

# **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`
  - `DELETE /session/:id?`

# **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 avoid as it is an exception
    - 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
  - User needs to correct input
- 401 - Unauthorized (Unauthenticated)
  - User needs to authenticate (log in)
- 403 - Forbidden
  - User is logged in, but isn't allowed
- 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 client can do (except 501)

# What to send in error body?

- Send enough detail on 4xx for user to correct error
- 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
  - 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 point of results?
- End point of results?

Does service accept (and how?)

- Start point for results?
- 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")

"session" is a resource

- We create, get, or delete "session"
- Arguably DELETE could use an id
  - I didn't because 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

# 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 token in a request header
  - Auth header is standard option
  - Could be a JWT
- Could be a parameter sent
  - Old school server-side only option

All forms of "bearer token"

- Trusted value (secret)
- Sent on every request because web stateless
- Minimize sending passwords

# What is CORS?

Cross-Origin Resource Sharing

CORS is a browser behavior

- based on headers from server
- allowing JS-based service calls
- to endpoints that are on a different domain/port
- than the currently loaded page

This is done for security reasons



# Before CORS: Wild West

Why CORS?

Consider life *before* CORS:

1st try: browser JS can do anything, anywhere

- Security problems, particularly with cookies
- Ex: my cat site calls services on your bank site
  - Would have your bank cookies, but is my JS

Browser represents huge security risk

# Same Origin Policy

2nd try: 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

# Same Origin Policy not enough

SOP Secure, but people WANTED Cross-Origin

- Including their own subdomains
  - **http://example.com, http://api.example.com**
- Workarounds included JSONP
  - Hides service call as a JS file to load and run
  - Which is NOT secure
    - Remote service runs JS on your page
    - Remote service may not be yours!

# Adopting CORS

3rd Try: CORS (Cross-Origin Resource Sharing)

- Response headers say what the service allows
  - Methods, Headers, Allowed Origins, etc
- Browser refuses to give data to JS if not allowed
- 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 custom headers is non-simple
- Sending auth headers (like cookies) is non-simple

Preflight:

- An OPTIONS (http method) request
  - checks response headers before real request
- Browser auto-sends and checks
  - bad check = no real request made

# 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 to add headers to response
- Can't turn security off just by asking
  - That would be bad security

# What about CORB?

CORB is related browser-enforced security block

Blocks a resource if it appears to be the wrong kind

- Try to load a CSS file that doesn't exist
- Express returns a 404 with `text/html` content-type
- Browser refuses to show 404 content because `text/html` isn't CSS

Fix: Show appropriate content-type

- Or make sure file exists



# 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 `Access-control-allow-origin` header
  - See CORS error vs non-error

# 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
- 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