Context and Goals

We've seen how Node.js enables JavaScript to interact with the computer. Express helps our Node application respond to HTTP requests. Let's apply these tools to create something consumer-focused: a simple Reddit/Hacker news clone. This project will dive deep into Express features.

Wizard News

We will build:

1. A homepage similar to Hacker News, that lists all posts.
2. A Details page, that shows the complete news content.
3. A form to send a new post. (not right now, but on a separate workshop)

Important note: Right now, there will be no persistent database in our application; we will simply be using a JavaScript array to hold objects in the server's RAM.

Setup

Download and open the zip file. Make sure in the terminal you `cd` into the correct directory and run `git init`. Then create a repository on github and push the code up

Project Structure

The initial project contains an `app.js` file and a `/public` folder with some assets. You can ignore the public folder for now - it's not currently being used in the app and you will work on it in a future step.

For now, let's investigate `app.js` - it contains a bare-bones express application:

```
const express = require("express");
const app = express();

app.get("/", (req, res) => res.send("Hello World!"));

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

Notice that we currently have a single defined route.

NPM

`npm install`

When cloning a node project, your first step is usually running `npm install` before testing. By invoking `npm install` without specifying any package, npm will look into the projects listed dependencies (inside `package.json`) and install all of them.

Npm scripts

Remember that one of `package.json`'s roles was to define scripts. We already provided a `start` script that handles the launch of the application, try it.

SIDE NOTE: You can imagine that having different scripts could be useful. For example, on our production server, we might prefer `node` to `nodemon`. We could use separate scripts for `serve` vs. `develop`. A `test` script would run our specs, and so on.

Logging Middleware

Your first task is to add some logging middleware that will fire *for every incoming request*. In the long run, this kind of utility will help us debug our application.

What is a Middleware again?

Middleware is any function that is invoked by the Express.js before your final request handler is, and thus sits in the **middle** between a raw request and the final intended route.

Here's the general form:

`app.use([[function here]])` registers some function to run for **each incoming request**.

---

morgan

One of the most popular logging middlewares is `morgan`, created by the express team. Passing it to `app.use()` makes it intercept all request and responses - every time you send a response, Morgan logs the request and response information. Morgan is also very configurable, with lots of "modes" (we recommend using the "dev" mode during development). For example, after installing morgan (`npm install morgan`): `app.use(morgan('dev'));`

Setting up a postBank.js Module

Great - The application is all set up and you have a sample route. We need data for our real routes, though. In the future, we will integrate Wizard News with a proper database management system, but for now, we will create a provisory way to store the information:

Make a `postBank.js` file in your project directory. This module will be responsible for holding all of the posts and giving us functions for interacting with them.

Sample Data

Copy the following code to your `postBank.js` file:

```
const data = [{ id: 1, upvotes: 257, title: "Fianto Duri, the complete tutorial", content: "Fianto Duri is a charm that was created to be combined with protective spells (Can be used with another person's shield spell)(When used on something else creates a explosion). As we already knows the (i.e.) Shield Charm needs the caster to stay focused on the spell in order to continue protecting him, so Fianto Duri allows the caster to keep a charm “alive” while he does some other work or casts some other spells.", name: "RubeusH", date: new Date(Date.now() - 15000000) }, { id: 2, upvotes: 221, title: "Untransfiguration classes to become compulsory at Hogwarts", content: "Learning untransfiguration is going to be mandatory at Hogwarts School of Witchcraft and Wizardry from 2017 onward. Untransfiguration will be covered in beginner-level spellbooks such as A Beginner's Guide to Transfiguration. Failure to at least attempt to untranfigure a wrongly-done transfiguration will be considered irresponsible.", name: "Baddock", date: new Date(Date.now() - 90000000) }, { id: 3, upvotes: 198, title: "Cracking the Aurologist Interview", content: "Now in the 5th edition, Cracking the Aurologist Interview gives you the interview preparation you need to get the top aura study jobs. The book is over 500 pages and includes 150 aurologist interview questions and answers, as well as other advice.", name: "Hetty", date: new Date(Date.now() - 900000) }, { id: 4, upvotes: 171, title: "ASK WN: What do you use to digitalize your scrolls?", content: "Some scrolls need conservation treatment before they can be safely transported, handled, and digitized.  After these questions are answered, Preservation and Information Technology Specialists assess the project requirements and create the digitilized version.", name: "Alphard", date: new Date(Date.now() - 5000) }, { id: 5, upvotes: 166, title: "The Pragmatic Dragon Feeder", content: "In The Pragmatic Dragon Feeder, the author Baruffio tell us how to give food to dragons in a way that we can follow. How did they get so smart? Aren't they just as focused on details as other dragon feeders? The answer is that they paid attention to what they were doing while they were doing it.", name: "Baruffio", date: new Date(Date.now() - 10000000) }, { id: 6, upvotes: 145, title: "The complete quidditch statistics", content: "This is the Complete source for quidditch history including complete player, team, and league stats, awards, records, leaders, rookies and scores.", name: "Hbeery", date: new Date(Date.now() - 5000000) }, { id: 7, upvotes: 126, title: "Ordinary Wizarding Levels study guide", content: "The Ordinary Wizarding Level (O.W.L.) is, as you know, going to determine whether or not you will be allowed to continue taking that subject in subsequent school years, and whether they might be successful in obtaining a particular job. This guide help direct you to the most important information you need to know to ace the test", name: "Alatar", date: new
```

