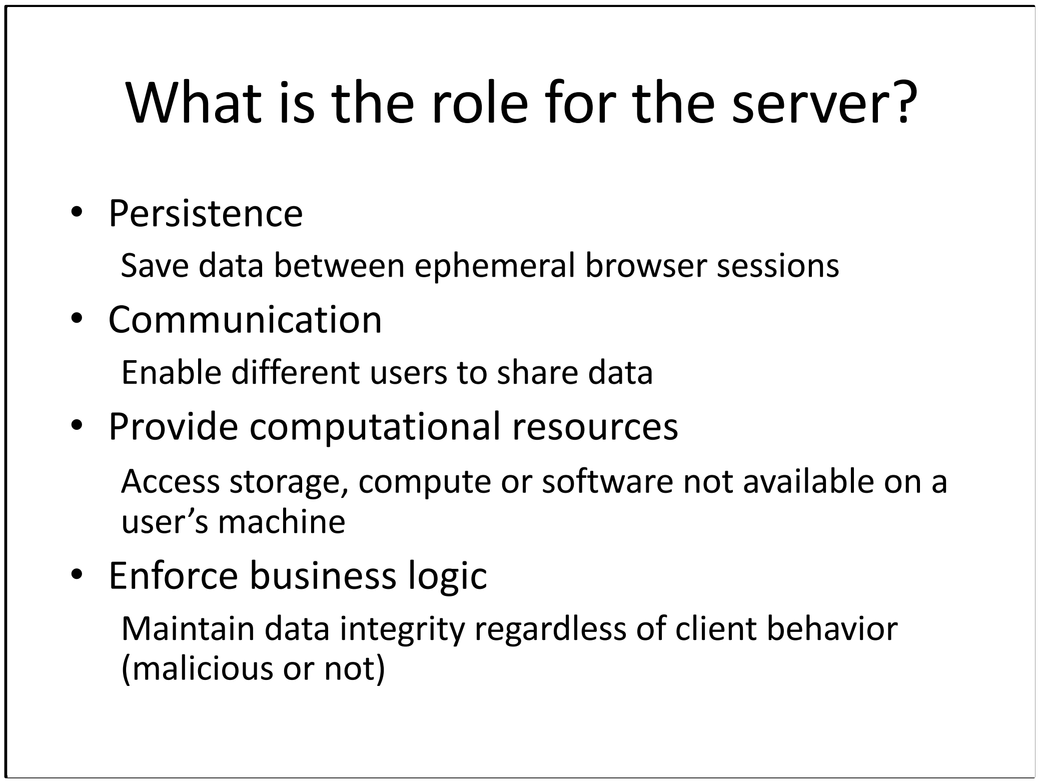


What is the role for the server?

- Persistence

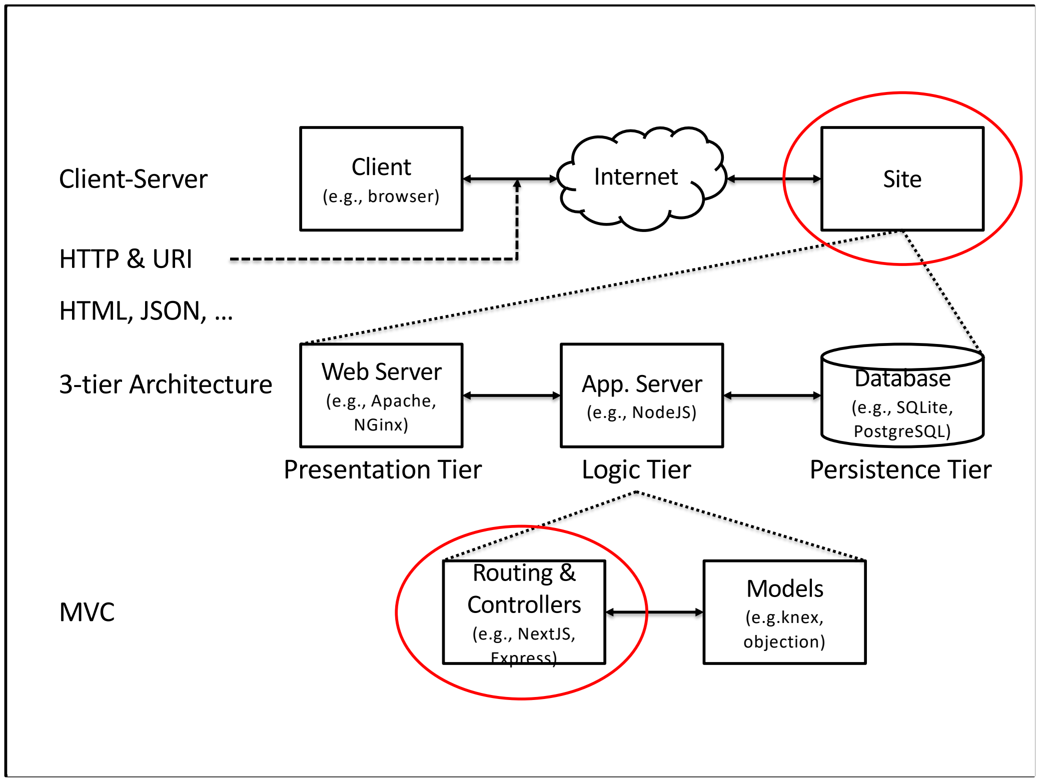
  Save data between ephemeral browser sessions
- Communication

