Backend Considerations

When offering/writing/updating services

- What does the Back End need to consider?

Deciding on Endpoints

- What are your "resources"
    - Likely have more than 1!
    - Students?
    - Cats?
    - Todos?
    - Todo Lists?
- Is anything in path a variable?
    - Very common in REST!
- What interactions do you have?

Naming is hard!

- Collections tend to be plural
  - `/cats`, not `/cat`
- Vaguely sentence-like
  - `GET /cats/Jorts`
- Can be a little abstract!
  - `GET /session`
  - `POST /session`
  - `DELETE /session`
  - Why not `DELETE /session/:id`? (Security)

Data Model is important

Can't know your Endpoints without a data model

- Identifiers in paths
- Resources are records/collections

Key questions

- What records/collections do you have?
- How do you identify a record?
- How do you search a collection?

Service endpoints vs page/asset URLs

Service **endpoints** different from **pages**

- Expect different inputs
- Give different responses

How does a user know which one a URL is?

How do you make sure one doesn't occupy a URL the other may want in the future?

- Go to add a service, but there is already a page?
- Go to add a page, there is already a service?
- Often run by different teams!

One Easy Answer: Dedicated Root Path

Example:

- All services start with `/api/`
- No pages will start with `/api/`
- (`/api/` is a common example, can be anything)

Easy for multiple teams to follow these rules

Not Found (404) Page common example

A service path offers solution to a common issue

- Browsers expect 404 HTML page
- Service calls with no matches respond 404
- Service calls that messed up url get…?

With Service Path, server can:

- Outside of Service Path:
    - Respond with 404 HTML
- In Service Path:
    - When no service, respond with clear 404 data
    - Service responds with clear 404 data

Implementing a Service Path in Express

Just have `/api/` at the start of your route paths

- It's that simple
- Express routers can "collect" routes together
    - But not needed for this course

Versioning Services

A web service can be used by MANY client applications

- Different applications
    - Or different versions of same app
        - (mobile/desktop)

Changing your API

- Changing input/output expectations
- Breaks clients

Clients simultaneously update when service does?

- Not possible

Version in Root Path

`/api/v1/` as root of all paths

- When `/api/v2/` rolls out with changes
    - `/app/v1/` keeps working as in past
    - Clients can move to new service at own pace
    - `v1` can be retired after all clients upgrade

Why not semver? (ex: `/v1.2.3/`)

- We only care about API breaking changes
- New versions are a major pain to roll out
    - While maintaining old
    - Want few version changes

Reporting Errors

- What to respond with and how

Common HTTP Success Status Codes

- 204 - No Content (No Body)
    - Success but no body sent
    - I generally avoid returning 204
        - Most services aren't written by me :)
- 206 - Partial Content
    - May be used with Pagination (see later)

Common HTTP Client Error Codes

- 400 - Bad Request
- 401 - Unauthorized, 403 - Forbidden
    - User needs to log in (401)
    - User is logged in, but isn't allowed (403)
- 404 - Not Found
    - No matching records (for service)
- 409 - Conflict
    - Request data conflicts with server data
- 429 - Too Many Requests
    - Used when services rate-limit clients

Common HTTP Server Error Codes

- 500 - Unexpected Server Error
  - Generic Server error
- 501 - Not Implemented
  - Wrong HTTP method
- 502 - Bad Gateway, 504 - Gateway Timeout
  - Failed to talk to some other server
- 503 - Service Unavailable
  - Temporary problem

Notably, not much user can do

What to send in error body?

- Send enough detail on 4xx for user to correct error
    - What data was bad?
    - Why why it bad?
- Be sure to include enough that *you* can debug
    - Knowing where the error came from is helpful
- Often better to send codes or brief messages
    - Don't hide meaning
    - Clients can change to text of choice
    - Need to document and share these!

What NOT to send in error body

- Do NOT send stacktrace-type details
  - Could reveal sensitive information
- Avoid echoing unsanitized data back to client

What format for error bodies?

