Upcoming Due Dates

- Tomorrow **(Friday)**: Sprint 0 Deliverables
  - *Please see the sprint 0 page on website for details*
  - Make sure that CRC cards/storyboards are visible in your backlog so that I can review them!
- Next Thursday: Practical 7
- Next Friday:

  FINAL DEADLINE for Assignments 1-4, Practicals 1-6

What is the role for the server?

- Persistence

  Save data between ephemeral browser sessions
- Communication

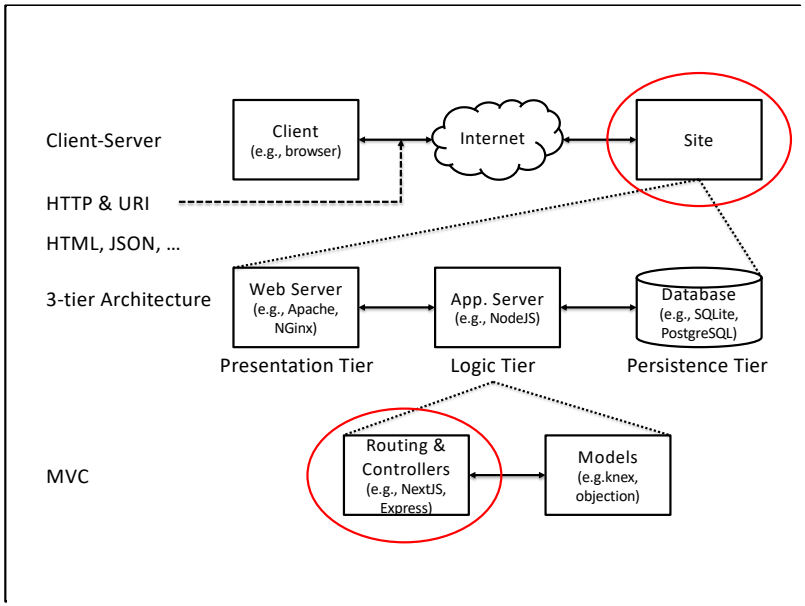
  Enable different users to share data
- Provide computational resources

  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)

This last one is critical. Most of work we have done so far, focused on code running on the client, i.e., in the user's browser. Recall though that they, the user, control that environment. They can do anything they want with the code and data you send. The server is only the environment you as the developer fully control (i.e., both the hardware and the software) and thus is the only place you can enforce the business logic (i.e., rules) for your application.

A concrete examples of the last is rejecting duplicate titles in Simplepedia. Even if we checked for duplicate titles on the client (in our code), nothing prevents a user from sending an article to the API (to the server) with a duplicate title.


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

For each request we need/want to respond with a status code (recall 2** is success, other ranges define different types redirects, errors, etc.) and typically a response body. Here that body is plain text, but since most of the servers we will build are intended to support "thick client" single-page applications, the request will return JSON encoded data (i.e., data encodes as a string using the JSON specification).

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. Recall we had talk about cookies and 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

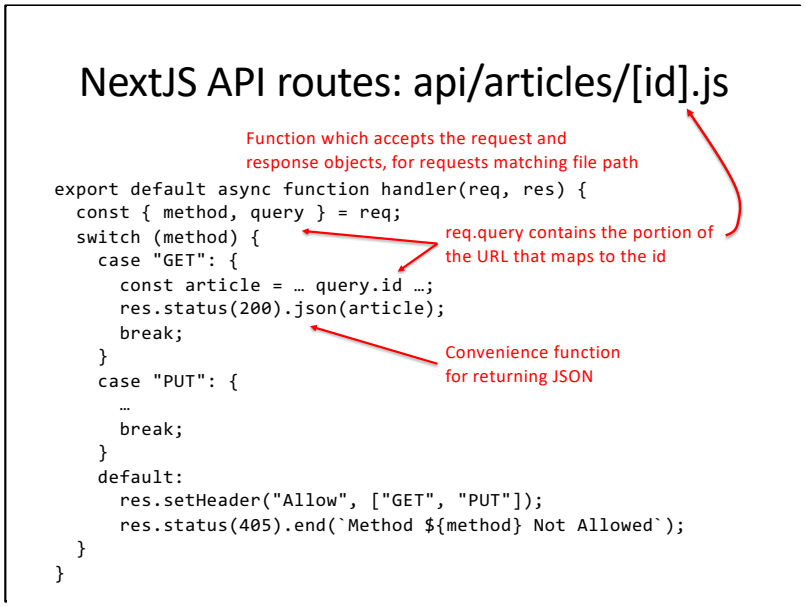
```
const http = require('http');
const server = http.createServer((request, response) => {
  const path = url.parse(request.url, true).query;
  if (
    path.match(/^\/api\/articles\/((?:[^\/]+?))(?:\/(?=$))?$/i) &&
    request.method === 'GET'
  ) {
   …
  }
}).listen(5042);
console.log('Listening 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. That is, we should be able to translate the high-level description of the endpoint, in the corresponding regex or other code for parsing the URL.

We will use tools built into NextJS. That is not the only approach. A common tool is Express (https://expressjs.com), a "minimalist" routing-oriented framework (and what we previously used in class). There is a counterpart to Express in most server-side languages (e.g., Sinatra for Ruby and Flask for Python). And in fact, we will use 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`);
  }
}
```

