



Welcome to Web Programming for Apps and Services

Welcome to the Web Programming for Apps and Services course. This document has information that helps you get started in the course.

Course introduction

In this course, you will learn to create web client (front end, in the browser) apps that work with a web service (back end, in the server). The apps will enable entry-level functionality, which can be hosted on-premise or in the cloud.

Throughout the learning process, you will learn foundational concepts, skills, and technologies that will enable you to create high-quality intermediate and advanced-level web applications in the future. These foundations will include:

- JavaScript
- Web API (web services) on a modern server stack (Node.js, Express.js, and MongoDB)
- The concept (and application) of front end web client development
- The React and Next.js libraries

How can you get started?

Get familiar with the course content online.

Using your own personal computer

The student will use a number of applications and development tools, including:

- A modern web app execution environment (Node.js, Express.js, MongoDB)
- Visual Studio Code
- Various code generators

During the course, the professor(s) will guide the student in the installation, configuration, and use of the software.

Please be aware of the following:

If you have problems or difficulties using your own personal computer for course work, your professor will not be able to provide technical support. In a problem scenario, you are still expected to complete your work on time. A problematic personal computer *cannot* be used as an excuse for delays in completing the course work.

How to use these course notes

Every class/session will reference the notes posted here

Before you come into a class, you are expected to read and process the topics covered in the notes.

The format and style of the notes pages will vary. At times, they will be terse, with headings and keywords that are intended to guide the student through the topics. At other times, they will be lengthy, with narrative that explains and supports the topics. Expect a full range of formats and styles between these extremes.

Class/sessions are important. The notes do not attempt to capture everything that must be communicated in the process of learning a topic.

What do we expect from you?

Before the class in which there's a test, we expect you to prepare for the class. This means:

- Read and study the notes
- Read and study the linked documents
- Make your own notes, including questions that you have

In other words, do not come into the classroom expecting somehow to soak up knowledge like a sponge. You need to prepare before class, so that you understand the topics and their context.

In the Lecture class, we expect you to be an engaged and actively-learning participant. This means:

- Listening effectively
- Asking and answering questions
- Writing notes
- Doing the in-class exercises and activities

Before the lab session of the week, we also expect you to prepare for the class.

This means:

- Being prepared to split your time between new topic learning, and working hands-on with the topic or the current assignment
- Asking and answering questions
- Writing notes
- Read and study the current assignment
- Practice its contents, and/or get started on its contents

Regarding the workload, it will simply not be possible to confine this course's learning experience to the scheduled class time. We expect you to spend some of the in-class time working on the assignments, but you must spend time out-of-class to complete the work.

That being said, you will encounter problems and delays. Please follow a general rule: If you cannot solve the problem within 20 to 30 minutes, then stop and set it aside. Seek help from your professor, during class time, or during the designated help time or office hours. Alternatively, seek help from a classmate *who knows the solution* to the problem.

Do not waste time. Do not attempt to wrestle the problem to the ground. Others will not think any less of you when you ask for help. You're here to learn, so take advantage of the course's resources and delivery to help you learn.

Developer tools used in this course

This document is a summary of the developer tools used in this course.

Dev tools summary

Some of the dev tools will be graphical user interface (GUI) apps that run on the *base* (device/host) operating system (including web browsers). Some will be command line interface (CLI) apps.

The following table shows the tasks to be done, and the apps that enable you to do the tasks, on macOS or Windows.

| Task | macOS | Windows |
|-----------------|------------------------|--------------------------|
| File system GUI | Finder | File Explorer |
| File system CLI | Terminal | Command Prompt (CMD) |
| Editor | Visual Studio Code | Visual Studio Code |
| Node.js, Git | Install these on macOS | Install these on Windows |
| Web | Chrome, Firefox, | Chrome, Firefox, |

| Task | macOS | Windows |
|--|--|--|
| browser(s)including dev tools,debugger | Opera, Safari | Opera, Edge |
| HTTP inspector | Visual Studio Code Extension: Thunder Client | Visual Studio Code Extension: Thunder Client |
| Data generator | mockaroo.com online | mockaroo.com online |

Dev tools usage notes

As you can see from the summary, you will be using GUI versions of the following apps. Each is an app that is designed for the base operating system.

- File system
- Code editor
- Browsers

Using terminal windows

Do not hesitate to use multiple terminal windows. During development, your professor typically uses a minimum of three, but often about five are opened. Each is opened at a different folder, and therefore is used for different purposes:

- One or more is focused on the parent folder of the current app
 - Used to create new apps and to run general commands

- Another is focused on the current app itself
 - It's used to create new files and to run app-specific commands (e.g. `npm start`)

Creating folders on macOS

In your Documents folder, create a folder to hold your apps, maybe named `dev`. (Then inside that folder, each app will be in its own folder.)

Creating folders on Windows

Using File Explorer, create a new folder (maybe named `dev`) in the root of drive C:. In other words, `c:\dev`. Inside that folder, we will be creating separate multiple apps (web APIs, React / Next.js apps, etc.).

Deleting old or unneeded app/project folders

As you know, when you delete a folder (using Finder or File Explorer), the folder is just "marked" as deleted, and is then managed by the operating system's Trash folder (Unix) or Recycle Bin folder (Windows). Later, you can "empty" the Trash folder, which actually and permanently deletes the contents.

Why does this matter to us? Well, a typical Node.js + Express.js, or React / Next.js app has thousands of files and is hundreds of megabytes in size. If you create and then discard five (for example) apps per week, then a month later, you have a huge amount of wasted storage space, which takes a long time to actually delete (it can be minutes).

Therefore, if you want to immediately and permanently delete an old/dead/unneeded app folder:

- On macOS Finder, use Option+Command+delete (instead of delete on its own)

- On Windows File Explorer, use Shift+delete (instead of delete on its own)

Web Services

Introduction

The goals of this short introduction are as follows:

1. Introduce the idea of a web service, and cover the relevant terminology
2. Explain the roles of the requestor (the client) and the responder (the server)
3. Discuss how a client uses an HTTP client object (e.g. XMLHttpRequest, or the fetch API) to contact a server, and handle (typically) JSON data responses
4. Promote the idea that we are building and working with a *distributed* computing system, that has two or more autonomous programs that pass messages (requests, responses) among the programs

What is a web service?

A web service is an application that runs on a web server, and is accessed programmatically.

When we parse this short sentence, and consider the meanings behind the simple words, we reveal some very important concepts:

- HTTP is the protocol, enabling wide use and scalability
- Humans don't use a web service directly - instead, the *application they are using* creates and sends a request to the web service, and handles the

response in a way that's meaningful to the application

- A web service's functionality is discoverable, and typically known as an application programming interface (API)

Web services can be developed on any web-connected technology platform, in any language, and can be used on any kind of device.

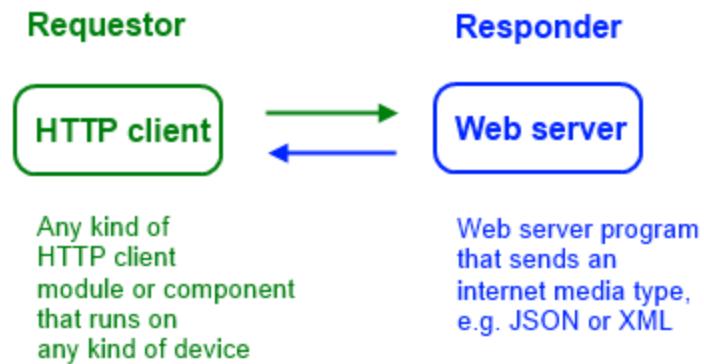
What's the difference between a *web app*, and a *web service*?

Study this diagram to understand the differences, and then be prepared to explain them to someone else:

Web app:



Web service:



Give me a brief history lesson on web services

With the rise of the web's use and popularity in the 1990s, efforts were made to define and specify web services.

This led to the *de facto* standardization of **SOAP XML web services**. Often described as "big web services", or "legacy web services", SOAP XML web services are the implementation of **remote procedure calls** on the web. This kind of web service typically has one specific endpoint address, and requestors

must create and send a data package (which conforms to a data format named SOAP), and then process the response (which also conforms to SOAP).

However, other efforts took advantage of the web and its existing features and benefits. In other words, they simply followed the [HTTP specification](#) and its *ex post facto* architecture definition, to create true and pure web services. These kinds of web services, often termed "Web API", "RESTful web services", or "HTTP services", exploded in use and popularity from about 2005 onwards, and are now the preferred design approach.

Web services development and deployment environment

This course will use the following frameworks and tools:

- Node.js
- Express.js
- MongoDB
- Visual Studio Code

Each web service that you create will be treated as a *separate and distinct* program, and not bundled in with the front end client (browser) app that you may be working on. It will be concerned only with listening then responding to requests that come in.

During development, you will use an *HTTP inspector* app, like [Thunder Client](#), to create and send requests to a web service.

| [Thunder Client Documentation](#)

Then, you will be creating *separate* web app *clients*, which send requests to your web service.

Each program can be individually and separately deployed to any device platform that meets your needs.

Terminology

As your web programming competency grows, it is important to know and be able to correctly use some common web programming terms.

Web app, and web service

If you create a server-based program, and it is intended to be used mostly by web browsers, we typically call that a *web app*.

Alternatively, if you create a server-based program, and it is intended to be used programmatically (by iOS and Android apps, or by JavaScript in a browser, or by an "HTTP client" module in a native Windows, macOS, or Linux app, we typically call that a *web service*.

Resource

A *resource* is a *digital asset*.

Familiar examples include a document, or an image.

How do you identify a resource? By using its URI (uniform resource identifier). The URI standard is described in [RFC 3986](#), and also in a [Wikipedia article](#).

What is the format, or representation, of the resource? Well, it depends on the design of the web service, and sometimes the needs of the requestor.

Representation

As defined above, a resource is a digital asset.

A *representation* is a *digital asset* that is *formatted as a specific internet media type*.

The concrete or real form of a representation "is a sequence of bytes, plus ... metadata to describe those bytes".

From Roy Fielding, "Architectural Styles and the Design of Network-based Software Architectures", [section 5.2.1.2](#)

Think about a scenario where a web service was used to manage students in a course. Each student is a resource - a digital asset - that can be identified by a URI.

If a user requested a specific resource through a web browser, you would expect that the resource would be *represented* by some HTML that included the student's name, student ID, and so on.

Alternatively, it's also possible to request the same specific resource - using the same URI - but also specify that it be returned in a data format (like JSON or XML, discussed later). The server will return a data *representation* of the resource.

Or, maybe the request specified that the student's photo be returned as the resource's *representation*. Again, the same URI is used.

So, in summary, a resource's representation can vary to meet the needs of the web service programmer or the web service user. The requestor and responder use a feature called [content negotiation](#) to make this happen.

Every representation is defined by an internet media type.

Same resource, same URI

Different representations

As HTML, or a fragment of HTML, often used in a browser

```
<div class=student>
  <h3>Peter McIntyre</h3>
  <p>Student ID: 012-345-678</p>
  <p>Age: 29</p>
</div>
```

As data in a specific format, e.g. JSON (shown) or XML

```
{
  "name": "Peter McIntyre",
  "studentId": "012-345-678",
  "age": 29
}
```

As an image



Internet media type

An *internet media type* is defined as a *data format* for a *representation* of a *resource* on the internet.

The data formats are standardized, **published**, and well-known, by the IANA (the Internet Assigned Numbers Authority).

[This Wikipedia article](#) is an acceptable introduction to internet media types.

For web service programmers, two important internet media types are used as

data formats, [JSON](#) and [XML](#).

Both are plain-text data formats. They are somewhat human-readable.

Later, you will probably learn how to work with non-text media types. From now on, we will typically work with plain-text JSON.

Get started with JSON

JSON is an initialism for [JavaScript Object Notation](#).

It is a lightweight data-interchange format. It is language-independent, however it uses conventions that were first suggested by the JavaScript object literal or initializer. JSON has become the *de facto* data-interchange format standard.

Here's an [overview of JSON](#) from Wikipedia.

Here's the [official web site for JSON](#), by Douglas Crockford.

Although JSON is historically derived from JavaScript object literals, there are a few notable differences to programmers who are new to JSON:

- In JSON, each property name must be surrounded by quotes (typically double-quotes). In pure JavaScript, this is optional for single-word property names.
- In JSON, there is no Date type. Dates are expressed as strings, almost always in ISO 8601 format. In contrast, JavaScript does have a Date object (not a *type*, but an *object*).

While we're on the topic of data types, the property values will be any of about five types:

1. string
2. number (integer or decimal)
3. object (i.e. {})
4. array (i.e. [])
5. null

String values must be surrounded by quotes.

The web services that you create in this course will rely on the JSON internet media type.

State

Let's discuss *state* in this section of the notes, and in the next section.

Data

In this section, we are interested in the state of the *data* elements in a distributed system. (In the next section, we will be interested in the state of the *interaction* among programs in a distributed system.)

In a web app or web service context, the meaning of the word *state* is the *current value of a resource*.

At the server, a resource could be in the form of an item in a persistent store (i.e. a database system), or it could be in the form of an item that is generated upon request. That difference doesn't matter; what matters is that when a resource is requested, its current state is sent, or *transferred*, as the response.

A resource's state can be *changed* (or modified), of course. The stimulus for the change can originate from anywhere in the distributed system, ranging

from server-based batch or automatic processes that modify the persistent store (apart from or separately from the web app or web service), through to client-sent requests that are received by a web app or web service.

Does this work the other way, when a client app changes the state of an resource?

Yes.

A client can send a request that includes a *representation* of a resource. (For example, an HTTP POST or PUT request.)

The new or updated *state* of the resource is *transferred*, from the client app to the server (web app or web service).

Upon acceptance (and validation etc.), the server appends to or updates the resource in the persistent store.

Interaction

In this section, we are interested in the state of the *interaction* among programs in a distributed system. (In the previous section, we were interested in the state of the *data* elements in a distributed system.)

One of the characteristics of a web app or web service is that it can be used by a single client app, or by theoretically unlimited numbers of client apps.

Does the server keep track of the interaction state with each client? No. This responsibility is borne by the client app. This design feature is one of most important parts of the web.

"Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is kept entirely on the client." (Roy Fielding, PhD thesis, [section 3.4.3](#))

The main point is that the server effectively treats every request as separate/

discrete/atomic, and *does not* actively maintain any notion of a logical session over time. In other words, no interaction state maintenance or management is done at the server.

Therefore, if the interaction state of a (message exchange) session is important to maintain, it's done by the client.

REST

Now we can circle back to REST. It is an acronym for **R**epresentational **S**tate **T**ransfer.

The implementation / description of a "REST API" can be simply described "*as a way to use the HTTP protocol with a standard message format to perform operations on data*". However, while that explanation is clear and understandable when first learning about REST, it embeds a much deeper understanding that can come only through more learning and experience. If a student is serious about a career that includes web programming, then it is essential to study Roy Fielding's PhD dissertation from the year 2000, [Architectural Styles and the Design of Network-based Software Architectures](#).

More information about [Roy Fielding](#) is here.

Will we be creating "REST APIs"? Yes, if we follow all the guidance and practices. However, that will come only with more knowledge and experience.

As a result, it may be more accurate and safer to use the term "web API" when naming or describing our web service work.

Web API

The term "web API" captures the essence of "REST API" above, but it uses a

more generic and easier to understand or explain word. Arguably, every computer user has a reasonably accurate knowledge of the word "web", so that word doesn't have to be explained. The expansion of the "API" initialism is also easy to understand or explain. It's just a bit better (and safer) than using "REST", and avoids the immediate need to cover large chunks of a [180-page PhD dissertation](#).

So, let's define "web API" as *an API to a web app or web service*.

In summary

In this document, we have defined and explained a web service.

Terminology was introduced, argued, and explained.

Now, you're ready to take the next step, and create web services that support the main focus of this course, which is front end client-side browser apps.

Creating and Testing a Web Service

The theme of this document is to design and create a simple web service.

The web service will deliver data to requestors. The data will be materialized in the app (i.e. as JavaScript objects). A future "version 2" web service will use a database.

Confirm that your tooling is ready

Before writing and running the app, confirm that your computer's tooling is ready.

If your system does not yet have the Node.js app dev ecosystem

Install Node.js (which also installs npm). Verify its status:

```
node --version  
npm --version
```

If your system does not have the developer tools

1. Install git
2. Install more browsers (assume that Safari is already there; add Chrome,

Firefox, and Opera)

3. Install Visual Studio Code (aka VS Code)

Create a project to hold the web service

Using Terminal, navigate to the file system location that will hold the project. Create a new folder to hold the project.

Navigate into that folder. Create an empty `server.js` file. Create an empty `index.html` file. It's also a good idea to create an empty `README.md` file.

Now, initialize the folder as a Node.js app:

```
npm init
```

Make sure you specify `server.js` as its entry point. Add your name as the author name, and add a description if you wish. The other settings can stay at the suggested default values.

Add Express.js:

```
npm install express
```

Now, edit the project.

```
code .
```

Open the `server.js` file for editing. Make it do something. For example:

```
console.log('Hello, world!');
```

Back in Terminal, run the app:

```
node server.js
```

It should respond with the console message, and then terminate.

Write a simple web server

Our goal is to create an app that will handle these requests:

- Get all
- Get one
- Add new
- Edit existing
- Delete item

This goal will work for a web service that handles ANY kind of data. Obviously, a more complex data model will have more request handlers, but they all share the same core design, and handle these five - or variants of these five - requests.

The core getting-started code looks something like the following.

```
// Setup
const express = require('express');
const path = require('path');
const app = express();
```

Edit `server.js`

Edit `server.js` so that it holds the code shown above.

Make a simple home page (HTML)

Edit `index.html`. Use the `html:5` Emmet snippet to help with this task.

Run the app

In Terminal, run the app.

Testing our API using the Thunder Client Extension

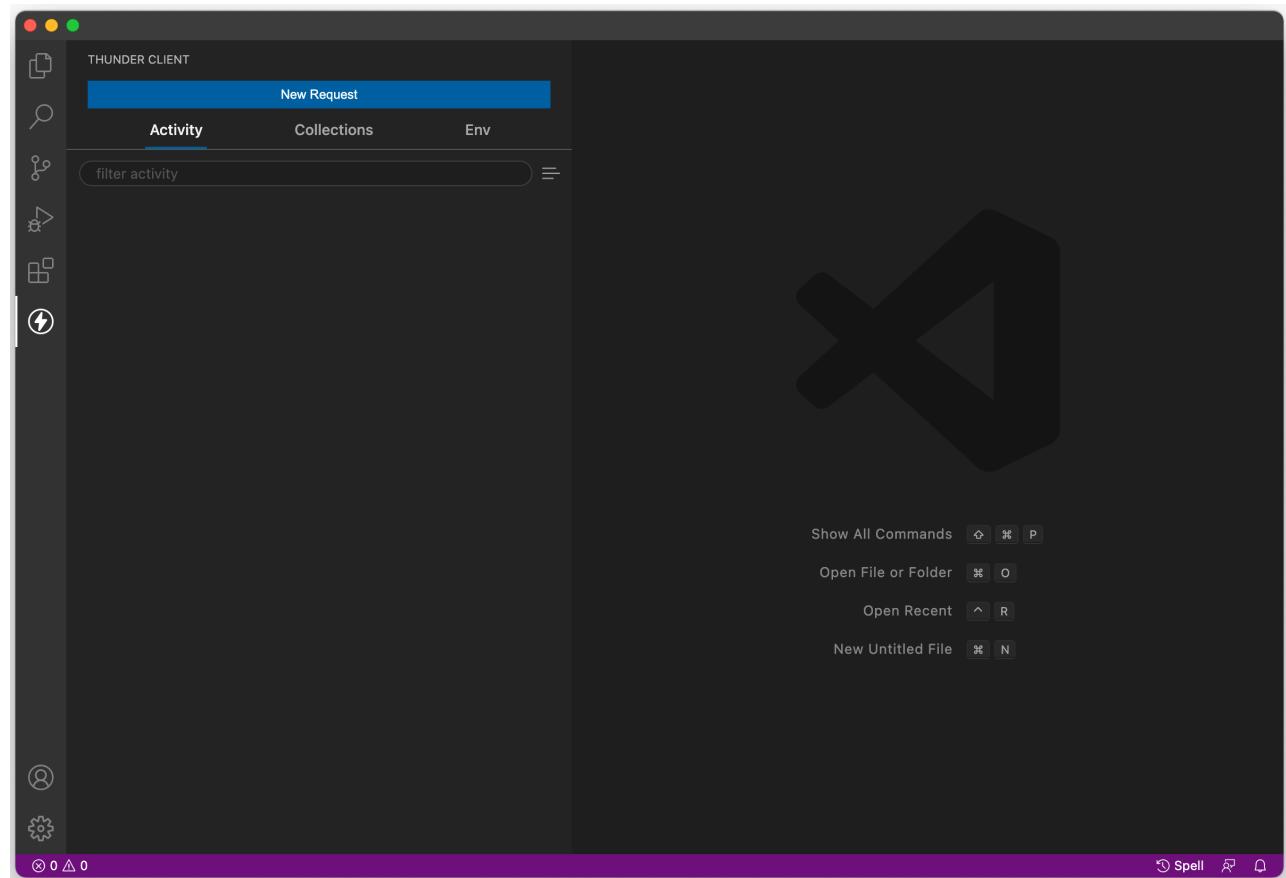
Thunder Client is a Visual Studio Code extension that enables web programmers to interact with a web service (aka web API) without the need to install additional software.

This kind of extension enables us to focus on web service creation - which is a server-focused development effort - without the additional burden of creating a separate client/requestor app that interacts with the web service. It *is* the client app as we test and interact with the web service.

Its user interface gives us the ability to "compose" a request. Then, we can send the request. Finally, we can inspect the response. The extension saves a history of requests, making it easy to see that you're making progress.

Start Up

After installing the extension, you can get started by clicking on the new "thunderbolt" icon in the left sidebar. This should show you the following screen, with the option to create a new request.



Important features

Once you click the "New Request" button, you will notice that the screen is populated with additional information. For beginners, there are two important features:

1. Request and response areas - enables you to inspect the contents of both

the request and the response (right pane)

2. List of past requests - saves all requests as a history under the "Activity" tab (left pane)



What is an HTTP request?

As you know, HTTP is a message-passing protocol, between two endpoints. One end will send a request, and the other will send a response.

The requester can be a browser (which is what all students have experience with), or it can be a component in an app. For example, almost every smartphone app includes a component - generically known as an HTTP client - that enables it to communicate with a web service.

