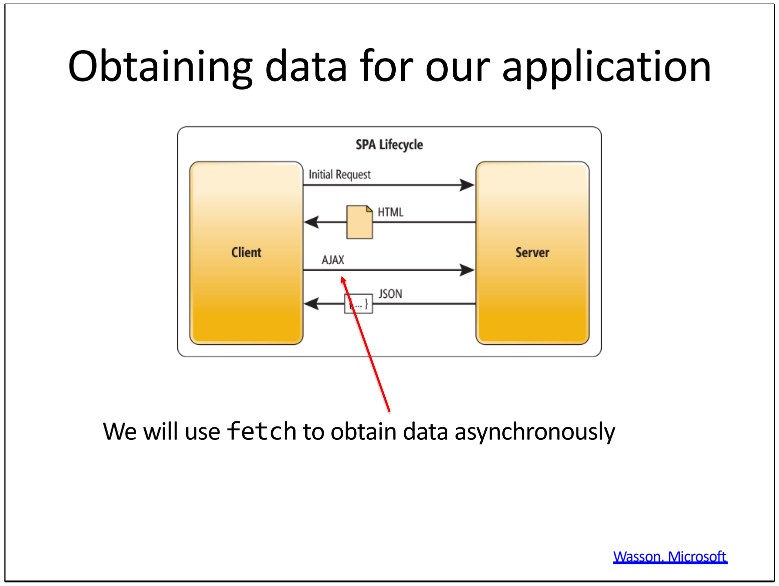


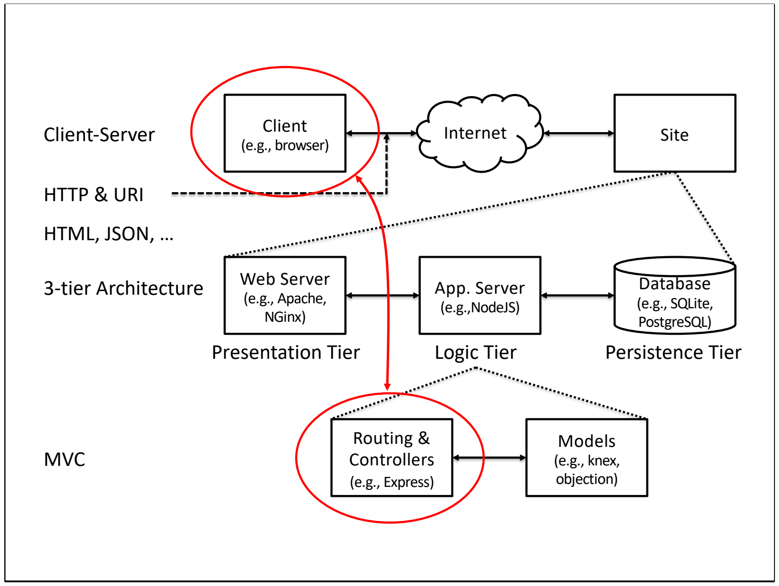
Obtaining data for our application

We will use `fetch` to obtain data asynchronously

Wasson, Microsoft

In our applications so far, such as Simpledia that you are working on right now, the data is “built into” the application and we ”load” it by import the JSON into the application. Why is that not desirable approach going forward?

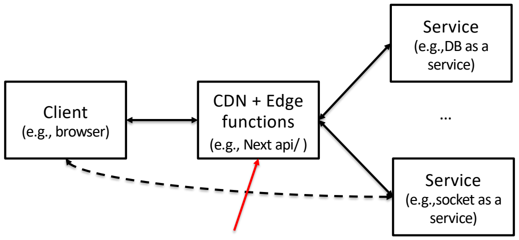
That data never changes! More typical is to fetch the data as needed from a server with an AJAX request and persist new or changed data by sending it back to the server (also via AJAX). AJAX is a technique (with multiple underlying implementations) to request data from a remote resource in the background without reloading the webpage.



Let’s break down the architecture of client server interactions. Our client, the browser, is connecting to a remote server using the HTTP protocol. A “design pattern” for the backend (server) of our site is the 3-tier architecture, where HTTP communication terminates at the web server which manages the connection itself and sends requests onto an application server, which runs the logic for our site. There is additional a persistence tier where we store the data. Our site logic is often composed of routing/controller later that specifies and implemented the interface of our application. The interface to the persistence tier is managed by the models.

[click] Our focus today is the communication between the client, that is the browser and the server, and specifically between the browser and the interface specified by the server routers/controllers. This enables us to get new/updated data (that is fetch data from the server to the client) and persist changes by sending data from the client to the server. Today are working from the client-side perspective, that is we will work with servers that already exist. In the future we will build our own servers.