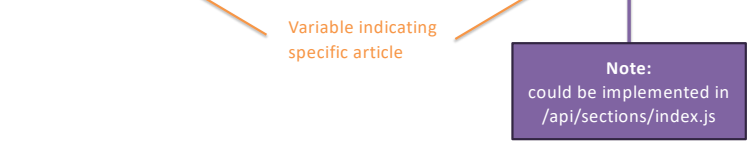
Next has support for API routes built in. So, when we deploy, it handles both serving the static files (the HTML and JavaScript), as well as providing the API endpoints. To add a new route, we add a new file to the API directory and implement in it a function that looks like this. As we saw with pages, 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, this code). That is the complex regular expression we saw previously is implemented in the file naming conventions.

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, that while this code is the same pages directory as the components you worked on in your assignments, by virtue of being in the api subdirectory, 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 a database (and vice-versa). By vice verse we mean the client code can use browser features that aren't available on the server.

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



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

Variables in URLs, e.g. [id], 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 nothing 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

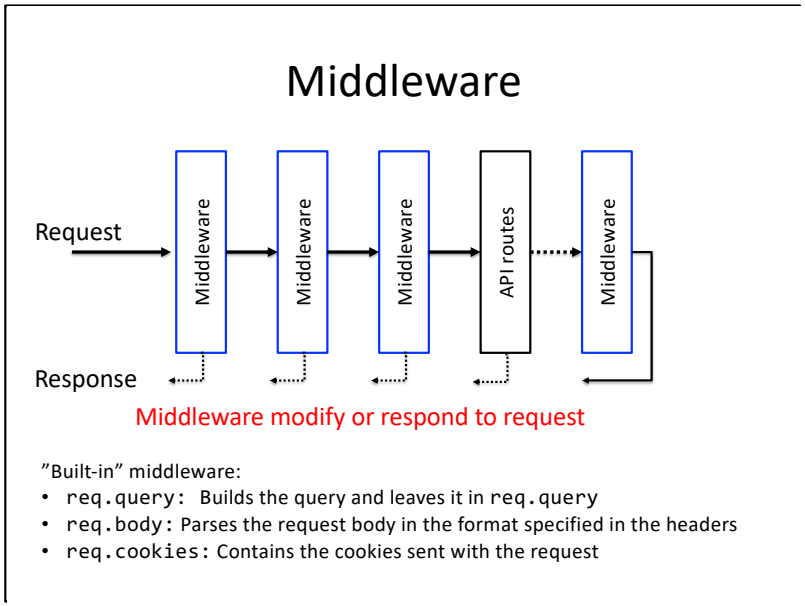
```
import { createRouter} from 'next-
connect';

const router = createRouter();

router.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 router.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 (even if a bit more verbose). And (although not shown here) next-connect also centrally handles non-matches, makes it easier to incorporate middleware, and more.

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



What do we mean by middleware. We can think of the server as a pipeline where the middleware are the stages in the 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 modify and pass the objects on to the next operation in the pipeline. Most of these middleware augment the request or the response and pass 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` or response methods (depending on the specific library API) 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 un-authenticated 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)

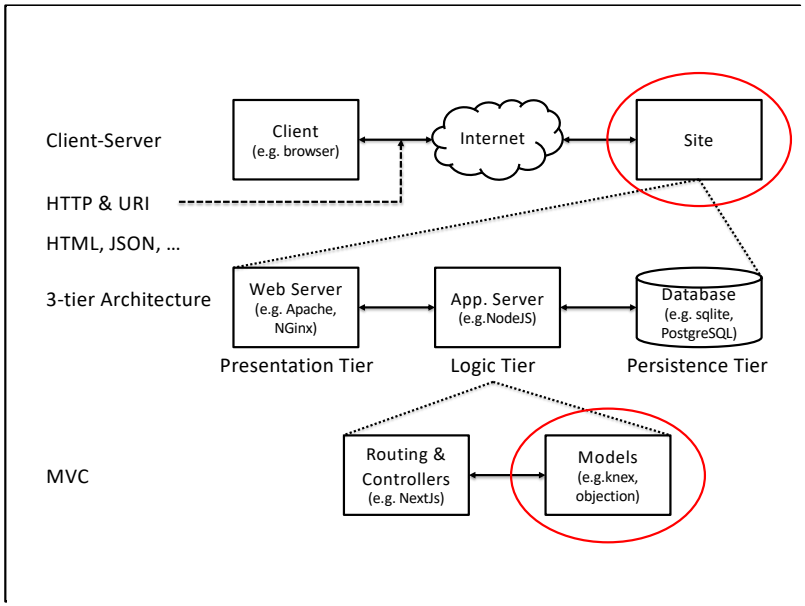
- Design pattern for implementing “cross-cutting” concerns
  - Middleware is an example of AOP
- “Cross cutting” concerns are those that affect many parts (or concerns) of the code
  - E.g., many requests require body parsing
- AOP is a general set of techniques for DRYing up “cross cutting” concerns

Middleware is an example of Aspect-oriented programming.

“Aspect-oriented programming is a technique for building common, reusable routines that can be applied applicationwide.” - https://www.sciencedirect.com/topics/computer-science/aspect-oriented-programming#:~:text=Aspect%2Doriented%20programming%20is%20a,error%20handling%2C%20etc.).

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

What is the/a criticism of AOP? A common criticism is “action at a distance”, that is when working on/debugging a specific route the issue may originate in middleware that is implemented and included the response chain somewhere else. That distance can make debugging tricky!.



What about the Model? This is both specific code we will implement to provide cross-cutting functionality for data validation, associations (links between different kinds of data), and to transparently support different persistence layers. But it also a general term for think about the "resources" or "nouns" in our application. That is think of it as the "data model" for our application. That data model influences the code we write for interacting with the database, but also determines what routes we create.

Where do routes come from?

Routes typically derive from RESTful actions on the models or resources, the "nouns", in our application.

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

A single model: Film