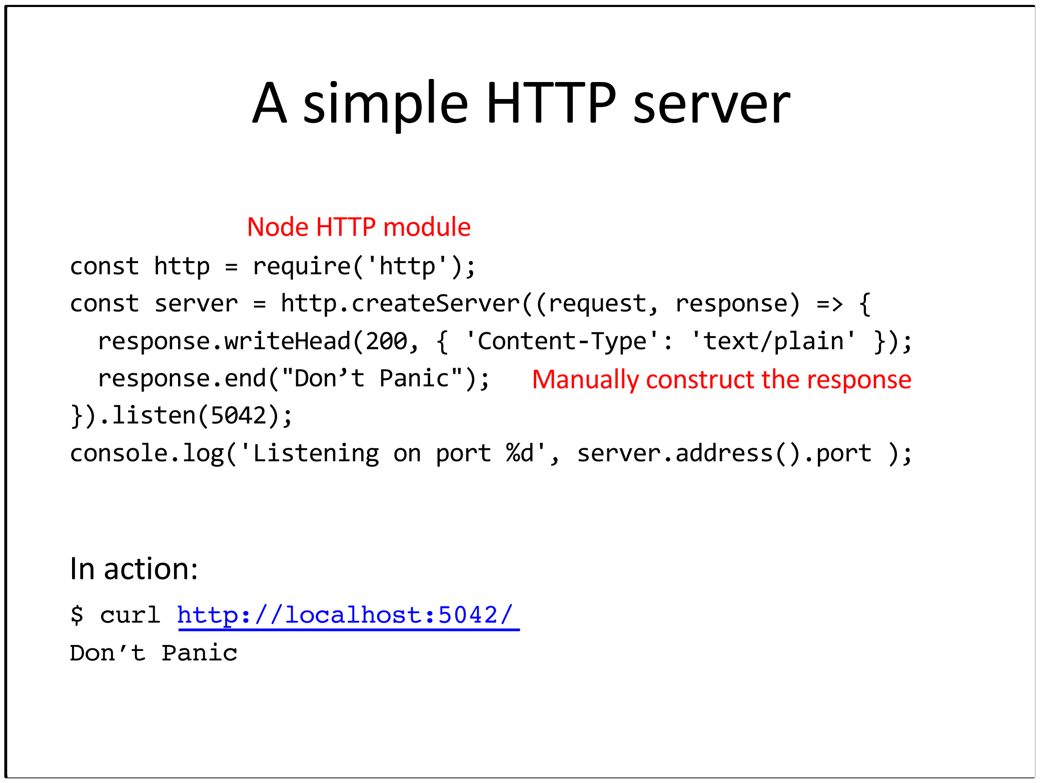
  Enable different users to share data
- Provide computational resources

  Access storage, compute or software not available on a user's machine
- Enforce business logic

  Maintain data integrity regardless of client behavior (malicious or not)

Examples of the last include rejecting duplicate titles in Simplepedia.


Client-Server
HTTP & URI
HTML, JSON, …
3-tier Architecture
MVC
Client
(e.g., browser)
Internet
Site
Web Server
(e.g., Apache, NGinx)
App. Server
(e.g., NodeJS)
Database
(e.g., SQLite, PostgreSQL)
Presentation Tier
Logic Tier
Persistence Tier
Routing & Controllers
(e.g., NextJS, Express)
Models
(e.g.knex, objection)



A simple HTTP server

Node HTTP module

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

Manually construct the response

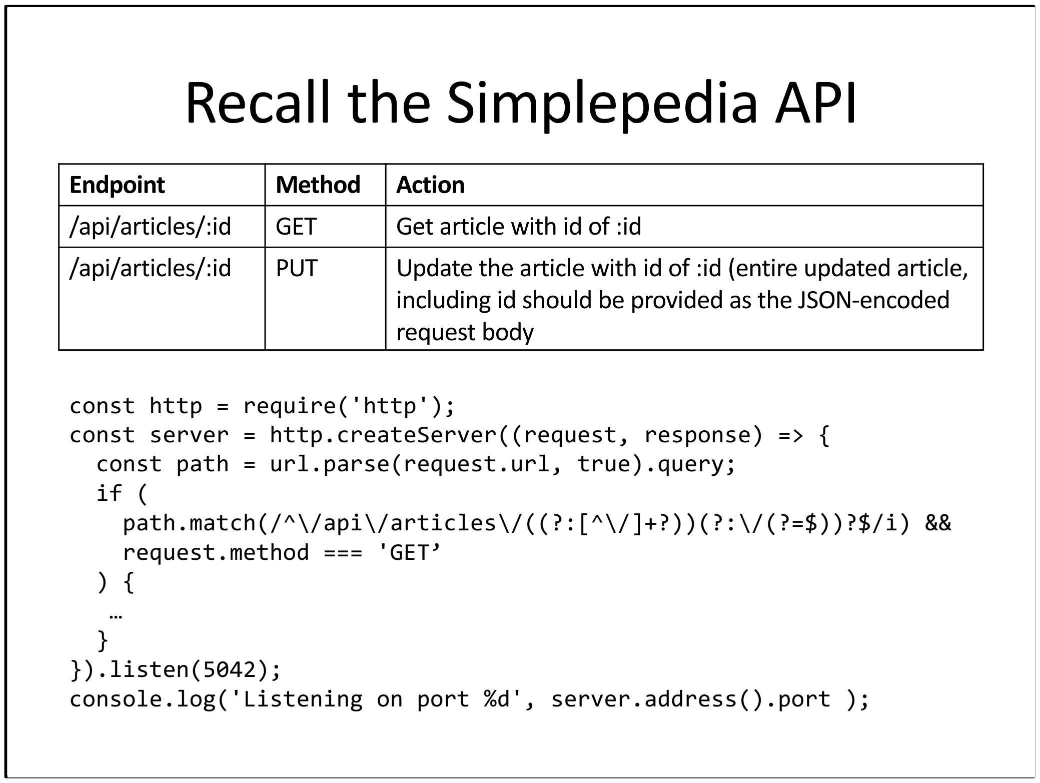
In action:

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

The server equivalent of Hello World using built-in Node modules.

