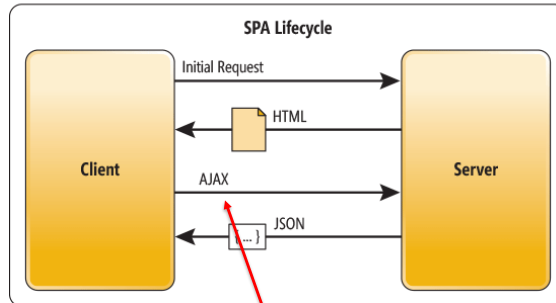


Obtaining data for our application

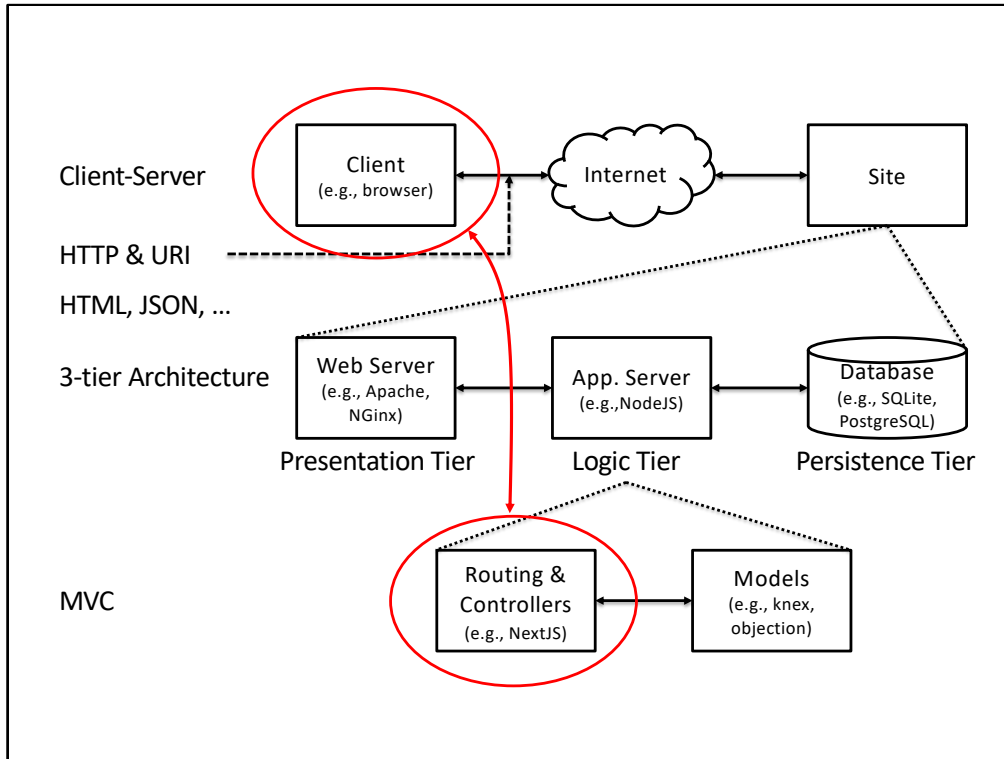


We will use `fetch` to obtain data asynchronously

[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 importing the JSON into the application. Why is that not desirable approach going forward?

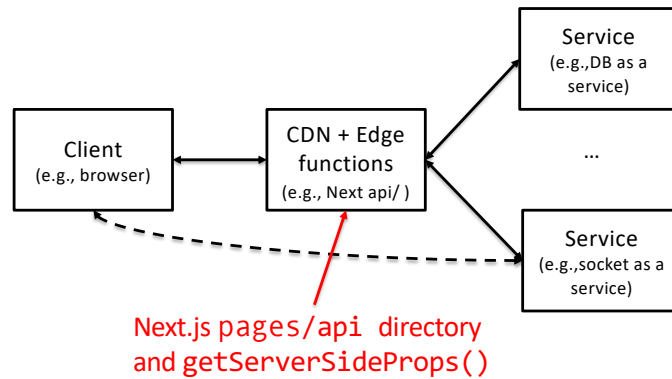
That data (can) never changes! More typical is to fetch the data as needed from a server with an AJAX request and persist new or changed data by sending it back to the server (also via AJAX). AJAX is a technique (with multiple underlying implementations) to request data from a remote resource in the 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 additionally 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 we 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.