As a second example, consider Simplepedia, most routes implement a RESTful interface for the "Article" resource/model. Is that the only resource? No. We also have a resource representing sections. The sections are derived from our article data but is implemented as a separate resource in our API.

Resources come from user stories

Independently rate a movie

As a user

I want to rate a movie

So that I can save my opinions of the 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 then often become the resources in your server API. For example, here movie becomes the film resource. These two user stories imply that we may need another resource representing the opinion of the movie, and a corresponding API route for creating that rating. To help us figure that out cheaply, before we write any code, we use CRC cards. CRC cards, which we saw in our spring 0 planning meeting last time are low-cost mechanism for figuring out this data model, and specifically figuring out the resources in our application and the relationships between them. For example…

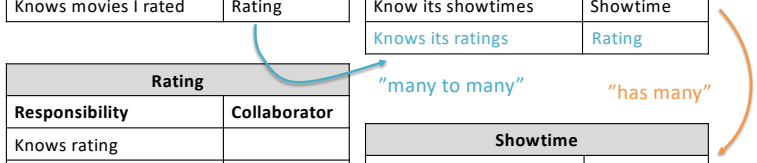
Lo-fi OO modeling: CRC cards*

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

Film	
Responsibility	**Collaborator**
Knows its title	
Knows its plot overview	
Know its showtimes	Showtime
Knows its ratings	Rating

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

Showtime	
Responsibility	**Collaborator**
Knows its theater	Theater
Knows its time	



*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.

<Fill in knows its rating, and as a collaborator Rating>

A preview of associations or how we talk about relationships between models. Here…
- A film has many showtimes
- There is a many-to-many relationship between Users and Films via the ratings. Often called a "has many through" association.

Closing the loop to define the API

Independently rate a movie

As a user

I want to rate a movie

So that I can save my opinions of the movie

POST /api/films/:film_id/ratings

PUT /api/films/:film_id/ratings/:rating_id

Working from our CRC cards, implementing the first user story might involve the following RESTful routes to create and update ratings associated with a film (i.e., a child resource of a film).

Closing the loop to define the API

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

GET /api/films/:film_id

The second user story would likely still be part of getting the data about a film, but would imply in that route we would want to obtain and summarize the associated ratings, i.e., we aren't creating a new API per se, but using the CRC cards to guide how we implement the route itself.

In-class application example

As an instructor, I want to launch a multiple-choice question to a room so that participants can respond.

As an instructor, I want to see the count and breakdown of participant responses in real-time so I can monitor response rates and understanding

In-class application example

As an **instructor**, I want to launch a **multiple-choice question** to a **room** so that **participants** can respond.

As an instructor, I want to see the count and breakdown of participant responses in real-time so I can monitor response rates and understanding

Room	
Responsibility	**Collaborator**
Knows name	
Knows members	User via Roster
Knows polls	Poll

Poll	
Responsibility	**Collaborator**
Knows start & end	
Knows results	
Knows rooms	Room

User	
Responsibility	**Collaborator**
Knows name	
Knows email	

Roster	
Responsibility	**Collaborator**
Knows role	
Knows room	Room
Knows user	User

Room	
Knows name	
Knows members	User via Roster
Knows polls	Poll

Poll	
Knows start & end	
Knows results	
Knows rooms	Room

User	
Knows name, email, etc.	

Roster	
Knows role (e.g., instructor, participant)	
Knows room	Room
Knows user	User

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

Demo: Testing Your API

Three methods:

- curl (command line)
- fetch (javascript)
- postman (web UI)

http://localhost:3000/api/articles/10