We pass a function to act as the main logic of the server. The server hands that function two objects, a request object and a response object. The request object contains everything we need to know about the request from the client. We then use the methods of the server to respond. Here, we are ignoring the request altogether. On every response, we issue a 200 (response ok), set a head to indicate we are sending plain text, and then send “Don’t Panic”.

Note that the server is *stateless*. The server doesn’t remember who you are. Every transaction is a new one. This is how servers can handle high volumes of requests. Later, we will talk more about implementing cookies or other similar techniques to create the notion of a session.



Recall the Simplepedia API

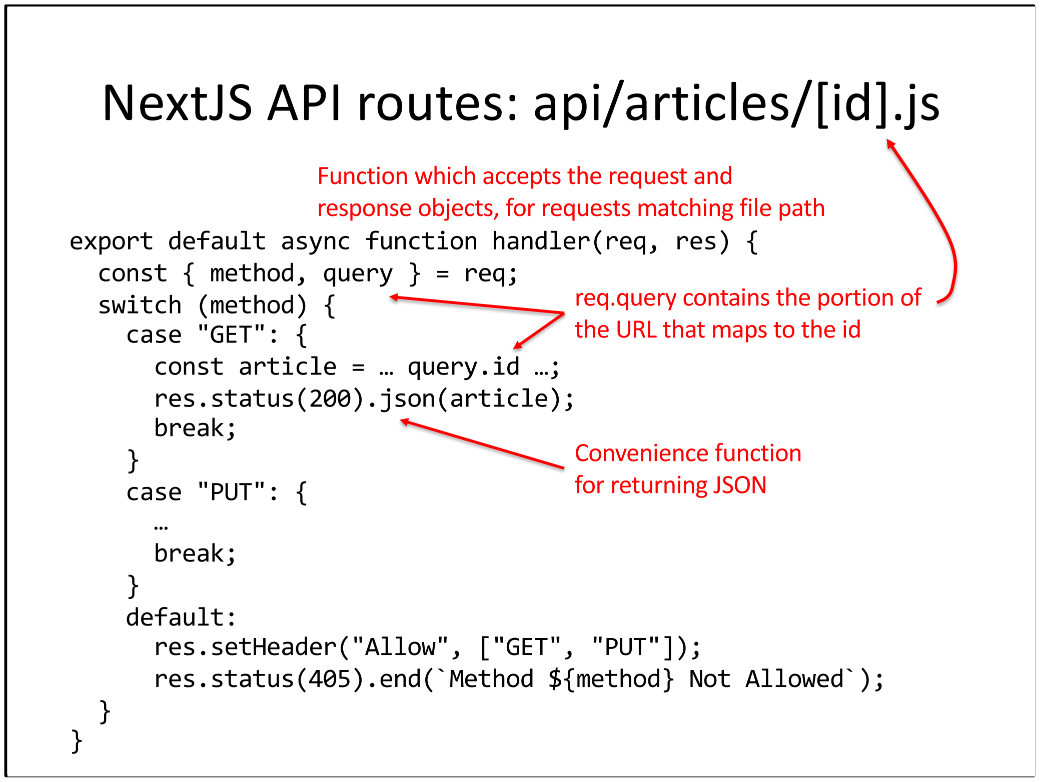
Endpoint	Method	Action
/api/articles/:id	GET	Get article with id of :id
/api/articles/:id	PUT	Update the article with id of :id (entire updated article, including id should be provided as the JSON-encoded request body

```
const http = require('http');
const server = http.createServer((request, response) => {
  const path = url.parse(request.url, true).query;
  if (
    path.match(/^\/api\/articles\/((?:[^\/]+?))(?:\/(?=$))?$/i) &&
    request.method === 'GET’
  ) {
   …
  }
}).listen(5042);
console.log('Listening on port %d', server.address().port );
```

Using what we just saw, we could implement this as… [click]

With this low-level interface we are responsible for everything, including 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. For example, constructing that regular expression to match and extract parameters from the URL.

We will use tools built into NextJS. But Express (https://expressjs.com), is another similar "minimalist" routing-oriented framework. There is a counterpart to Express in most server-side languages (e.g., Sinatra for Ruby and Flask for Python). We will see libraries that help make NextJS more express-like.



NextJS API routes: api/articles/[id].js

```
export default async function handler(req, res) {
  const { method, query } = req;
  switch (method) {
    case "GET": {
      const article = … query.id …;
      res.status(200).json(article);
      break;
    }
    case "PUT": {
      …
      break;
    }
    default:
      res.setHeader("Allow", ["GET", "PUT"]);
      res.status(405).end(`Method ${method} Not Allowed`);
  }
}
```