- Be consistent
- I recommend same format as success body
    - Ex: JSON
- Some use text instead

Remember

- Goal is for service to be **consumed**
    - Including errors
- Goal is NOT just to send data
    - Make it easy and convenient to USE

Returning Data on Success

JSON is most common format

- But you CAN send any format
    - XML, HTML, text, YAML, .ini, etc

What do you return?

A GET has an obvious return

- But what to return for other methods?

General guidelines

- If you created a new record
    - Return either ID or URL for that resource
- If you changed a record
    - Return the changed record
- Don't return big data unless requested
- Don't return data outside resource

Considering Slow Queries

Queries are usually talking to databases

- select, update, etc

Queries can take a long time

> *Find the birth dates of all authors that had cats whose names started with 'J'*

Service requests can timeout

- Also, users are impatient

One solution: Check Back

Server can create a "query"

- A resource

Server responds to request creating query

- Responds quickly
- Responds with an ID/URL for the created query

Client can check back later

Client can check back later

- Using query resource URL
- Response if query not yet done
- Response if query done
    - Might be results
    - Might just be yes/id/resource URL

Once query complete Client can request results

- Query removed by request/time/some process

Pagination

Too many results

- Lots of bandwidth
- May make slow queries

How often do you look at Page 3 of Google results?

- Yet could be millions of results

What is Service Pagination?

Service returns partial results

- Indicates which part
- Client can request different parts

NOT SAME AS CLIENT PAGINATION

- Often both happen in sync
- Not always though
    - Client can make multiple requests
    - Client can have all and show only pages

Pagination through Storage

How to paginate server data depends on storage

- Can tell DB to return only some results
- Could store full results in a caching layer
    - Only return partial results through service

Depending on storage, server might need to know

- Start/end points
- Page "number"
- Start point + Number of results/page
- a "cursor" to the cached results

Pagination Request/Response

Does the service return (and how?)

- HTTP Status code 206 (Partial Content)?
- Cursor id?
- Start/End point of results?

Does service accept (and how?)

- Start point for results?
    - What if records can delete/reorder?
- Cursor id?
- Number of results/page?

```
https://www.google.com/search?q=cat+videos&start=40
```

Service Authorization

- How would we write services to DO authorization?
- How do service calls check your authorization?

Sample Authentication endpoint

- **POST** `/api/v1/session` - sets cookie ("logged in")
- **GET** `/api/v1/session` - client can see if logged in
- **DELETE** `/api/v1/session` - clears cookie ("logout")

Here a "session" is a resource

- We create, get, or delete "session"
- DELETE *could* use an id
    - But session-ids are secret
    - Keep secret data out of urls

Using Auth Endpoint

- Set/clear cookie on response
- No Redirect!
    - Because it isn't navigation
- What data in response?
    - Should be limited to session
    - Probably shouldn't include user records
    - App can expand in purpose and records

Checking Auth on Service Call

- **GET** `/api/v1/cats`
    - Requires the cookie be set
    - ...with a value the server knows is valid
    - Returns a 401 value if cookie not set
    - Returns a 403 value if cookie is bad value
    - Other endpoints also make these checks
- No redirects/forms on response
    - Service call is not navigation

Other ways of authorizing service calls

We use a cookie with a session id in this course

- Could have a JWT in cookie
- Could have a token in a non-cookie header
    - Auth header is standard option

All forms of "bearer token"

- Trusted value (secret)
- Sent on every request because web stateless
    - May not be using your intended order of calls
- Minimize sending passwords

What is CORS?

Cross-Origin Resource Sharing (CORS)

CORS is a **browser behavior**

- Based on **headers from server**
- Allowing JS-based service calls
- To endpoints on different origin than current page
    - Origin = protocol + domain + port

This is done for security reasons

- Because Browsers are security nightmares

Without any policy: Wild West

Why CORS?