HTTP defines several kinds of requests. At a minimum, a simple "get me some data" request must include the *HTTP method* (e.g. GET, POST, etc.), and a *URL*. Other kinds of requests must include other metadata.

Remember, [RFC 7231](#) is the authoritative resource for the HTTP protocol.

Compose a request - GET

To compose an HTTP GET request, two settings are required:

1. HTTP method, GET
2. URL of the collection or object

Other settings may be required (e.g. security-related info, etc.), and we'll see those in the future.



The screenshot shows the Postman application interface. The title bar says "tc New Request X". The main area has a "GET" dropdown and a URL input field containing "https://reqres.in/api/users?page=2". A blue "Send" button is to the right. Below this, there are tabs for "Query", "Headers 2", "Auth", "Body", and "Tests". The "Query" tab is selected. Under "Query Parameters", there are two entries: "page" with value "2" and "parameter" with value "value".

Compose a request - POST

A POST request is designed to enable the requestor to "add" a new item to the collection that is represented by the URL. Obviously, when compared to a GET request, more is required.

To compose an HTTP POST request, four settings are required:

1. HTTP method, POST
2. URL of the collection
3. A header that defines the content type of the data we're sending
4. Data (the new item)

As above, other settings may be required, but these will enable you to get started.

The image below shows a simple post request, with metadata for the first three settings. Notice that you must select the "Headers" tab to get to the data-entry panel. The header we want is `Content-Type`, and its value will be `application/json`, because that's what we're sending

The screenshot shows the Postman interface with a dark theme. At the top, it says "TC New Request X". Below that is a search bar with "POST" and a URL field containing "https://reqres.in/api/users". To the right of the URL is a blue "Send" button. Underneath the search bar are tabs: "Query", "Headers 1", "Auth", "Body", and "Tests". The "Headers" tab is currently selected and highlighted in blue. Below the tabs, there is a section titled "Http Headers". It contains two entries: one with a checked checkbox labeled "Content-Type" and a value "application/json", and another with an unchecked checkbox labeled "header" and a value "value". There is also a "Raw" checkbox. On the far right, there are three dots (...).

The image below shows how to enter metadata for the fourth setting. Select the "Body" tab and then in the text entry area, add the JSON that correctly defines a new item for the collection.



More about Thunder Client

The official help docs on Github will enable you to become more comfortable using the extension:

[Getting started](#)

Add some data (simple arrays)

You obviously noticed that the server's functions were not really handling any data. The requests and responses were simple strings. We'll change that now, and use real data.

We have to do two tasks:

1. Add some data

2. Edit the functions

Add simple string data

Let's add a super-simple array of strings, for example the names of colours. Somewhere in the `server.js` file, create a variable to hold the data, something like this:

```
// Array of strings
let colours = ['Red', 'Green', 'Blue', 'Yellow', 'Aqua',
'Fuschia'];
```

Change the "get all" function, and this time return the data.

What about the other functions? Yes, they need some work.

If we assume that `itemId` is the array element or index, then we can code a "get one". For example, the function body looks like the following:

```
// Extract the item identifier
let itemId = req.params.itemId;
// Make sure it's valid
if (itemId > colours.length) {
  res.status(404).send('Resource not found');
} else {
  res.json(colours[itemId]);
}
```

Next, if we assume that we pass in a simple JSON object with one key-value pair (and the key name is "colourName"), then we can code an "add new". For example, the function body looks like the following:

```
// Extract the incoming data from { "colourName": "Purple" }
let newItem = req.body.colourName;
// Add another item to the array
colours.push(newItem);
// Return the result; RFC 7231 tells us that it must return HTTP
status 201
res.status(201).json({ message: `added ${newItem} as item
identifier ${colours.length}` });

```

Add some data (objects)

We will leave this as an in-class hands-on task (in our computer-lab session).

Generate a substantial amount of data

One of the new skills that a web programming student should add is the ability to generate and use a large amount of data. This is especially important for web programmers, because the result of the work we do is so visual in nature. It is always a good idea to use and show good solid credible data, instead of the simple placeholder data (e.g. abc, 123, foo, bar, etc.) that is so common in entry-level programming work.

Here, we will introduce you to, or remind you about, the [Mockaroo service](#):

Need some mock data to test your app?

Mockaroo lets you generate up to 1,000 rows of realistic test data in CSV, JSON, SQL, and Excel formats.

Problem solved.

Generate some data. Make sure you configure enough fields to give you a rich variety of data. For the professor's in-class demonstration example, here are a

few notable choices we made:

- Birth date was within a range and in ISO 8601 format
- Credit score was an integer within a range
- Rating was a number within a range with two decimal places
- 150 rows were returned
- Data format was JSON

Here's a screen capture. (Right-click and then open it in a new tab/window to see it full size.)

Need some mock data to test your app? Mockaroo lets you generate up to 1,000 rows of realistic test data in CSV, JSON, SQL, and Excel formats.

Download data using your browser or sign in and create your own [Mock APIs](#).

Need more data? [Plans start at just \\$50/year](#).

| Field Name | Type | Options |
|-------------|---------------|--|
| id | Row Number | blank: 0 % fx x |
| firstName | First Name | blank: 0 % fx x |
| lastName | Last Name | blank: 0 % fx x |
| gender | Gender | blank: 0 % fx x |
| birthDate | Date | 1/8/1992 to 1/8/2000 in ISO 8601 (UTC) blank: 0 % fx x |
| email | Email Address | blank: 0 % fx x |
| web | URL | include: <input checked="" type="checkbox"/> protocol <input checked="" type="checkbox"/> host <input type="checkbox"/> path <input type="checkbox"/> query string blank: 0 % fx x |
| creditScore | Number | min: 200 max: 800 decimals: 0 blank: 0 % fx x |
| rating | Number | min: 1 max: 20 decimals: 2 blank: 0 % fx x |

[Add another field](#)

Rows: Format: [JSON](#) [array](#) [include null values](#)

Hint: Use "." in column names to generate nested json objects, brackets to generate arrays. [More information...](#)

[Download Data](#)

[Preview](#)

| [More](#)

Want to save this for later? [Sign up for free](#).

Use the data in your web service

The data comes to your system as a standard download. Open it for editing.
Copy it.

In your code, similar to what you did above, assign the JSON as the value of a new variable.

```
let people = // your pasted JSON goes here
```

Don't forget the statement's trailing semicolon.

Now you have a rich body of data, with which you can do the standard get all, get one, add new, edit existing, and delete item tasks.

Update the code in the functions

Obviously, the code in the functions needs updating, to work with the new data.

Get all is easy to fix.

If you want sorted data - for example by last name then first name - more work needs to be done. The built-in JavaScript array `sort()` function mutates the existing array. Maybe that's not such a good idea. Therefore, before sorting, copy the array to a new array, with something like this:

```
let c = data.map((p) => p);
```

Then, the "c" array can be mutated, with something like this:

```
c.sort(function (a, b) {
  return a.lastName.localeCompare(b.lastName) ||
  a.firstName.localeCompare(b.firstName) || 0;
});

// or, using arrow function syntax...
c.sort((a, b) => {
  return a.lastName.localeCompare(b.lastName) ||
```

Finally, the sorted "c" array can be returned.

Do you want to sort an array of numbers?

The "compare" is an arithmetic subtraction compare.

Look at the [MDN docs](#) for more, but briefly:

```
let numbers = [4, 2, 5, 1, 3];
numbers.sort((a, b) => a - b);
```

Or, if the numbers were in a *property* of an array of objects:

```
// assume the array "c" of people/person objects
// and we want to sort on credit score
c.sort((a, b) => a.creditScore - b.creditScore);
```

Get one is also relatively easy to fix.

Compared to the "colours" array above, where it was easy to "find" the value we wanted, this "people" array has *objects* as elements in the array. We want to support finding a matching identifier (i.e. the "id" field).

This needs a function (an arrow function, specifically). For example:

```
let o = data.find((p) => p.id == itemId);
```

Then, if "o" is "undefined", return HTTP 404. Otherwise return the object "o".

You can do **the others** on your own (i.e. add new, edit existing, delete item). Doing this will remind you of your work with JavaScript arrays, and enhance those skills.

Looking for a more challenging task?

Try adding another function to handle a request for people with a high credit score (e.g. a `creditScore` value over 600). We can suggest the `filter()` function can help with that.

Try adding another function to return only the full names of the people objects. We suggest the `map()` function can help with that. (Bonus, sort the results.)

Try adding another function that will enable a *find* by last name (case insensitive).

Doing these additional tasks will prepare you for more interesting work with data and its transformations.

Enjoy!

Bootstrap Introduction

You have likely worked with Bootstrap before, either in previous courses or for your personal or professional projects. It is an extremely popular front-end toolkit used to build websites / user interfaces.

Here, we will be discussing [Bootstrap 5.1](#) - this is important because Bootstrap 5 is the first version to [drop support for Internet Explorer](#) as well as no longer require [jQuery](#) as a dependency.

NOTE: The full [Bootstrap 5.1 Documentation](#) provides a ton of great resources expanding on what is discussed below. The following documentation describes enough of the core components to get started and build a simple user interface only. Please refer back to the original documentation for examples and guides on how to use the **many** other great UI components provided by Bootstrap 5.1 not mentioned here.

Including Bootstrap 5

The simplest way to include Bootstrap in our projects is to add the CDN links directly in our HTML files. Alternatively, you can download the [Source Files](#) and [customize](#) them yourself using Sass, or use a package manager (ie: npm) to [install Bootstrap](#).

For simplicity, we will include the CDN links directly in our HTML document, ie:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <!-- Bootstrap 5.1 CSS-->
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
      rel="stylesheet"
      integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
      crossorigin="anonymous"
    />

    <!-- Bootstrap 5.1 JS Bundle -->
```

Grid System

Arguably one of Bootstrap's best features has been the responsive utilities and the responsive grid system. Once you have planned the layout for your application, this is typically the first place to start placing UI elements and organizing your layout.

If you have used Bootstrap before, the core concept should be familiar: Place your main content within a "container" element, each logical row of your layout within a "row" element and each column within a "col" element, for example:

```
<div class="container">
  <div class="row">
    <div class="col">Column</div>
    <div class="col">Column</div>
  </div>
</div>
```

The above code will render a single "row" with two columns of equal width within a "container". The width of the columns will always remain side-by-side and of equal width regardless of the size of the browser window.

Notice how we did not have to specify the number of "columns", each div takes up (from a total of 12). Since we have only 2 columns in the above example, Bootstrap automatically assigns each column a width of "6" (totaling 12 columns). If we wish to change this behaviour and have one of our columns shorter or longer than the other, then we can use the familiar "column-counting" pattern, ie:

```
<div class="col-3">Column</div>
<div class="col-9">Column</div>
```

Using the above code, you can see that the 2nd column takes up 9 (of 12) columns, while the other column takes the remaining 3.

NOTE: To achieve two equal width columns, two "col-6" classes would work as well.

You will also notice at certain points the layout "jumps" and the width of both columns are either *increased* or *decreased*, while remaining in the center of the container. This is because bootstrap 5 uses **6 Responsive Breakpoints**, essentially representing 6 typical viewport / device sizes. From

the documentation, you can see that these breakpoints (the places that the layout "jumps" when resizing) fall at the following viewport widths:

| Breakpoint | Class infix | Dimensions |
|-------------------|-------------|------------|
| X-Small | None | <576px |
| Small | sm | ≥576px |
| Medium | md | ≥768px |
| Large | lg | ≥992px |
| Extra Large | xl | ≥1200px |
| Extra Extra Large | xxl | ≥1400px |

From the above table, we can see that Bootstrap 5 identifies a viewport width of anything less than 576px as "X-Small", while everything between 576px and 767px is considered "Small" and so on. As we have done in previous versions of Bootstrap, we can use the "infix" value with a specified class name, to target certain viewport sizes in order to change the layout of our grid.

For example, say we wish for our two-column "3, 9" layout from above to change its column widths at a specific viewpoint. A common, simple case is to have the columns "collapse" into a single column at the "Small" size, thus making the UI easier to read on mobile browsers.

To achieve this using the above table for reference, we can use the following code to specify that we only wish our columns to remain at the "3,9" layout for sizes "Medium" and above:

```
<div class="col-md-3">Column</div>
<div class="col-md-9">Column</div>
```

Additional Options

The above code will give us all of the flexibility that we require for simple layouts, however Bootstrap 5 offers additional options such as:

- **Mix and Match** - The "col-" classes can be applied multiple times to a div, allowing finer control over how the div will display within the grid at different viewport widths.

- **Row Columns** - Used to specify how many rows should be created when multiple "col" classes are used, allowing columns to carry over to additional rows
- **Nesting** - As in previous versions of Bootstrap, columns can be nested depending on your layout needs

Navigation Bar

Another common starting point for any application is to design and code a navigation bar to make traversing your app simple and intuitive. Once again, Bootstrap provides an excellent starting point to create a clean, *responsive* navigation bar. To begin, you only need a single <nav> element (typically placed **above** our main "container") containing a "container-fluid":

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container-fluid"></div>
</nav>
```

Notice how our <nav> element has a number of classes added to it. Each of these classes provides the following look / functionality:

- **navbar** - This is the main class defining / styling the navigation bar
- **navbar-expand-lg** - This class is used to set the viewport minimum width for a full sized navigation bar. Anything smaller than the "lg" size (992px) will cause the content of the navigation bar to collapse into a sub menu with a "**Hamburger**" **Icon**
- **navbar-light** - This controls the colour / brightness of the items on the navigation bar. "navbar-light" will give you darker text (to be used on a lighter background), whereas "navbar-dark" will give you lighter text (to be used on a darker background)
- **bg-light** - This controls the colour / brightness of the background of the navigation bar. "bg-light" will give you a light coloured background, whereas "bg-dark" will give you a darker one. You may also use one of Bootstrap's predefined theme colours, ie:
 - "bg-primary"
 - "bg-secondary"
 - "bg-success"
 - "bg-danger"
 - "bg-warning"

- "bg-info"

Navigation Items

All of the various items placed on the navigation bar will be located within the "container-fluid" <div>. Why don't we expand our navigation bar to contain the items listed in the Bootstrap documentation:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button
      class="navbar-toggler"
      type="button"
      data-bs-toggle="collapse"
      data-bs-target="#navbarSupportedContent"
      aria-controls="navbarSupportedContent"
      aria-expanded="false"
      aria-label="Toggle navigation"
    >
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav me-auto mb-2 mb-lg-0">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="#">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Link</a>
        </li>
        <li class="nav-item dropdown">
          <a
            class="nav-link dropdown-toggle"
            href="#"
            id="navbarDropdown"
            role="button"
            data-bs-toggle="dropdown"
            aria-expanded="false"
          >
            Dropdown
          </a>
          <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
            <li><a class="dropdown-item" href="#">Action</a></li>
            <li><a class="dropdown-item" href="#">Another action</a></li>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

This is a great starting point for building your own navigation, since it contains starter code for many of the elements that you may wish to use in your own navigation bar, such as:

navbar-brand

```
<a class="navbar-brand" href="#">Navbar</a>
```

This is the large text appearing at the left of the navigation bar (typically the title of the site / app)

navbar-toggler

```
<button  
  class="navbar-toggler"  
  type="button"  
  data-bs-toggle="collapse"  
  data-bs-target="#navbarSupportedContent"  
  aria-controls="navbarSupportedContent"  
  aria-expanded="false"  
  aria-label="Toggle navigation"  
>  
  <span class="navbar-toggler-icon"></span>  
</button>
```

This is the "Hamburger" button that will toggle the visibility of the navigation items (see "collapse" below) if the viewport width falls below the "navbar-expand-' value" (in this case it is "navbar-expand-lg" - from above).

collapse

```
<div class="collapse navbar-collapse" id="navbarSupportedContent">...</div>
```

This `<div>` is the container for the main navigational elements of our navigation bar. Notice the "id" value matches the "data-bs-target" property of the "navbar-toggler", allowing this container to be shown/hidden if the "Hamburger" button is toggled.

navbar-nav

```
<ul class="navbar-nav me-auto mb-2 mb-lg-0">  
  ...
```

Nested within the "collapse" container (above) is the unordered-list defining all of our navigation items. The additional classes added to the element are to help with the **spacing**.

nav-item

```
<li class="nav-item">...</li>
```

These are all of the actual links contained within the navigation bar. They may take the form of either:

- <a> elements with the class "nav-link" (optionally "active" for the current element)
- <a> elements with the class "nav-link" and "dropdown-toggle" - to be used to create a "dropdown" menu within your navigation bar. The full code for the dropdown looks like:

```
<a  
  class="nav-link dropdown-toggle"  
  href="#"  
  id="navbarDropdown"  
  role="button"  
  data-bs-toggle="dropdown"  
  aria-expanded="false"  
>  
  Dropdown  
</a>  
<ul class="dropdown-menu" aria-labelledby="navbarDropdown">  
  <li><a class="dropdown-item" href="#">Action</a></li>  
  <li><a class="dropdown-item" href="#">Another action</a></li>  
  <li><hr class="dropdown-divider" /></li>  
  <li><a class="dropdown-item" href="#">Something else here</a></li>  
</ul>
```

form

```
<form class="d-flex">  
  <input class="form-control me-2" type="search" placeholder="Search" aria-  
  label="Search" />  
  <button class="btn btn-outline-success" type="submit">Search</button>  
</form>
```

Finally (outside of the "navbar-nav" unordered list), It is possible to have <form> elements nested

within the navbar. This may be used if we wish to implement search or login functionality, for example.

Buttons

The bootstrap "btn" and "btn-{colour}" classes allow us to easily create consistent, clean buttons within the user interface that match current theme.

To create a button in Bootstrap, simply use the class "btn" and select a "theme" colour (these will be the same colours as the navbar background options), ie:

```
<button class="btn btn-primary">Button</button>
```

will render a button with the "primary" colour as its background. If we want to use one of the other colours, one of the predefined theme colours can be used as a suffix for the "btn-" class, ie:



```
<button class="btn btn-primary">Primary</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-success">Success</button>
<button class="btn btn-danger">Danger</button>
<button class="btn btn-warning">Warning</button>
<button class="btn btn-info">Info</button>
<button class="btn btn-light">Light</button>
<button class="btn btn-dark">Dark</button>
```

In addition to the standard look above, Bootstrap also offers some customization options, including:

- Rendering the buttons using only their **outline**
- Modifying the **size** of the buttons
- Showing the button in a **disabled** state
- Rendering full-width **block** buttons

Dropdowns

One extremely useful *variation* on the button is the "dropdown" button, effectively creating a button that, when clicked, shows a menu containing more links / buttons. Dropdowns are widely used and allow us to preserve space on the user interface. To create a dropdown button in Bootstrap, the following code can be used:

NOTE: both <button> and <a> can be used in the following example

```
<div class="dropdown">
  <button
    class="btn btn-secondary dropdown-toggle"
    type="button"
    id="dropdownMenuButton1"
    data-bs-toggle="dropdown"
    aria-expanded="false"
  >
    Dropdown button
  </button>
  <ul class="dropdown-menu" aria-labelledby="dropdownMenuButton1">
    <li><a class="dropdown-item" href="#">Action</a></li>
    <li><a class="dropdown-item" href="#">Another action</a></li>
    <li><a class="dropdown-item" href="#">Something else here</a></li>
  </ul>
</div>
```

As with regular buttons, there are some options for customization for dropdowns, including:

- Using different **sizes**
- Creating **Split Buttons**, where only a portion of the button is pressed to show the dropdown menu
- ["Dark"] dropdowns, where the background of the dropdown menu is a "dark" colour to match darker themes
- Showing the dropdown menu in different **directions**

Tables

Like the above components, tables in Bootstrap are extremely easy to create, and provide lots of configuration options. All that is required is that you add the class "table" to your existing

<table> element, ie:

```
<table class="table">
  <thead>
    <tr>
      <th>Column One</th>
      <th>Column Two</th>
      <th>Column Three</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Cell A</td>
      <td>Cell B</td>
      <td>Cell C</td>
    </tr>
    <tr>
      <td>Cell D</td>
      <td>Cell E</td>
      <td>Cell F</td>
    </tr>
  </tbody>
</table>
```

This will provide the default table style that is used by Bootstrap. However, there are a ton of configuration options such as:

- Adding **background colours** to the entire, or specific cells / rows using the class name "table-{colour}", where {colour} can be one of the theme colours, ie "primary" etc
- Rendering the table using **striped rows** by adding the class "table-striped" alongside the "table" class
- **Highlighting rows** when the user moves their mouse over the table by adding the class "table-hover" alongside the "table" class
- Adding / removing **borders** to tables using the "table-bordered" / "table-bordereless" classes
- Configuring the **size** and **responsive behaviour** of tables

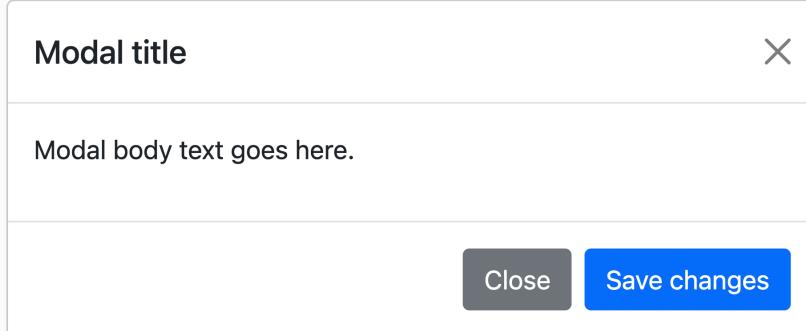
Modal Windows

The final component that we will discuss here is the "modal" window. This is essentially an in-page dialog box that focuses the users attention on a specific task or piece(s) of data.

To create a modal window in Bootstrap, you can use the following code (typically placed at the bottom of your <body> element):

```
<div class="modal fade" tabindex="-1" id="exampleModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Modal title</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-
label="Close"></button>
      </div>
      <div class="modal-body">
        <p>Modal body text goes here.</p>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-bs-
dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>
```

This creates a small, *invisible* user interface element that looks like the following:



in order to actually *show* the modal window, a few different options may be used:

- Add the data attributes: `data-bs-toggle="modal" data-bs-target="#exampleModal"` to an HTML element. This will instruct bootstrap to show the modal with id "exampleModal" (using

the default options) when the element is clicked

- Using JavaScript, the following function can be used, ie:

```
letmyModal = new bootstrap.Modal(document.getElementById('exampleModal'), {  
    backdrop: 'static', // default true - "static" indicates that clicking on the  
    // backdrop will not close the modal window  
    keyboard: false, // default true - false indicates that pressing on the "esc" key  
    // will not close the modal window  
    focus: true, // default true - this instructs the browser to place the modal  
    // window in focus when initialized  
});  
  
myModal.show();
```

If we wish to hide the modal, `myModal.hide()` may be used.