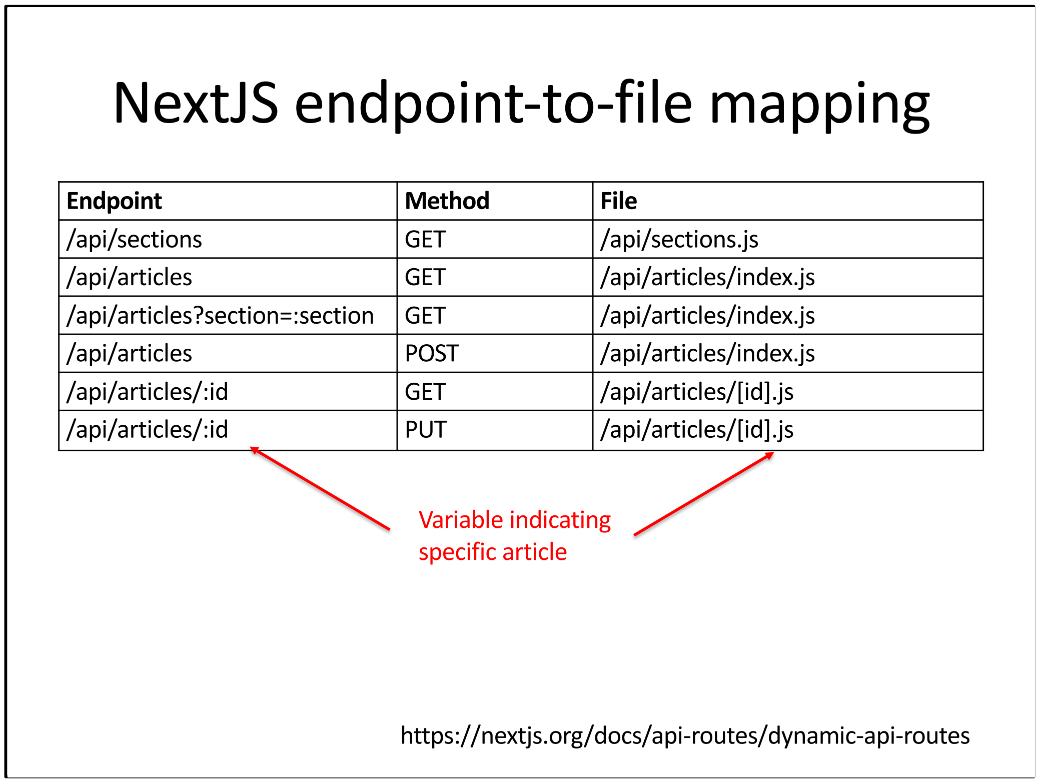
Function which accepts the request and response objects, for requests matching file path
req.query contains the portion of the URL that maps to the id
Convenience function for returning JSON

Next has support for API routes built in. So, when we deploy, it handles both serving the static files (the HTML and JavaScript), as well as providing the API endpoints. Adding a new route is as easy as adding a new file to the API directory and implementing in a function that looks like this. As we saw with pages, the directory structure supports dynamic routing, i.e., a single file matches multiple routes, and we can extract variables from the URL (e.g., id).

There is a one-to-one mapping between API endpoints and file-case combinations.

The req and res are not quite standard http.IncomingMessage and http.ServerResponse we sew previously; they have some extra built-in methods for common operations.

Note, this code runs on the *server*, while the rest of your code is running on the *client.* That is this code can access resources and do things the client-side code can't, e.g., access the database (and vice-versa)



NextJS endpoint-to-file mapping

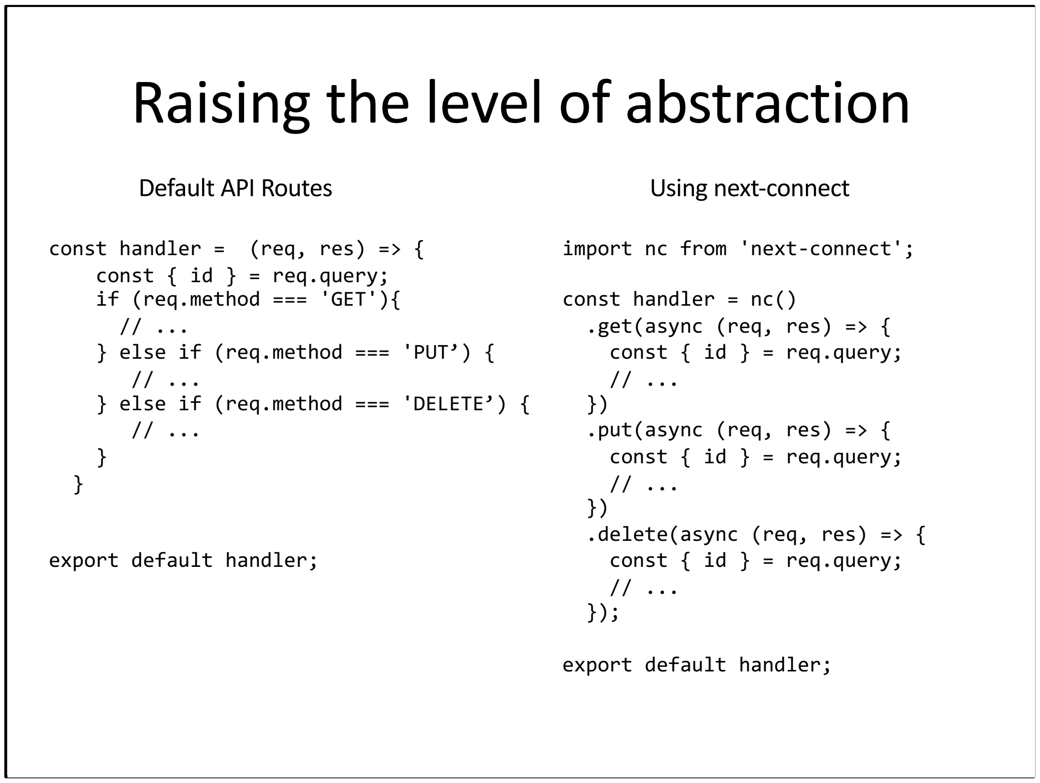
Endpoint	**Method**	**File**
/api/sections	GET	/api/sections.js
/api/articles	GET	/api/articles/index.js
/api/articles?section=:section	GET	/api/articles/index.js
/api/articles	POST	/api/articles/index.js
/api/articles/:id	GET	/api/articles/[id].js
/api/articles/:id	PUT	/api/articles/[id].js

Variable indicating specific article

https://nextjs.org/docs/api-routes/dynamic-api-routes

Variables in URLs become NextJS dynamic API routes. The relevant parameters are extracted by middleware (stay tuned…)

Note that /api/sections could map to /api/sections.js as shown or /api/sections/index.js. The directory name (with notihing else) maps to the index.js file. When might we use or the other? If we have multiple nested routes under a prefix, like we do with /api/articles, we will likely need multiple files and thus want the latter approach with a directory as opposed to a single file.