Interlude: Modern architectures can be more distributed



Client
(e.g., browser)

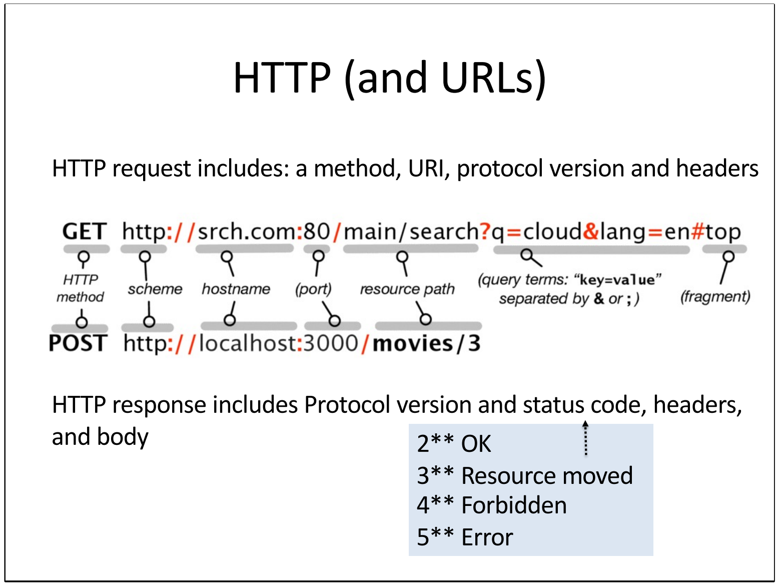
CDN + Edge functions
(e.g., Next api/)

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

HTTP method
scheme
hostname
(port)
resource path
(query terms: "**key=value**" separated by **&** or **;**)
(fragment)

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

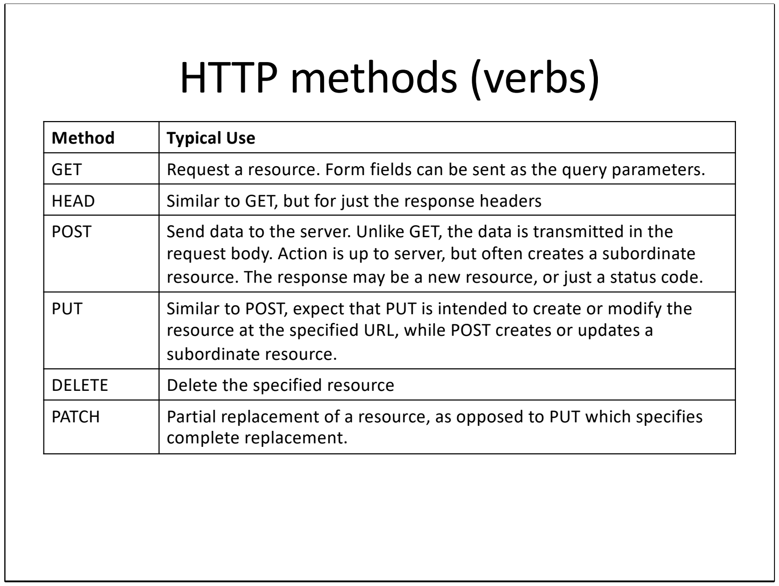
HTTP response includes Protocol version and status code, headers, and body

2** OK
3** Resource moved
4** Forbidden
5** Error

HTTP is a request-response protocol implemented on top of TCP/IP. The hostname is translated to the IP address (via DNS or other mechanism). The optional port specifies with TCP port to use. TCP ports enable multiple applications on the same node to use TCP/IP concurrently and independently. Optional because many protocols have specified "well-known" ports (e.g., 22 for SSH, 80 for HTTP) that will be used by default if the port is not specified (which is why we typically don't see URLs like the first example). I want to highlight the vocabulary for the different parts of the request, including the method (sometimes called the HTTP verb), the resource path and the query parameters (the latter is the part after a question mark, is a set of key-value relationships).

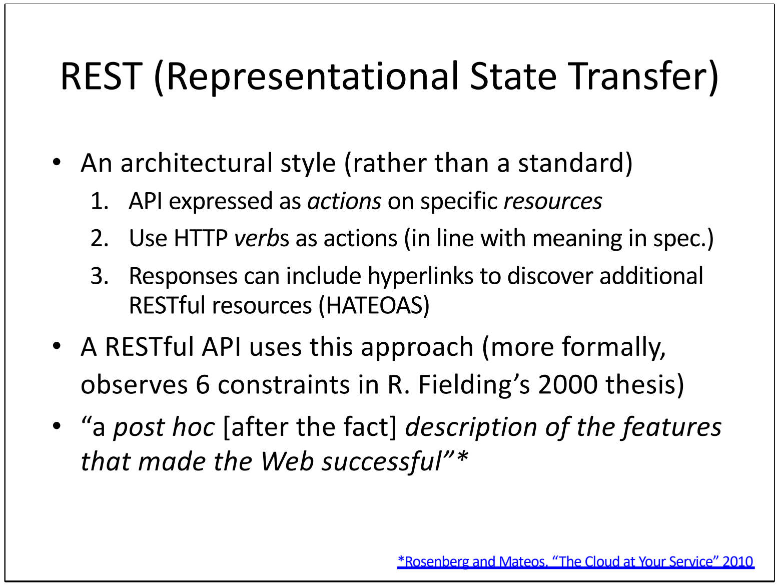
Vocabulary: URI is a superset of URLs (URLs are URI that describe both a protocol, e.g., http, and a resource)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



HTTP methods (verbs)

Method	**Typical Use**
GET	Request a resource. Form fields can be sent as the query parameters.
HEAD	Similar to GET, but for just the response headers
POST	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.
PUT	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.
DELETE	Delete the specified resource
PATCH	Partial replacement of a resource, as opposed to PUT which specifies complete replacement.