```
Date(Date.now() - 600000) }, { id: 8, upvotes: 114, title: "Is muggle-baiting ever acceptable?", content: "Muggle-baiting can be a manifestation of anti-Muggle sentiments and is not acceptable according to the International Statute of Wizarding Secrecy - But, are there any circumstances under which it could be acceptable?", name: "Falco", date: new Date(Date.now() - 60000000) }, { id: 9, upvotes: 102, title: "Conserving waterplants cheatsheet.", content: "This Cheat Sheet is dedicated to providing wizards the information they want in an approachable, entertaining way.", name: "Otto", date: new Date(Date.now() - 3000000) }, { id: 10, upvotes: 59, title: "Could wizards prevent WW3?", content: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vitae fermentum enim. Pellentesque sodales ut risus eu porta. Duis dictum rhoncus semper. Proin accumsan mollis ligula, eget elementum nibh dignissim quis. Proin augue risus, mollis non neque in, molestie rutrum purus. Morbi pretium nisl a commodo.", name: "Cuthbert", date: new Date(Date.now() - 6000000) }, { id: 11, upvotes: 46, title: "Show WN: Wand-Extinguishing Protection", content: "This spell extinguishes the wand the caster is holding, a counter-charm to Lumos.", name: "Humphrey22", date: new Date(Date.now() - 50000) }, { id: 12, upvotes: 30, title: "Do you still use Alarte Ascendare?", content: "You've got levicorpus and Ascendio and wingardium leviosa, so is anyone still using Alarte Ascendare, too? (That is, unless you find wingardium leviosa too difficult to pronounce.)", name: "Bellatrix1", date: new Date(Date.now() - 6000000) }, { id: 13, upvotes: 21, title: "Mailing lists WN readers ought to know about?", content: "I love to subscribe to information feeds through mailing list subscription. What do you subscribe to that you think others would benefit by if they were to as well?", name: "Dracod", date: new Date(Date.now() - 60000) }, { id: 14, upvotes: 10, title: "How to tell which spell used on a bug?", content: "Question: Are ther any non-jinx incantations available to detect which spell used on a bug?", name: "Lupin", date: new Date() }];
```

Defining the postBank.js Functions

You now have a `data` array with all the posts, but we do not want to make this array directly accessible to the rest of our app; it will safely remain as a private variable inside the postBank module.

What we *will* make available, using `module.exports`, are functions for listing and finding posts. Add the code below to the bottom of your `postBank` module.

```
const list = () => {

 return [...data] // Notice that we're returning a copy of the array, so the original data is safe. This is called 'immutability'.

};
```

```
const find = (id) => {

 const post = data.find(post => post.id === +id);

 return {...post}; // Again, we copy the post data before returning so the original information is safe.