Rendering Data

Now that we have seen a sampling of what Bootstrap has to offer, let's use what we have learned to build a user interface to explore the data from a popular test API: [{JSON} Placeholder](#). This will involve utilizing the "Fetch" API to make AJAX requests, using native DOM methods to wire up user events and ES6 techniques such as [Template Literals](#) to format the data and generate HTML.

Dependencies

Before we obtain the data and attempt to render it in the browser, we should first include any dependencies that are required. To begin, we will start with an HTML5 skeleton that includes Bootstrap 5.1

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <!-- Bootstrap 5.1 CSS-->
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
      rel="stylesheet"
      integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
      crossorigin="anonymous"
    />

    <!-- Bootstrap 5.1 JS Bundle -->
    <script
      src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
      integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENB00LRn5q+8nbTov4+1p"
      crossorigin="anonymous"
    ></script>

    <title>Bootstrap Practice</title>
  </head>
```

The Data

Next, it is a good idea to examine the data that we will be rendering on the page before using any other boilerplate code. For our purposes we will use the [posts collection](#). This dataset provides the following functionality that we can use in our user interface:

- <https://jsonplaceholder.typicode.com/posts> - This is a full list of "Posts" using the format:

```
{  
  "userId": 1,  
  "id": 1,  
  "title": "sunt aut facere repellat provident occaecati excepturi optio  
  reprehenderit",  
  "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\n  
  reprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet  
  architecto"  
}
```

- <https://jsonplaceholder.typicode.com/posts?userId=3> - This is the same list filtered by "userId" (in this case userId=3)
- <https://jsonplaceholder.typicode.com/posts/1> - This is the same list, filtered by "id" (in this case id=1)
- <https://jsonplaceholder.typicode.com/comments?postId=1> - Here, we have all the comments for a given post (in this case postId=1) using the format:

```
{  
  "postId": 1,  
  "id": 1,  
  "name": "id labore ex et quam laborum",  
  "email": "Eliseo@gardner.biz",  
  "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\\ntempora quo  
  necessitatibus\\ndolor quam autem quasi\\nreiciendis et nam sapiente accusantium"  
}
```

UI Elements

With the dependencies in place and the structure ("shape") of the data known, we can begin to create the UI elements for our application.

Navbar

The best place to begin is with the navigation bar (navbar). Since it is possible to filter our data by "userId" let's create a navbar that also has a "search" bar:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Posts Dataset</a>
    <button
      class="navbar-toggler"
      type="button"
      data-bs-toggle="collapse"
      data-bs-target="#navbarSupportedContent"
      aria-controls="navbarSupportedContent"
      aria-expanded="false"
      aria-label="Toggle navigation"
    >
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse justify-content-end"
      id="navbarSupportedContent">
      <form class="d-flex" id="searchForm">
        <input
          class="form-control me-2"
          type="search"
          placeholder="User ID (Number)"
          id="userId"
          aria-label="Search"
        />
        <button class="btn btn-outline-success" type="submit">Search</button>
      </form>
    </div>
  </div>
</nav>
```

Here, we have only included the "navbar-brand", "navbar-toggler" / "collapse" and "form"

elements. The form has been given an id of "searchForm" and the "collapse" <div> has been given an additional class "**justify-content-end**" to ensure that the search bar appears on the right of the navigation bar.

Main Container & Data Table

The primary display for our data will be in a table format. This will display all "posts" by including their "userId", "title" and "body" attributes, ie:

```
<div class="container">
  <div class="row">
    <div class="col">
      <table class="table table-hover" id="postsTable">
        <thead>
          <tr>
            <th>User ID</th>
            <th>Title</th>
            <th>Body</th>
          </tr>
        </thead>
        <tbody></tbody>
      </table>
    </div>
  </div>
</div>
```

Notice how we have included a regular table with the classes "table" and "table-hover" as well as the id "postsTable". This is where we will eventually render all of our "Posts" data from the API.

Modal Window

Finally, we will include a "modal" window to show a specific post as well as the related comments. This window will be shown once a user clicks on a specific row of the table. For now, we will simply include the "skeleton" and update the contents on demand later:

```
<div class="modal fade" tabindex="-1" id="commentsModal">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Comments</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-
```

Here, we have given the "modal" an id of "commentsModal" as well as omitted the "modal-body" content. Lastly, since the user will not be entering any data, we have also omitted the "Save Changes" button

The JavaScript

The next step in the development effort is to start working with JavaScript to obtain and render the data to the table as well as make the table searchable and interactive. For this example, we will place our JavaScript logic within a <script> element on the same HTML page as the rest of the example, rather than using an external .js file:

```
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Bootstrap 5.1 CSS-->
  <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
    rel="stylesheet"

  integrity="sha384-1BmE4kBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
  crossorigin="anonymous"
  />

  <!-- Bootstrap 5.1 JS Bundle -->
  <script
    src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/
bootstrap.bundle.min.js"

  integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENB00LRn5q+8nbTov4+1p"
  crossorigin="anonymous"
  ></script>

  <script>
    // Custom JS - included beneath Bootstrap
  </script>

  <title>Bootstrap Practice</title>
</head>
```

Fetching and Rendering the Data

The first step here is to write a function that will actually pull the data from the API. This will be done using "fetch()", in it's simplest form, ie:

```
function populatePostsTable(userId = null) {
  let url = +userId // attempt to convert userId to a number
  ? `https://jsonplaceholder.typicode.com/posts?userId=${+userId}`
  : `https://jsonplaceholder.typicode.com/posts`;

  fetch(url)
    .then((res) => res.json())
    .then((data) => {
      console.log(data);
    });
}
```

For this function we have included a single parameter "userId", which is set to a default value of *null*. This is because the URL that we fetch our data from will change depending on whether or not we have a numeric "userId" value. Also, we are not yet generating any HTML or updating the DOM - this is purely a test to ensure that the function works as expected.

Next, since we wish the table to show the data once the page is first loaded, we must execute this function when the "DOM is Ready" - if you are familiar with jQuery, this would be the `$(function() { ... })` function. However, since we have removed the dependency on jQuery, we can use the following code instead:

```
// Execute when the DOM is 'ready'
document.addEventListener('DOMContentLoaded', function () {
  populatePostsTable();
  populatePostsTable(4); // test with User ID 4 (to be removed after testing)
});
```

Generating HTML

Once we are confident that our "populatePostsTable" is functioning as expected, the next step is to take the returned data, ie:

[

and transform it into HTML to be included in the DOM, ie:

```
<tr data-id="1">
  <td>1</td>
  <td>"sunt aut facere repellat provident occaecati excepturi optio
reprehenderit"</td>
  <td>
    "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\
nreprehenderit molestiae ut
      ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
  </td>
</tr>
<tr data-id="2">
  <td>2</td>
  <td>"qui est esse"</td>
  <td>
    "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea
dolores neque\nfugiat
      blanditiis voluptate porro vel nihil molestiae ut reiciendis\nqui aperiam non
debitis possimus
        qui neque nisi nulla"
  </td>
</tr>
```

This is where knowledge of [Template literals](#) will come in handy.

You will recall (from examples above / online) that the syntax for Template literals is the following:

```
`string text ${expression} string text`;
```

where "expression" is a valid JavaScript [expression](#), ie: "any valid unit of code that resolves to a value". Therefore, if our task is to loop through our array of results and generate HTML, we can use the [map\(\)](#) method of an array within our "expression" to process the post objects one at a time and generate html.

To see this strategy in action, let's first try a simple example, where we take an array of strings and convert them to a single string showing the html for an unordered-list:

```
let numbers = ['one', 'two', 'three'];
let numberList = `<ul>${numbers.map((num) => `<li>${num}</li>`)}</ul>`;
```

This should show the following in the console:

```
<ul><li>one</li>,<li>two</li>,<li>three</li></ul>
```

This is very close, however you will notice that we have unnecessary commas (",") in our string output. This is because the `map()` method always returns an array and when that array is implicitly converted to a string, commas (',') are inserted. To overcome this, we must make one small change to our template literal, ie:

```
let numberList = `<ul>${numbers.map((num) => `<li>${num}</li>`).join('')}</ul>`;
```

By chaining the `join("")` method at end of the "map" operation, we can ensure that the array is joined using empty strings ("") instead of commas (',').

With this in mind, let's attempt to write a template string that will solve our problem, by converting the data from jsonplaceholder into a series of rows (`<tr>` elements) to be used in our "postsTable":

```
let postRows = ` ${data.map(post => (`<tr data-id=${post.id}>
    <td>${post.userId}</td>
    <td>${post.title}</td>
    <td>${post.body}</td>
</tr>`)).join('')}`;
```

Updating the DOM

With our `postRows` showing valid HTML, our next step is to add it to the DOM. Fortunately, all that needs to be done is for the correct DOM element to be selected and its `.innerHTML` property set to our newly generated `postRows` HTML string:

```
document.querySelector('#postsTable tbody').innerHTML = postRows;
```

Click Events

Once the elements are added to the DOM, the last thing that we must do is to associate each row

with a "click" event (we will be using this later). This involves selecting the newly created <tr> elements, looping through each one and (using the familiar "addEventListener" function) registering a "click" event. For the time being, we will test this by outputting "clicked" to the console:

```
// add the "click" event listener to the newly created rows
document.querySelectorAll('#postsTable tbody tr').forEach((row) => {
  row.addEventListener('click', (e) => {
    console.log('clicked');
  });
});
```

Filtering the Table

Our next major task is to give the user the ability to search for a user ID using our search form in the navigation bar. To achieve this, we must first register an event to trigger when the "searchForm" is submitted:

NOTE: Make sure this code is executed only when the "DOM is Ready"

```
document.querySelector('#searchForm').addEventListener('submit', (event) => {
  // prevent the form from officially submitting
  event.preventDefault();
});
```

Here, we use the `event.preventDefault()`; function to ensure that the event's default action is not taken - ie: submitting the form by attempting to send a request back to the server.

What we actually want to do is to get the value of the only <input> field (id: "userId"). In this case, we can obtain its "value" by simply using the `value` attribute of the form field element. Once we have this, we can invoke the "populatePostsTable()" method with the value, which will refresh the table:

```
document.querySelector('#searchForm').addEventListener('submit', (event) => {
  // prevent the form from officially submitting
  event.preventDefault();
  // populate the posts table with the userId value
  populatePostsTable(document.querySelector('#userId').value);
});
```

Populating / Showing the Modal Window

The final piece of interactivity that we will add to the table is to enable the user to click on a specific *row* to obtain additional information for a given post - in this case we will show all of the "comments" for a given post.

Getting the Data on "Click"

To begin, we will be placing our code within the "click" eventListener callback for the "row", which currently displays the text "clicked" in the console. However, instead of displaying "clicked", we will instead display the "data-id" value of the row that was clicked. This can be accomplished by using the `getAttribute()` method of the element (row):

```
// add the "click" event listener to the newly created rows
document.querySelectorAll('#postsTable tbody tr').forEach((row) => {
  row.addEventListener('click', (e) => {
    let clickedId = row.getAttribute('data-id');
    console.log(clickedId);
  });
});
```

Once we have confirmed this works and the correct "clickedId" is displayed in the console, we can use this value to get all of the comments for a current post using:

```
https://jsonplaceholder.typicode.com/comments?postId=ID
```

To test this functionality, add the following "fetch()" call to the above logic and confirm that the correct comments are indeed output to the console:

```
fetch(`https://jsonplaceholder.typicode.com/comments?postId=${clickedId}`)
  .then((res) => res.json())
  .then((data) => {
    console.log(data);
  });
});
```

You should see that comments for "postId" 1 are shown for the first row, comments for "postId" 5 are shown for the fifth row, etc.

Generating the List

Next, we must convert this data into an HTML representation and add it to the DOM - specifically the "modal-body" `<div>` element of our "commentsModal". A similar operation was required above when first [converting the initial post data](#) to valid `<tr>` elements and we will use the same logic here. However, instead of generating `<tr>` elements, we will instead generate an unordered list using Bootstrap's `list-group` and `list-group-item` classes:

```
let commentsList = `<ul class="list-group">
  ${data.map(comment => (`<li class="list-group-item">
    ${comment.body}<br /><br />
    <strong>Name:</strong> ${comment.name}<br />
    <strong>Email:</strong> ${comment.email}<br />
  </li>
`)).join('')}
</ul>
`;
```

Populating the Modal

Finally, with the `commentsList` containing the correct HTML, we can populate the modal window and show it to the user. This will involve adding the `commentsList` to the "modal-body":

```
document.querySelector('#commentsModal .modal-body').innerHTML = commentsList;
```

and using using the `bootstrap.modal()` function; open the modal window:

```
let modal = new bootstrap.Modal(document.getElementById('commentsModal'), {
  backdrop: 'static',
  keyboard: false,
});

modal.show();
```

Example Code

You may download the sample code for this topic here:

| [Bootstrap-UI-Implementation](#)

React Introduction

So far, we have seen how to work with / render API data using native JavaScript code with [Bootstrap 5.1](#). This technique focused on modifying the DOM directly by working with [Elements](#) and their properties such as [innerHTML](#) and methods like [addEventListener\(\)](#). We also made regular use of [Template literals](#) and [Array](#) properties / methods in order to generate new HTML to be added to the DOM and rendered.

The strategy employed was to design a complete page, examine its contents in the DOM, and write JavaScript code to manipulate it based on user actions. To store data for the page, it would be declared in the associated JavaScript file, or added to elements of the page using "[data-*](#)" attributes.

This works well for smaller web applications such as our example, however as applications grew more complicated, a modern, scalable approach needed to be developed. This led to the creation of the "MVVM" or "Model View ViewModel" design pattern and tools such as [Knockout](#), [Ember](#) and [Angular.js](#) gained popularity.

MVVM

From the official [Knockout](#) documentation:

Model: your application's stored data. This data represents objects and operations in your business domain (e.g., bank accounts that can perform money transfers) and is independent of any UI. When using KO, you will usually make Ajax calls to some server-side code to read and write this stored model data.

View Model: a pure-code representation of the data and operations on a

UI. For example, if you're implementing a list editor, your view model would be an object holding a list of items, and exposing methods to add and remove items.

Note that this is not the UI itself: it doesn't have any concept of buttons or display styles. It's not the persisted data model either - it holds the unsaved data the user is working with. When using KO, your view models are pure JavaScript objects that hold no knowledge of HTML. Keeping the view model abstract in this way lets it stay simple, so you can manage more sophisticated behaviors without getting lost.

View: a visible, interactive UI representing the state of the view model. It displays information from the view model, sends commands to the view model (e.g., when the user clicks buttons), and updates whenever the state of the view model changes.

This concept drastically changed how to think about the designing sites / applications on the web. By introducing this "Separation of Concerns", we can create code that is modular, reusable and testable.

In 2011 Facebook employee Jordan Walke created an early prototype of React called "FaxJS" which focused on creating "components", essentially providing the "view model" and "view" for reusable pieces of a User Interface (UI). Multiple components would then be combined to create a functional web site / app.

Quick Note: "[Web Components](#)" are now standardized and [available in modern browsers](#). Please refer to the [MDN Documentation](#) for more information.

In 2013, React was released as open source and has steadily grown in use among developers, even surpassing popular frameworks like Angular and Vue in metrics such as [Questions per Month on Stack Overflow](#) and [Most commonly](#)

used Web Framework in the 2021 Stack Overflow Survey of over 67,000 professional developers.

Getting Started

To get started creating applications with React, technically all we need to do is to add some scripts to an existing HTML page and start creating and rendering our components, ie:

```
<h1>React</h1>

<!-- We will put our React component inside this div. -->
<div id="main_container"></div>

<!-- Load React. -->
<!-- Note: when deploying, replace "development.js" with
"production.min.js". -->
<script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
<script src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js" crossorigin></script>
<script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>

<script type="text/babel">
  'use strict';

  // Hello Component
  function Hello(props){
    return <p>message: {props.message}</p>
  }

  // Main Component
  function Main() {
```

However, we will be building larger apps and while [adding React to an Existing Project](#) is indeed possible (as we have seen above), we would prefer to use something that scales better and gives a superior development experience. Here, the idea of "Toolchains" comes into play, ie:

"A set of programming tools that is used to perform a complex software development task or to create a software product"

When working with React, this will typically be things like [Webpack](#), [Babel](#), [ESLint](#) and [Jest](#) among many others. Fortunately, there are frameworks available for us to use that have these "toolchains" correctly configured and optimized. For example:

- [React Router](#): "The most popular routing library for React and can be paired with Vite to create a full-stack React framework. It emphasizes standard Web APIs and has several ready to deploy templates for various JavaScript runtimes and platforms."
- [Next.js](#): "A popular and lightweight framework for static and server-rendered applications built with React. It includes styling and routing solutions out of the box, and assumes that you're using Node.js as the server environment."
- [Gatsby](#): "The best way to create static websites with React. It lets you use React components, but outputs pre-rendered HTML and CSS to guarantee the fastest load time."
- Many others including: [Remix](#), [Expo](#), etc.

Next.js

For this course, we will be choosing [Next.js](#), which describes itself as "the best developer experience with all the features you need for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more. No config needed". This sounds perfect for our purposes,

so why don't we get started and see what the starter app looks like for Next.js.

To begin, create a new folder on your machine and open it using Visual Studio Code. Next, open the Integrated Terminal and (assuming that you want to create a new app (and folder) named “my-app”) execute the command:

```
npx create-next-app@15 my-app --use-npm
```

You will then be asked to make some decisions regarding what to include in your new app (for now, we will chose the **following values** (ie: the only option that we will say "Yes" to, is the ESLint option):

? Would you like to use TypeScript? ... **No** / Yes

? Would you like to use ESLint? ... No / **Yes**

? Would you like to use Tailwind CSS? ... **No** / Yes

? Would you like your code inside a `src/` directory? ... **No** / Yes

? Would you like to use App Router? (recommended) ... **No** / Yes

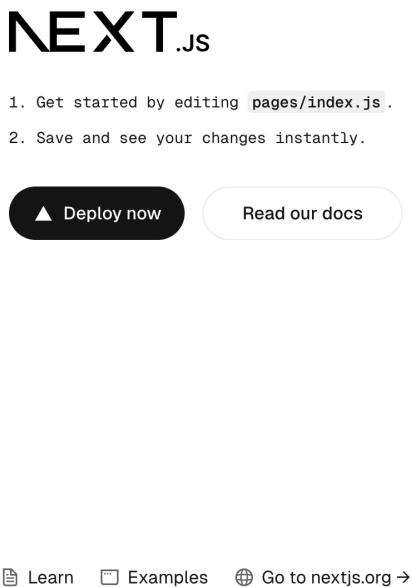
? Would you like to use Turbopack for `next dev`? ... **No** / Yes

? Would you like to customize the import alias (`@/*` by default)? ... **No** / Yes

The process will create a new folder called “my-app”, with the code needed to get started using Next.js. Once it finishes its initial setup (downloading dependencies, etc), you can view the starter site immediately by executing the commands:

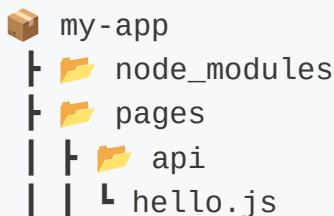
```
cd my-app
```

This will start a development server on localhost on port "3000" (<http://localhost:3000>) and when opened in the browser, should look something like this:



File Structure

Now that we know how to create a "Next.js" app, let's walk through some of what was created during the "create-next-app" process. If you open the file explorer tab in Visual Studio Code, you should see the following:



At first, we will not be touching many of these files and will be **adding** more folders and files to this structure in the near future. For the time being however, let's discuss the following areas already created for us:

- **"pages"** Folder: This folder is responsible for containing the components that will act as "routes" for our application, ie: any file added to the "pages" directory is automatically available as a route. For example, currently we have an "index" route, ie "/" - this file (index.js) is where we find the "Home" component, which contains the code to render our "Welcome to Next.js!" page.

Additionally, we have an "api" folder, currently containing the file "hello.js" with the code to return `{"name": "John Doe"}` at the route "/api/hello".

- **"public"** Folder: This is where we will keep any static resources for our app / site. For example, you can see that one of the files contained therein is "next.svg" - this file is referenced in the "Home" component (index.js) with the code:

```
<Image
  className={styles.logo}
  src="/next.svg"
  alt="Next.js logo"
  width={180}
  height={38}
  priority
/>
```

- **"styles"** Folder: As the name suggests, this is where we will be storing our .css files for the application. At the moment, you will find two files:
 - "globals.css": referenced in the "_app.js", the file containing the top-level "App" component (discussed later)

- "Home.module.css": referenced in the "index.js" file, used by the "Home" component.

NOTE: The ".module.css" extension identifies the file as a "**CSS Module**" which will "locally scope CSS by automatically creating a unique class name. This allows you to use the same CSS class name in different files without worrying about collisions."

Components

Before we can start working with some of the more interesting features of Next.js, we must first learn some of the basics of React; specifically "Components" and "JSX":

So far, we have seen one important component: "Home" (defined in index.js):

```
import Head from "next/head";
import Image from "next/image";
import { Geist, Geist_Mono } from "next/font/google";
import styles from "@/styles/Home.module.css";

...

export default function Home() {
  return (
    <>
      <Head>
        ...
      </Head>
      <div className={`${styles.page} ${geistSans.variable}
${geistMono.variable}`}>
        <main className={styles.main}>
          ...
        </main>
        <footer className={styles.footer}>
          ...
        </footer>
      </div>
    </>
  );
}
```

This is known as a "functional component", since it's defined as a "function". The name of the function corresponds to the tag used to render the component, in this case `<Home />`. This tag is said to be "self-closing", but we could also write the component in this form: `<Home></Home>`. This is not as common however, and it's unnecessary unless we wish to write a component that acts as a wrapper that renders other components.