REST (Representational State Transfer)

- An architectural style (rather than a standard)
  1. API expressed as *actions* on specific *resources*
  2. Use HTTP *verb*s 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

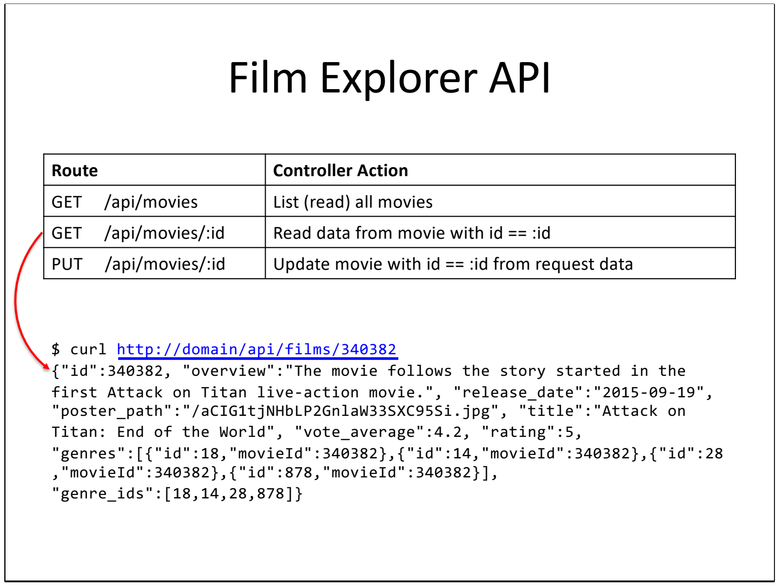
We aim to have the URI just be nouns, and the verbs provided by HTTP methods. That is our understanding of the resources in our application will drive the design of the API.

We won’t get much deeper in our current understanding of REST.

HATEOAS – Hypertext As The Engine Of Application State

An example of non-REST API would be a single endpoint that accepted multiple different input messages that were effectively remote procedure calls.

https://martinfowler.com/articles/richardsonMaturityModel.html
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

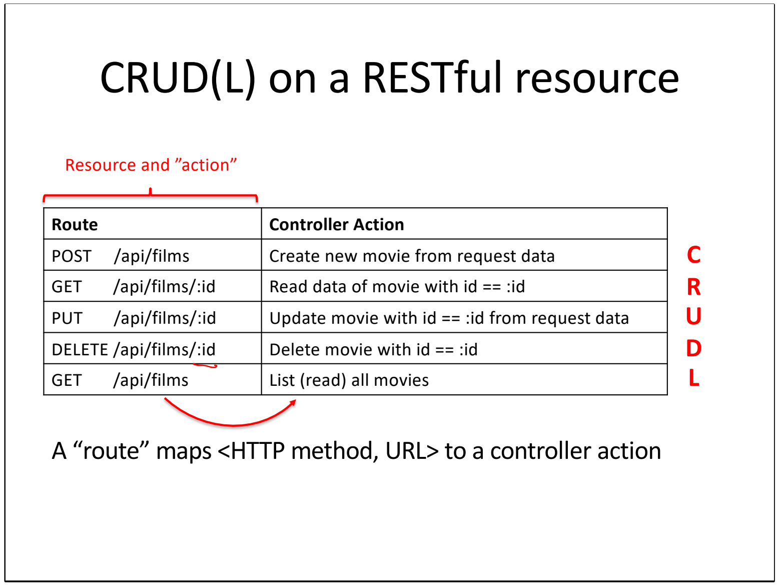


Film Explorer API

Route	Controller Action
GET /api/movies	List (read) all movies
GET /api/movies/:id	Read data from movie with id == :id
PUT /api/movies/:id	Update movie with id == :id from request data

```
$ 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":"/aCIG1tjNHbLP2GnlaW33SXC95Si.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]}
```

What is the key resource in the film explorer? Films. We see that resource reflected in the API. We observe the URLs are nouns and the HTTP methods specify the verbs on those nouns (resources).



CRUD(L) on a RESTful resource

Resource and ”action”

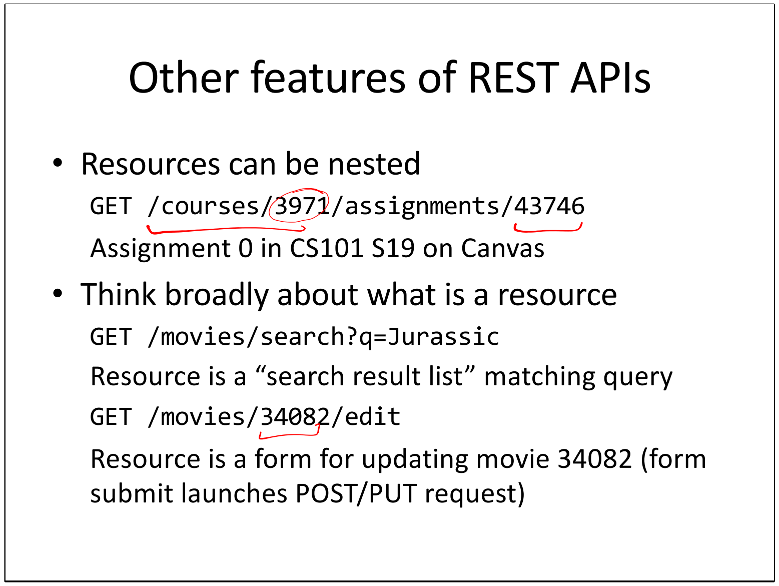
Route	Controller Action	
POST /api/films	Create new movie from request data	C
GET /api/films/:id	Read data of movie with id == :id	R
PUT /api/films/:id	Update movie with id == :id from request data	U
DELETE /api/films/:id	Delete movie with id == :id	D
GET /api/films	List (read) all movies	L