Raising the level of abstraction

Default API Routes

```
const handler =  (req, res) => {
    const { id } = req.query;
    if (req.method === 'GET'){
      // ...
    } else if (req.method === 'PUT’) {
       // ...
    } else if (req.method === 'DELETE’) {
       // ...
    }
  }


export default handler;
```

Using next-connect

```
import nc from 'next-connect';

const handler = nc()
  .get(async (req, res) => {
    const { id } = req.query;
    // ...
  })
  .put(async (req, res) => {
    const { id } = req.query;
    // ...
  })
  .delete(async (req, res) => {
    const { id } = req.query;
    // ...
  });

export default handler;
```

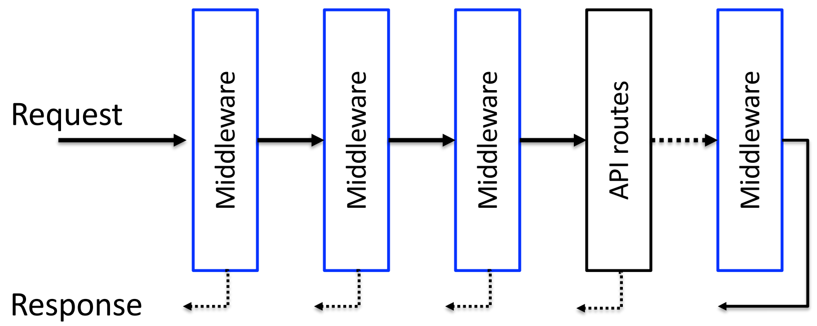
We will often use next-connect. This is a library that makes our routes a little simpler to write. It is also based on the same approach as Express, which is the library we would use for implementing servers if NextJS didn’t support API routes.

These do the same thing, but with next-connect it is clearer which endpoint we are implementing. Although not shown here, next-connect also handles non-matches, makes it easier to incorporate middleware, etc.

Interlude: Other NextJS server functionality

- API routes are just one form of server-side functionality
- NextJS also supports different types of server-side rendering (SSR)
  - Provide `getServerSideProps` function with your Page component to execute on server for each request. Return value injected into component as props
  - Provide `getStaticProps` to render page statically during build

Middleware



Middleware modify or respond to request

"Built-in" middleware:

- `req.query:` Builds the query and leaves it in `req.query`
- `req.body:` Parses the request body in the format specified in the headers
- `req.cookies:` Contains the cookies sent with the request

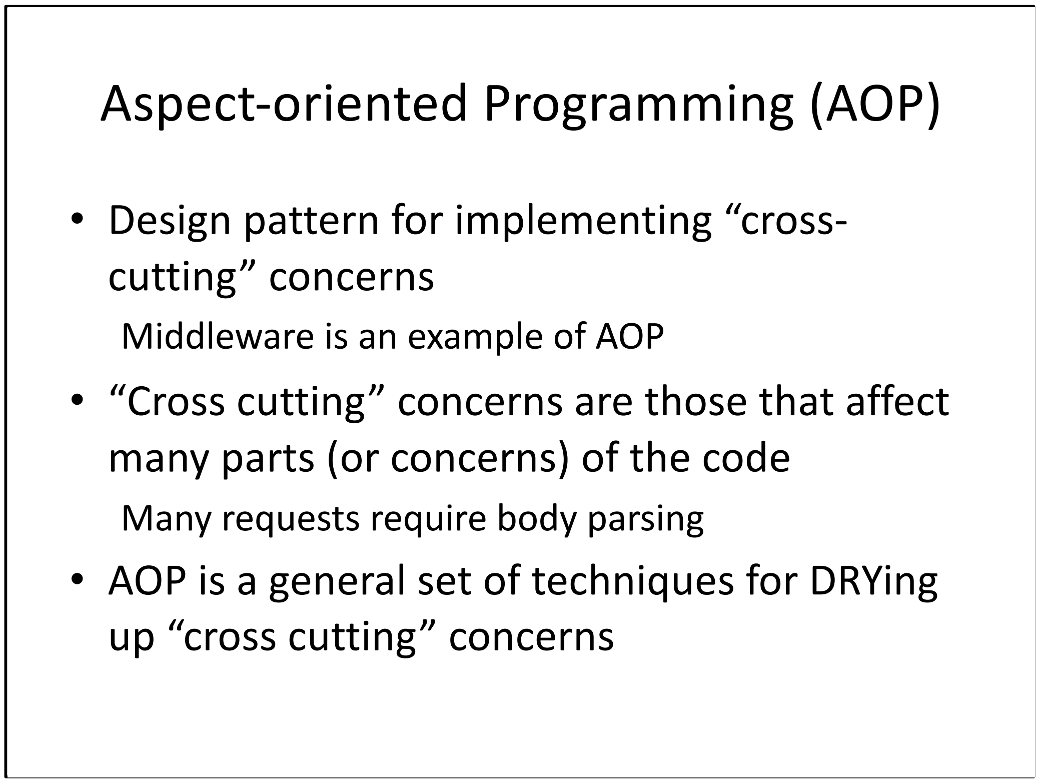
What do we mean by middleware. We can think of the server as a pipeline. Each piece of middleware in the chain takes in the request and the response. It can then end the request/response cycle, or it can pass the objects on to the next element in the pipeline. Most of these middleware augment the request or the response and passing it on.

Notice that the endpoints that we write are just another piece of the middleware: We can end the chain, or we can pass the request and response along to the next layer.

Response 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.