Additionally, you will notice that our function is preceded by four import statements. In this case, they provide the following functionality:

- `import Head from 'next/head'` - This is the import statement for the "`<Head>...</Head>`" component. This is essentially a built-in component that Next.js provides to append elements, such as title and meta tags, to the `<head>` element in the document.
- `import Image from 'next/image'` - This is the import statement for the "`<Image />`" component. This component is described as: "an extension of the HTML `` element, evolved for the modern web. It includes a variety of built-in performance optimizations to help you achieve good [Core Web Vitals](#)".
- `import { Geist, Geist_Mono } from "next/font/google"` - This is the import statement used to set up the Next.js [font optimization feature](#) for the "[Geist](#)" and "[Geist Mono](#)" Google Fonts. For more information on font optimization in Next.js and how to use Google fonts, see "[Google Fonts](#)" in the official documentation.
- `import styles from '@/styles/Home.module.css'` - This loads the "Home" CSS Module from our "styles" directory. If you inspect elements for this component in the browser, you will notice that many of the values in the "class" attributes have extra text, ie: "Home_" followed by some random characters. This is the "CSS Module" functionality mentioned earlier, which is achieved by including the classes in the CSS Module using the following

syntax:

```
className={styles.someClass}
```

instead of simply:

```
className="someClass"
```

Finally, you will notice the "export" statement: `export default` is stated before the function definition. This is required because our Home component exists in its own file (module).

Creating our own Component

Now that we have seen what a basic component consists of, let's create our own component using the same pattern to explore the unique syntax and functionality that can be achieved using functional components.

Start by creating a new folder called "components" in the "my-app" directory. Next, within this folder, create a file called "Hello.js" (Our component will be named "Hello"). Inside the component, we will add everything required for a basic component that outputs (you guessed it: "Hello World"), ie:

```
export default function Hello() {
  return <p>Hello World!</p>
}
```

To actually see this component working, we need to render it on the page somewhere. Since we really only have one route / page at the moment (ie: "Home", defined in "index.js"), let's place it there.

Before we do so however, we should first *wipe out* most of the code for our "Home" component, leaving a "blank slate" for us to start fresh. Go ahead and wipe out all of the code within the "return" statement, as well as the "include" statements, leaving only:

```
export default function Home() {  
  return (  
    )  
}
```

Next, at the top of the file, add the following "import" statement:

```
import Hello from '@/components/Hello';
```

Once that is complete, include the "Hello" component as the only item in the return statement:

```
<Hello />
```

This should cause your index.js file to look like the following:

```
import Hello from '@/components/Hello';  
  
export default function Home() {  
  return (  
    <Hello />  
  )  
}
```

Once you have saved everything, head back to your browser to see the

changes - "Hello World!"

Introducing JSX

With our `<Hello />` component now displaying correctly within our "Home", let's discuss the strange syntax within the return value of "Home" and other components. It is responsible for how the component is rendered and looks like a String or HTML, but it is in fact, neither:

"It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript".

Returning a Single Element

Whenever we use JSX, we must ensure that whatever we return is wrapped in a *single element*. This is because part of the build process for our Next.js apps is **Babel** compiling the **JSX** code into a **React.createElement(component, props, ...children)** call, ie:

```
const element = <p className="greeting">Hello, world!</p>
```

becomes:

```
const element = React.createElement('p', { className: 'greeting' }, 'Hello, world!');
```

NOTE: If you *do not* wish to include a "wrapper" component (ie: a `<div>...</div>`, `...`, etc) you may instead use a "**JSX Fragment**" (ie: `<>...</>`), which will not create an extra node in the DOM.

Empty Elements

Also, When using JSX, there is no notion of an "**empty element**", so be careful when using tags like:

```
<br>
```

as this will actually cause a problem and your component will not compile, due to the error "**JSX fragment has no corresponding closing tag.**". Instead, you must use the "self-closing" syntax, ie:

```
<br />
```

Embedding Expressions in JSX

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

```
export default function Hello() {
  const name = 'Josh Perez';
  return <p>Hello {name}!</p>
}
```

You can put any valid **JavaScript expression** inside the curly braces in JSX. For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into a `<p>` element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

export default function Hello() {
  const user = {
    firstName: 'Harper',
    lastName: 'Perez',
  };

  return (
    <p>
      Hello, {formatName(user)}!
    </p>
  );
}
```

We often split JSX over multiple lines for readability (as above). While it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of [automatic semicolon insertion](#).

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <p>Hello, {formatName(user)}!</p>
  }
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl} />
```

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but *not both* in the same attribute.

Warning:

Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.

For example, class becomes `className` in JSX, and tabindex becomes `tabIndex`.

Accepting "Props"

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

For example, we can extend our “Hello” component to accept one or more

"props" by including the "props" parameter to our function definition and accessing each named "prop" as an attribute using the same name, ie:

```
export default function Hello(props) {
  return (
    <p>
      Hello {props.fName} {props.lName}!
    </p>
  );
}
```

will allow us to provide "fName" and "lName" values to the component using the straightforward syntax:

```
<Hello fName="Jason" lName="Perez" />
```

NOTE: If we wish to have *default* values for props (in this case: **fName** and **lName**), we can update the function definition to use object destructuring with **default values**:

```
export default function Hello({ fName = 'First Name', lName =
  'Last Name' }) {
  // NOTE: props will be accessed using fName and lName instead of
  // props.fName and props.lName
}
```

Introducing "Hooks".

As of version 16.8, React has introduced a feature known as "hooks". Using this syntax will open up some new, interesting possibilities to our functional components, including working with the "**state**" as well as performing "**side**

effects" during the lifetime of the component (ie: "Data fetching, setting up a subscription, and manually changing the DOM in React components").

Basically, by using certain built-in "hooks" (functions), React components are able to store and manage data internally to the component (ie, its "state" values). When this data changes, a refresh (render) of the component will occur and the user interface will be updated. This allows us to create components that work with data internally that changes over time.

To actually see this in action, let's create a new component called **Clock**:

First, create a new file in "components" called "Clock.js". Once this is done, add the following code:

```
import { useState, useEffect } from 'react';

export default function Clock(props) {
  return (
    <p>
      Locale: {props.locale}: <mark>TODO: Render Locale-Dependant
      Clock Here</mark>
    </p>
  );
}
```

So far, this looks very similar to our "Hello" component above; it is defined as a function that accepts props and it returns some JSX to be rendered. However, there is one key difference: we have imported both the **useState** and the **useEffect** hooks from 'react'. Soon, we will use these functions within our component.

For now, let's just add the Clock component to our Home so that we can see what it outputs:

Open the **index.js** file and add the following "import" statement:

```
import Clock from '@/components/Clock';
```

Next, include the "Clock" component *beneath* the `<Hello />` tag using its associated "self-closing" tag, as well as some code to include the locale as its only "prop":

```
<Clock locale="en-CA" />
```

Currently, we should see the locale next to some highlighted text stating "TODO: Render Locale-Dependant Clock Here".

Adding "state"

As mentioned above, the "state" of a component is a way to store data within the component (ie: the "date" data for our clock) that is synchronized with the UI of the component. This is a very powerful concept and one of the core ideas behind designing apps using components.

For our example, let's add a "state" value to our `<Clock />` component, so that we can keep the UI of the component in sync with the current time. In this way, we can say that each `<Clock />` component keeps track of its own internal Date value. It will also be responsible for updating its UI every second to reflect changes in this data.

Here is where we will use our first hook: **useState()**. In the first line of your "Clock" function, add the line:

```
const [date, setDate] = useState(null); // Note: Never set this to
```

Here, we can see that "useState" is a function, which:

- Accepts a parameter that allows us to set the *initial value* of a "state" variable
- Returns an array consisting two values: the "state" variable itself and a **function to update it**. We use a *destructuring assignment* to assign each of those values to a pair of constant variables - in this case: "date" and " setDate". In the above case, this is *shorthand* for:

```
const dateState = useState(null);
const date = dateState[0];
const setDate = dateState[1];
```

We use the "const" keyword here since we **must** use the " setDate" function to modify the state value "date" - we cannot modify "date" directly. By invoking the " setDate" method (ie: " setDate(**New Date Value**)"), we not only update the value of "date", but also trigger our component to re-render!

Now, let's add some code to attempt to render the "date" value within our component. Here, we will be using the **toLocaleTimeString** function, ie:

```
return (
  <p>Locale: {props.locale}:
{date.toLocaleTimeString(props.locale)} </p>
);
```

However, if we refresh the page now, we should see an error since at this point since "date" is currently **null**, ie:

```
TypeError: Cannot read properties of null (reading  
'toLocaleTimeString')
```

For the mean time (until we have a real date object that we can use), we can avoid this error by using the **Optional Chaining Operator** on our date object, ie:

```
{date?.toLocaleTimeString(props.locale)}
```

Quick Note: "state" vs. "props"

While "state" & "props" both hold information that can be used to influence the output of the rendered component, they are different in one important way: props get *passed to* the component whereas state is managed *within* the component.

One interesting thing to note about "props" is that we can pass anything as a property, including functions! This can be very helpful if we wish to send a message from a "child" component to a "parent" component. For example, if we define a function (ie: `handleMessage(msg)`) in the "Parent" component, we can pass it in to the "Child" component using a custom property, ie "`sendMessage`". Whenever the child wishes to send a message back to the parent, it can invoke the callback function from "props" and pass the data:

Parent Component

```
function handleMessage(msg) {  
  console.log(`Child Says: ${msg}`);  
}  
return <Child sendMessage={handleMessage} />
```

Child Component

```
props.sendMessage('Hello');
```

Updating the <Clock /> Component using the "useEffect" Hook

At the moment, our <Clock /> component has a "date" value in the state, but it's currently set to "null" so we are unable to see any values output in the browser. If we cannot set a new date value as the the *initial value* of the "state" variable (potentially causing a "**Hydration Error**" in this case), where do we initialize it?

This is where the **useEffect** Hook comes into play. This hook allows us to provide a function that *only executes* under certain conditions, for example when the component is "first rendered". To see this in action, place the following code above your return statement in the "Clock" function:

```
useEffect(() => {
  setDate(new Date());
}, []);
```

Here, you will notice that the **useEffect** hook actually accepts two parameters: a callback function and an array of "dependencies". The callback function is simply the code to be executed once the component is first "mounted" and rendered, while the dependency array is a list of variables that, when changed, will cause the effect to execute again. Since we only want this effect to execute **once**, we can provide an empty array.

Now if we refresh the page, we should see a clock value showing the current time!

However, if we wish our `<Clock />` component to function as a proper clock and update the UI every second, we must add some additional logic. As expected, this will involve the `setInterval()` function to update the date value every second. To achieve this functionality, update the code in your `useEffect` hook to set a new date once every second, ie:

```
useEffect(() => {
  setDate(new Date());
  // update the date once every second
  const timerID = setInterval(() => {
    setDate(new Date());
  }, 1000);
}, []);
```

NOTE: If the new value of your state variable depends on the previous value, consider using an "updater function". For more information, see: [Updating state based on the previous state / Is using an updater always preferred?](#)

When / How to Stop the interval?

At the moment, our code has no mechanism to **stop** the interval using `clearInterval()` when it is no longer needed. This would be part of a clean-up process and should execute when the component is "unmounted" or removed from the DOM.

Fortunately, we can handle this situation within the **return value** of the callback function provided to `useEffect`, ie:

```
useEffect(() => {
  setDate(new Date());
  // update the date once every second
```


Example Code

You may download the sample code for this topic here:

| [React-NextJS-Introduction](#)

Handling User Events

NOTE: Some of the below code examples and explanations have been reproduced from sections of the [official online documentation](#) for React.

Handling events in React is very similar to handling events on DOM elements using properties like `onclick`. However, there are some differences, ie:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="processClick()">Click Me!</button>
```

is slightly different in React:

```
<button onClick={processClick}>Click me!</button>
```

To see this in action, let's code a simple "click counter" component that renders a single button that shows a number that increases by one (1) every time it's clicked. To achieve this, we'll create a new component called "ClickCounter":

```
import { useState } from 'react';

export default function ClickCounter(props) {
  const [numClicks, setNumClicks] = useState(0);

  function increaseNumClicks(e) { // 'e' is the current event
    object
```

NOTE: Be careful when updating state based on a previous value (ie: `numClicks + 1` in the example above), as it may not always work as expected on types that are not **primitive**, ie: "string", "number", "boolean", etc. For example, if we use the same logic to add an element to a state value holding an *array*, we may be tempted to use the following code:

```
setMyArray(myArray.push('new element'));
```

However, updating state in this manner **will not** cause the component to re-render. Instead, we must provide a **new array**, ie:

```
setMyArray([...myArray, 'new element']);
```

For more information see: [Updating arrays in state](#) and [Updating objects in state](#)

Here, you will notice that we have added a couple new concepts to the construction and rendering of a typical functional component, ie:

- We have declared a function to handle the event. It receives a single parameter 'e' which is a "**synthetic event**" - "a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers."
- On our button element, we use "onClick" (instead of "onclick") to reference the event handler and "wire up" the event.

NOTE: For a full list events please refer to the official documentation for [supported events](#).

Adding Parameters

As you can see from the above example, our callback function "increaseNumClicks" is registered to the onClick event by *passing the function only* - the function is not actually *invoked* anywhere in our JSX. This works fine, but what if we wish to pass one or more parameters to the function, in addition to the SyntheticEvent (above)?

This can actually be achieved by registering the event as an anonymous function declared within the JSX, which *invokes* the callback function. For example:

```
function increaseNumClicks(e, message) { // 'e' is the current
  event object
  console.log(message);
  setNumClicks(numClicks + 1);
}

return <button onClick={() => { increaseNumClicks(e, "Hello") }}>Clicks: {numClicks}</button>
```

Here, we declare the callback function in place. It accepts a single parameter "e" as before, but the body of the function *invokes* the callback function. This allows us to continue to pass the SyntheticEvent (e) to our event handler "increaseNumClicks" as well as add any other parameter values.

Adding API Data

If you inspect the source code that is returned from the development server for a given page in Next.js, you will notice that in addition to the `<script>` tags that provide the functionality for the application / site, we also have parts of our components *already rendered* in the `<body>`. This is in stark contrast to other toolchains / frameworks such as "Create React App" which only contain a **single** element in the `<body>`, ie:

```
<div id="root"></div>
```

The bulk of the content (ie: your Components) would then be dynamically added to the body using JavaScript and the full page would be rendered. Unfortunately, this method of rendering content means that it is more difficult for search engines to index your pages and **SEO (Search Engine Optimization)** suffers as a result. Fortunately, since we are using Next.js, this is less of a problem:

From the [official Next.js documentation](#)

By default, Next.js pre-renders every page. This means that Next.js generates HTML for each page in advance, instead of having it all done by client-side JavaScript. Pre-rendering can result in better performance and SEO.

Each generated HTML is associated with minimal JavaScript code necessary for that page. When a page is loaded by the browser, its JavaScript code runs and makes the page fully interactive. (This process is called hydration.)

Hydration

Notice how the documentation mentioned "hydration" in its description of what occurs when a page is loaded by the browser. This is not the first time that we have seen this mentioned in these notes - recall the code to initialize the "date" state for our `<Clock />` component :

```
const [date, setDate] = useState(null); // Note: Never set this to  
unknown data obtained at run time (ie: new Date(), a random  
number, etc.) - see: https://nextjs.org/docs/messages/react-hydration-error
```

We were forced to set the initial value of "date" to "null" to avoid a potential **hydration error**. This is because the pre-rendered HTML is actually generated when the app does a new "build" (in "dev" mode, this is every time a change is made to your code). If we initialize the state to a dynamic value (ie: "new Date()"), then the page will be pre-rendered with a value that will be instantly out-of-date. When the page is then loaded at a different time, a hydration error occurs:

```
Unhandled Runtime Error  
Error: Text content does not match server-rendered HTML.
```

This is because the pre-rendered body of the page looks something like this:

```
<p>  
  Locale: <!-- -->: <!-- -->1:43:08 PM<!-- --> <!-- -->  
</p>
```

which instantly disagrees with the code generated on the first render, ie after "hydration" sometime later.

You will recall that to fix this issue, we placed our code to initialize the date value within the body of the **"useEffect"** hook's callback function:

```
useEffect(() => {
  setDate(new Date());
}, []);
```

The reason that this worked to solve the hydration error was because code in the callback defined in the `useEffect` hook only gets executed once the component is first "mounted" (ie: added to the DOM) after "hydration". It is not executed when the pre-rendered HTML is being generated. This causes the pre-rendered body of the page to look like the following:

```
<p>
  Locale: <!-- -->: <!-- --> <!-- -->
</p>
```

which avoids the content mis-match when the component is rendered after "hydration".

Fetching API Data after Hydration

Now that we are familiar with the concepts of "pre-rendering" and "hydration", it follows that a request for API data that must occur *after* hydration should be done within the `"useEffect"` hook as well. For example, consider the following "Post" component which fetches data from our familiar [JSON&125; Placeholder](#) dataset:

```

import { useState, useEffect } from "react";

export default function Post() {

  const [post, setPost] = useState();

  useEffect(() => { // NOTE: The callback function must not be
    "async" - Results in error: TypeError: destroy is not a function
    fetch(`https://jsonplaceholder.typicode.com/
  posts/1`).then(res => res.json()).then(data => {
    setPost(data);
  })
}, []);

return (
  <>
    <strong>User ID:</strong> {post?.userId}<br />
    <strong>Title:</strong> {post?.title}<br />
    <strong>Body:</strong> {post?.body}<br />
  </>
)
}

```

This component does not set any value for "post" in the state (leaving it 'undefined') and instead relies upon the callback function defined within "useEffect" to pull in the data and update the "post" value. This results in the pre-rendered HTML looking like the following:

```

<strong>User ID:</strong> <!-- --><br>
<strong>Title:</strong> <!-- --><br>
<strong>Body:</strong> <!-- --><br>

```

Once "hydration" occurs, the effect is executed and the "post" value is set (causing a render). This gives us:

```
<strong>User ID:</strong> <!-- -->1<!-- --><br>
<strong>Title:</strong> <!-- -->sunt aut facere repellat provident
occaecati excepturi optio reprehenderit<!-- --><br>
<strong>Body:</strong> <!-- -->quia et suscipit
recusandae consequuntur expedita et cum reprehenderit molestiae ut
ut quas totam nostrum rerum est autem sunt rem eveniet
architecto<!-- --><br>
```

NOTE: When fetching data on the client-side (as above, in the "useEffect" hook), Next.js recommends that **SWR** be used instead, as it handles "caching, revalidation, focus tracking, refetching on intervals, and more".

Using SWR, the above component would look like:

```
import useSWR from 'swr';

// define the "fetcher" function. This can also be defined
// globally using SWRConfig (https://swr.vercel.app/docs/global-configuration)
const fetcher = (url) => fetch(url).then((res) =>
  res.json());

export default function Post() {
  const { data, error } =
    useSWR('https://jsonplaceholder.typicode.com/posts/1',
    fetcher);

  return (
    <>
      <strong>User ID:</strong> {data?.userId}<br />
      <strong>Title:</strong> {data?.title}<br />
      <strong>Body:</strong> {data?.body}<br />
    </>
  );
}
```

For more information on using SWR, refer to the [official SWR documentation](#).

Fetching API Data for Pre-Rendered HTML

If the data that is coming back from the API is not likely to change, we may wish to include it in the pre-rendered HTML to speed up load times and provide greater SEO.

Next.js provides this functionality via a mechanism called [getStaticProps](#). This is essentially a function that Next.js runs on the server when the app is built in order to obtain data required to pre-render your pages. From our point of view, it is a function that we can export from any "page" component to provide data to any components on that page via "props".

Warning: This will *not work* with custom components defined within the "components" folder.

At the moment, we really only have one "page" component - the `<Home />` component declared in **index.html**. It should contain the `<Post />` component, ie:

```
import Post from '@/components/Post';

export default function Home() {
  return (
    <>
      <Post />
    </>
  );
}
```

if we wish to have the data for the <Post /> component fetched at build time, we must make use of the "asynchronous" `getStaticProps()` function in this file, ie:

```
import Post from '@/components/Post';

// This function gets called at build time
export function getStaticProps() {
  // Call an external API endpoint to get posts
  return new Promise((resolve, reject) =>{
    fetch('https://jsonplaceholder.typicode.com/posts/1')
      .then(res => res.json())
      .then(data =>{
        resolve({ props: { staticPost: data } })
      })
    })
  }
}

export default function Home(props) {
  console.log(props); // props.staticPost should contain our data
  return (
    <>
      <Post />
    </>
  );
}
```

Here, we have exported an extra function above our "Home" component definition. The purpose of this function is to provide the exported page component (ie: "**Home**", in this case) with additional props that contain data to be pre-rendered by the component and/or the child components. The function always returns a promise which resolves with an object that contains one of the following properties:

- **props**: "a key-value pair, where each value is received by the page

component. It should be a serializable object so that any props passed, could be serialized with `JSON.stringify()`.

- **redirect**: "The redirect object allows redirecting to internal or external resources. It should match the shape of `{ destination: string, permanent: boolean }`.
- **notfound**: "allows the page to return a 404 status and 404 Page. With `notFound: true`, the page will return a 404 even if there was a successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author."

NOTE: an optional "**revalidate**" is also available, which allows you to update static pages after you've built your site. See: [Incremental Static Regeneration \(ISR\)](#) for more information

Finally, since this function always returns a promise is often written using the `async / await` syntax, ie:

```
// This function gets called at build time
export async function getStaticProps() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://jsonplaceholder.typicode.com/posts/1');
  const data = await res.json();

  return { props: { staticPost: data } };
}
```

Passing the `staticPost` prop to "Post"

Now that we know we can fetch data at build time for "page" components, the

final step is to refactor any components contained on the page that use that data to accept it as a property (props). In our case, this is the `<Post />` component. At the moment, it is in charge of fetching its own data on demand at runtime (per request) by placing the "fetch" code within the "useEffect" callback:

```
import { useState, useEffect } from "react";

export default function Post() {

  const [post, setPost] = useState();

  useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/
posts/1`).then(res => res.json()).then(data => {
      setPost(data);
    })
  }, []);

  return (
    <>
      <strong>User ID:</strong> {post?.userId}<br />
      <strong>Title:</strong> {post?.title}<br />
      <strong>Body:</strong> {post?.body}<br />
    </>
  )
}
```

since we know that the same data is available in the parent component, we can refactor this code to use "props" instead:

```
export default function Post(props) {
  return (
    <>
```

Finally, we must ensure that the `<Post />` component actually receives the props from the parent "page" component:

```
export default function Home(props) {
  return (
    <>
      <Post post={props.staticPost} />
    </>
  );
}
```

Now, if you try viewing the component again you should see the data as before, however now the pre-rendered content of the page contains the data:

```
<strong>User ID:</strong> <!-- -->1<!-- --><br>
<strong>Title:</strong> <!-- -->sunt aut facere repellat provident
occaecati excepturi optio reprehenderit<!-- --><br>
<strong>Body:</strong> <!-- -->quia et suscipit
recusandae consequuntur expedita et cum reprehenderit molestiae ut
ut quas totam nostrum rerum est autem sunt rem eveniet
architecto<!-- --><br>
```

Conditionally Displaying Data

So far, we have seen how we can render a value in JSX by placing an expression within curly braces `{...}`. This expression is then evaluated and used in place within our JSX, either to:

- render the data in place, ie:

```
{date.toLocaleTimeString()}
```

- provide a value to a property, ie:

```
<img src={user.avatarUrl} />
```

However, we actually have a great deal of control over how the data is displayed using this syntax. Since the content between the curly braces `{...}` is an *expression*, we can use well known JavaScript syntax and functions to control our output.

Before we move on to the examples, let's assume that we have the following static collection of data hardcoded in the state of a component:

```
const [users, setUsers] = useState([
  { user: 'fred', active: false, age: 40 },
  { user: 'pebbles', active: false, age: 1 },
  { user: 'barney', active: true, age: 36 },
]);
```

Logical && Operator (If)

First, let's take a look at a situation where we may only want to render some data under a specific condition. For example, say we only want to show the 'user' name if the user is "active". To accomplish this, we can leverage the **&&** Operator.