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

[click] We will use the term route to describe the mapping between the <HTTP method, URL> (i.e., the action and resource) to specific controller behavior.

[click] CRUD(L) is a shorthand for the common operations in a RESTful API shown here. A resource that provides those operations in this style is often called a RESTful resource. The description “CRUD app” is describing an application focused on implementing these operations for a set of resources. It is often used pejoratively to imply an application is trivial, but in practice building and deploying an application in the real-world is hardly trivial!

Note the colons. :id is a common notation for indicating a variable named id extracted from the URL. It is derived from the way URLs are specified in server libraries, i.e., how we specify that /api/films/:id should match /api/films/1, /api/films/2, …



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

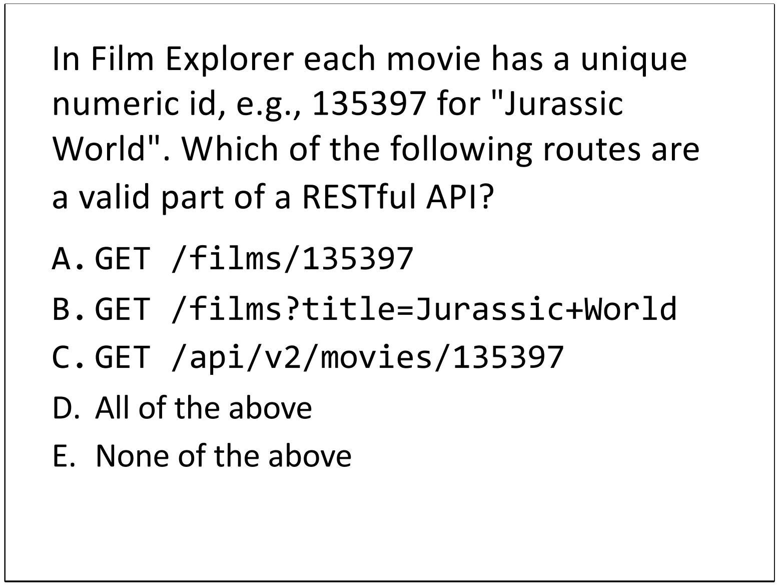
  ```
  GET /movies/search?q=Jurassic
  ```

  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)

For last. APIs intended for traditional web applications will likely have additional routes to obtain UI (e.g., editing form).

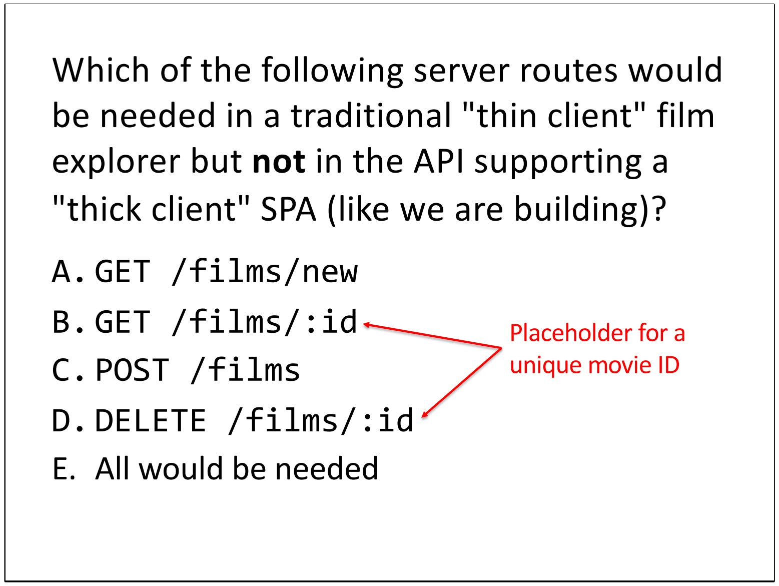


In Film Explorer each movie has a unique numeric id, e.g., 135397 for "Jurassic World". Which of the following routes are a valid part of a RESTful API?

A. `GET /films/135397`
B. `GET /films?title=Jurassic+World`
C. `GET /api/v2/movies/135397`
D. All of the above
E. None of the above

Answer: D

All the above describe resources and a corresponding action. The difference is A & C are a specific movie while B is presumably all the movies whose title matches the filter in the query parameters (which could be 0 or more).



Which of the following server routes would be needed in a traditional "thin client" film explorer but **not** in the API supporting a "thick client" SPA (like we are building)?

