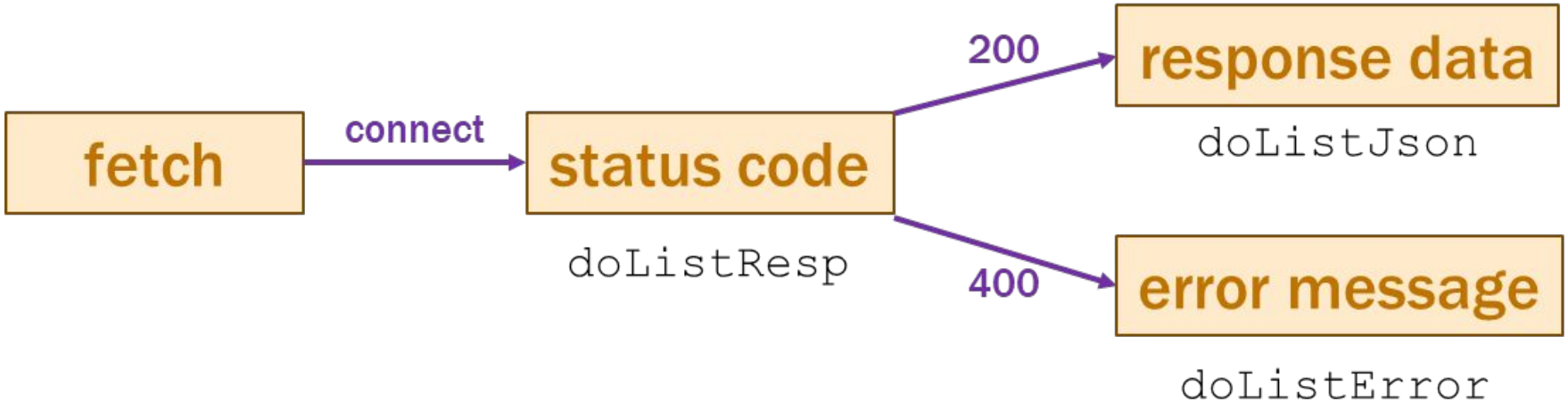
     – official store of the data

  3. Connect the client to the server

     – use fetch to update data on the server before doing same to client

Fetch Request methods

1. Method that makes the fetch
2. Handler for fetch Response
3. Handler for fetched JSON
4. Handler for errors



Making an HTTP Request (Review)

- **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"))
```

- **Fetch** returns a "promise" object, has `.then` & `.catch` methods
  - – then handler is called if the request **can** be made
  - – catch handler is called if **could not connect** to the server at all or if "then" handler **throws exception**

Handling HTTP Response (Review)

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

```
if (res.status === 200) {
        res.json().then(this.doListJson)
            .catch(() => this.doListError("200
        response is not JSON"));}
```

  - with **400 status code**, use `res.text()` to get error message

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

React Lifecycle Methods (Review)

- **React includes events about its "life cycle"**
  - `componentDidMount`: UI is now on the screen
  - `componentDidUpdate`: UI was just changed to match render (also called when props changes)
  - `componentWillUnmount`: UI is about to go away
- **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");
};
```

Type Checking of Request/Response

- All our 200 responses are records, so start here

  –the `isRecord` function is provided for you

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

- Fields of the record can have any types

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

- For Arrays, call `Array.isArray` and then loop through the elements to check `typeof`

Client-Server Communication Debugging Steps

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 Debugging Steps

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

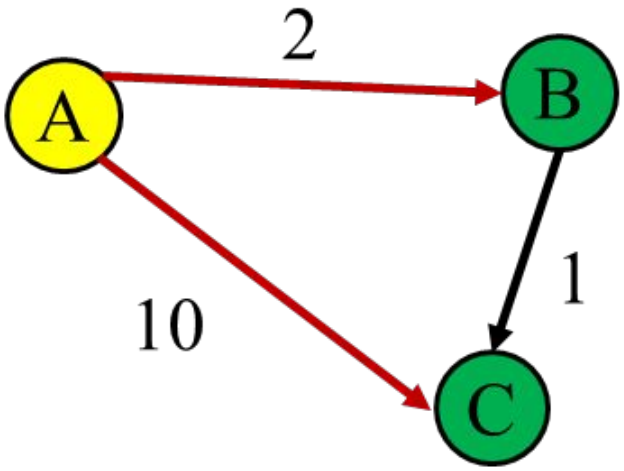
5. **Does the request say "pending" forever?**
   - your server forgot to call `res.send` to deliver a response