```
return <div>{users[0].active && <p>{users[0].user} is  
Active!</p>}</div>
```

Ternary Operator (If-Else)

Next, let's look at how we can use the **ternary operator**, ie: `(age > 18) ? "adult" : "minor"` to render a different `<p>` element depending on whether or not the user is "active".

```
return (  
  <div>  
    {users[0].active ? <p>{users[0].user} is Active!</p> :  
     <p>{users[0].user} is Inactive!</p>}  
  </div>  
);
```

Array.map() (Iteration)

One extremely common task is iterating over a collection and outputting each element using a consistent format. This could be rows in a table, items in a list,

other components, etc. To achieve this within our JSX code, we can use the `Array.map()` method, ie:

```
return (
  <table>
    <thead>
      <tr>
        <th>User</th>
        <th>Active</th>
        <th>Age</th>
      </tr>
    </thead>
    <tbody>
      {users.map((user) => (
        <tr>
          <td>{user.user}</td>
          <td>{user.active ? 'yes' : 'no'}</td>
          <td>{user.age}</td>
        </tr>
      )))
    </tbody>
  </table>
);
```

While this does work to render each user in its own `<tr>` element, we actually have one small problem. If you open the console in the browser, you will see an error: "Warning: Each child in a list should have a unique "key" prop." According to the documentation:

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.

Normally, we would have a unique id to work with (ie `"_id"` from MongoDB), however with our list we don't have any "stable" id's to work with. In this case,

we can make use of the 2nd parameter to the "map()" method - the "index". This requires us to change our JSX to use the following code:

```
{users.map((user, index) => (
  <tr key={index}>
    <td>{user.user}</td>
    <td>{user.active ? 'yes' : 'no'}</td>
    <td>{user.age}</td>
  </tr>
))}
```

Returning Null

Finally, we can actually choose not to render anything by returning **null**, for example:

```
if (!loading) {
  return <p>Done Loading! - TODO: Show the data here</p>
} else {
  return null; // don't render anything - still loading
}
```

Example Code

You may download the sample code for this topic here:

| Handling-Events-Rendering-Data

Layouts & Pages

Until now, we have only worked with a single "page" in our Next.js app: "Home", defined by the "Home" component, located within the `index.js` file in the "pages" folder. In this section, we explore how to add additional routes (pages) to the application.

Adding a Route

Each "route" in our application / site is defined by the name of the file within the "pages" folder. In this case, we only have one route "/" because there is only one file in the "pages" folder: **index.js** (not including `_app.js` which is used to initialize pages - discussed later on).

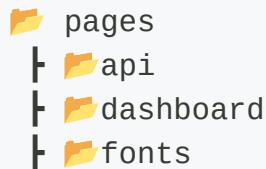
We will be adding another route at "/about". When complete, our application will have two routes:

- Route: "/" - Renders component from "pages/index.js"
- Route "/about" - Renders component from "pages/about.js"

Here's how to add the /about route:

1. In the project's `pages` folder, add a file called: `about.js`

Pages folder structure



2. In the `about.js` file, add this code:

```
/pages/about.js
```

```
export default function About() {
  return (
    <>
      <p>About</p>
    </>
  );
}
```

 **CAUTION**

The name of the exported component does *not* need to match the file name, since it is the file name that defines the route. However, the name of the component **must be capitalized** as usual.

3. In the browser, visit the `/about` route. The page contents will appear.

Nested Routes

It is sometimes necessary to define routes that are "nested" ie: `"/dashboard/preferences"`. To achieve this, we simply need to recreate the nested route as a nested folder structure within the `pages` folder. Additionally, if we place an **index.js** file nested within another folder, Next.js will automatically serve that component as the default route for that folder.

Let's expand our current list of routes to add two "dashboard" routes:

- Route: `"/dashboard"` - Renders component from `"pages/dashboard/index.js"`

- Route **"/dashboard/preferences"** - Renders component from "pages/dashboard/preferences.js"

1. In the `pages` folder, add a folder called `dashboard`

```
📁 pages
  |- 📁 api
  |- 📁 dashboard <-- create this folder
  |- 📁 fonts
  |- index.js
  |- ... etc
```

2. Within the `pages/dashboard` folder, add two new components:

- **File:** "pages/dashboard/index.js"
- **File:** "pages/dashboard/preferences.js"

```
📁 pages
  |- 📁 api
  |- 📁 dashboard
    |- index.js      <-- add this file
    |- preferences.js <-- add this file
  |- 📁 fonts
  |- index.js
  |- ... etc
```

3. In each file, add the code for a new endpoint:

- Code snippet for the `dashboard/index.js` page:

```
pages/dashboard/index.js
```

```
export default function DashboardHome() {
  return (
    <>
      <p>Dashboard Home</p>
    </>
  );
}
```

- Code for the `dashboard/preferences` page:

pages/dashboard/preferences.js

```
export default function DashboardPreferences() {
  return (
    <>
      <p>Dashboard Preferences</p>
    </>
  );
}
```

3. When complete, the application has this updated list of routes:

- Route: "/" - Renders component from "pages/index.js"
- Route "/about" - Renders component from "pages/about.js"
- Route: "/dashboard" - Renders component from "pages/dashboard/index.js"
- Route "/dashboard/preferences" - Renders component from "pages/dashboard/preferences.js"

Layouts

If we have components that are re-used across multiple pages (ie: a common

"navbar" / "footer"), we can place these in a custom "Layout" component, which can then be specified for the whole application. Let's begin by creating a very simple layout without any extra components.

1. In the project's `components` folder, add a new file called `layout.js`:

```
my-app
├── components
│   └── layout.js      <-- add this file
├── pages
├── public
├── styles
├── README.md
└── package-lock.json
... etc
```

⚠ CAUTION

Since layouts are not "pages", they **must not** be placed within the "pages" folder - instead, place them in a "components" folder, as we have previously done.

2. In the `components/layout.js` file, add this code:

```
components/layout.js
```

```
export default function Layout(props) {
  return (
    <>
    <h1>Pages / Routing in Next.js</h1>
    <a href="/">Home</a> | <a href="/about">About</a> | <a
    href="/dashboard">Dashboard</a> | <a href="/dashboard/
    preferences">Dashboard Preferences</a>
```

You will notice that this looks exactly like any other custom component that we have created before, except instead of rendering specific data passed to props (or within the "state" of the component), it renders "`props.children`". This is essentially a placeholder that renders the children of the component at a specific point in the layout. When using "`props.children`", we will be including the component using the form "`<Layout></Layout>`" instead of the "self-closing" notation (ie: `<Layout />`) that we have used so far.

3. To include this layout in all our pages, we must open the aforementioned `_app.js` file located within the "pages" folder and "wrap" the `<Component {...pageProps} />` with our new layout:

```
pages/_app.js
```

```
import Layout from '@/components/layout';
import '@/styles/globals.css';

export default function App({ Component, pageProps }) {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  );
}
```

4. Navigate to the website. You will see a common navigation bar with links to all our newly created routes.

! **INFO**

It is also possible to configure layouts on a page-by-page basis. See the [official documentation](#) for more information.

Client-Side Page Transitions

When browsing the site, you may have noticed that each route takes a moment to load, despite the fact there is little difference between the UI / DOM for each route. This is because the entirety of the app (ie: all .js files) is downloaded each time a new route is accessed - this can be confirmed by viewing the network activity in the dev tools. Fortunately, Next.js provides a mechanism to transition between pages without reloading every .js file for the application. This is known as "client-side routing" / "client-side route transitions".

Link Component

Client-side routing can be achieved by replacing anchor tags (...) with custom **"Link" Components** (...) containing the "href" attribute. Let's test this by refactoring our Layout to use "Link":

components/layout.js

```
import Link from 'next/link';

export default function Layout(props) {
  return (
    <>
      <h1>Pages / Routing in Next.js</h1>
      <Link href="/">Home</Link> | <Link
      href="/about">About</Link> | <Link
      href="/dashboard">Dashboard</Link> | <Link href="/dashboard/
      preferences">Dashboard Preferences</Link>
      <hr />
      <br />
```

You will find that the page transitions are much faster and that only the relevant .js is loaded when each route is first accessed. Once that route is accessed for a second time, nothing is download from the server - loading the required .js files on demand is known as "[Lazy Loading](#)".

It is also important to note that the "Link" component accepts the following props (from the [official documentation](#)):

- **href** - The path or URL to navigate to. This is the only required prop
- **as** - Optional decorator for the path that will be shown in the browser URL bar.
- **legacyBehavior** - Changes behavior so that child must be <a>. Defaults to false.

NOTE: This is important to use if [the child is a custom component that wraps the <a> tag](#)
- **passHref** - Forces Link to send the href property to its child. Defaults to false
- **prefetch** - Prefetch the page in the background. Defaults to true. Any <Link /> that is in the viewport (initially or through scroll) will be preloaded. Prefetch can be disabled by passing prefetch=. When prefetch is set to false, prefetching will still occur on hover. Pages using Static Generation will preload JSON files with the data for faster page transitions. Prefetching is only enabled in production.
- **replace** - Replace the current history state instead of adding a new url into the stack. Defaults to false
- **scroll** - Scroll to the top of the page after a navigation. Defaults to true

- **shallow** - Update the path of the current page without rerunning getStaticProps, getServerSideProps or getInitialProps. Defaults to false
- **locale** - The active locale is automatically prepended. locale allows for providing a different locale. When false href has to include the locale as the default behavior is disabled.

useRouter Hook

If we wish to achieve the same effect from within our component logic (such as in an "onClick" event handler), we must make use of the "**useRouter**" hook from "next/router":

```
import { useRouter } from 'next/router';
```

This hook will be used to obtain a "**router**" object from within our component by invoking "useRouter()":

```
const router = useRouter();
```

The **router object** itself has *many useful properties*, such as:

- **pathname**: The current route - the path of the page in /pages.
- **query**: The query string parsed to an object, including dynamic route parameters. It will be an empty object during prerendering if the page doesn't have data fetching requirements. Defaults to

However, at the moment we are most interested in the "**push**" method to transition to a new route:

```
router.push('/'); // navigate to the home route "/"
```

This method accepts three arguments, ie:

- **url**: The URL to navigate to (either as a string or **urlObject**)
- **as**: Optional decorator for the path that will be shown in the browser URL bar.
- **options**: Optional object with the following configuration options:
 - **scroll**: Optional boolean, controls scrolling to the top of the page after navigation. Defaults to true
 - **shallow**: Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to false
 - **locale**: Optional string, indicates locale of the new page

Dynamic Routes

For certain applications, using predefined paths (such as "/", "/about", "/dashboard", etc.) may not be enough. We may also wish to have paths that support "route" parameters, as well as "query" parameters, such as:

- "/product/**id**" - where **id** can be any value, ie: 5
- "/products?**query**" - where **query** can be an optional query string, ie: "page=1&category=stationary"

While there is no change to our filenames to support optional query strings, we do have to use a special convention if we wish to accommodate "route" parameters. For example the file:

```
"pages/product/[id].js"
```

can be used to create a "product" page that accepts an "id" route parameter.

Reading Route Parameters

In order to actually read and display a route parameter (product "id" in the above case), the "**router**" object once must again be obtained using the "**useRouter**" hook from "next/router":

File: "pages/product/[id].js"

```
import { useRouter } from 'next/router';

export default function Product() {
```

Here, we have made use of the "query" property of router to obtain an object containing a property: "id" that matches the filename: [id].js"

Reading Query Parameters

Reading query parameters follows the exact same procedure. Let's see how we go about reading the "page" and "category" query parameters as specified above:

```
import { useRouter } from 'next/router'

export default function Products() {
  const router = useRouter()
  const { page } = router.query;
  const { category } = router.query;

  if (page && category) {
    return <p>Products for page: {page} & category: {category}</p>
  } else {
    return <p>Page and/or Category Parameters Missing</p>
  }
}
```

Once again we have made use of the "query" property of router to obtain an object containing properties that match each query parameter. However, since these are technically optional parameters (it is possible to navigate to the route without them), we should check to make sure they exist before we render any data related to them.

NOTE: if there are duplicate values for a query parameter, ie: "category=stationary&category=office", then the "category" property will contain an array containing the values: ["stationary", "office"]

Dynamic Routes with 'getStaticProps'

If you wish to use 'getStaticProps' with dynamic routes, things become slightly more complicated. In a route that always fetches data from the same API endpoint, having the data available for pre-rendering is straightforward since the "fetch" statement always pulls data from the same location. However, if the route is dynamic, it becomes more difficult to pull the data ahead of time using 'getStaticProps'. To solve this problem, Next.js has an asynchronous "`getStaticPaths`" function that must be used in *addition* to "getStaticProps". To see how this works, consider the following example:

File: "pages/post/[id].js"

```
export async function getStaticPaths() {
  // pre-render and support post/1 through post/5 only
  return {
    paths: [
      { params: { id: "1" } },
      { params: { id: "2" } },
      { params: { id: "3" } },
      { params: { id: "4" } },
      { params: { id: "5" } }
    ], fallback: false // any pages not identified above, will
result in a 404 error, ie post/6
  }
}

export async function getStaticProps(context) {

  const res = await fetch(`https://jsonplaceholder.typicode.com/`)
```

Here, we have a component "Post" that is located within the "pages" directory at "pages/post/[id].js". We also have "getStaticProps" as we have seen it before, except in this case it accepts a "context" parameter that provides the function with one of the id's in the list of paths returned from the "getStaticPaths" function (ie: context.params.id).

All of the supported route parameters are included in the "paths" property of the return value for the "getStaticPaths" function - if the user tries to access a route containing a parameter that is not listed in "paths", a 404 error will be returned. This is because an additional "fallback" property has explicitly been set to false. If this functionality is not desired, then fallback can be set to true (see: [the official documentation for "fallback: true"](#) for more information).

Finally, you will notice that in our "Post" component, we do not have to explicitly read the route parameter using the "[useRouter](#)" hook (as above). The "post" property in "props" will automatically contain the data for the correct "id" parameter.

Custom Error Pages

Finally, you may have noticed by now that Next.js provides its own **404** and **500** error pages. However, it is possible to create your own as well. In this case all that is required is that you create a "404.js" or "500.js" file in the "pages" directory, ie:

File: "pages/404.js"

```
export default function Custom404() {
  return <h1>404 - Page Not Found</h1>
}
```

File: "pages/500.js"

```
export default function Custom500() {  
  return <h1>500 - Server-side error occurred</h1>  
}
```

Example Code

You may download the sample code for this topic here:

| Pages-Routing

"API" Routes

Introduction

One of the more interesting features of Next.js is the ability specify routes for a Web API to be executed on the same server serving your site. This extends the functionality of the server, in that it not only pre-renders and serves your files but also can act as a back end API for your application!

Recall, when we first created a boilerplate Next.js app, we were given an "index.js" file within the "pages" folder that rendered the exported "Home" component at the default route ("/"). We were also provided with an "api" folder, containing a single "hello.js" file. Therefore, if we follow the structure of the "pages" folder, we should be able to access a "hello" route from the "api" folder, ie: "<http://localhost:3000/api/hello>". This is indeed the case, and the server will respond with:

```
{ "name": "John Doe" }
```

Route Definitions

If you open the "hello.js" file, you will see some code that looks very similar to how [routes are defined in "Express"](#), ie:

File: "pages/api/hello.js"

```
export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' });
```

'req' and 'res'

You will see that both "req" and "res" objects are available to the exported callback function in order to give us access to the HTTP request / response. However, it is important to note that these are **not** the same as the "**Request**" and "**Response**" objects provided by "**Express**", though they serve the same purpose.

Additionally, "middleware" functions have been built in to parse the incoming request, which gives the "req" object the following *additional* properties:

- **req.cookies** - An object containing the cookies sent by the request.
Defaults to
- **req.query** - An object containing the query string. Defaults to "" - Note: route parameter values also included.
- **req.body** - An object containing the body parsed by content-type, or null if no body was sent

Similarly, some "helper functions" have been made available on the "res" object to provide *additional* functionality. These are similar to what is offered by "Express":

- **res.status(code)** - A function to set the status code. code must be a valid HTTP status code
- **res.json(body)** - Sends a JSON response. body must be a serializable object
- **res.send(body)** - Sends the HTTP response. body can be a string, an object or a Buffer

- **res.redirect([status,] path)** - Redirects to a specified path or URL. status must be a valid HTTP status code. If not specified, status defaults to "307" "Temporary redirect".
- **res.revalidate(urlPath)** - Revalidate a page on demand using getStaticProps. urlPath must be a string.

HTTP Methods

At the moment, the "hello" API route responds to "GET" requests only. If we wish to extend this to match other HTTP methods (ie: "POST"), we can leverage the **"method"** property of the "req" object:

```
export default function handler(req, res) {
  const { method } = req;

  switch (method) {
    case 'GET':
      res.status(200).json({ name: 'John Doe' });
      break;
    case 'POST':
      // return the 'name' value provided in the body of the
      // request
      res.status(200).json({ name: req.body.name });
      break;
    default:
      // send an error message back, indicating that the method is
      // not supported by this route
      res.setHeader('Allow', ['GET', 'POST']);
      res.status(405).end(`Method ${method} Not Allowed`);
  }
}
```

Dynamic Routes

As with regular routing, API routes may also contain "route parameters". These must be defined in a similar way, in that they must exist in their own .js file with the desired route parameter as the file name. For example, if we wish to match the route "/api/users/**id**" (where **id** is the unknown parameter), we would create the following file:

File: "/pages/api/users/[id].js"

```
export default function handler(req, res) {
  const { id } = req.query; // "id" route parameter
  res.status(200).json({ name: `user ${id}` });
}
```

If we wish to reference the route parameter in the route definition, it can be accessed using **req.query**.

Web API Structure

Using the above techniques, it is possible to create routes that match that of a typical Web API. For example, consider the following files:

File: "/pages/api/users/index.js"

```
export default function handler(req, res) {
  const { name } = req.body;
  const { method } = req;

  switch (method) {
```

File: "/pages/api/users/[id].js"

```
export default function handler(req, res) {
  const { id } = req.query;
  const { name } = req.body;
  const { method } = req;

  switch (method) {
    case 'GET':
      // Read data from your database
      res.status(200).json({ message: `TODO: Get User with id: ${id}` });
      break;
    case 'PUT':
      // Update data in your database
      res.status(200).json({ message: `TODO: Update User with id: ${id} - Set Name: ${name}` });
      break;
    case 'DELETE':
      // Delete data in your database
      res.status(200).json({ message: `TODO: Delete User with id: ${id}` });
      break;
    default:
      res.setHeader('Allow', ['GET', 'PUT', 'DELETE']);
      res.status(405).end(`Method ${method} Not Allowed`);
  }
}
```

Here, we have accounted for each of the major operations, ie:

| Route | HTTP Method | Description |
|------------|-------------|-------------------|
| /api/users | GET | Get all the users |

| Route | HTTP Method | Description |
|----------------|-------------|------------------------------------|
| /api/users | POST | Create a user |
| /api/users/:id | GET | Get a single user |
| /api/users/:id | PUT | Update a user with new information |
| /api/users/:id | DELETE | Delete a user |

Using MongoDB (Mongoose)

If we wish to extend the API structure to work with real data (ie: using MongoDB Atlas), we can use the familiar "[Mongoose](#)" ODM.

To get started, install it as a dependency:

```
npm install mongoose
```

Next, since the database connection will be shared in multiple files (ie: "/pages/api/users/[id].js" and "/pages/api/users/index.js"), we should place the database **model** and **connection** logic in a separate file (or files) somewhere within our project folder. Since this code is not responsible for rendering a specific route or component, we will create a new "**lib**" folder and place it there:

File: lib/dbUtils.js

```

import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true,
  },
});

mongoose.models = {};
export const UserModel = mongoose.model('users', userSchema);

export async function mongooseConnect() {
  if (mongoose.connections[0].readyState) {
    return true;
  }

  try {
    await mongoose.connect(`Your MongoDB Connection String Here`);
    return true;
  } catch (err) {
    throw new Error(err);
  }
}

```

You can see from the above code, that we are exporting both the **UserModel** and the **mongooseConnect()** function to be used in our API routes. You will also notice that we have created our "Schema" and reset the "models" before defining the "UserModel" (failing to do so may result in an "OverwriteModelError", ie: "Cannot overwrite 'users' model once compiled.").

Additionally, you will notice that our "mongooseConnect()" function checks to see if our connection is in a "**ready state**". If this property is *falsy* ie: **0**, then our connection is **disconnected** and we must create a new connection using

"mongoose.connect()". If the ".readystate" property is *truthy*, then this function does not need to create a new connection and simply returns true.

With our "dbUtils.js" file complete, we can focus on adding the remainder of the **CRUD Operations** to our API routes, specifically:

File: "/pages/api/users/index.js"