A. `GET /films/new`

B. `GET /films/:id`

C. `POST /films`

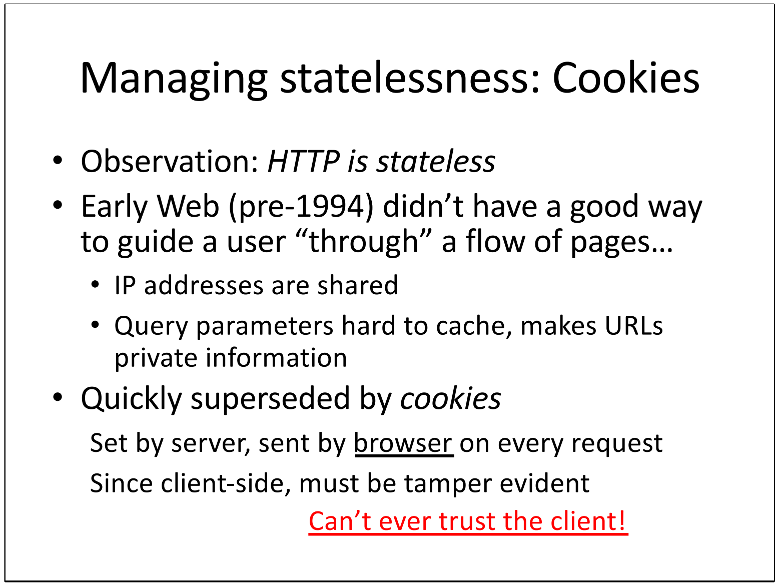
D. `DELETE /films/:id`

E. All would be needed

Placeholder for a unique movie ID

Answer: A

GET /films/new would typically be needed to return the form that a user would fill in to create a new movie. That information would be sent to the server as a POST request. In a "thick client", the form is built into the client (not fetched from the server). For the latter applications, we only need the server API to support operations on data, not provide UI.



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

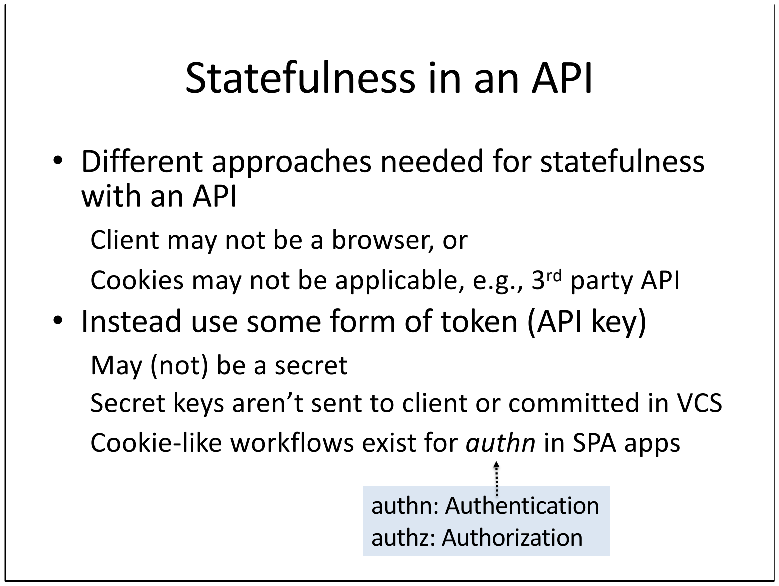
  Can't ever trust the client!

What do we mean by stateless? Each request is treated independently. The advantages? No need to maintain client's previous interactions, and thus different servers can handle different requests. But the reality we is often need statefulness. A common use case is authentication (i.e., you authenticate and then are allowed to do additional authenticated actions), but there are many more (preferences, tracking, etc.).

We can't have the server record IPs (since those are shared), and if we try to embed the information into the URL, e.g., as a query parameters, we will break caching and potentially start exposing private information in the URL itself. These problems motivated the development of cookies.

Cookies are originally sent by the sever and the sent back by the client with every request (done transparently by the browser). Allows the server to introduce associate state with a particular request (link it to other requests by the same user).

But since the cookie is sent to the client it is under their control. We must make sure we can know if they tampered with the cookie (e.g., to pretend to be someone they are not). This is the first of many reminders of one of our key security mantras. We can't trust the client.



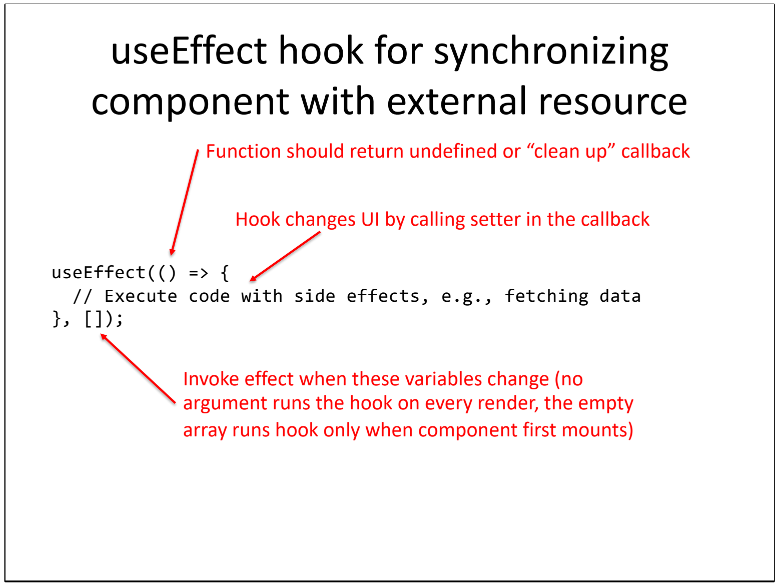
Statefulness in an API

- 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

Fetch, for example, doesn't send cookies by default. Instead, we will often use various forms of cryptographic tokens. Some examples:

Not secret keys: Some uses of Google Maps API key. Instead, you restrict that key to be just used from your domain.
Secret keys: Keys need access Facebook API and other services on behalf of user. These are intended to remain on the server and are used when that server forwards request s to Facebook on behalf of that user. Since these keys are secret, they shouldn't be sent to the client or committed to your repository (we will learn about this more in relevant practical exercises).



useEffect hook for synchronizing component with external resource

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)

useEffect is one of trickier and more controversial hooks. As described in the documentation, it is an “escape hatch from the React paradigm” that used to ”synchronize your components with an external resource”. Often that external resource is an API, i.e., you want to synchronize your components with data obtained via the network from external API.

[click] 3X

What don’t you need useEffect for: Transforming data during rendering or responding to user events. Both are better handled within the React paradigm (e.g., within the component function or event handlers).

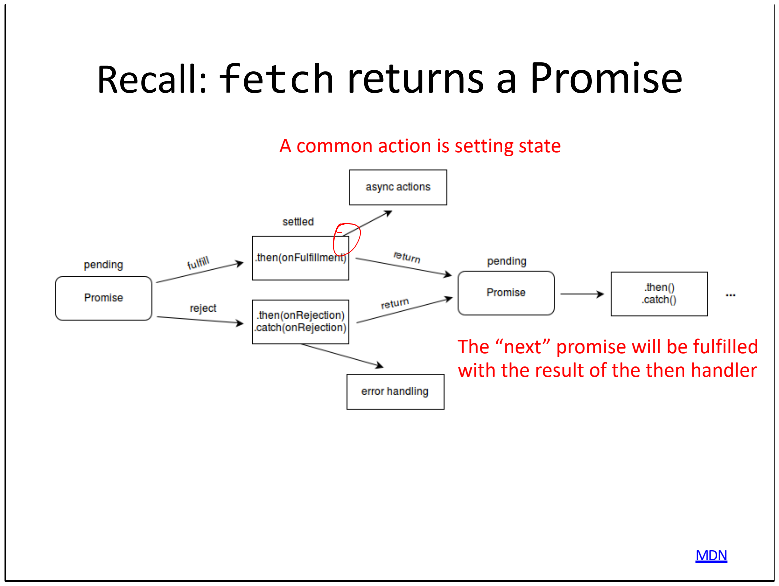
https://react.dev/learn/you-might-not-need-an-effect

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.

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

Use conditional rendering



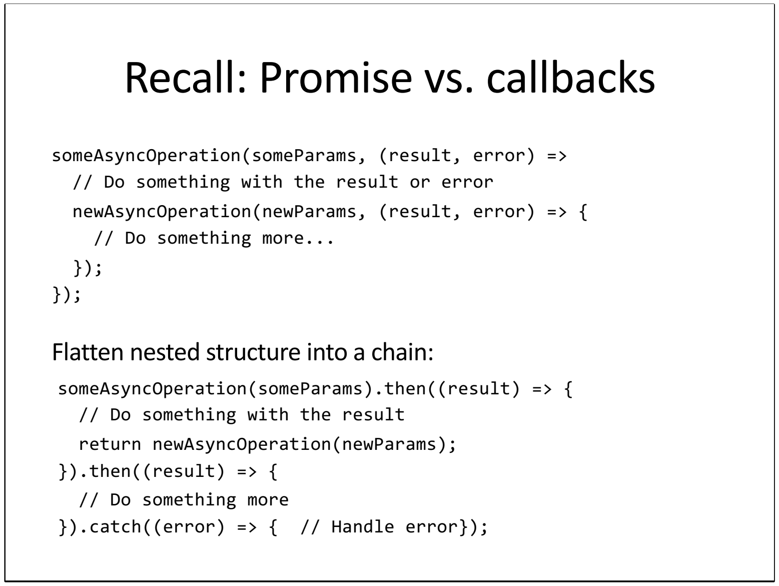
Recall: `fetch` returns a Promise

A common action is setting state

The “next” promise will be fulfilled with the result of the then handler

MDN

We will use fetch to obtain data from external resource, that is use fetch in the body of the useEffect hook. Fetch is asynchronous and returns a promise that resolves when the data is available. We will launch the request in the hook and update state when the value becomes available, i.e., in a then callback applied to the Promise.



