In Simplepedia and the stand alone version of Film Explorer, we let webpack take care of the messy business of bundling in our data. While convenient, this is not a typical way to manage our data. Why not?
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.
We also want to go the other direction and have data persistence.

"As a user, I want to rate the movies myself so that I can record and review my opinion of the movie"

Will need persistence to maintain data over time (through restarts, etc). Will need a server. That interface, from the client’s perspective, is our focus for today.
HTTP (and URLs)

<table>
<thead>
<tr>
<th>HTTP method</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td><a href="http://localhost:3000/movies/3">http://localhost:3000/movies/3</a></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2** OK</td>
<td>OK</td>
</tr>
<tr>
<td>3** Resource moved</td>
<td></td>
</tr>
<tr>
<td>4** Forbidden</td>
<td></td>
</tr>
<tr>
<td>5** Error</td>
<td>Error</td>
</tr>
</tbody>
</table>

HTTP is 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.

HTTP is a request-response protocol

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)

<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>
We aim to have the URI just be nouns, and the verbs provided by HTTP methods.

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**

<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>


```json
{
  "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":"/aC1G1tjNHbLP2Gn1aW33SXC958i.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) are common operations in a RESTful API. A resource that provides those operations in this style is often called a RESTful resource.

:id represents a variable (the movie id)
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) that generate PUT requests, etc. In our SPA model, those forms are built into the client and so the API would only need the routes for modifying the object.
Now we need to worry about the mechanics of communicating with the server from JavaScript.

Why in that method? Partly future proofing (https://github.com/reactjs/reactjs.org/issues/302), partly to ensure DOM access if needed (after mounting) and then prevent un-rendered changes to the views.
Recall that the browser is asynchronous
A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

Then can take both fulfillment and rejection handlers, although typically just used with fulfillment handler.

How can you terminate a chain of promises? Basic ES6 API doesn’t actually have a method for forced synchronization. That reminds that we are really performing asynchronous operations.

An aside, ES8 (2017) includes async/await declaration for creating functions that return Promises. That functionality is actually built on top of Promises. To keep scope under control we will use Promises directly.
Example of how the promise returned by then is replaced by any promise return by the then _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
Why do we need the second then handler, why can’t we do: `this.setState({ movies: response.json() })`?
const prom1 = fetch('/api/movies/');
const prom2 = prom1.then((response) => {
  return response.json();
});
prom2.then((data) => {
  this.setState({ movies: data });
})
// Do something after