}



module.exports = { list: list, find: find };
```

Listing Posts

Now that we have a data module with lots of posts, your next job is to create a route that lists all post titles and authors.

First, **Remove the given initial route** from `app.js`. Keep any middleware (app.use), of course.

Then, create a route to list the titles and author names of all posts.

A HINT IS ON THE NEXT PAGE. Try to attempt it yourself


```
const express = require("express");
const morgan = require("morgan");
const postBank = require("./postBank");

const app = express();

app.use(morgan('dev'));

app.get("/", (req, res) => {
  // First, get the list of posts.
  const posts = postBank.list();

  // Then, prepare some html to send as output.
  const html = `<!DOCTYPE html>
    <html>
    <head>
      <title>Wizard News</title>
    </head>
    <body>
      <ul>
        ${posts.map(post => `<li>${/* You have access to the post data here*/}</li>`)}
      </ul>
    </body>
    </html>`;

  // Finally, send a response
  res.send(html);
});
```

You might notice a stray comma between posts. `arr.map()` returns an array of elements, but we're injecting that array into a string. This means that the array elements have to be JOINED together. By default, JavaScript joins elements with a comma separator between elements. If you'd like to get rid of those pesky commas, consider running join manually. Here are the MDN docs if you'd like a nudge in the right direction.

In a bonus step at the end of this workshop, we'll use a fancy *HTML template tag*, and we will not need to manually join arrays.

Static Routing

Next, we need to make sure that the express application serves up the contents of the files it finds in the `/public` folder. Notice that by default this folder is being completely ignored by your application - if you want express to look for files in this folder and serve them, you have to configure it to do so.

Read the documentation for `express.static` and incorporate *static routing* into your application for the `public` directory.

Now, everything we put in `public` will be automatically accessible via URI path, as if it was actually a filepath (remember, normally it is not!). That includes `public/style.css`, which the browser can request with `GET /style.css`.

What other reasons might we prefer this kind of static routing? Here are two frequent use cases:

> A folder of dozens of images that form part of our website's presentation
> A folder of javascript files, so that code can be downloaded & run on the client side

Imagine having to write individual routes to serve up every one of those potential files. Static routing takes care of that for us, automatically; now all we need to do is drop a file into `public` somewhere, and Express will automatically route requests for it.

Is it working?

When its time to see if everything went as planned, visit http://localhost:1337/logo.png and you should see the Wizard News logo.

Styling the initial route

Great job - you have a main route that lists all of the posts and static route is serving everything in the public folder.

Let's combine those two to make a nice-looking display of news, shall we? Edit your main route so that it returns some HTML like this:

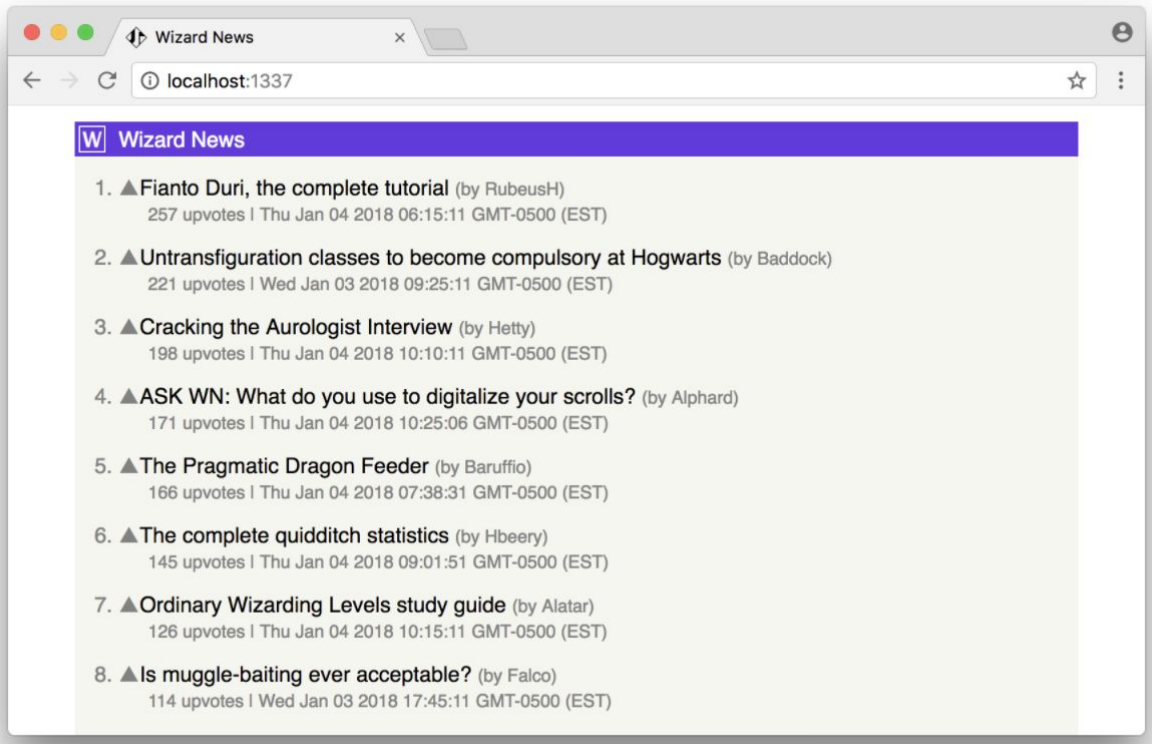
```
`<!DOCTYPE html>
 <html>
 <head>
   <title>Wizard News</title>
   <link rel="stylesheet" href="/style.css" />
 </head>
 <body>
   <div class="news-list">
     <header><img src="/logo.png"/>Wizard News</header>
     ${posts.map(post => `
       <div class='news-item'>
         <p>
           <span class="news-position">${post.id}.
▲</span>${post.title}
           <small>(by ${post.name})</small>
         </p>
         <small class="news-info">
           ${post.upvotes} upvotes | ${post.date}
         </small>
       </div>`
     ).join('')}
   </div>
 </body>
</html>`
```

The main differences between this and what you had before are:

1. Added `<style>` tag to load the `style.css` file.
2. Added `<header>` tag to display the “Wizard News” logo.

3. More detailed markup to display each post, including more information (upvotes, date and id).

If your style.css is being correctly server from the public folder and you edited everything correctly, you should see something like this:



Dynamic Routing with Parameters

Right now, our server has one route. This means users can do just one thing: see a feed of all posts. We want more routes, starting with a route that allows the user to see the complete details of one post.

Request Parameters

Another way of thinking of routes is that they "catch" requests.