Recall: Promise vs. callbacks

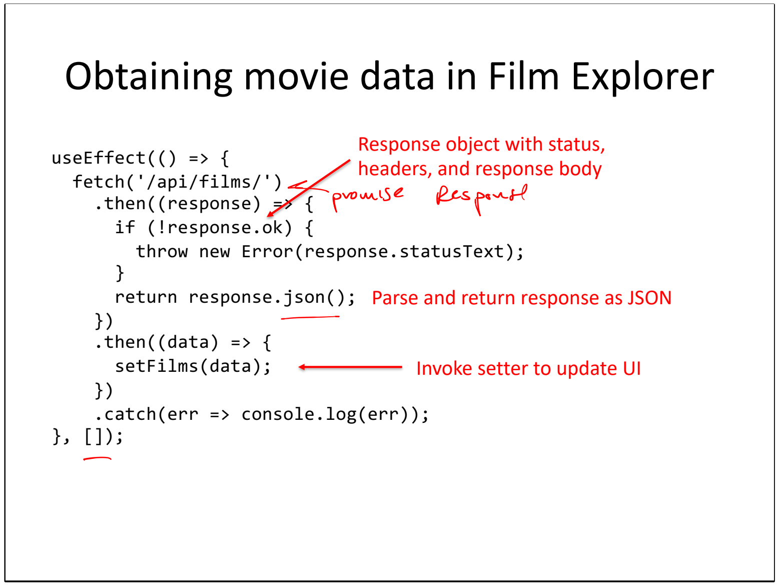
```
someAsyncOperation(someParams, (result, error) =>
  // Do something with the result or error
  newAsyncOperation(newParams, (result, error) => {
    // Do something more...
  });
});
```

Flatten nested structure into a chain:

```
someAsyncOperation(someParams).then((result) => {
  // Do something with the result
  return newAsyncOperation(newParams);
}).then((result) => {
  // Do something more
}).catch((error) => {  // Handle error});
```

One of the key advantages of Promises is flattening a deeply nested set of callbacks into a linear chain of promises. In our example here the first then (invoked on the Promise returns by someAsyncOperation) returns a Promise. That promise is eventually replaced by the Promise created by `newAsyncOperation in its handler.`

If instead of executing steps in sequence, you want to execute a set of synchronous operations in parallel, use:
Promise.all: If you care when they are all fulfilled
Promise.race: If you just care when the first Promise fulfills/rejects



Obtaining movie data in Film Explorer

```
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

promise Response

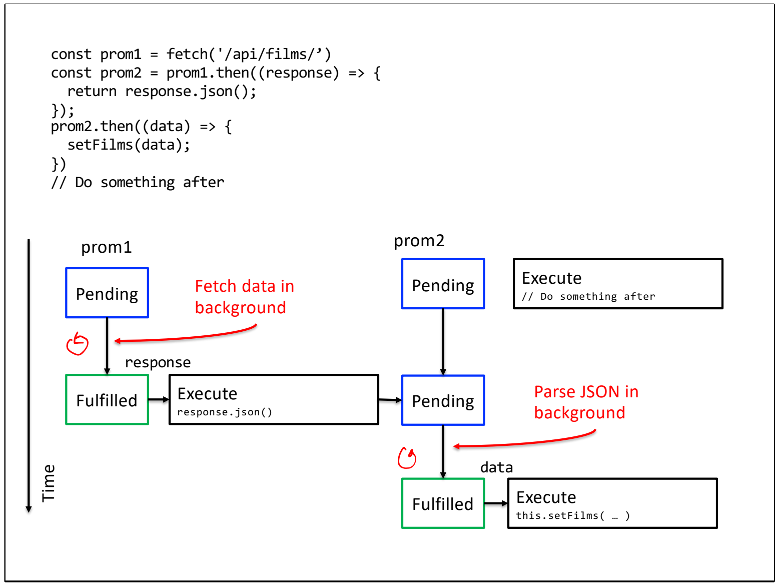
Parse and return response as JSON

Invoke setter to update UI

Here we see that linear structure applied to obtaining data from an API and updating the application state. `fetch` returns a promise that will eventually resolve in the response object with the status, body (data), etc. The response is processed by the then callback.

Why do we need the second then handler, why can't we do: setFilms(response.json()); ? If we checkout the documentation for response.json() we see it returns a Promise. That is JSON parsing is an asynchronous operation.

Note that here are only logging the errors. In practice we would want to provide more meaningful feedback to the user when something failed.