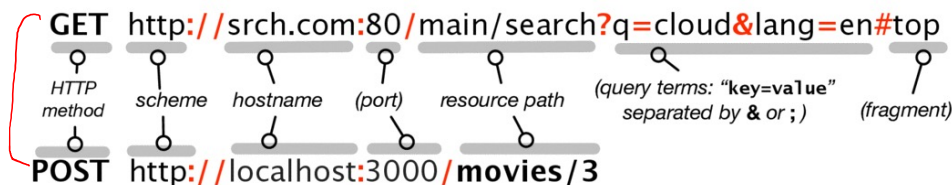
Interlude: Architectures can be more distributed



Next is produced by Vercel. Their business is providing a “serverless” deployment platform, in which there is not a persistent backend, but instead the routes execute as ephemeral “edge” functions and the persistence tier, e.g., the database, is implemented as an external service. We will not use this “serverless” approach, but I wanted to include it here for context and reference.

HTTP (and URLs)

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



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

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

While I suspect we are familiar with some aspects of HTTP and URLs, I want to highlight some of the less familiar parts of a HTTP request and the URL. Each HTTP request includes a method (sometimes called the HTTP verb), the resource path and the optional query parameters (the latter is the part after a question mark, is a set of key-value relationships) and fragment. When we type a URL into browser that automatically triggers a GET request, but when interacting with an API we will use more of the available methods.

Notice these addresses explicitly specify the port. Ports enable multiple applications on the same node to use TCP/IP (the networking protocol underlying HTTP) concurrently and independently. Optional because many protocols have specified "well-known" ports (e.g., 80 for HTTP) that will be used by default if the port is not specified. In the latter example, we are running the development server on a non-standard port and so need to explicitly specify it.

HTTP specifies both the request and possible responses. One of the relevant features of the latter is the response code, ranges of which are associated with success or failure (think "404" error...).

Vocabulary: URI identifies a resource while URLs locate (URLs contain a URI, i.e., the resource, but all the protocol, hostname, etc., where that resource is located).

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).

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.

Each verb has a typical role, e.g., reading or "get"ing a resource, creating a new resource (POST), updating etc. While we don't always have to use the verbs this way, being consistent with this convention will enable us to take best advantage of the available tools/libraries (which assume that pattern) and ensure our code is readily understood by others.

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](#)

Similarly, while there are many patterns for designing APIs (that is combination of HTTP methods and URLs in use), a widely used approach is REST. In a RESTful approach we aim to have the URI just be nouns, i.e., resources in our application, and the verbs provided by HTTP methods. That is our understanding of the resources in our application, e.g., articles in Simplepedia or films in the FilmExplorer, will drive the design of the API. As a contrast, an example of non-REST API would be a single endpoint, i.e., a single URL and verb, that accepted multiple different input messages where the action was determined by the message body.

We won't get much deeper in our current understanding of REST, for our purposes I want to us start thinking in "resources". Keeping these ideas in mind will help understand why/how existing APIs are designed and define and implement our own APIs.

Vocabulary: *HATEOAS – Hypertext As The Engine Of Application State*

<https://martinfowler.com/articles/richardsonMaturityModel.html>

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


Resource(s): A definition

- Fielding: “a document or image, a temporal service (e.g., ‘today's weather in Los Angeles’), a collection of other resources, a non-virtual object (e.g., a person), and so on”
- Many of our resources are the data used in our application, e.g., in Simplepedia an *Article* or *list of Articles*
- A resource can have subordinate resources, e.g., a *course* may have *assignments*

https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Film Explorer API

Route	Controller Action
GET /api/films	List (read) all movies
GET /api/films/:id	Read data from movie with id == :id
PUT /api/films/: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 (an IMDB-like application)? 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). Per convention, `films` describes all the films, while `films/:id` describes a single specific film, e.g., the film with id 34082.

CRUD(L) on a RESTful resource

Resource and "action"

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

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

[click] We will use the term "route" to describe the mapping between the combination of <HTTP method, URL> (i.e., the action and resource) to a 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 1 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 the last. APIs intended for traditional web applications will likely have additional routes to obtain various interfaces (forms), e.g., an “editing” form pre-filled with the existing data for that particular movie.