Consider life without a security policy:

Browser JS can send service call to any origin

- Security problems, particularly with cookies
- Ex: My cat site calls services on your bank site
    - Would have your bank cookies, but is my JS
    - Service calls with your bank session id/JWT
        - But doing what my site JS said to do

Browser represents huge security risk

Same Origin Policy

Same Origin Policy (SOP)

- Pages can only load resources from same "origin"
    - Origin = (protocol + domain + port)
- Except for images, JS, and CSS files
    - Don't break the existing web
    - Pages can load cross-origin images/CSS/JS

Same Origin Policy not enough

SOP more Secure, but people WANTED Cross-Origin

- Including their own subdomains
    - **http://example.com**, **http://api.example.com**
- "Software as a Service" (SaaS)
    - Sell use of remote service
    - Cross origin

Adopting CORS

Cross-Origin Resource Sharing (CORS)

- Response headers say what the service allows
    - Methods, Headers, Allowed Origins, etc
- Browser refuses to give data to JS if not allowed
    - Even if Browser GOT data
- ENFORCED BY BROWSER
    - No browser, no CORS enforcement
    - Full security requires server-side enforcement

CORS Preflight

Non-"simple" requests send a "preflight" request

- Not GET/POST is non-simple
- Sending most custom headers is non-simple
- Sending auth headers (like cookies) is non-simple

Preflight:

- When non-simple request is made by JS
- Browser sends OPTIONS (http method) request
    - This is the "preflight" request
- Checks response before making real request
- Response headers not give okay = no real request

Triggering CORS

Simply load a page, then run some JS that makes a `fetch()` call to a different origin.

```
$ serve public/
```

In browser `Devtools > Console`:

```
fetch('http://example.com/api/');
```

What are the origins of:

- The loaded page?
- The request url in the fetch?

Misleading CORS message

Access to fetch at '**http://example.com/api/**' from origin '**http://127.0.0.1:9000**' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

I hate this message.

- `no-cors` is not what you want
    - You will NOT see the response ("opaque")
- Error is because response lacked CORS headers
- Fix is: Server adds needed headers to response
- Can't turn CORS off just by asking
    - That would be awful security

CORS workarounds

Don't try to "get around" CORS when it blocks you

- CORS is security
- Any "workaround" will be fixed

Options:

- (best) Have the server side send CORS headers
- (okay) Have a backend proxy
    - Write/Find a service you CAN call
    - It makes the cross-origin request
    - It gives you the data

Easy CORS practice

- Set up a server running a service on one port
- Call that service from a page on a different port

Test different combinations:

- Simple calls vs non-simple calls
  - See OPTIONS preflight call in the Network tab
- Add response header:
  - `Access-Control-Allow-Origin: *`
  - See CORS error vs non-error
- Add OPTIONS endpoint with CORS headers
  - Handles preflight

CORS Headers

This course won't dive further into CORS

- Goal: You CAN understand instructions for CORS
- Not: You KNOW instructions for CORS

Multiple headers could be involved. Common ones:

- `Access-Control-Allow-Origin`
    - Note: `*` not allowed if cookies/auth headers
- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Headers`
- `Acesss-Control-Allow-Credentials`

OPTIONS method makes no changes to resource

Common CORS issues

Issue 1: No `access-control-allow-origin` header

- Fix: Add header to allow origin `*` (or see Issue 2)

Issue 2: origin `*` is allowed, but still errors

- Why: Auth headers aren't allowed with origin `*`
- Fix: get origin from req, allow that origin in res

Issue 3: CORS set up, but get CORS error

- Why: Was response 200? CORS headers on errors?
- Fix 1: CORS error is distraction, fix actual error
- Fix 2: Add CORS headers on error responses

CORS Takeaways

- CORS is enforced by the browser
- CORS exists for good security reasons
- "Fix/workaround" is to follow the protocol
    - You can't "turn CORS off"
- CORS error messages can be misleading
    - Make sure you know the problem
- Backend folks often don't know CORS
    - Because browser-side only
    - Service will work for them
        - Using non-browser tests