```
import { UserModel, mongooseConnect } from '@/lib/dbUtils';

export default async function handler(req, res) {
  const { name } = req.body;
  const { method } = req;

  try {
    await mongooseConnect();

    switch (method) {
      case 'GET':
        let users = await UserModel.find().exec();
        res.status(200).json(users);
        break;
      case 'POST':
        const newUser = new UserModel({ name });
        await newUser.save();
        res.status(200).json({ message: `User: ${name} Created` });
        break;
      default:
        res.setHeader('Allow', ['GET', 'POST']);
        res.status(405).end(`Method ${method} Not Allowed`);
    }
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
}
```

File: "/pages/api/users/[id].js"

```
import { UserModel, mongooseConnect } from '@/lib/dbUtils';

export default async function handler(req, res) {
  const { id } = req.query;
  const { name } = req.body;
  const { method } = req;

  try {
    await mongooseConnect();

    switch (method) {
      case 'GET':
        let users = await UserModel.find({ _id: id }).exec();
        res.status(200).json(users[0]);
        break;
      case 'PUT':
        await UserModel.updateOne({ _id: id }, { $set: { name: name } }).exec();
        res.status(200).json({ message: `User with id: ${id} updated` });
        break;
      case 'DELETE':
        await UserModel.deleteOne({ _id: id }).exec();
        res.status(200).json({ message: `Deleted User with id: ${id}` });
        break;
      default:
        res.setHeader('Allow', ['GET', 'PUT', 'DELETE']);
        res.status(405).end(`Method ${method} Not Allowed`);
    }
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
}
```

In both files, we wait for mongooseConnect() to complete (either creating a connection, or reusing the current one) before we use the "UserModel" to perform our operations.

Middleware

If you are familiar with the popular "[Express.js](#)" web framework for Node.js, you will be familiar with the concept of "Middleware"

Middleware functions are functions that have access to the request object (req), the response object (res), and the next() function in the application's request-response cycle. The next() function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

<http://expressjs.com/en/guide/writing-middleware.html>.

Essentially, middleware allows us to execute functions in the 'middle' of a request/response cycle typically before a matching route (api / page) handler function is executed.

Next.js has a similar concept:

[In Next.js] middleware allows you to run code on the server before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Middleware executes before routes are rendered. It's particularly useful for implementing custom server-side logic like authentication, logging, or handling redirects.

<https://nextjs.org/docs/pages/api-reference/file-conventions/middleware>

Here, we have a function that is automatically executed as part of the request / response cycle in Next.js. It can be configured to apply logic to a specific route,

or conditionally with multiple routes.

Getting Started

To see how middleware in Next.js is implemented, create a "**middleware.js**" file within the root of your application folder (ie: "my-app/middleware.js"):

File: /middleware.js

```
export function middleware(request) {
  console.log('requested: ', request.url);
}
```

NOTE: If the middleware does not modify the response (discussed further down), you do not need to explicitly return or invoke a `.next()` method. Next.js will automatically continue processing the request.

If you test the server now and navigate to the default route "/", you will see that the middleware function has been executed once for every resource sent from our local server (`http://localhost:3000`) for the "/" page:

```
requested: http://localhost:3000/vercel.svg
requested: http://localhost:3000/next.svg
requested: http://localhost:3000/_next/static/media/
2AAF0723e720e8b9-s.p.woff2
requested: http://localhost:3000/_next/static/chunks/react-
refresh.js?ts=1693361554048
requested: http://localhost:3000/_next/static/chunks/
main.js?ts=1693361554048
requested: http://localhost:3000/_next/static/chunks/
webpack.js?ts=1693361554048
requested: http://localhost:3000/_next/static/chunks/
```

We are able to access the "url" property on the "request" object, because request is technically an instance of "**NextRequest**", which itself, is an extension of the native "**Request**" object.

Matching Paths

Now that we know that the "middleware" function is behaving correctly (ie: invoked as a part of the request / response cycle - before the request is completed), we should consider only applying it to certain paths, such as pages or api routes. To achieve this, we must update our "middleware.js" file to also export a "config" object with a "matcher" property:

```
export const config = {  
  matcher: '/',  
};
```

In the above case, having a matcher value of "/" will restrict the middleware function to *only* run on the "/" route. If we open the console with the current configuration, we will only see:

```
requested: http://localhost:3000/
```

Multiple Paths

Say we have a second route: "/about" that we would also like to match. This can be done by passing an **array of matchers** to the "matcher" property:

```
export const config = {  
  matcher: ['/','/about'],
```

Nested Paths (Wildcard)

There are many cases where we have nested paths, such as "/api/users". In addition to matching "/api/users", we may want to match all "/api/users" routes, such as "/api/users/123". This can be done using the `:path*` (which will also match routes such as "/api/users/a/b/c"):

```
export const config = {
  matcher: ['/api/users/:path*'],
};
```

NOTE: The matcher config allows full regex so matching like negative lookaheads or character matching is supported. For example:

`'/((?!api|_next/static|_next/image|favicon.ico).*)'` will match all request paths except for the ones starting with:

- api (API routes)
- _next/static (static files)
- _next/image (image optimization files)
- favicon.ico (favicon file)

[Next.js Docs - "matcher config"](#)

Conditionally

Finally, we may wish to perform different actions depending on which path is matched. In this case, we do not include the "matcher" config, and instead rely on the "request" parameter. Recall: this is an instance of ["NextRequest"](#), which gives us access to the "nextUrl" property, which itself includes "an extended, parsed, URL object that gives you access to Next.js specific properties such as

pathname, basePath, trailingSlash and i18n". This appears to be exactly what we need (i.e. manually examine the *pathname* and respond with the intended logic):

```
export function middleware(request) {
  if (request.nextUrl.pathname.startsWith('/about')) {
    console.log('Visiting About');
  }

  if (request.nextUrl.pathname.startsWith('/api/users')) {
    console.log('Visiting the Users API');
  }
}
```

Practical Examples

Now that we are able to add middleware functionality to a certain route / set of routes, let's see what kind of practical benefits this provides.

NOTE: When using middleware, we have access to a "NextResponse" object from "next/server" (ie: `import { NextResponse } from 'next/server'`). Using this object, we can perform some useful actions from our middleware, such as:

- redirect the incoming request to a different URL
- rewrite the response by displaying a given URL
- Set request headers for API Routes, getServerSideProps, and rewrite destinations
- Set response cookies
- Set response headers

Using Cookies

As we know, a "cookie" is a small chunk of data that is sent by a server and stored in the client's web browser using the header "**Set-Cookie**". Similarly, the data is automatically sent from the client back to the server, using the "**Cookie**" header, often used to manage session information. In Next.js we can implement this functionality in our middleware functions by using the "cookies" property on both the request ("NextRequest") and response ("NextResponse") objects.

In the below example, we have two routes: "/setCookie" (which is expecting a query parameter: "message") and "/getCookie". When the middleware matches the "/setCookie" route, it reads the query parameter "message" and adds it to the "Set-Cookie" response header. If the middleware matches the "/getCookie" route, it simply outputs the "message" cookie value to the console. It uses the "**next()**" function to continue routing:

```
import { NextResponse } from 'next/server';

export function middleware(request) {
  const response = NextResponse.next();

  if (request.nextUrl.pathname.startsWith('/setCookie')) {
    let cookieMessage =
      request.nextUrl.searchParams.get('message');
    response.cookies.set('message', cookieMessage);
  }

  if (request.nextUrl.pathname.startsWith('/getCookie')) {
    let cookie = request.cookies.get('message');
    console.log(cookie);
  }
}
```

NOTE: We can also manually set headers using `response.headers.set()`, ie:

```
response.headers.set('x-hello-from-middleware', 'hello');
```

URL Rewrites

It may also be beneficial to map a specific path in Next.js to a different one, either temporarily (ie: during testing or development of a new bug fix / feature) or permanently depending on a condition such as the users language preference. This is possible using URL "rewrites":

Rewrites allow you to map an incoming request path to a different destination path.

Rewrites act as a URL proxy and mask the destination path, making it appear the user hasn't changed their location on the site. In contrast, redirects will reroute to a new page and show the URL changes.

For example, the following code will check the "Accept-Language" header value for the existence of "fr". If it is found, the url will be "rewritten" to the "/fr/about" route. To the user, they will still be at "/about", but the page rendered will be from "/fr/about"

```
import { NextResponse } from 'next/server';

export function middleware(request) {
  const language = request.headers.get('Accept-Language');

  if (language.includes('fr')) {
    return NextResponse.rewrite(new URL('/fr/about',
request.url));
  }
}
```


Example Code

You may download the sample code for this topic here:

| [API-Routes-Middleware](#)

React Forms

A common workflow for working with web forms has typically involved correctly setting the "action", "method", and optionally "enctype" attributes of a <form> element to reference a specific route on a server using "POST". Upon form submission, the browser would generate a HTTP "POST" request, with the body of the request containing the "urlencoded" data for the form, ie: "the keys and values are encoded in key-value tuples separated by '>', with a '=' between the key and the value." It would be the responsibility of the server to decode / parse this data into an object, such that the form data can be persisted to a data store, etc.

When the back-end is a Web API however, this process has to change slightly to accommodate both the client-side requirements (likely submitting the form via an AJAX request) and server requirements (typically requiring the data to be encoded using JSON). In this scenario, certain things must happen on the client-side, ie:

- Explicitly handling the "submit" event for the form and preventing the default action of automatically sending a HTTP "POST" request
- Obtaining updated form values and generating JSON formatted data to reflect the current values of the form fields.

Controlled Components

React has identified this need for greater control over form components and has introduced the term "**Controlled Components**":

"In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In

React, mutable state is typically kept in the state property of components, and only updated with `setState()`.

We can combine the two by making the React state be the 'single source of truth'. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a 'controlled component'".

A very simple example of a controlled component can be seen in the below example. Here, we have included a single "input" control that not only obtains its initial value from the state ("user_name"), but also updates the state with any changes via the "onChange" event:

```
import { useState } from 'react';

export default function SimpleForm() {
  const [user_name, setUserName] = useState('Homer Simpson');

  function submitForm(e) {
    e.preventDefault(); // prevent the browser from automatically
    // submitting the form
    console.log(`form submitted - user_name: ${user_name}`);
  }

  return (
    <form onSubmit={submitForm}>
      User Name: <input value={user_name} onChange={(e) =>
        setUserName(e.target.value)} />
      <br />
      <br />
      <button type="submit">Update User Name</button>
    </form>
  );
}
```

By ensuring that the initial value of the form control is set to the value *currently in the state*, as well as updating the state with the current value of the form field whenever it is *changed*, we can say that the state is the "single source of truth". If the state always holds an "up-to-date" representation of the form, it can be used in the form submission handler to send the correct information to a Web API via an AJAX "POST" / "PUT", etc. request.

While this certainly works for smaller forms, it does not necessarily scale well and adding common features such as validation, keeping track of visited fields, handling varied types of input fields / select-multiple, etc. adds extra complexity to the code.

It is for these reasons that complete, third-party solutions have been created as an alternative to working directly with "Controlled Components" in React. Some popular solutions include:

- **Formik:** <https://formik.org>
- **React Hook Form:** <https://react-hook-form.com>

React Hook Form

For our purposes, we will be working with "**React Hook Form**", as it provides excellent performance, greatly simplifies working with forms and has great support for schema / HTML standard validation and handling errors.

Getting Started

To get started working with React Hook Form, we first must install it as a dependency from **npm**:

```
npm i react-hook-form
```

After this, we simply have to import the "useForm" hook and we can get started writing forms:

```
import { useForm } from 'react-hook-form';
```

As a simple example, let's begin by re-writing the above "Controlled Component" code, to use "react-hook-form" and the "useForm" hook:

```
import { useForm } from 'react-hook-form';

export default function UserForm() {
  const { register, handleSubmit } = useForm({
    defaultValues: {
      userName: 'Homer Simpson',
    },
  });

  function submitForm(data) {
    console.log(`form submitted - userName: ${data.userName}`);
  }

  return (
    <form onSubmit={handleSubmit(submitForm)}>
      User Name: <input {...register('userName')} />
      <br />
      <br />
      <button type="submit">Update User Name</button>
    </form>
  );
}
```

At first, you will notice one important difference - we did not import, nor use the "useState" hook at all in our solution. This is because React Hook Form manages its own internal state and if we wish to set an initial value for a form field, it can be done using either the "defaultValue" property on a specific form element, or by using the "defaultValues" property in the argument passed to the `"useForm"` hook (as above).

NOTE: While React Hook Form encourages the use of default values, it is also very likely that any initial form data will not be available until it has been obtained from a Web API. In this case it would be common practice to update the form from within the `"useEffect()"` hook, once the data has materialized. If this is the case, `"setValue"` may be used with a default value of `""` for `userName`:

```
const { register, handleSubmit, setValue } = useForm({
  defaultValues: {
    userName: '', // the documentation encourages default
    values
  },
});

useEffect(() => {
  // set the userName value
  setValue('userName', 'Homer Simpson');
}, []);
```

You will also notice that our `"submitForm"` function has changed, as it no longer accepts the submit event `"e"`. With React Hook Form, the submit function is instead invoked by `"handleSubmit"`, which automatically passes the form data to the callback function (ie: `"submitForm"`).

NOTE: It is still possible to obtain the submit event, by referencing a 2nd parameter `"e"`, in the `"submitForm"` function, ie:

```
function submitForm(data, e) {}
```

Finally, we made use of the "register" method to associate this control with React Hook Form. By "registering" a control with React Hook Form, we're essentially registering the onBlur and onChange callbacks, as well as setting the name property for the form control and assigning a "ref". From the React Hook Form documentation:

```
const { onChange, onBlur, name, ref } = register('firstName');

<input
  onChange={onChange} // assign onChange event
  onBlur={onBlur} // assign onBlur event
  name={name} // assign name prop
  ref={ref} // assign ref prop
/>

// same as above
<input {...register('firstName')} />
```

Registering Multiple Form Controls

As a more complete example, let's use the above methods to show a form containing all of the basic form controls (<input />, <select>...</select>, <textarea>...</textarea>). We will also set the initial values of the form controls from an object defined in the "useEffect" hook:

```
import { useForm } from 'react-hook-form';
import { useEffect } from 'react';

export default function UserForm() {
```

As you can see from the above code, there is no special syntax for registering one type of form control over another. All that is required is that the code:

```
{...register(fieldName)}
```

is placed on the form control, where "fieldName" is the name of the field that references the control.

Additionally, you will notice a small addition to the code in the "useEffect" hook:

```
// set the values of each form field to match "data"
for (const prop in data) {
  setValue(prop, data[prop]);
}
```

Here, we are looping through every property in the "data" object (using a **"for..in" loop**) and explicitly setting the value for every field in our form with the value of the matching property. This works, since the registered "fieldname" values in the form match the property names in the data object.

"Watching" Form Values

If you need to obtain the value of a form field as it's updated, it is possible to **"watch"** it and be updated with any changes. For example, if we wish to "watch" the "userName" field, such that we can show it to the user or use it to hide / show other fields in the form, all that is required is that we include "watch" and create a variable to hold the value, ie:

```
const { watch } = useForm();
```

```
const watchUserName = watch('userName');
```

```
<p>User Name: {watchUserName}</p>
```

For more information / special cases, see [the official documentation for "watch"](#)

Form Validation

React Hook Form makes simple form validation very straightforward by aligning with "browser native validation" attributes. As such, the following basic validation rules are provided:

- **required**: Indicates that the input must have a value before the form can be submitted.
- **min**: The minimum value to accept for this input (number)
- **max**: The maximum value to accept for the input (number)
- **minLength**: The minimum length of the value to accept for the input
- **maxLength**: The maximum length of the value to accept for the input
- **pattern**: The regex pattern for the input.

Adding Validation Rules to "register"

In addition to "registering" our form controls, the "register" function also accepts a second "options" parameter to configure how the control behaves, including setting validation rules.

For example, we will start with the following simple form:

```
import { useForm } from 'react-hook-form';
import { useEffect } from 'react';

export default function FormWithValidation() {

  const { register, handleSubmit, setValue } = useForm({
```

Here, we ask the user to modify their first name, last name and age, starting from initial values obtained in the "useEffect" hook. At the moment, there are no restrictions on what the user may enter, apart from the restriction built in to the input type "**number**".

If we wish to add specific validation rules for each form field individually, we can specify each validation rule as a *property* of the *RegisterOptions* parameter of the "**register**" method. For example, we can modify the 3 form fields above to use specific validation rules such as "required", "maxLength", "pattern", "min", "max", etc.

```
<form onSubmit={handleSubmit(submitForm)}>
  First Name: <br />
  <input {...register("firstName", { required: true,
  maxLength: 20 })} /><br /><br />

  Last Name: <br />
  <input {...register("lastName", { pattern: /^[A-Za-z]+$/i
})} /><br /><br />

  Age: <br />
  <input type="number" {...register("age", { min: 18, max: 99
})} /><br /><br />

  <button type="submit">Update User</button>
</form>
```

After this change has been made, the form should still appear the same, however if we try to break one of the validation rules (ie: removing the first name value) before submitting the form, we will notice that:

1. The (first) form field that is in violation of a validation rule is focused
2. The "submitForm" function does not run

Custom Validation Rules

In addition to the above "native validation", React Hook Form also allows us to create our own validation rules by specifying one or more callback functions as properties within a "validate" parameter to *RegisterOptions*. As a simple example, say we wish to ensure that we are only accepting "Age" values that are even, in the above form. We can add the following **custom** "onlyEven" validation rule to our "number" field:

```
<input type="number" {...register("age", { min: 18, max: 99, validate: { onlyEven: v => v % 2 == 0 } })} /><br /><br />
```

Showing Errors

While restricting the form submission and automatically focusing the problematic field is an effective way to perform client side validation, we are missing an important piece: showing a *description* of the error to the user, so that they can correct the mistake.

In order to actually show the errors, we must first add the "errors" object from "**formState**", to give us access to any errors in the form:

```
const { register, handleSubmit, setValue, formState: { errors } } = useForm({...});
```

This will give us access to the "errors" object, which will contain each form field that contains an error, as a property. Additionally, each specific validation error will also be identified for each form field using a "type" property. If there are no invalid fields, this object will be empty, ie "".

Using this, we can easily inspect a specific form control to see if it contains any errors and if so, which rules have been violated. For example, if we wish to show errors for the "firstName" field, we could use the following code:

```
<input {...register("firstName", { required: true, maxLength: 20 })} />
{errors.firstName?.type === "required" && <span><br />First Name
is required</span>}
{errors.firstName?.type === "maxLength" && <span><br />First Name
Cannot contain more than 20 characters</span>}
<br /><br />
```

Notice how we use the "**optional chaining operator**" before checking the "type" of the error message. This ensures that we do not receive the following error if the form field is currently valid (ie: no errors):

```
TypeError: Cannot read properties of undefined (reading 'type')
```

Also, if you try running the example again, you should see that the error messages show up only after the form is first submitted. After this first submit, the errors are shown / hidden as the user modifies the form. This lets the user know immediately if they have corrected the error.

NOTE: In addition to the "errors" object, you can include additional objects such as "dirtyFields" and "touchedFields" to monitor which fields have been modified and visited.

Highlighting Fields

Given that the "errors" object contains property names of all form fields that are currently in violation of a validation rule, it is a simple task to conditionally add a CSS class to a form field that is in error. For example, if we had the class

"inputError", we could conditionally add it to the field using the code:

```
className={errors.firstName && "inputError"}
```

Here, we check if the "errors" object contains a *firstName* property, and if it does, add the "inputError" class.

Disabling the "submit" Button

Another interesting UI improvement that can be made using "errors" is conditionally disabling the "submit" button if the form is "in error" (has one or more fields with validation errors). This can be done by counting how many properties exist on the "errors" object using "[Object.keys\(\)](#)", since only form fields that are in error should have a corresponding property within the errors object, ie:

```
<button type="submit" disabled={Object.keys(errors).length > 0}>Update User</button>
```


Example Code

You may download the sample code for this topic here:

| [Forms-Introduction](#)

Shared State with Props & Context

So far, we have discussed "**Component State**", ie: managing data / values that are associated with a specific *component*. These values can change over time and when this happens, the component is re-rendered to reflect the updated "state" data. It is this feature that allows us to create dynamic components that change over time, such as our original "clock" example.

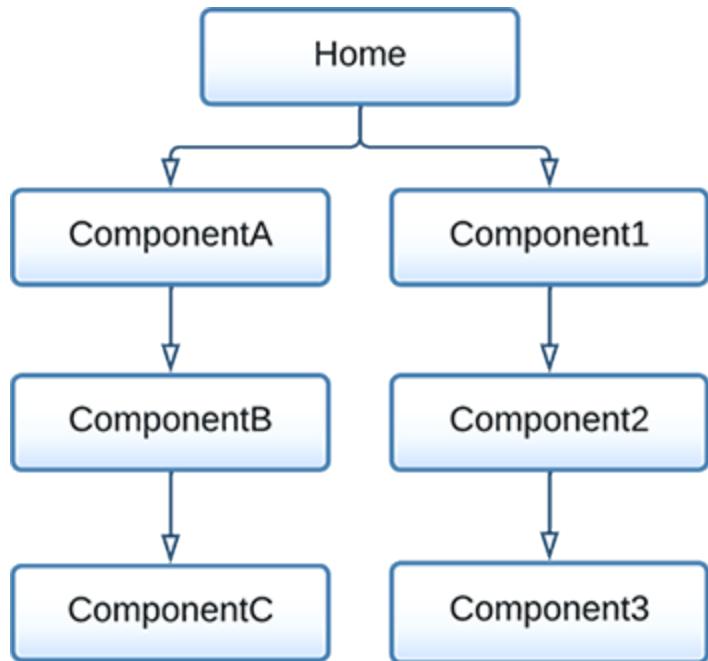
However, there are many circumstances in which the notion of "state" extends beyond an individual component. Consider a typical e-commerce site that allows users to pick and choose products to add to a "cart". Once the user is ready, they can modify the cart and / or purchase the items. In this situation, the concept of the "cart" and the items it contains must be shared by more than one component. This is because the navbar may show how many items are in the cart, or specific products may show an "added to cart" message. Additionally, the button to actually add an item to a cart would exist in a separate component from the list of items in the cart.

Essentially, what we would like to do is have the notion of an "application level" state for the "cart" that may be used by one or more components in the "component tree" (the components used in your application / site, leading all the way back to a single component).

Prop Drilling

Using what we know so far, there is a way to implement the concept of an application level state in our application / site. What we must do, is declare the state in a top-level component and pass it down from component to

component via "props", so that it may be accessed by the nested component that requires it. This is informally known as "prop drilling", since we're "drilling" through multiple components via "props" to deliver the state to the nested component. For example, consider the following tree of components:



In this case, both "ComponentA" and "Component1" are rendered by the "Home" component. "ComponentA" renders "ComponentB", which renders "ComponentC" and likewise, "Component1" renders "Component2", which renders "Component3".

Now, say there's a button on "ComponentC" that increments a counter value by 1. We have seen this before when introducing "user events" with "onClick". However, the difference here is that the component *responsible for displaying* the counter value as it increases is actually "Component3", **not** "ComponentC", where the button is rendered.

To solve this, we do not declare the "count" state in "ComponentC" with the

button, but instead declare it in a *top level* component.

You will recall from the discussion on "Layouts & Pages" that Next.js actually has a **high-level component** declared in "pages/_app.js" - this is where we placed our `<layout>...</layout>` component so that it will be available on all pages. This is also where we will declare our counter state and pass it to page components via props, ie:

File: "/pages/_app.js"

