Obtaining data for our application

We will use fetch to obtain data asynchronously

Wasson, Microsoft
Client (e.g., browser) → Internet → Site

HTTP & URI

HTML, JSON, ...

3-tier Architecture

Presentation Tier

Web Server (e.g., Apache, NGinx)

Logic Tier

App. Server (e.g., NodeJS)

Persistence Tier

Database (e.g., SQLite, PostgreSQL)

Routing & Controllers (e.g., Express)

Models (e.g., knex, objection)

Client-Server

MVC
Interlude: Modern architectures can be more distributed

Client (e.g., browser) —> CDN + Edge functions (e.g., Next api/)

Service (e.g., DB as a service)

Service (e.g., socket as a service)

Next.js pages/api directory and getServerSideProps()
HTTP (and URLs)

HTTP request includes: a method, URI, protocol version and headers

GET  http://srch.com:80/main/search?q=cloud&lang=en#top

POST  http://localhost:3000/movies/3

HTTP response includes Protocol version and status code, headers, and body

2** OK
3** Resource moved
4** Forbidden
5** Error
## HTTP methods (verbs)

<table>
<thead>
<tr>
<th>Method</th>
<th>Typical Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Request a resource. Form fields can be sent as the query parameters.</td>
</tr>
<tr>
<td>HEAD</td>
<td>Similar to GET, but for just the response headers</td>
</tr>
<tr>
<td>POST</td>
<td>Send data to the server. Unlike GET, the data is transmitted in the request body. Action is up to server, but often creates a subordinate resource. The response may be a new resource, or just a status code.</td>
</tr>
<tr>
<td>PUT</td>
<td>Similar to POST, expect that PUT is intended to create or modify the resource at the specified URL, while POST creates or updates a subordinate resource.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete the specified resource</td>
</tr>
<tr>
<td>PATCH</td>
<td>Partial replacement of a resource, as opposed to PUT which specifies complete replacement.</td>
</tr>
</tbody>
</table>
REST (Representational State Transfer)

• An architectural style (rather than a standard)
  1. API expressed as actions on specific resources
  2. Use HTTP verbs as actions (in line with meaning in spec.)
  3. Responses can include hyperlinks to discover additional RESTful resources (HATEOAS)

• A RESTful API uses this approach (more formally, observes 6 constraints in R. Fielding’s 2000 thesis)

• “a post hoc [after the fact] description of the features that made the Web successful”*

*Rosenberg and Mateos, “The Cloud at Your Service” 2010*
# Film Explorer API

<table>
<thead>
<tr>
<th>Route</th>
<th>Controller Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /api/movies</td>
<td>List (read) all movies</td>
</tr>
<tr>
<td>GET /api/movies/:id</td>
<td>Read data from movie with id == :id</td>
</tr>
<tr>
<td>PUT /api/movies/:id</td>
<td>Update movie with id == :id from request data</td>
</tr>
</tbody>
</table>

$ curl http://domain/api/films/340382

{"id":340382, "overview":"The movie follows the story started in the first Attack on Titan live-action movie.", "release_date":"2015-09-19", "poster_path":"/aCIG1tjNHbLP2Gn1aW33SXC95Si.jpg", "title":"Attack on Titan: End of the World", "vote_average":4.2, "rating":5, "genres":[{"id":18,"movieId":340382},{"id":14,"movieId":340382},{"id":28,"movieId":340382},{"id":878,"movieId":340382}], "genre_ids":[18,14,28,878]}
# CRUD(L) on a RESTful resource

A “route” maps <HTTP method, URL> to a controller action.

<table>
<thead>
<tr>
<th>Route</th>
<th>Controller Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /api/films</td>
<td>Create new movie from request data</td>
</tr>
<tr>
<td>GET /api/films/:id</td>
<td>Read data of movie with id == :id</td>
</tr>
<tr>
<td>PUT /api/films/:id</td>
<td>Update movie with id == :id from request data</td>
</tr>
<tr>
<td>DELETE /api/films/:id</td>
<td>Delete movie with id == :id</td>
</tr>
<tr>
<td>GET /api/films</td>
<td>List (read) all movies</td>
</tr>
</tbody>
</table>
Other features of REST APIs

• Resources can be nested
  GET /courses/3971/assignments/43746
  Assignment 0 in CS101 S19 on Canvas

• Think broadly about what is a resource
  GET /movies/search?q=Jurassic
  Resource is a “search result list” matching query
  GET /movies/34082/edit
  Resource is a form for updating movie 34082 (form submit launches POST/PUT request)
Managing statelessness: Cookies

• Observation: *HTTP is stateless*

• Early Web (pre-1994) didn’t have a good way to guide a user “through” a flow of pages…
  • IP addresses are shared
  • Query parameters hard to cache, makes URLs private information

• Quickly superseded by *cookies*
  Set by server, sent by *browser* on every request
  Since client-side, must be tamper evident

  *Can’t ever trust the client!*
Statefulness in an API

- Different approaches needed for statefulness with an API
  - Client may not be a browser, or
  - Cookies may not be applicable, e.g., 3rd party API
- Instead use some form of token (API key)
  - May (not) be a secret
  - Secret keys aren’t sent to client or committed in VCS
  - Cookie-like workflows exist for authn in SPA apps

authn: Authentication
authz: Authorization
**useEffect hook for operations with side effects**

Function should return undefined or “clean up” callback

Hook changes UI by calling setter in the callback

```
useEffect(() => {
  // Execute code with side effects, e.g., fetching data
}, []);
```

Invoke effect when these variables change (no argument runs the hook on every render, the empty array runs hook only when component first mounts)
Interlude: Rendering a view while waiting for the effect

We can now have renders where the data, e.g., films, is undefined. Our view must handle both situations.

```javascript
let filmContents = (<h2>Loading...</h2>);
if (films) {
    filmContents = (<FilmTableContainer films={films} ... />);
}
```

Use conditional rendering
Recall: `fetch` returns a Promise

A common action is setting state

The “next” promise will be fulfilled with the result of the then handler
```javascript
useEffect(() => {
  fetch('/api/films/').then((response) => {
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    return response.json();
  }).then((data) => {
    setFilms(data);
  }).catch(err => console.log(err));
}, []);
```

- **Response object with status, headers, and response body**
- **Parse and return response as JSON**
- **Invoke setter to update UI**
const prom1 = fetch('/api/films/');
const prom2 = prom1.then((response) => {
    return response.json();
});

prom2.then((data) => {
    setFilms(data);
})
// Do something after
Recall: REST is not just for servers

src/pages/
  _app.js
  articles/
    [[...id]].js // http://domain/articles/[42]
    [id]/
      edit.js  // http://domain/articles/42/edit
  edit.js    // http://domain/edit

https://nextjs.org/docs/routing/dynamic-routes