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/:id/edit

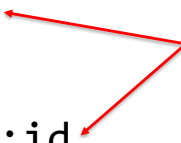
B. GET /films/:id

C. POST /films

D. DELETE /films/:id

E. All would be needed

Placeholder for a
unique movie ID



*SPA: Single Page Application

Answer: A

As we noted before, GET /films/:id/edit would typically be needed to return the form that a user would fill in to edit movie. That information would be sent to the server as a PUT request. In a "thick client", the form is built into the client (not fetched from the server), e.g., as one of our React components. For the latter applications, we only need the server API to support operations on the underlying 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 in practice we often need stateful-ness. 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.

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

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



In the situation where we can't use cookies, we will need another mechanism, typically some form of cryptographic token. As a practical matter that might manifest in additional parameters we provide when making the request so we can send the relevant tokens, etc.

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 needed to 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 more managing secrets in the relevant practical).

Fetching from an API in React with the useEffect hook

```
useEffect(() => {  
  // Execute code with side effects, e.g., fetching data  
  fetch("/api/films")  
  ...  
  .then((data) => {  
    setFilms(data);  
  });  
}, []);
```

Function should return undefined or "clean up" callback

Hook ultimately changes UI by calling state setter in the effect function

Invoke effect when these variables change (no argument runs the hook on every render, the empty array runs hook only when component first mounts)

To actually fetch data in our React applications we will use the `useEffect` hook. `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. That is how we will use it here (although that is not the only role for useEffect). The idea is that some change in our component (prop or state) triggers the effect function. That function obtains whatever data it needs and ultimately calls a state setter. Here we are using the fetch function to obtain data from an external server...

[click] 3X

Recall that only state changes trigger a re-render in React. Thus, if your `useEffect` function is not ultimately changing state (in some way) then you are not changing your application UI (and something has probably gone awry design-wise).

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 (we are waiting for the request). Our view must handle both situations.

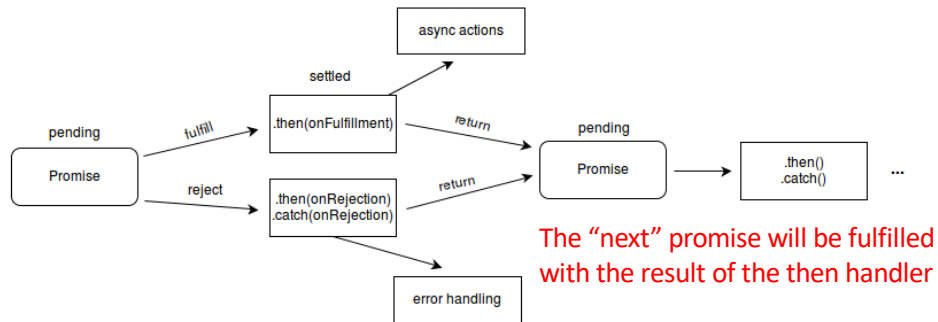
```
let filmContents = (<h2>Loading...</h2>);  
if (films) {  
  filmContents = (<FilmTableContainer films={films} ... />);  
}
```



Use conditional rendering

fetch returns a Promise

A common action is setting state



[MDN](#)

We will use the fetch function to obtain data from external resource. It will be the main operation in body of our useEffect hook.

Fetch is asynchronous (getting data from outside world takes time!) and so returns a promise that resolves when the data is available (like we talked about previously). We will launch the request in the hook and update state when the value becomes available, i.e., in a function passed to the then method.

