Obtaining data for our application

```javascript
import React, { Component } from 'react';

import movieData from './movies.json';
import MovieTableContainer from './components/MovieTableContainer';
import SearchBar from './components/SearchBar';

class FilmExplorer extends Component {
  constructor(props) {
    super(props);
    this.state = {
      searchTerm: '',
      sortType: 'title',
      movies: movieData,
    };
  }
}
```
Obtaining data for our application

We will use `window.fetch` to obtain data asynchronously.
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
### 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://basin.cs.middlebury.edu:5042/api/movies/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]}
**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 /movies</td>
<td>Create new movie from request data</td>
</tr>
<tr>
<td>GET /movies/:id</td>
<td>Read data of movie with id == :id</td>
</tr>
<tr>
<td>PUT /movies/:id</td>
<td>Update movie with id == :id from request data</td>
</tr>
<tr>
<td>DELETE /movies/:id</td>
<td>Delete movie with id == :id</td>
</tr>
<tr>
<td>GET /movies</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)
let movieContents = (<h2>Loading...</h2>);
if (this.state.movies) {
    movieContents = (
        <MovieTableContainer movies={this.state.movies} ... />
    );
}
Recall that the browser is asynchronous.
**fetch** returns a Promise

A common action is **setState**

The “next” promise will be fulfilled with the result of the then handler
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});
Obtaining movie data in Film Explorer

```javascript
fetch('/api/movies/').
  .then((response) => {
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    return response.json();
  })
  .then((data) => {
    this.setState({ movies: data });
  })
  .catch(err => console.log(err));
```

Response object with status, headers, and response body

```
Response object with status, headers, and response body
```

```
Parse and return response as JSON
```

```
fetch('/api/movies/').
  .then((response) => {
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    return response.json();
  })
  .then((data) => {
    this.setState({ movies: data });
  })
  .catch(err => console.log(err));
```
const prom1 = fetch('/api/movies/');
const prom2 = prom1.then((response) => {
    return response.json();
});
prom2.then((data) => {
    this.setState({ movies: data });
})
// Do something after