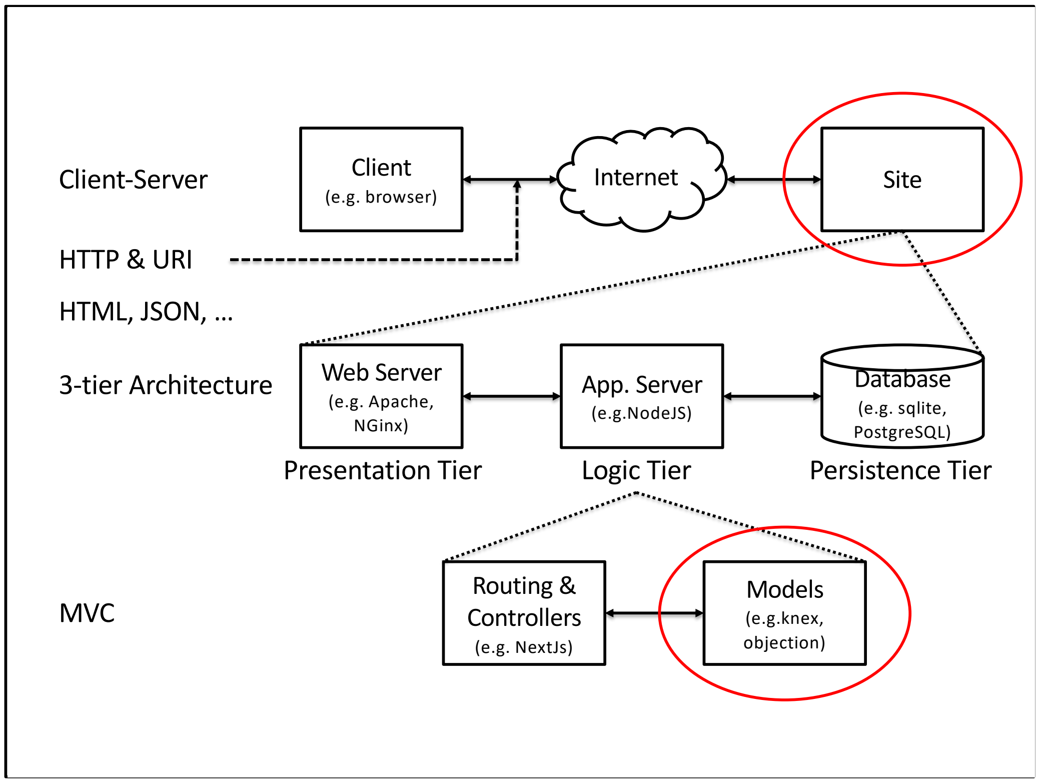
Example middleware:
body-parser: Parse JSON request body
static: Return static assets, like HTML or CSS files



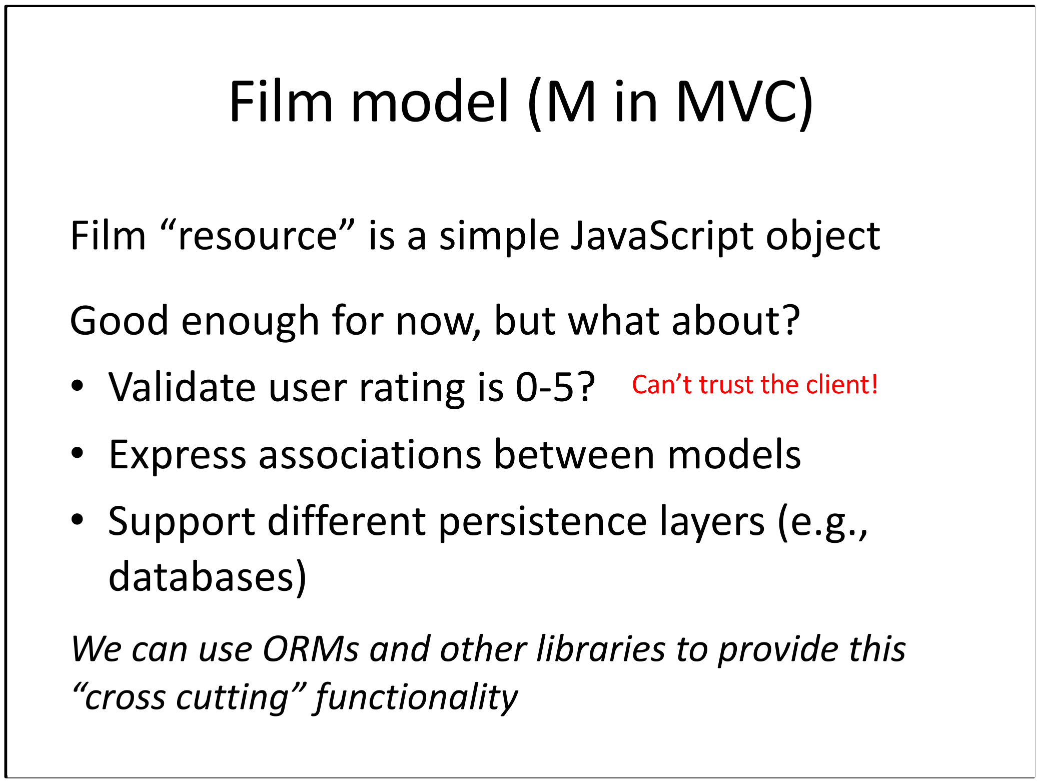
Aspect-oriented Programming (AOP)

- Design pattern for implementing “cross-cutting” concerns
  - Middleware is an example of AOP
- “Cross cutting” concerns are those that affect many parts (or concerns) of the code
  - Many requests require body parsing
- AOP is a general set of techniques for DRYing up “cross cutting” concerns

We will see other examples of "cross cutting" concerns soon, notably in implementing validations for models (in the MVC sense).



The Express routes often function at the controller (in the MVC sense). What about the Model?