```
import '@/styles/globals.css';
import { useState } from 'react';

export default function App({ Component, pageProps }) {
  const [count, setCount] = useState(0); // declare high-level
  "count" state
  return <Component {...pageProps} count={count}
  setCount={setCount} /> // pass it as props to the page components
}
```

Here, we declare the state in "App" and ensure that it can be accessed by passing the values as props "count" and "setCount", respectfully.

Now, in the "Home" component when we render "ComponentA" and "Component1", we will continue to pass the props so that "ComponentB" and "Component2" have access, and so on. We are passing ("drilling") the state through every component via "props" until it reaches a component that requires it.

File: "/pages/index.js"

```
<Component1 count={props.count} />
<ComponentA setCount={props.setCount} />
```

The "Home" component (declared in "index.js") simply passes the "count" and "setCount" properties to the appropriate components, so that in addition to rendering "Component2", "Component1" takes the "count" prop and passes it on, until it can be used by "Component3". The same is true for "ComponentA", however it takes the "setCount" prop and passes it on, until it can be used by "ComponentC".

Let's skip ahead and look at the final components; "ComponentC" and "Component3" to see how they make use of the "count" and "setCount" props that have been passed down to them:

File: "/components/Component3.js"

```
export default function Component3(props) {
  return <>Value: {props.count}</>
}
```

File: "/components/ComponentC.js"

```
export default function ComponentC(props) {
  return <button onClick={(e) => props.setCount(n => n +
  1)}>Increase Value</button>
}
```

NOTE: If the new state is computed using the previous state, you can **pass a function to setState**, which receives the previous value.

As you can see, "count" and "setCount" can be accessed directly from the "props" object, since it has been passed down from component to component.

Problems with Prop Drilling

Depending on the complexity of the layout / application, "prop drilling" can add extra properties to components that do not actually need them; the only reason they were passed the props, is to hand them off to their child components. This makes components harder to reuse and adds an extra dependency between components that we would like to avoid. This can also get very cumbersome if we have many application level state values to manage.

Additionally, there is an impact on performance when using this method of passing state from component to component. Every time the state is accessed / changed in a child component, **every** parent component in the "prop" chain (ie: "Home", "Component1", "Component2", "Component3", "ComponentA", "ComponentB", "ComponentC") **also** gets rendered.

Context

As we have seen, our first approach to solving "application level" state works, but causes organizational and performance problems with our application / site. It is for these reasons that as of React 16, "**Context**" was introduced:

"In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree."

This is certainly an improvement to "prop drilling", so let's implement it in our scenario above. As before, we will declare a "count" state in App, but instead of

passing various "props" we will instead create "Context" objects and wrap our `<Component {...pageProps} />` with "Provider" components:

File: "/pages/_app.js"

```
import '@/styles/globals.css';
import { useState, createContext } from 'react';

export const CountContext = createContext();
export const SetCountContext = createContext();

export default function App({ Component, pageProps }) {
  const [count, setCount] = useState(0);

  return (
    <>
      <CountContext.Provider value={count}>
        <SetCountContext.Provider value={setCount}>
          <Component {...pageProps} />
        </SetCountContext.Provider>
      </CountContext.Provider>
    </>
  );
}
```

Notice how we create and export a new "Context" object for every value that we wish to make "global" to our application (ie: accessible by deeply nested components, such as "Component3" and "ComponentC"). We then wrap our `<Component {...pageProps} />` with the associated "Provider" components with a "value" prop, to make the context values available to child components (pages).

This eliminates the need for "prop drilling", so we do not need to update any components *except* the components that must make use of the context, ie:

"Component3" and "ComponentC":

File: "/components/Component3.js"

```
import { useContext } from 'react';
import { CountContext } from '@/pages/_app';

export default function Component3() {
  const count = useContext(CountContext);
  return <>Value: {count}</>
}
```

File: "/components/ComponentC.js"

```
import { useContext } from 'react';
import { SetCountContext } from '@/pages/_app';

export default function ComponentC() {
  const setCount = useContext(SetCountContext);
  return <button onClick={(e) => setCount(n => n + 1)}>Increase
Value</button>;
}
```

For both functions, we must import "useContext" from 'react', as well as the specific "Context" object that is required by the component, ie: "CountContext" or "SetCountContext". We invoke the "useContext" function with a specific "Context" object to retrieve the value from the "Provider" component (included in "App"), which can then be used within our component.

Problems with Context

Unfortunately, while this does avoid the need to pass props through unrelated components ("prop drilling"), it still suffers from some organizational and

performance issues. For example, what happens when the "application state" gets complicated, causing the providers to build up? This can result in what is known as "provider hell", ie:

```
<AContext.Provider value={"A"}>
  <BContext.Provider value={"B"}>
    <CContext.Provider value={"C"}>
      <DContext.Provider value={"D"}>
        <EContext.Provider value={"E"}>
          <Component {...pageProps} />
        </EContext.Provider>
      </DContext.Provider>
    </CContext.Provider>
  </BContext.Provider>
</AContext.Provider>
```

Additionally, the same performance problem exists, ie: every time the state is accessed / changed in a child component using context, **every** parent component back to the "Provider" (ie: "Home", "Component1", "Component2", "Component3", "ComponentA", "ComponentB", "ComponentC") **also** gets rendered.

Alternatives

If neither of the above built-in strategies work for your specific application, don't worry; there exist **many** 3rd party alternatives. Some of the more popular state management libraries include:

- **Redux / Redux Toolkit**: "The official, opinionated, batteries-included toolset for efficient Redux development. Includes utilities to simplify common use cases like store setup, creating reducers, immutable update logic, and more."

- **Recoil**: "A state management library for React. Recoil works and thinks like React. Add some to your app and get fast and flexible shared state."
- **Jotai**: "Jotai takes a bottom-up approach to React state management with an atomic model inspired by Recoil. One can build state by combining atoms and renders are optimized based on atom dependency. This solves the extra re-render issue of React context and eliminates the need for the memoization technique."
- **Zustand**: "A small, fast and scalable barebones state-management solution. Has a comfy API based on hooks, isn't boilerplatey or opinionated, but still just enough to be explicit and flux-like."
- **MobX**: "MobX is a battle tested library that makes state management simple and scalable by transparently applying functional reactive programming (FTRP)."
- **HookState**: "The most straightforward, extensible and incredibly fast state management that is based on React state hook"

Introducing Jotai

As we have seen, there are a number of popular 3rd party state management solutions for React. However, for our purposes we will be going with the relatively new "Jotai", for its simplicity, efficiency and support for Next.js. It also makes use of Typescript, which is excellent, but not currently necessary for our implementations going forward.

From the [Jotai Documentation](#):

Jotai was born to solve extra re-render issues in React. An extra re-render is when the render process produces the same UI result, where users won't see any differences.

To tackle this issue with React context (`useContext + useState`), one would require many contexts and face some issues.

- Provider hell: It's likely that your root component has many context providers, which is technically okay, and sometimes desirable to provide context in different subtree.
- Dynamic addition/deletion: Adding a new context at runtime is not very nice, because you need to add a new provider and its children will be re-mounted. Traditionally, a top-down solution to this is to use a selector interface. The [use-context-selector](#) library is one example. The issue with this approach is the selector function needs to return referentially equal values to prevent re-renders, and this often requires a memoization technique.

Jotai takes a bottom-up approach with the atomic model, inspired by [Recoil](#). One can build state combining atoms, and optimize renders based on atom dependency. This avoids the need for memoization.

Jotai has two principles.

- Primitive: Its basic interface is simple, like useState.
- Flexible: Derived atoms can combine other atoms and enable useReducer style with side effects. Jotai's core API is minimalistic and makes it easy to build utilities based upon it.

As you can see from the main concepts outlined above, Jotai was inspired by [Recoil](#) (an experimental library created by Dave McCabe, a Software Engineer at Facebook) and was designed to solve some of the problems such as "provider hell" and unnecessary re-renders that we discussed when reviewing "Prop Drilling" and "Context" in the previous section. This makes it a perfect alternative for us to use.

Getting Started

To begin working with Jotai, all we need to do is install it using npm, ie:

```
npm i jotai
```

Defining Application Level State

In Jotai, state values are defined as "atoms", essentially units of state that are both updatable and subscribable. When an atom is updated, any subscribed component will be re-rendered with the new value. This makes working with atoms very familiar, as the syntax and behaviour very closely resembles working with local state in components using the ["useState"](#) hook.

To define atoms in Next.js, we will place them in a separate file, ie: "store.js". Since each atom represents a different unit of state, we can define as many as

we wish in this file, ie:

File: "my-app/store.js"

```
import { atom } from 'jotai';

export const countAtom = atom(0);
export const countryAtom = atom('Japan');
export const citiesAtom = atom(['Tokyo', 'Kyoto', 'Osaka']);
export const mangaAtom = atom({ 'Dragon Ball': 1984, 'One Piece': 1997, 'Naruto': 1999 });
```

Here, we have defined 4 atoms with varying default values from numbers, strings, arrays and objects. Each of these atoms can be directly referenced from any component in the tree and may be used just like "useState" (see below).

Async Default Values

There may be situations where you cannot hard-code default values into your atoms and instead must fetch them from an API, file, etc. To accommodate this, Jotai allows atoms to be defined using an **"async function"**, ie:

```
export const postAtom = atom((async () => {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts/1');
  const data = await res.json();

  return data;
}))();
```

Reading / Writing State

To use any of the atoms defined in our "store.js" file, we must import both the "`useAtom`" function (from 'jotai') as well as the specific atom that we wish to read from / write to. For example, if we wish to reference the "countryAtom" (defined above with a default value of "Japan"), we can use the following code:

```
import { useAtom } from 'jotai';
import { countryAtom } from '@/store';

export default function Country() {
  const [country, setCountry] = useAtom(countryAtom);

  return <>Country: {country}</>
}
```

Notice how "`useAtom`" functions in a very similar way to "`useState`". We can access the state directly from the atom and when it's updated (using "`setCountry`", in this case) any other components rendering the value from the `countryAtom` will also get updated.

"Component Tree" Example

Now that we understand how state management works in Jotai, let's update our "Component Tree" example from the previous section to use it.

NOTE: we do *not* need to modify the file "/pages/_app.js" as in previous examples. Instead we will create a new file: "store.js".

File: "my-app/store.js"

```
import { atom } from 'jotai';

export const countAtom = atom(0);
```

File: "/components/Component3.js"

```
import { useAtom } from 'jotai';
import { countAtom } from '@/store';

export default function Component3() {
  const [count, setCount] = useAtom(countAtom);
  return <>Value: {count}</>
}
```

File: "/components/ComponentC.js"

```
import { useAtom } from 'jotai';
import { countAtom } from '@/store';

export default function ComponentC() {
  const [count, setCount] = useAtom(countAtom);
  return <button onClick={(e) => setCount(count + 1)}>Increase Value</button>
}
```

As you can see, we only had to create / modify 3 files:

- **store.js**: Defines our atoms (global state with optional default values)
- **component3.js**: The component using the atom to display its value
- **componentC.js**: The component using the atom to modify its value

This is much cleaner than our previous approaches, with the added bonus of

having the syntax feel very familiar. Additionally, our application / site no longer suffers from the performance hit caused by re-rendering all of the components in the tree; only "Component3" and "ComponentC" are re-rendered when the state changes.

For more information including handling special cases, etc. please reference the ["official Jotai documentation"](#).

Implementation: Shopping Cart

As we have seen, Jotai greatly improves and simplifies working with "application level" state, as compared to built in methods such as "Prop Drilling" and "Context".

When the need for "application level" state was first discussed, one possible use case was an e-commerce site that implements a shopping cart. As an extended example for Jotai, let's use it to implement a simplified shopping cart for a site that pulls a list of products from [DummyJSON](#).

Getting Started

To begin, we will use the "shopping-state-missing" example from the sample code: [Managing-Application-State](#) as a starting point.

Once you have the source code downloaded:

1. Open the "shopping-state-missing" folder in your code editor (ie: "Visual Studio Code")
2. Open the "my-app" folder in the integrated terminal
3. Run the command "npm install" (alternatively: "npm i") to install the dependencies
4. Build / Run the site with the usual command: "npm run dev"
5. Browse the site

File Structure

The project currently contains the following "components" / "pages" structure:

- **components/Layout.js:** The main / shared layout for the site. This contains the navbar as well as the headline "Online Shopping".
- **components/ProductBox.js:** This is the component responsible for rendering a specific product on the "/products" page. It takes a product as a property and renders the details (image, description, price, etc.) in a `<div>...</div>` element with a maximum width of "300px". Additionally, it contains buttons that either link to the specific product details page ("/`products/[id]`"), or invoke an "addToCart()" function with the current product object. Currently, "addToCart()" simply outputs the product to the console with the message: "TODO: Add to Cart".
- **pages/Products/[id].js:** This page renders additional details for a specific product (brand, rating, stock, etc.), based on the "id" parameter. Like "ProductBox", it contains an "addToCart" function that has not yet been implemented as well as a button that links back to the product list ("/`products`"). Additionally, it makes use of `"getStaticPaths()"` and `"getStaticProps()"` in order to pre-render the 30 potential products available.
- **pages/Products/index.js:** This is the page that renders a single "ProductBox" for all 30 available products in a grid using **Flexbox**. Like the "Products/[id]" page, it makes use of `"getStaticProps()"` in order to pre-render the 30 products to be displayed.
- **pages/_app.js:** Contains the boilerplate code for a Next.js app, with the addition of the `<Layout>...</Layout>` component.

- **pages/cart.js:** Currently only shows the text "Cart" - this is where we will eventually render the products currently contained within the "cart"
- **pages/index.js:** Simply renders the "Home" component on the default route "/" - currently contains a short description of the demo.

Adding "Cart" state with Jotai

Before we begin, we must install Jotai using the command:

```
npm i jotai
```

Next, if would like to make our "cart" (ie: a list of "products" that the user wishes to purchase) available anywhere within the site, we should create an "atom" to store the values. Additionally, let's also include Product 1 (iPhone 9) and Product 2 (iPhone X) as default values for the cart:

File: "/my-app/store.js"

```
import { atom } from 'jotai';

async function defaultValues() {
  const results = [];

  // Fetch Product 1

  const prod1Result = await fetch('https://dummyjson.com/products/1');
  const prod1 = await prod1Result.json();

  results.push(prod1);

  return results;
}
```

As before, we have created a "store.js" file, imported the "atom" function from 'jotai' and defined and exported an atom ("cartListAtom"). The default value for the atom is an array of products, obtained by invoking the asynchronous "defaultValues()" function.

Updating "Layout"

The first component that we would like to update to use the newly created "cartListAtom" is "Layout". Here, we will show how many products have been added to the cart in parentheses next to the "Shopping Cart" link:

File: "/components/Layout.js"

```
import Link from 'next/link';
import { useAtom } from 'jotai';
import { cartListAtom } from '@/store';

export default function Layout(props) {

  const [cartList, setCartList] = useAtom(cartListAtom);

  return (
    <>
      <div style={{ padding: "10px" }}>
        <h2>Online Shopping</h2>
        <Link href="/">Home</Link> | <Link
        href="/products">Products</Link> | <Link href="/cart">Shopping
        Cart <span>({cartList.length})</span></Link>
        <hr />
        {props.children}
      </div>
    </>
  )
}
```

Notice how we updated the component to use both "useAtom" from 'jotai' and "cartListAtom" from '@/store' (our store.js file containing the atom definition). In the component, we use "useAtom" in the same way that we use "useState" only the "default value" is the atom "cartListAtom". This gives us full read/write access to the atom, shared by the rest of the site.

If we refresh the site after making this change, we should see that the "Shopping Cart" link has been updated to read "Shopping Cart (2)".

Updating "addToCart()" Functions

The next piece that we should update is the "addToCart()" functions that exist in both "/pages/products/[id].js" and "/components/ProductBox.js" files.

In both cases, we must add the cartListAtom from the correct "store" location, as well as the useAtom function:

```
import { useAtom } from 'jotai';
import { cartListAtom } from '@/store';
```

Once we have the atom, we must reference it in the component using the syntax:

```
const [cartList, setCartList] = useAtom(cartListAtom);
```

Finally, we can update the "addToCart()" function to add the product to the cart:

```
function addToCart(product) {
  setCartList([...cartList, product]);
```

Since we are modifying the current list of items, we must use "spread syntax" to include all of the previous products in the list, in addition to the product to specify the state.

If we refresh the site now, we should be able to click any "Add to Cart" button, and see the "Shopping Cart" number increase in the navigation bar.

Updating the "cart" Page

The final piece of functionality that we must add is to show all of the products currently within the cart on the "cart" page, as well as a total cost for all of the products within the cart.

Once again, to gain access to the products in the cart, we must add the `cartListAtom` from the "store" as well as the `useAtom` function:

```
import { useAtom } from 'jotai';
import { cartListAtom } from '@/store';
```

Followed by the code to reference it in the component:

```
const [cartList, setCartList] = useAtom(cartListAtom);
```

With this in place, the following JSX code can be added to render the products in a list:

```
return (
  <>
  <br />
  <ul>
    {cartList.map((product, index) =>(
```

With this complete, we can now browse the site and add products to the cart. To view all of the items in the cart the "cart" page should now show an updated list with a total cost.

Example Code

You may download the sample code for this topic here:

| [Managing-Application-State](#)

Web API With Authentication

Before we can begin learning about JWT and how to secure a Web API, we must first create a simple Node.js server to handle our API requests. To speed this along, we have included a simple Web API in the [Example Code](#) for this week (See the "simple-API" folder). Currently, the primary function of this Web API is to return a hard-coded, static list of vehicles from its `data-service.js` module, using the route `/api/vehicles`.

Once you have extracted the "simple-API" folder from the example code, open it in Visual Studio Code and execute the following command to fetch the dependencies (currently, only `express` & `cors`):

```
npm install
```

Once this is complete, execute the command:

```
node server.js
```

This will start the server and enable you to test the `/api/vehicles` route on `localhost:8080`. You should see an array of JSON objects, consisting of 5 vehicles.

Quick note on "CORS"

At this point, you may be asking "What is 'cors' and why do we need this module?". CORS stands for "Cross-Origin Resource Sharing" and it is essentially a way to enable JavaScript to make an AJAX call from one origin (domain) to a server on a **different** domain. This is not permitted by default, as browsers restrict these types of requests for security reasons. If we did not enable CORS, we could not use AJAX to make requests from our localhost to our API, if our API is running online.

In addition to simply allowing all AJAX requests from outside domains, the CORS module also allows you to "whitelist" certain domains, thereby allowing access for specific domains, while restricting access from all others.

More details can be found on MDN under "[Cross-Origin Resource Sharing \(CORS\)](#)" and the "["cors" module on NPM](#)

Account Management & Security

With our extremely simple "vehicles" API in place and producing data, we can now move on to discuss how we might *protect* this data from unwanted (unauthorized) access. This will include creating routes that allow users to register accounts (persisted in MongoDB with encrypted passwords) as well as logging in to the system.

MongoDB Atlas & MongoDB

As mentioned above, we will be persisting registered accounts using MongoDB in the cloud database platform: [MongoDB Atlas](#). If you're not familiar with MongoDB Atlas at this point, you may review the basic setup here:

[MongoDB Atlas Tutorial](#)

Once your cluster is up and running correctly (ie: A "Database User" has been created, IP Access is "Allowed from Anywhere", etc):

1. In MongoDB, setup a new "simple-API-users" database with a "users" collection for the simple API.
2. Obtain a copy of the connection string - this should look something like:

```
mongodb+srv://user:<password>@cluster0-abc1.mongodb.net/?retryWrites=true&w=majority
```

3. Add the text: **simple-API-users** in the above connection string after the last part of the url before the query parameters, ie: **mongodb.net/simple-API-users**. In addition, you must update the values for **user** and **<password>** to match the credentials that you created for the "Database User". For example:

```
mongodb+srv://user:<password>@cluster0-abc1.mongodb.net/simple-API-
users?retryWrites=true&w=majority
```

4. Keep track of your connection string, as we will be using it in the next piece:

Updating the "user-service"

To keep our DB authentication piece clean, we will be making use of the promise-based "user-

service" module, defined in the `user-service.js` file.

In the `user-services.js` file, you will see a space for your MongoDB connection string. **Enter it now before proceeding.**

Observe the definition of the "user" Schema (`userSchema`). The schema consists of 4 simple fields:

- **userName:** A (unique) string representing the user's login/user name
- **password:** The user's password
- **fullName:** Ths user's full name
- **role:** The user's role, ie "administrator", "data-entry", "maintenance", etc. (the user's role will define exactly what in the API the user has access to. For our example we will not be using this field, as every user will have access to all vehicles)

Below this, you should note that there are 3 exported functions:

- **connect():** This function simply ensures that we can connect to the DB and if successful, assign the "User" object as a "User" model, using the "users" collection (specified by `userSchema`).
- **registerUser(userData):** Ensures that the provided passwords match and that the user name is not already taken. If the userData provided meets this criteria, add the user to the system.
- **checkUser(userData):** This function ensures that the user specified by "userData" is in the system and has the correct password (used for logging in)

Lastly, before we can move on to test the application (below) we must update our "server.js" to "connect" to our user service before we start the server.

To do this, go to the `server.js` file, and wrap the existing `app.listen()` function within the `userService.connect()` function:

server.js

```
// wrap the existing app.listen in a userService.connect() call:  
userService.connect().then(()=>{  
    app.listen(HTTP_PORT, ()=>{console.log("API listening on: " + HTTP_PORT)});  
})
```

Hashed Passwords with bcrypt (bcryptjs)

Up to this point, our user service has been designed to store passwords as plain text. This is a serious security concern as passwords must **always** be encrypted. To achieve this, we will be making use of the password-hashing function: "bcrypt".

For our project, we will use the "bcryptjs" library, a npm package that implements the bcrypt algorithm in Javascript.

In your project, use **npm** to install **bcryptjs**:

Terminal