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});
```

Recall that one of the key advantages of Promises is flattening a deeply nested set of callbacks into a linear chain of promises. This was not actual code but is an actual situation. Fetch creates a nearly identical sequence of asynchronous operations! And as we will hopefully see, it is easier to reason about that sequence in a chain.

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

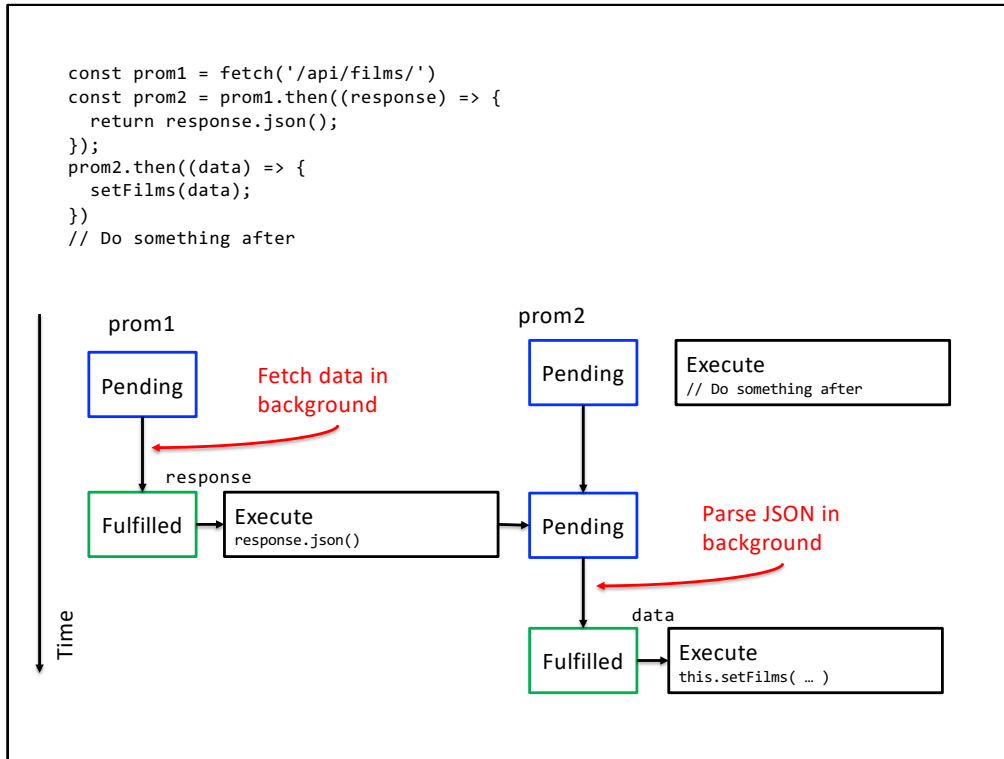
Parse and return response as JSON

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, when it is available, is processed by the function provided to `then`.

Why do we need the second then handler, i.e., why can't we do: `setFilms(response.json());` ? If we check out the documentation for `response.json()` we see it too returns a Promise. That is JSON parsing is itself an asynchronous operation. Here the initial promise return by the first `then` is subsumed by the promise returned from `response.json()`. When it resolves, we update the component state (`setFilms`).

<click> If there is an error anywhere in the chain, the `catch` callback will be executed. Note that here are only logging the errors. In practice we would want to provide more meaningful feedback to the user when something failed.



As we did before, let's transform this chain into a more discrete set of steps: ``prom1`` and ``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	<pre>useEffect(() => { (async () => { const resp = await fetch('/api/films'); if (resp.ok) { setFilms(resp.json()); } })(); }, []);</pre>
B	<pre>useEffect(() => { (async () => { const resp = await fetch('/api/films'); if (resp.ok) { setFilms(await resp.json()); } })(); }, []);</pre>
C	<pre>useEffect(() => { (async () => { const resp = await fetch('/api/films'); if (!resp.ok) { setFilms(await resp.json()); } })(); }, []);</pre>
D	<pre>useEffect(() => { (async () => { const resp = fetch('/api/films'); if (await resp.ok) { setFilms(await resp.json()); } })(); }, []);</pre>

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 logic, 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`. One approach to mitigate that is to use additional libraries 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 (and PA3) 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 design the URL structure within our front-end application around that resource. That is, we are using a RESTful pattern even though there is no server involved. Recall these URLs are managed entirely within the front-end application running on the browser, but we still benefit from using a similar design approach.

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 : Read a form for Updating an article (the form is one resource, the article the other)

/edit.js : Read a form for Creating a new article (the form is one resource, the article the other).

In PA4 will be extending those pages to use an external API for retrieving data and creating or updating articles (instead of just modifying state). The forms will make POST or PUT requests to create or update article resources.