Film model (M in MVC)

Film “resource” is a simple JavaScript object

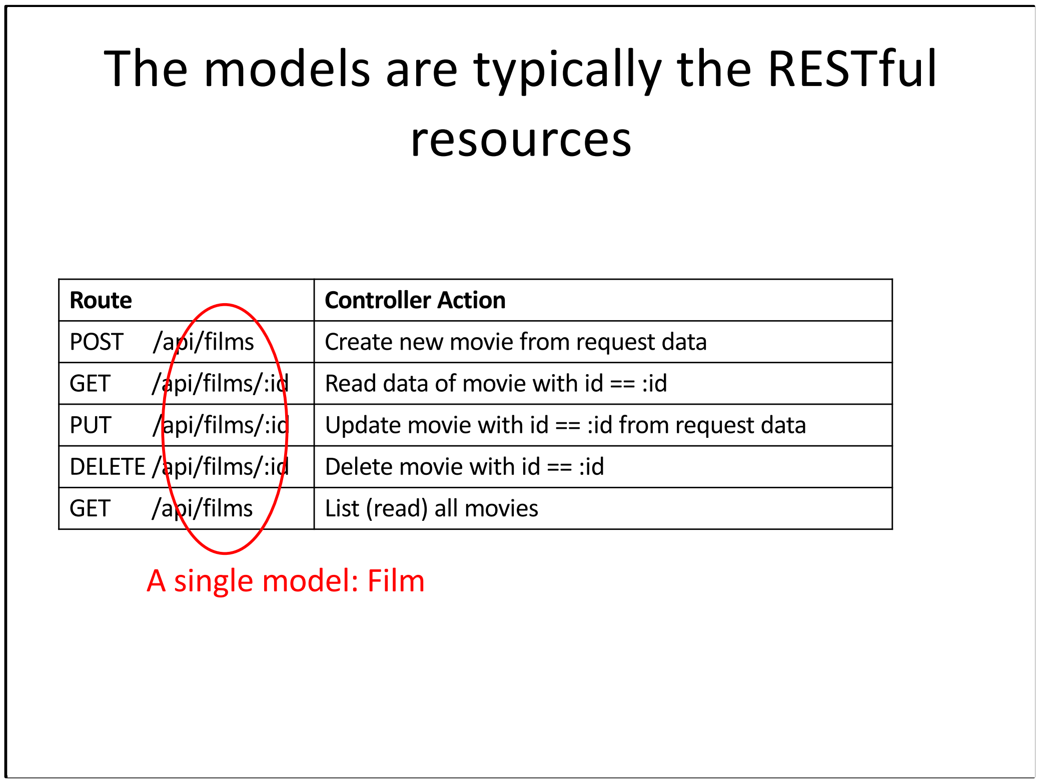
Good enough for now, but what about?

- 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

In that context, the model is a film. So far there is no explicit model class, just a JavaScript object. And for a simple application 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.

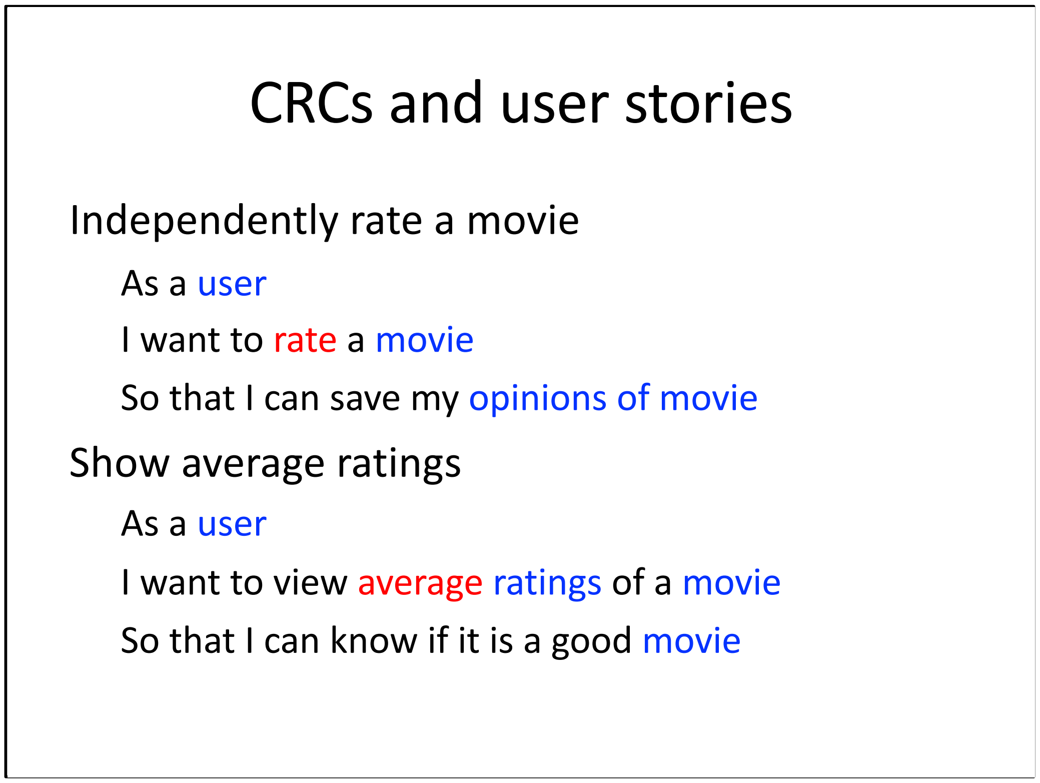
Object-relational Mapping (ORM) libraries provide much of the above "cross-cutting" (or aspect oriented) functionality (the parts that are the same) and thus we will often use ORM libraries to implement our models. The choice of a specific library will often depend on what kind of database we plan to 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.