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

HW 3 Prep: Dijkstra's Algorithm

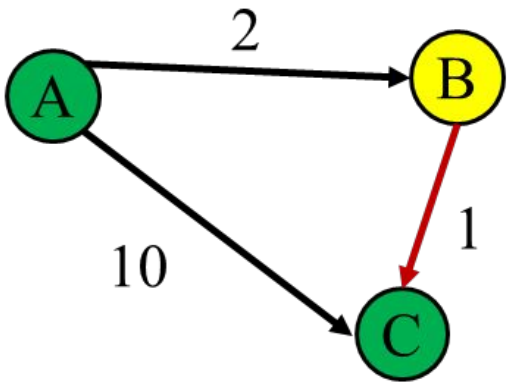
- **Main idea:** Start at the source node and find the shortest path to all reachable nodes.
- **Input:** graph with no negative edge weights, start node s
  - When a node is the closest undiscovered thing to the start, we have found its shortest path

Node	Finished	Cost	Prev
A	False	0	-
B	False	∞	
C	False	∞	

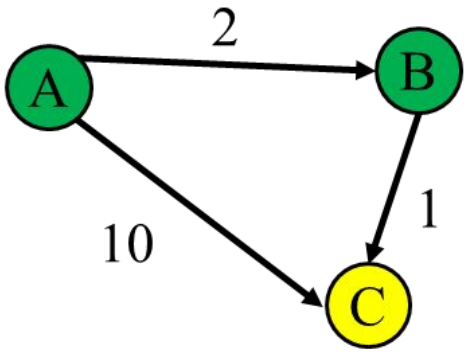


HW 3 Prep: Dijkstra's Algorithm

Node	Finished	Cost	Prev
A	True	0	-
B	False	~~∞~~ 2	A
C	False	~~∞~~ 10	A



Node	Finished	Cost	Prev
A	True	0	-
B	True	2	A
C	True	~~10~~ 3	~~A~~ B

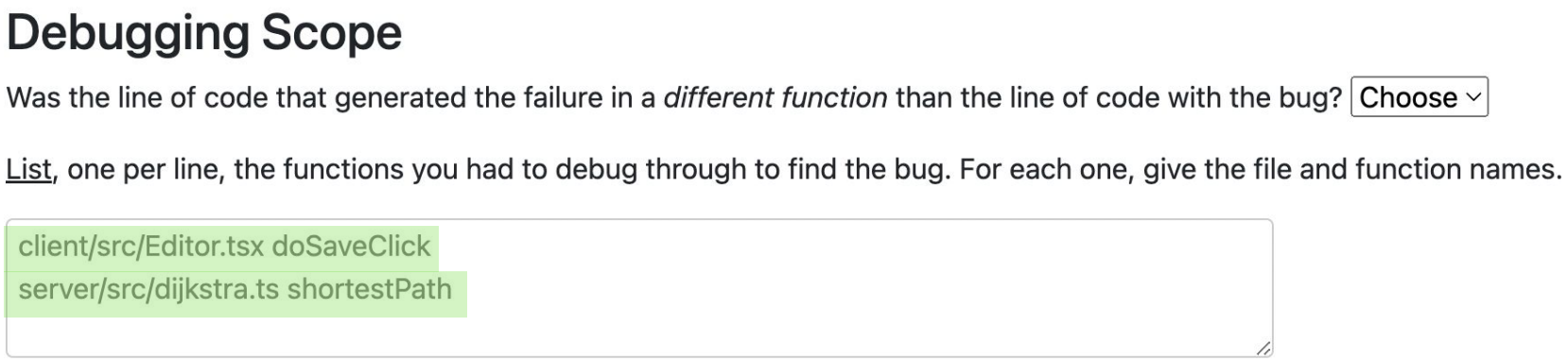


Dijkstra's algorithm – pseudocode

```
active = priority queue of paths.
finished = empty set of nodes.
add a path from start to itself to active
<inv: All paths found so far are shortest paths>
while active is non-empty:
    minPath = active.removeMin()
    minDest = destination node in minPath
    if minDest is dest:
        return minPath
    if minDest is in finished:
        continue
    for each edge e = ⟨minDest, child⟩:
      if child is not in finished:
        newPath = minPath + e
        add newPath to active
    add minDest to finished
```

Debugging Log

- https://comfy.cs.washington.edu/service/hw3-practice
- Be sure to keep track of each function you work on as you debug (ex. client/server, file name, function name)
- **Example:**



sec-debug coding exercise

debugging practice !!