```
const prom1 = fetch('/api/films/’)
const prom2 = prom1.then((response) => {
  return response.json();
});
prom2.then((data) => {
  setFilms(data);
})
// Do something after
```

prom1

Pending

Fetch data in background

response

Fulfilled

Execute
response.json()

prom2

Pending

Execute
// Do something after

Pending

Parse JSON in background

data

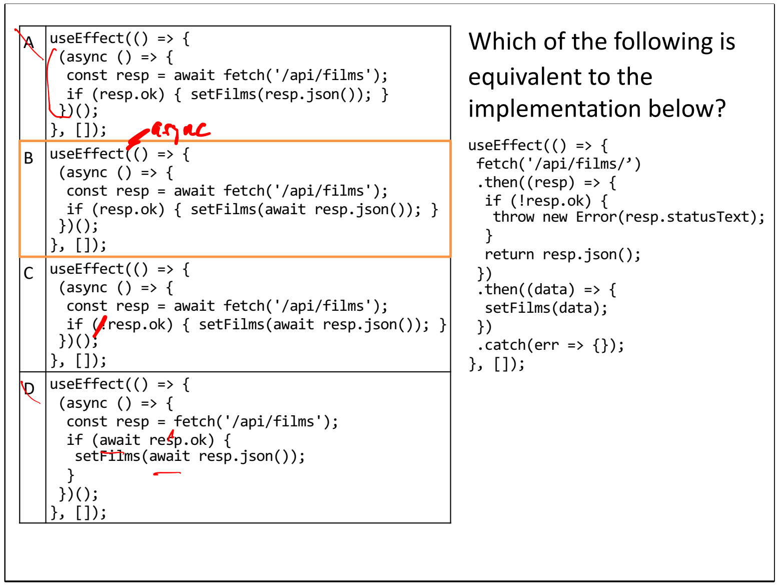
Fulfilled

Execute
this.setFilms(…)

Time

This Promise chain is a common source of confusion. Let’s rewrite in into more discrete steps.

- prom1, prom2 are effectively defined immediately, that is fetch and the `then` method return immediately with promises that will be resolved in the future.
- Thus, before the network request has completed, we start executing “Do something after”
- In the meantime, the browser is performing the network request. When the request is completed, the promise resolves with the response object and we invoke the first then callback. It immediately returns a promise that will eventually resolve with the parsed JSON. That newly returned promise subsumes the original prom2.
- When that second promise resolves we perform the state update.



A

```
useEffect(() => {
 (async () => {
  const resp = await fetch('/api/films');
  if (resp.ok) { setFilms(resp.json()); }
 })();
}, []);
```

async

B

```
useEffect(() => {
 (async () => {
  const resp = await fetch('/api/films');
  if (resp.ok) { setFilms(await resp.json()); }
 })();
}, []);
```

C

```
useEffect(() => {
 (async () => {
  const resp = await fetch('/api/films');
  if (!resp.ok) { setFilms(await resp.json()); }
 })();
}, []);
```

D

```
useEffect(() => {
 (async () => {
  const resp = fetch('/api/films');
  if (await resp.ok) {
   setFilms(await resp.json());
  }
 })();
}, []);
```

Which of the following is equivalent to the implementation below?

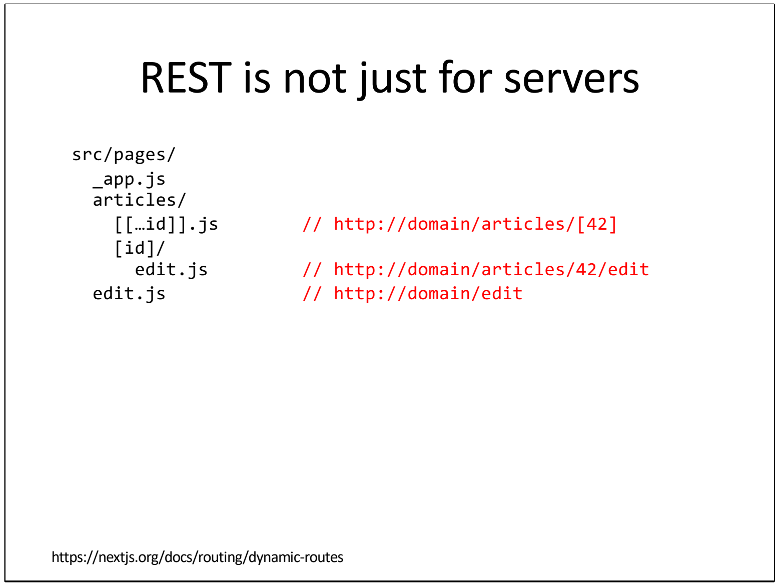
```
useEffect(() => {
 fetch('/api/films/’)
 .then((resp) => {
  if (!resp.ok) {
   throw new Error(resp.statusText);
  }
  return resp.json();
 })
 .then((data) => {
  setFilms(data);
 })
 .catch(err => {});
}, []);
```

What is the funky-ness with the immediately evaluated function? useEffect is expecting a function that either returns nothing or function that ”cleans up” any side effects (e.g., disconnects from a chat server). But an async function returns a Promise and so can’t be used directly as the function argument to useEffect. Instead, we need to create the async function inside of useEffect.

Answer: B

Answer A is missing an await for the JSON parsing, answer C has the incorrect login, we only want to call the setter if the response is “ok”, and answer D has the first await in the wrong place (is a Promise, not resp.ok).

More generally, we get the sense there can be a bit a boilerplate involved with fetch. On approach to mitigate that is to use additional libraries, e.g., axios, that provide some of the functionality already. Another is to encapsulate the common code in a custom hook.



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

Recall from the slides about Next that we utilize Next's dynamic routing features to manage which view are showing, that is we use the URL to maintain a portion of application state, specifically which article we are showing. The design of those URLs is intentional. What is the resource in Simplepedia? An article. We express resource with our URL structure:

What is the correspondence between the pages and CRUDL operations we saw earlier?

articles/[[...id]].js : Read (a single article) or list all articles (within sections)
articles/[id]/edit.js : Get form for Updating an article
/edit.js : Get form for Creating a new article

Our next assignment will be extending those pages to use an external API for retrieving data and creating or updating articles (instead of just modifying state).