CSE 331

Full Stack Apps

Kevin Zatloukal

Administrivia

- HW8 section materials on website
- HW8 released today
  - start early!
  - this is a debugging assignment
- Concise notes on today's topic on website

Reminders

"Engineers are paid to think and understand"

— James Wilcox

- thinking required for reasoning about code

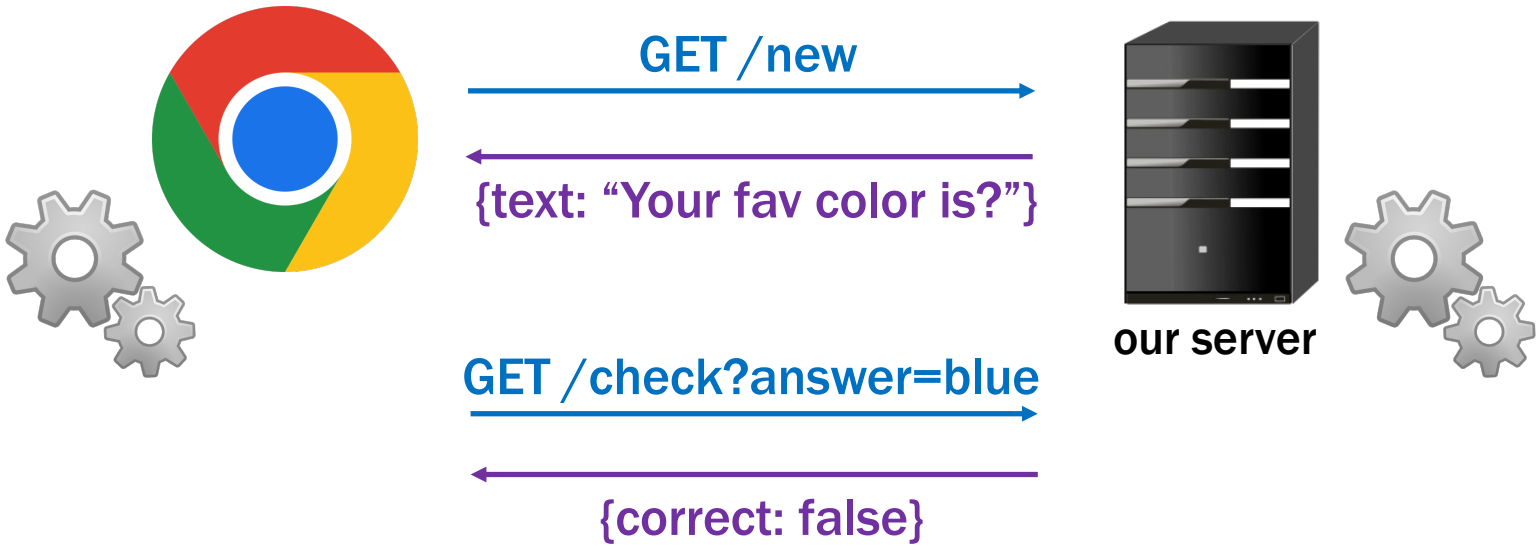
  saw that in HW3-7 (see, e.g., `splitWords` in HW7)

- understanding required for debugging

  will see that in HW8-9

Recall: Client-Server Interaction

- Client needs to update the UI after getting response
  - need a way to update the UI without a reload



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

How do we get new data from the server?

Client-Server Interaction

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

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 **catch every error**
  - **painful** debugging if an error occurs and you don't see it!

Making HTTP Requests

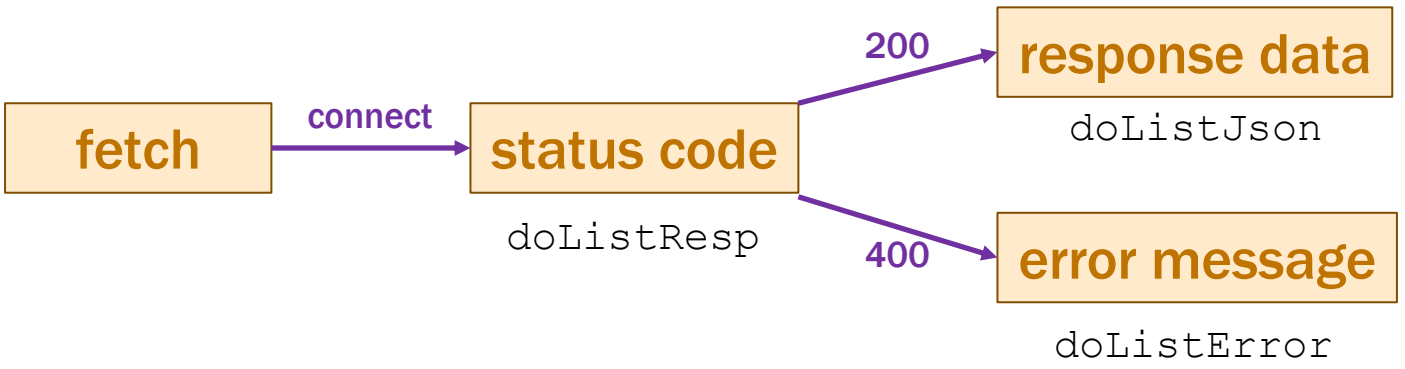
```
doListResp = (res: Response): void => {
  if (res.status === 200) {
    res.json().then(this.doListJson);
       .catch(() => this.doListError("not JSON");
  } else if (res.status === 400) {
    res.text().then(this.doListError);
       .catch(() => this.doListError("not text");
  } else {
    this.doListError(`bad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

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

```
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 – 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}`);
  }
};
```

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)

Lifecycle Events

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

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!
  }
};
```

  - **better to avoid this if possible**

    good setup for **painful** debugging

Debugging Client-Server

Client-Server Communication

- Client-server communication can fail in many ways
  - almost always requires debugging

- Here are steps you can use when
  - the client should have made a request
  - but you don't see the expected result afterward

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