```
app.get( '/posts/:id', someFunction );
```

would catch the request `GET /posts/7` (and then call `someFunction`, passing in `req` and `res`).

What's this new `:id` part of the URI? The colon `:` is a trick that Express provides to define particular portions of the URI string as variables. In other words, in `posts/:id`, the `:id` portion can be anything. The variable and its value are stored on the `req.params` object.

```
// say that a client GET requests the path /posts/7
app.get( '/posts/:id', (req, res) => {
  console.log( req.params.id ); // --> '7'
});
```

Here is another example to make this clear:

```
// say that a client GET requests the path /users/nimit
app.get( '/users/:name', (req, res) => {
  console.log( req.params.name ); // --> 'nimit'
});
```

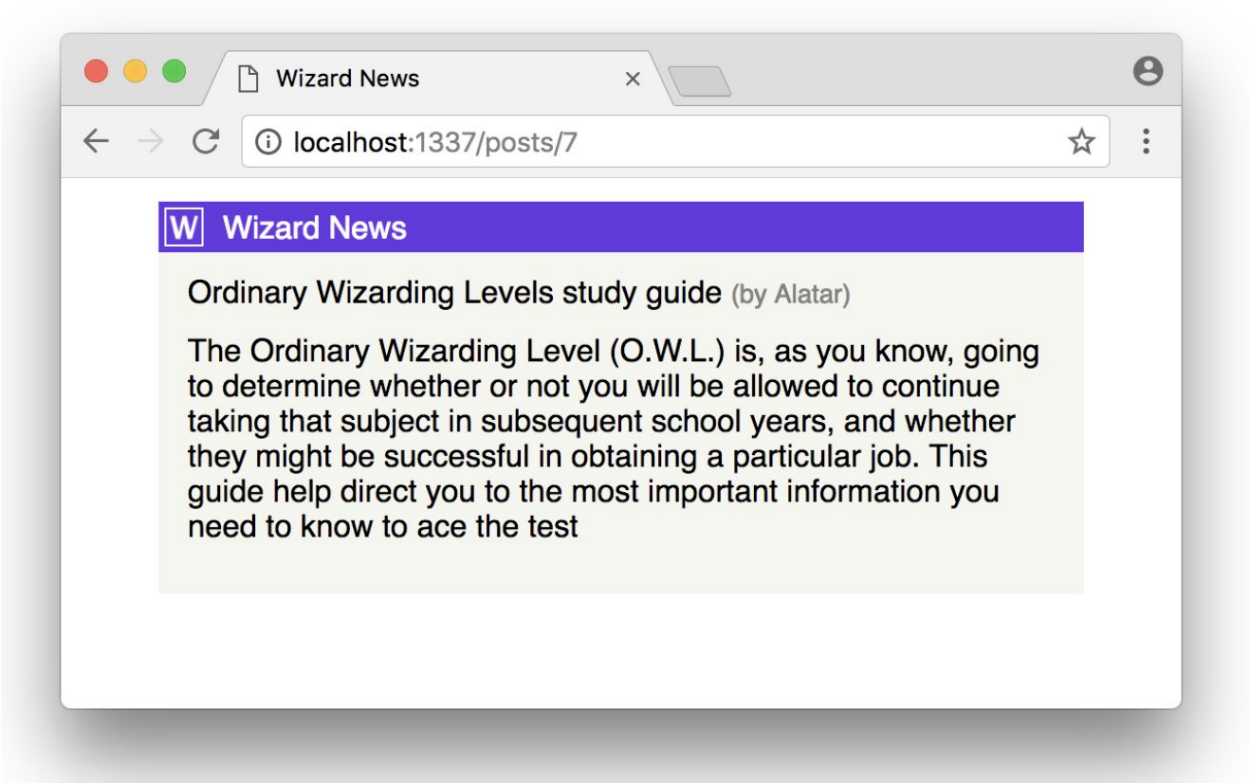
Add a Single-Post Route

In order to display details from a single post, your route should look (almost) like this:

```
app.get('/posts/:id', (req, res) => {
  const id = req.params.id;
  const post = postBank.find(id);
  res.send(/* The HTML document string here */);
});
```

For the HTML document string, you can reuse most of the string used in the original route and add the post details (title, author name, date and content).

At the end, if you manually visit the URL `/posts/7` in your browser you should see something like this: (next page)



Adding links to post details in the main route

We can link to this new page in the main route. Add links to the loop in the html document string so that each post title links to the correct post details view.

An example of how you could do so is below:

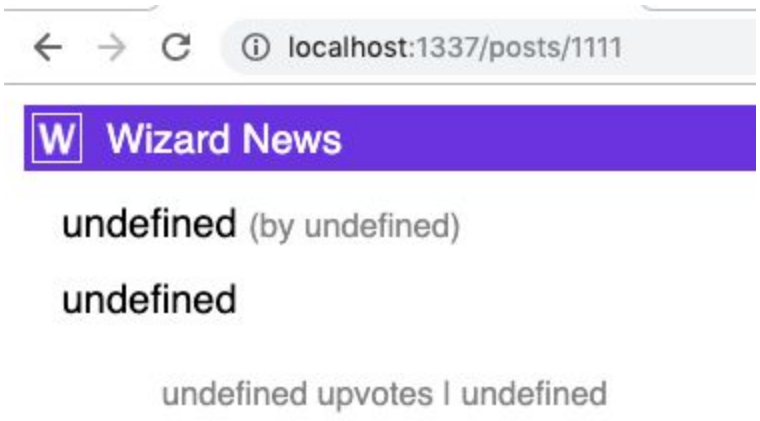
```
<a href="/posts/${post.id}">${post.title}</a>
```

Now we can click on post titles and view their details.

Custom Error Handler

We're in good shape! We can list all posts. When we click on a post, it takes us to the detail view of that post.

But let's suppose that a user mistypes the URL for a post. Go to http://localhost:1337/posts/1111 and you'll see something like this:



It would be better if users were given a friendly "404 Not Found" page. We've got a couple of different ways to accomplish this.

> **Option 1:** Check to see if `postBank.find()` returned an actual post and if not, send them a Not Found page instead of the post detail HTML.
> **Option 2:** Check to see if `postBank.find()` returned an actual post and if not, *throw an error*, to be caught by an Express error handler.
> **Option 3:** Check to see if `postBank.find()` returned an actual post and if not, *create an error*, and pass that error to the `next` callback to be handled by an Express error handler.

Option 1 would look something like this:

```
app.get('/posts/:id', (req, res, next) => {
 const id = req.params.id
 const post = postBank.find(id)
 if (!post.id) {
   // If the post wasn't found, set the HTTP status to 404 and send Not
Found HTML
   res.status(404)
   const html = `
   <!DOCTYPE html>
   <html>
   <head>
     <title>Wizard News</title>
     <link rel="stylesheet" href="/style.css" />
   </head>
   <body>
     <header><img src="/logo.png"/>Wizard News</header>
     <div class="not-found">
       <p>Accio Page! 🧙♀ ... Page Not Found</p>
       <img src="/dumbledore-404.gif" />
     </div>
   </body>
   </html>`
   res.send(html)
 } else {
 // ... Otherwise, send the regular post detail HTML
```

This approach works just fine. But it might get repetitive to have to handle errors separately for each individual route.

Option 2 would look something like this:

```
app.get('/posts/:id', (req, res) => {
 const id = req.params.id
 const post = find(id)
 if (!post.id) {
   // If the post wasn't found, just throw an error
   throw new Error('Not Found')
 }
 // ... Otherwise, send the regular post detail HTML
```

Try this out yourself. You'll notice a few things:

1. The server didn't shut down. It's still listening for requests. Phew!
2. The error was logged in the terminal, including a stack trace. That's useful.
3. The error is displayed in the browser, including a stack trace. That's... not so great.

We certainly don't want to send the server's stack traces to the browser. Not only does it make for bad user experience, but it may also pose a security vulnerability.

This is the built-in Express error handler at work. It's good to have some default error-handling middleware built into Express. Otherwise, a single bad request could shut down the server. But we may want to provide our *own* error handler.

Create an error handler, placed somewhere after all the other routes (e.g. just above `app.listen()`). It should respond with a 404 status code and some kind of friendly "Not Found" page.

Remember that error handlers are Express middleware, much like `morgan` or `express.static`. But they're special in that they take *four* parameters: `(err, req, res, next)`. Read the docs for some guidance.

Option 3 is particularly useful for catching asynchronous errors. We don't have any asynchronous code yet, since all of our data is stored in memory. We'll get a chance to use that `next` callback in a future workshop.

BONUS

Congratulations on reaching this far - you just went through all the basics in creating a server application using Express - from the basic setup, through middlewares and all kinds of routes (from a simple regular route to dynamic routes with parameters).

Over the next steps, you will further refine both the Wizard News application and refactor the codebase (so it's going be easier to maintain and expand this project with more features in the future).

Let's get started with some nice date formating, on the next page.

Date Formatting

Right now, the post dates are being displayed like this:

```
Wed Jan 03 2018 09:25:11 GMT-0500 (EST)
```

That's the default formatting applied to a date object in JavaScript when you convert it to a string, but instead, we want to display some nice, human-readable string with relative time - something like:

> Just Now
> A Few minutes ago
> An hour ago
> Yesterday
> And so on...

There are a few possible approaches to tackle this:

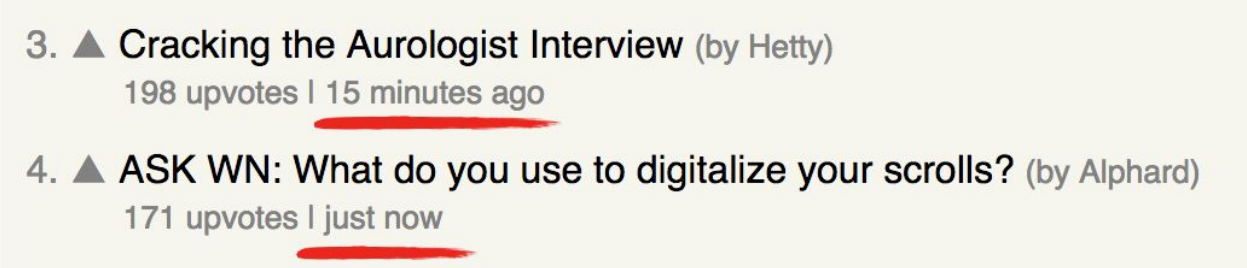
Do it yourself

1. The `getTime()` method on the `Date` prototype returns a numerical value corresponding to the date. The nice thing about having a plain numerical value is that you can make calculations with it ;)
2. Knowing how to convert dates into a numerical representation, you could create a new date with the current time get the difference (subtract) between right now and each post's times.
3. Knowing that the difference is in milliseconds, you can then use a few conditionals to establish the magnitude of difference and return the appropriate, human-readable wording.

...OR...

Knowing that working with dates is a very common task that developers do over and over again, instead of reinventing the wheel - what about checking npm for some existing module that does just that? We'll cut you some slack: Check Node time-ago

At the end, your list should look like this:



NOTES:

1. It is perfectly fine to "reinvent the wheel" for fun =]
2. Malicious packages can exist in any package distribution system manager for any language. The npm registry does a decent job of spotting and removing malicious code, but ultimately YOU are responsible for what you install. Make sure that any packages that you are installing are well known in the community and, when in doubt, inspect the source before you npm install it.

HTML Document String refactor

You just finished all functionality in Wizard News part 1 - we will keep working on this project in the future and add more routes and functionality. But before wrapping up, it's time for some housekeeping:

So far, you have two routes, but because we're producing entire html document string for each route, we already have around 80 lines of code in `app.js` - can you imagine how quickly this can get out of hand when we add a few additional routes and functionality?

As you know, Node provides a module system. Among other things, a module system can help developers organize code by splitting it into multiple files - and that's exaclty what you are going to do next: you're going to move the html document string generation to individual functions in their own modules. The goal is for your `app.js` to look like this:

