A simple HTTP server

Node HTTP module

```javascript
const http = require('http');
const server = http.createServer((request, response) => {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.end("Don’t Panic");  // Manually construct the response
}).listen(5042);
console.log('Listening on port %d', server.address().port);
```

In action:

```bash
$ curl http://localhost:5042/
Don’t Panic
```

With this low-level interface we are responsible for everything, interpreting the request and building the entire response. As you expect there is an opportunity for frameworks that implement the common features of a web server. We will use Express (https://expressjs.com), a "minimalist" routing oriented framework. There is a counterpart to Express in most server-side languages (e.g. Sinatra for Ruby and Flask for Python).
Recall that each route in our REST API include a method, e.g. GET, and a URL (possibly parameterized). These reflected in Express’ routing-oriented interface in which the application is composed from a series of route handlers responsible for a particular method and URL.

Express provides `response.send`, which automatically sends the headers, sets the correct content type, and calls `end()` (so it will only be called once during a response). It will even convert JavaScript objects to JSON for us! Think of this as like “return” in a function context.

In fact we can think our API like specifying a function. The method and URL are analogous to the function name and the parameterized components of the URL are like the arguments.
The Express middleware is an example of a design pattern for implementing "cross cutting" concerns. Each middleware has access to the request, the response and the next middleware in the chain. Invoking `send` terminates the chain (and sends a response), while calling `next()` passes the request (and response) objects to the next middleware in the chain. With the middleware pattern we build up a complex application from many small transformations to the request (or response).

Unlike the routes we just saw, which are invoked for only a specific request, the middleware handlers are invoked for all requests. For example, in many applications most routes require the user to login. Instead of introducing this check in each route, we can do so with a middleware that will redirect all but a few specific unauthenticated requests to the login page.

Example middleware:
- body-parser: Parse JSON request body
- static: Return static assets, like HTML or CSS files
We will see other examples of "cross cutting" concerns soon, notably in implementing validations for models (in the MVC sense).

*Advice* is a specific piece of code that implements a cross-cutting concern

*Pointcuts* are the places you want to “inject” advice at runtime

Advice+Pointcut = Aspect
The Express routes often function at the controller (in the MVC sense). What about the Model?
In this context, the model is a film. There is no explicit model class, just a JavaScript object. And for such a simple application in which we are not persisting any changes (that is ratings don't persist through restarting the server), we might not need much more. But as we want to add features, we will quickly find that we could benefit from established design patterns and library support.

Object-relational Mapping (ORM) libraries provide much of the above "cross-cutting" (or aspect oriented) functionality (the parts that are the same) and thus we will often use ORM libraries to implement our models. The choice of a specific library will often depend on what kind of database we plan to use (e.g. SQL vs. NoSQL). We will discuss those choices more in subsequent classes. For now we will focus on the data modeling itself.

**Film model (M in MVC)**

Film “resource” is a simple JavaScript object

Good enough for now, but what about?

- Validate user rating is 0-5?  
  *Can’t trust the client!*
- Express associations between models
- Support different persistence layers (e.g. databases)

*We can use ORMs and other libraries to provide this “cross cutting” functionality*
The models are typically the RESTful resources

<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>

A single model: Film
OO modeling: A tiny bit of UML

- **Unified Modeling Language**: A visual language for describing artifacts in a OO system
- UML is expansive, we are focused just on class modeling features

![Diagram](image)

- Car is a subclass of Vehicle
- Engine is a *component* of Car
- Engine class includes `start()`, `stop()` methods

Relationships are shown by the arrows (and fill)...

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
CRC cards are like user stories, but for classes. Each index card contains:

- On top of the card, the class name
- On the left, the responsibilities of the class, i.e. what this class "knows" and "does". For example, a "car" class may know how many seats and doors it has, and could "do" things like stop and go.
- On the right, the collaborators (other classes) with which this class interacts to fulfill its responsibilities

Like User Stories, using an index card limits complexity and helps designers focus on the essentials of the system.

A preview of associations, that is how we talk about relationships between models. Here...

- A film has many genres
- There is a many-to-many relationship between Users and Films via the ratings. Often called a “has many through” association.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
The nouns in the user stories (blue) often correspond to models, while the verbs (red) correspond to associations between models and/or methods on the models. As you start to define the user stories for your application, e.g. your project, start to look for shared nouns that will become your models.

**CRCs and user stories**

**Independently rate a movie**

As a **user**
I want to **rate** a **movie**
So that I can save my **opinions of movie**

**Show average ratings**

As a **user**
I want to view **average ratings of a movie**
So that I can know if it is a good **movie**
Student Advice: CRC cards and designing up front

- “Having a solid design & schema saved us a lot of pain”
- “MVC's separation of concerns really made for a nice app structure”
- “Designing rich client-side and server-side in SOA made it easy to decouple development”
- “We wish we had designed the object model and schema more thoroughly”

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