The models are typically the RESTful resources

Route	Controller Action
POST /api/films	Create new movie from request data
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 single model: Film



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

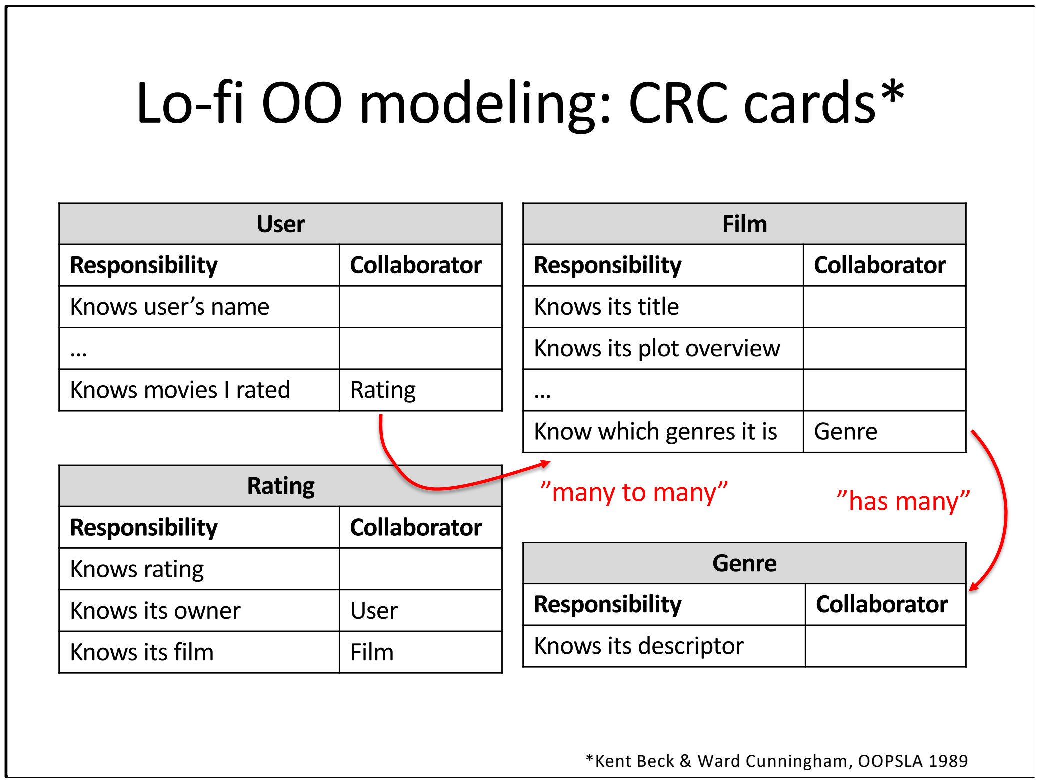
As a user

I want to view average ratings of a movie

So that I can know if it is a good movie

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.

The models will likely then become the resources in your server API.



Lo-fi OO modeling: CRC cards*

User	
Responsibility	**Collaborator**
Knows user's name	
…	
Knows movies I rated	Rating

Film	
Responsibility	**Collaborator**
Knows its title	
Knows its plot overview	
…	
Know which genres it is	Genre

Rating	
Responsibility	**Collaborator**
Knows rating	
Knows its owner	User
Knows its film	Film

Genre	
Responsibility	**Collaborator**
Knows its descriptor	

*Kent Beck & Ward Cunningham, OOPSLA 1989

CRC cards are like user stories, but for classes. Each index card contains:

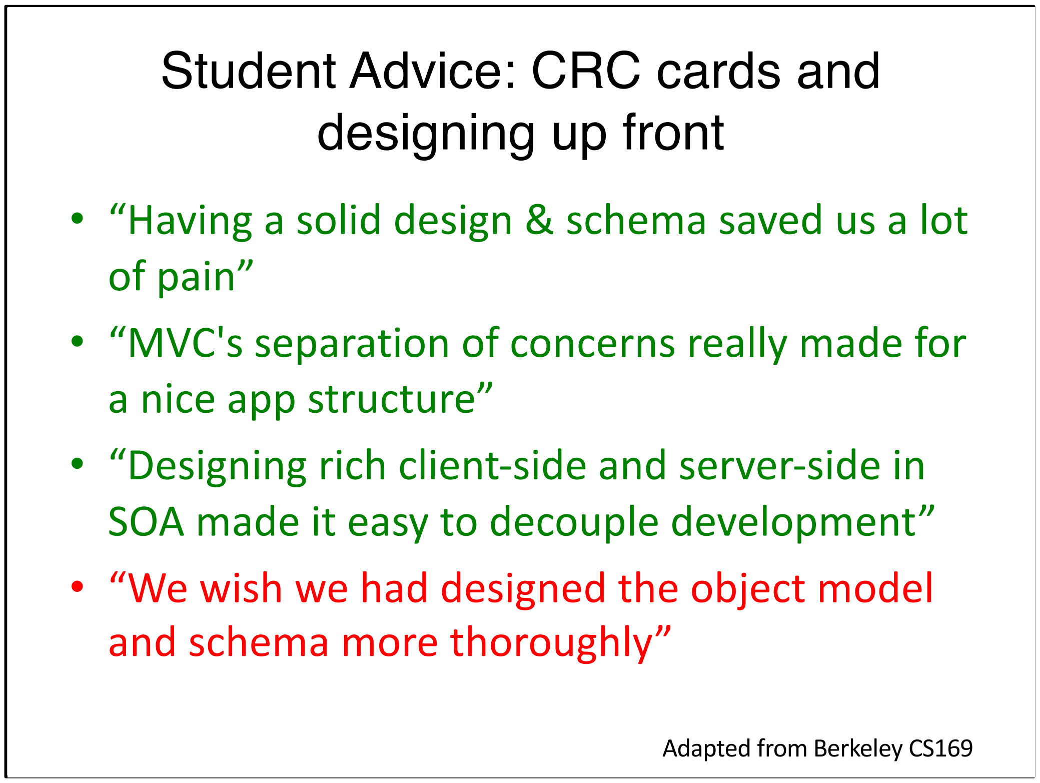
- On top of the card, the class name
- On the left, the responsibilities of the class, i.e., what this class "knows” and "does". For example, a "car" class may know how many seats and doors it has and could "do" things like stop and go.
- On the right, the collaborators (other classes) with which this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



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 Berkeley CS169