```
const express = require("express");
const morgan = require("morgan");
const postBank = require("./postBank");
const postList = require("./views/postList");
const postDetails = require("./views/postDetails");

const app = express();

app.use(morgan('dev'));
app.use(express.static(__dirname + "/public"));

app.get("/", (req, res) => {
  const posts = postBank.list();
  res.send(postList(posts));
});

app.get("/posts/:id", (req, res) => {
```

Tagged Template Literals

We've been using ES6 template literals a lot throughout the whole curriculum and specially now in our HTML document templates. They're very useful because they allow us to write multi-line strings and do value interpolation (${expression}).

One feature that comes along with template literals and that we didn't use so far, is the ability to tag them.

Tagging a template literal gives additional control on how to interpret & process the template using any additional logic - it looks like this:

```
// Regular template literal

`Hello ${name}!`



// Tagged template literal



tag`Hello ${name}!`
```

By default, the JavaScript language doesn't provide any tags, but it gives developers the ability to create their own - consequently there are tons of tag functions available on NPM.

We're not going to dive into creating your own tagged template literals right now, instead we're going to install an HTML tagged template literal to help working with HTML templates.

First, install "html-template-tag"

```
npm install html-template-tag
```

The html-template-tag package adds the following functionality to the default template literals:

1. Autoescaping - Escaping is the process of converting values that will be interpolated to be properly displayed as plain text (turning angle brackets into < and >, for example) so they are not interpreted as tags. Interpolating values without escaping can lead to cross-site scripting (XSS) vulnerabilities.
2. Array joining - The html tagged template literal automatically detects when you're trying to interpolate an array, and will automatically join it without commas or spaces.

Finally, as a side-effect (that has nothing to do with the html-template-tag package in particular), some code editors will recognize when the tag in a template literal has the name of a recognizable syntax (like "html", "css", "sql" etc) and correctly apply syntax highlight and autocomplete:

Your turn:

Install, require and use the html-template-tag in your HTML document templates.

...

NOTE: EDITOR SUPPORT FOR SYNTAX HIGHLIGHT INSIDE TAGGED TEMPLATE LITERALS:

Editor	Supported
Atom	Yes (nativelly)

VSCode	No, but there are extensions available
Sublime	No, but there are some alternatives

Review

We've seen some more robust use of *server-side page generation* in this workshop using Express (running on Node.js). Concepts and techs covered include:

- `npm init` to generate a `package.json` for a new Node project
- installing Express and Morgan with `npm install`
- using `require` to import modules
- instantiating a new Express `app` with `express()`
- Express middleware like Morgan intercepts all requests
- registering middleware using `app.use`
- passing control to following middleware using `next`
- starting a server with `app.listen`
- using `nodemon` and a `start` script to standardize & streamline dev work
- Express routing definitions like `app.get` and `app.post`
- static routing from a file folder using `express.static`
- dynamic routing using route parameters (`req.params`)