```
npm install bcrypt
```

At the top of `user-service.js`, "require" the bcrypt module:

user-service.js

```
const bcrypt = require('bcryptjs');
```

After importing bcryptjs into the user-service.js file, we can use library's `hash()` function to create a hashed version of a plan text password. Here's how the function works:

```
// Encrypt the plain text: "myPassword123"
bcrypt.hash('myPassword123', 10).then((hash) => {
  // Hash the password using a Salt that was generated using 10 rounds
  // TODO: Store the resulting "hash" value in the DB
});
```

Apply this process to our "registerUser" function (thereby *hashing* the provided password when registering the user), our code will look like this:

user-service.js

```
module.exports.registerUser = function (userData) {
  return new Promise(function (resolve, reject) {
```

This makes the code a little longer and harder to follow, but we are really only adding the **bcrypt.hash()** method to our existing function.

If we wish to **compare** a plain text password to a **hashed** password, we can use bcrypt's **compare()** method with the following logic:

```
// Pull the password "hash" value from the DB and compare it to "myPassword123"
(match)
bcrypt.compare('myPassword123', hash).then((res) => {
  // if res === true, the passwords match
});
```

If we apply this to our "checkUser" function (thereby comparing the DB's *hashed* password with the provided password), our code will look like this:

user-service.js

```
module.exports.checkUser = function (userData) {
  return new Promise(function (resolve, reject) {

    User.find({ userName: userData.userName })
      .limit(1)
      .exec()
      .then((users) => {

        if (users.length == 0) {
          reject("Unable to find user " + userData.userName);
        } else {
          bcrypt.compare(userData.password, users[0].password).then((res) => {
            if (res === true) {
              resolve(users[0]);
            } else {
              reject("Incorrect password for user " + userData.userName);
            }
          });
        }
      }).catch((err) => {
        reject("Unable to find user " + userData.userName);
      });
  });
};
```

Not much has changed here. Instead of simply comparing `userData.password` with

users[0].password directly, we use the **bcrypt.compare()** method.

Adding & Testing Authentication Routes

Now that we have a working "user" service that will handle registering and validating user information, we will add new API routes (/api/) for authentication to our server application.

Since our new routes will be accepting input (via JSON, posted to the route), we will need to configure our server to correctly parse "JSON" formatted data. This can be accomplished by adding `express.json()` built-in middleware before our route definitions:

server.js

```
app.use(express.json());
```

With the middleware correctly configured, we can reliably assume that the "body" property of the request (req) will contain the properties and values of the data sent from the AJAX request.

ⓘ INFO

We do not yet have a UI to gather user information for registration and validation, so we must make use of an API testing tool such as the [Thunder Client Extension](#) to make requests and provide POST data when testing our new routes.

Creating a register user route

Start by creating a route for registering new users: **/api/register**

This route simply collects user registration information sent using POST to the API in the form of a JSON-formatted string, ie:

```
{
  "userName": "bob",
  "password": "myPassword",
  "password2": "myPassword",
  "fullName": "Robert Wiley",
  "role": "administrator"
}
```

Fortunately, our **userService.registerUser()** function is perfectly set up to handle this type of

data. It will validate whether password & password2 match and check that the user name "bob" is not taken. If the data meets these requirements, the provided password will be hashed and the user will be entered into the system. Therefore, our new /api/register route is very simple; it must simply pass the posted data to the userService for processing and report back when it has completed, ie:

server.js

```
app.post('/api/register', (req, res) => {
  userService
    .registerUser(req.body)
    .then((msg) => {
      res.json({ message: msg });
    })
    .catch((msg) => {
      res.status(422).json({ message: msg });
    });
});
```

ⓘ INFO

The 422 error code communicates back to the client that the server understands the content type of the request and the syntax is correct but was unable to process the data (see: [422 Unprocessable Entity](#)).

To test this new route:

1. Stop and start your API (server.js) again and proceed to make the following request in your API testing tool (example: Thunderclient):
 - Make sure **POST** is selected in the request type dropdown
 - In the address bar, type: "**http://localhost:8080/api/register**"
 - In the **Headers** tab, ensure that "Content-Type" is selected with a value of "application/json"
 - In the **Body** tab, copy and paste our information for user "bob" in the provided text area:

```
{
  "userName": "bob",
  "password": "myPassword",
  "password2": "myPassword",
  "fullName": "Robert Wiley",
  "role": "administrator"
```

- Once the request is processed, it should return with a status 200 and the JSON:

```
{  
  "message": "User bob successfully registered"  
}
```

- In the database, a user document will be inserted into collection.

Creating a login route

In addition to **adding** users to the system, we must also be able to **authenticate** users and allow them to "login" before being granted access to the data: **/api/login**

In this case, all of the work required for authenticating user data is done in the "dataAuth.checkUser()" method. So like "/api/register", our "/api/login" route will once again pass the posted data to the userService for processing and report back when it has completed, ie:

server.js

```
app.post('/api/login', (req, res) => {  
  userService  
    .checkUser(req.body)  
    .then((user) => {  
      res.json({ message: 'login successful' });  
    })  
    .catch((msg) => {  
      res.status(422).json({ message: msg });  
    });  
});
```

To test this new route:

- Stop and start your API (server.js) and make another request using your API testing tool. We will keep most of the values the same, with the following exceptions:
 - In the address bar, type: "**http://localhost:8080/api/login**"
 - In the **Body** tab, copy and paste our information for user "bob" in the provided text area:

```
{  
  "userName": "bob",  
  "password": "myPassword"  
}
```

- When you're sure you've entered everything correctly and your server is running, hit the blue **Send** button to send the POST data to the API.

3. Once the request is processed, it should return with a status 200 and the JSON:

```
{  
  "message": "login successful"  
}
```

4. Try entering incorrect credentials in the request body (ie: a different "userName", or an incorrect "password") to verify that our service is indeed functioning properly and will not send the "login successful" message to unauthorized users.

- In your API testing tool, update the **Body** with username bob, but an incorrect password:

```
{  
  "userName": "bob",  
  "password": "1234"  
}
```

5. Once the request is processed, it should return with a status 422 (Unprocessable Entity) and the JSON:

```
{  
  "message": "Incorrect password for user bob"  
}
```

JSON Web Tokens (JWT)

With our new authentication routes tested and working correctly, we can now concentrate on leveraging this logic to actually **secure** the vehicle data in our simple API.

Currently, the `/api/vehicles` route is available to anyone, regardless of whether they've been authenticated or not. You can see this by:

1. Executing a POST request to `"/api/login"` with an incorrect password for "bob"
2. Following the POST request, make a GET request to `"/api/vehicles"`.
3. Observe that the `"/api/vehicles"` endpoint displays the vehicle data.

The fact that we did not provide correct credentials during the "login" phase, had no effect on whether or not we can access the data on the `"/api/vehicles"` route.

So, how can we solve this problem? Essentially, what we need is some kind of secure, tamper-proof "logged in" identifier that we can **send** back to the client once they have been authenticated ("logged in"). This identifier could be stored on the client and sent with subsequent requests, serving as both identification and proof that they have been authenticated by the server.

JSON Web Token (JWT) to the rescue

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

This is perfect for our purposes. We can generate a JWT on the server (only) once the user has been **successfully authenticated** and send it back to the client along with the "login successful" message. It will contain digitally-signed information about the authenticated user such as their "userName", "fullName" & "role" (but **never** their password). The client can then read this information and **send the JWT back to the server** in an "Authorization" header with every subsequent request to be verified on the server. Since it is digitally signed on the server using a "secret", we can verify that the data has not been tampered with and that the JWT did indeed come from our simple API server and we can send the requested data.

When should you use JSON Web Tokens? Here are some scenarios where JSON Web Tokens are useful:

Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

For more information about JWT, including the signature & structure of the payload, see the excellent documentation at <https://jwt.io/introduction/>

Securing routes with JWT

We have now identified that we would like to work with JWT to secure our routes. However, how do we go about implementing JWT generation and verification in our server.js? This will involve 3 key modules, obtained from NPM:

jsonwebtoken

The "jsonwebtoken" module (available using `npm install jsonwebtoken --save` & added to server.js using: `const jwt = require('jsonwebtoken');`). In our application, this module is used primarily to "**sign**" our JSON payload with a 'secret' and generate the token, ie:

```
let token = jwt.sign({ userName: 'bob' }, 'secret');
```

We can also use a 3rd parameter to specify options such as **expiresIn** (A numeric value is interpreted as a seconds count):

```
jwt.sign({
  userName: 'bob'
}, 'secret', { expiresIn: 60 * 60 });
```

For more information on the usage of this function including additional options, methods and errors/codes see [the documentaiton for jsonwebtoken on npm](#)

passport

The "passport" module (available using `npm install passport --save` &

added to server.js using `const passport = require("passport");`) is described as the following:

Passport is Express-compatible authentication middleware for Node.js. Passport's sole purpose is to authenticate requests, which it does through an extensible set of plugins known as strategies. Passport does not mount routes or assume any particular database schema, which maximizes flexibility and allows application-level decisions to be made by the developer. The API is simple: you provide Passport a request to authenticate, and Passport provides hooks for controlling what occurs when authentication succeeds or fails.

In our application, we will be using the following methods:

- "**initializeapp.use() method, ie:**

```
app.use(passport.initialize());
```

- "**authenticate**

```
app.get("/api/vehicles", passport.authenticate('jwt', {  
  session: false }), (req, res) => {  
  // ...  
})
```

- "**use**

```
passport.use(strategy);
```

NOTE: Passport currently supports over 480 different authentication strategies including [Facebook](#), [Google](#) and [many others](#)

passport-jwt

Since we will be using JWT, our strategy will be "[passport-jwt](#)" (available using `npm install passport-jwt --save` & added to server.js using `const passportJWT = require("passport-jwt");`). Using "passportJWT", we can specify the "strategy" using a set of **options** (defined in our server as an object called **jwtOptions**: `let jwtOptions = {};`), such as the "secretOrKey", as well as specifying how to read the jwt from the authentication header.

For example, once we have a reference to the "passport-jwt" module (ie: "passwordJWT", from above), we can use the following code to configure our strategy:

```
// JSON Web Token Setup
let ExtractJwt = passportJWT.ExtractJwt;
let JwtStrategy = passportJWT.Strategy;

// Configure its options
let jwtOptions = {
  jwtFromRequest: ExtractJwt.fromAuthHeaderWithScheme('jwt'),
  secretOrKey:
  '&0y7$noP#5rt99&GB%Pz7j2b1vkzaB0RKs%^N^0z0P89NT04mPuAM!&G8cbNZ0tH',
};

// IMPORTANT - this secret should be a long, unguessable string
// (ideally stored in a "protected storage" area on the web
// server).
```

There's a lot going on in the above code, but the key pieces involve first defining the **jwtOptions** (using the **jwtFromRequest** and **secretOrKey** properties) and then defining the "strategy" as a (**JwtStrategy**) middleware function using the **jwtOptions** and providing a callback function. The callback function simply checks that there is indeed a valid **jwt_payload** and if so, invoke the **next()** method with the payload data as it's second parameter. If the **jwt_payload** is invalid, the **next()** method will be called without the payload data, which will cause our server to return a **401 (Unauthorized)** error.

Adding the code to server.js

With all of the individual pieces of our JWT solution identified, it's now time to update **server.js**:

Step 1: Installation

As you will recall (from above), our JWT enabled **server.js** will require 3 modules: "**jwt**", "**passport**" & "**passport-jwt**" to function correctly. Using **npm**, install them into the project:

Terminal

```
npm install jsonwebtoken --save
npm install passport --save
npm install passport-jwt --save
```

Step 2: Requiring the Modules

After installation, add the modules to our list of imports at the top of **server.js**:

server.js

```
const jwt = require('jsonwebtoken');
const passport = require('passport');
const passportJWT = require('passport-jwt');
```

Step 3: Configuring the "Strategy"

With our modules added, we can now add the code to configure the JWT "strategy". Recall, this involves creating a **jwtOptions** object that we can pass to the **jwtStrategy** constructor, along with a callback function that looks at the "jwt_payload" parameter. For our purposes, we can use the code exactly as it has been identified above, placed before our first "app.use()" statement. However, a **new** "secretOrKey" property should be generated (optionally using the ["Generate Password" Tool](#) from LastPass).

NOTE: If the "user" has different properties (ie, something *other* than, "_id", "userName", "fullName" and "role"), the data passed in the **next()** function should be modified to reflect the correct properties.

How to add the strategy:

In server.js, add this code **before** the first usage of `app.use()`:

server.js, before the first app.use() statement

```
// JSON Web Token Setup
let ExtractJwt = passportJWT.ExtractJwt;
let JwtStrategy = passportJWT.Strategy;

// Configure its options
let jwtOptions = {
```

Step 4: Set the Strategy & Add the Middleware

The last step needed to tell our server that we wish to use Passport (with the "JWT" strategy), by adding the function as middleware to our server using `app.use()`:

server.js

```
// tell passport to use our "strategy"  
passport.use(strategy);  
  
// add passport as application-level middleware  
app.use(passport.initialize());
```

Step 5: Generating & Sending the JWT

At this point, we're all set to work with JWT. We have the correct modules added and the Passsport middleware is configured and added to our application. However, before we can *protect* our routes (see below), we need to first **send** the token back to the client. Currently, our api/login route simply sends the following data with a 200 status code to indicate that the login was indeed successful:

```
{ "message": "login successful" }
```

If we wish to grant this user access to our (soon to be) protected routes, we must also provide the JWT as a means of identification. Using the **sign()** method of the included **jsonwebtoken** module, we can generate it and send it back to the client alongside the "message".

To accomplish this, we need to add the following code to our "/api/login" route at the top of our **userService.checkUser(req.body).then(...)** callback:

server.js

```
let payload = {  
  _id: user._id,  
  userName: user.userName,  
  fullName: user.fullName,  
  role: user.role,  
};  
  
let token = jwt.sign(payload, jwtOptions.secretOrKey);
```

This will generate a JWT for us using the user's "_id", "userName", "fullName" and "role" properties, encrypted with our "secretOrKey" (identified when we configured our passport strategy in jwtOptions).

Once we have the token, we can send it back along with the message to the user using **res.json()** (typically using the property: "token"):

server.js

```
res.json({ message: 'login successful', token: token });
```

Step 6: Protecting Route(s) using the Passport Middleware

In order to restrict access to our /api/vehicles route, we need to employ the Passport middleware "authenticate" function (identified above in our **authenticate()** example).

This simply involves adding the code:

```
passport.authenticate('jwt', { session: false })
```

as a middleware function to any routes that we wish to protect (ie: our /api/vehicles route):

server.js

```
app.get("/api/vehicles", passport.authenticate('jwt', { session: false }), (req, res) => {
    // ...
})
```

You will notice that we provide the option "session: false". This is because we require credentials to be supplied with each request, rather than set up a session. For more information and configuration options, see the Passport.js documentation, under ["Authenticate"](#).

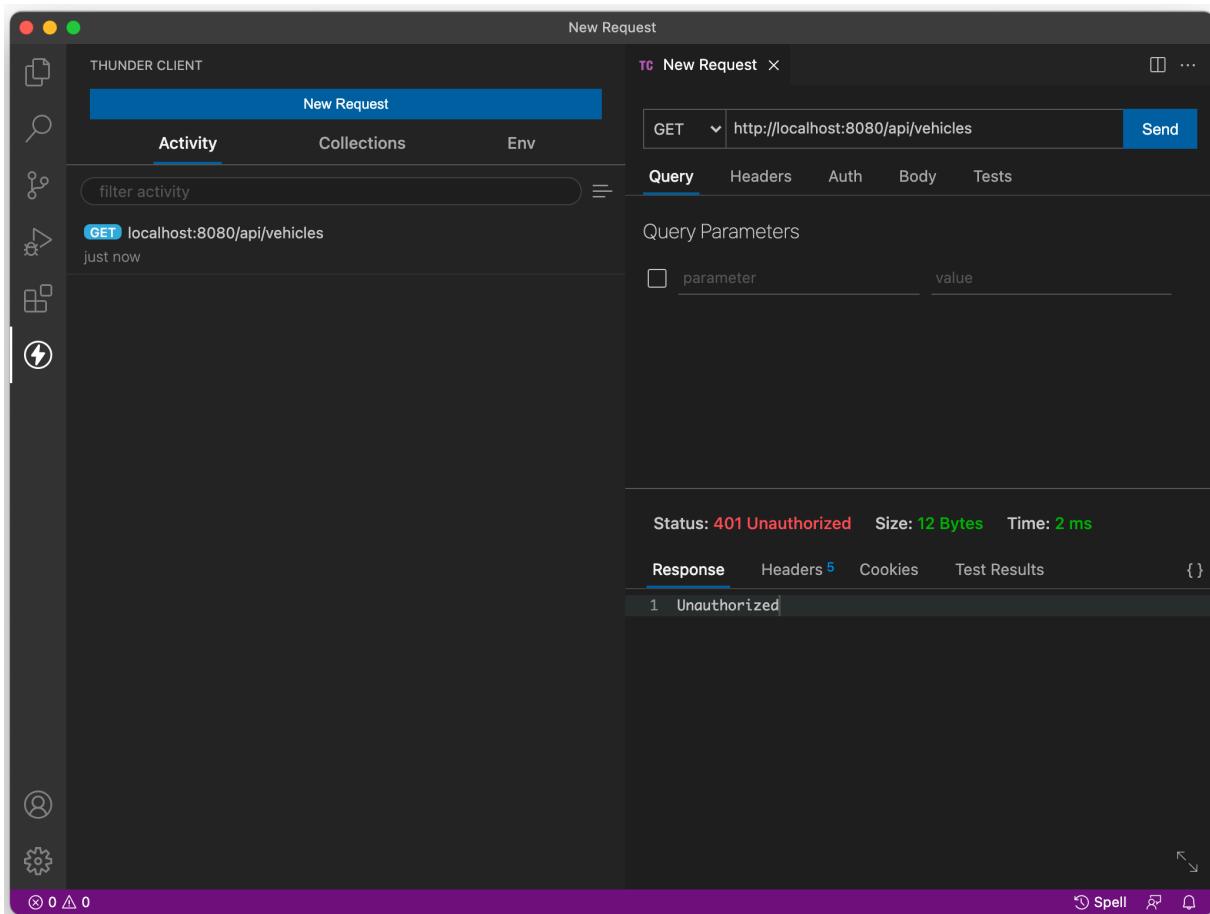
Testing the New Functionality

We have now completed all of the changes that are required on our server.js and are ready to test our Simple API and see if this technology really works to protect our routes.

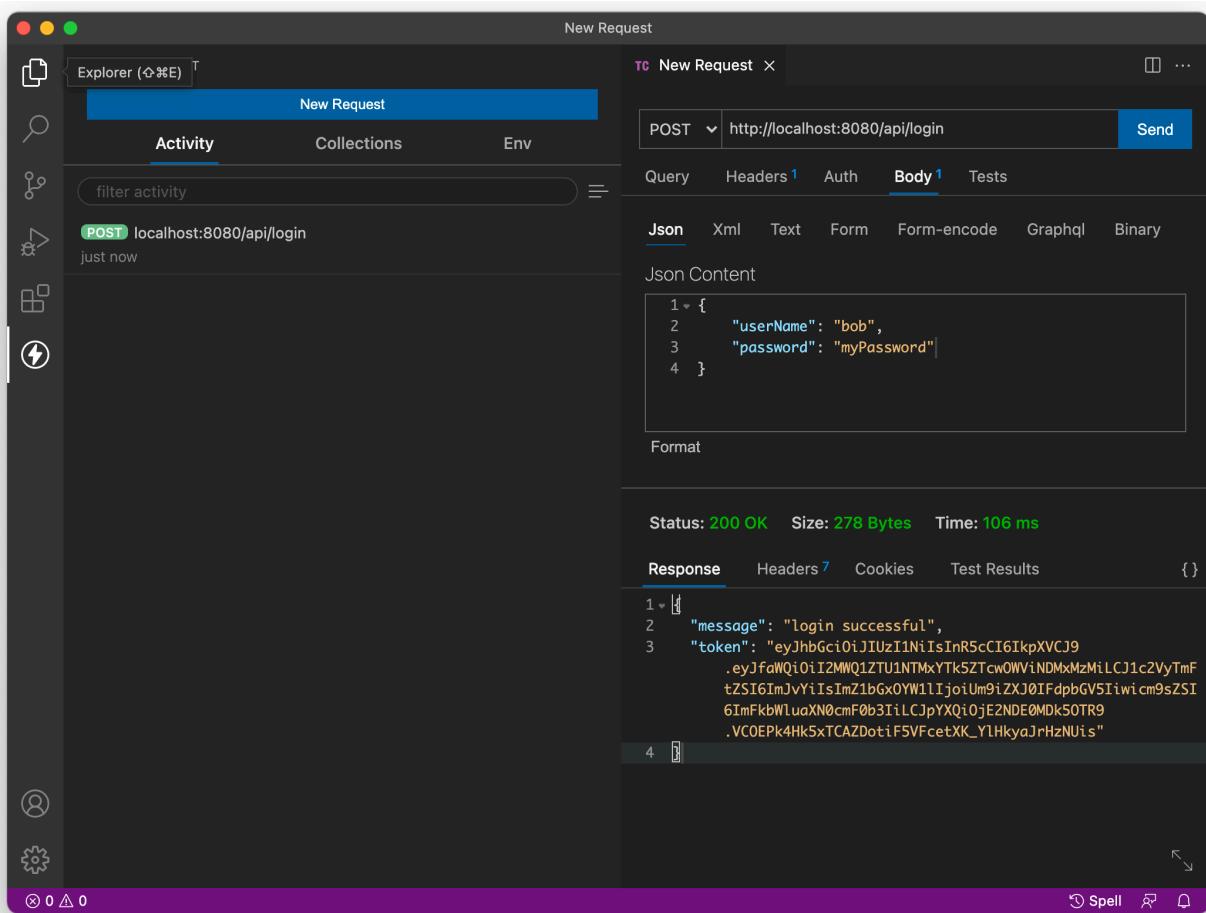
To test this, we must insure the following series of actions yields the expected results (listed below):

- **Action:** Attempt to access the route /api/vehicles as before (without supplying a JWT).

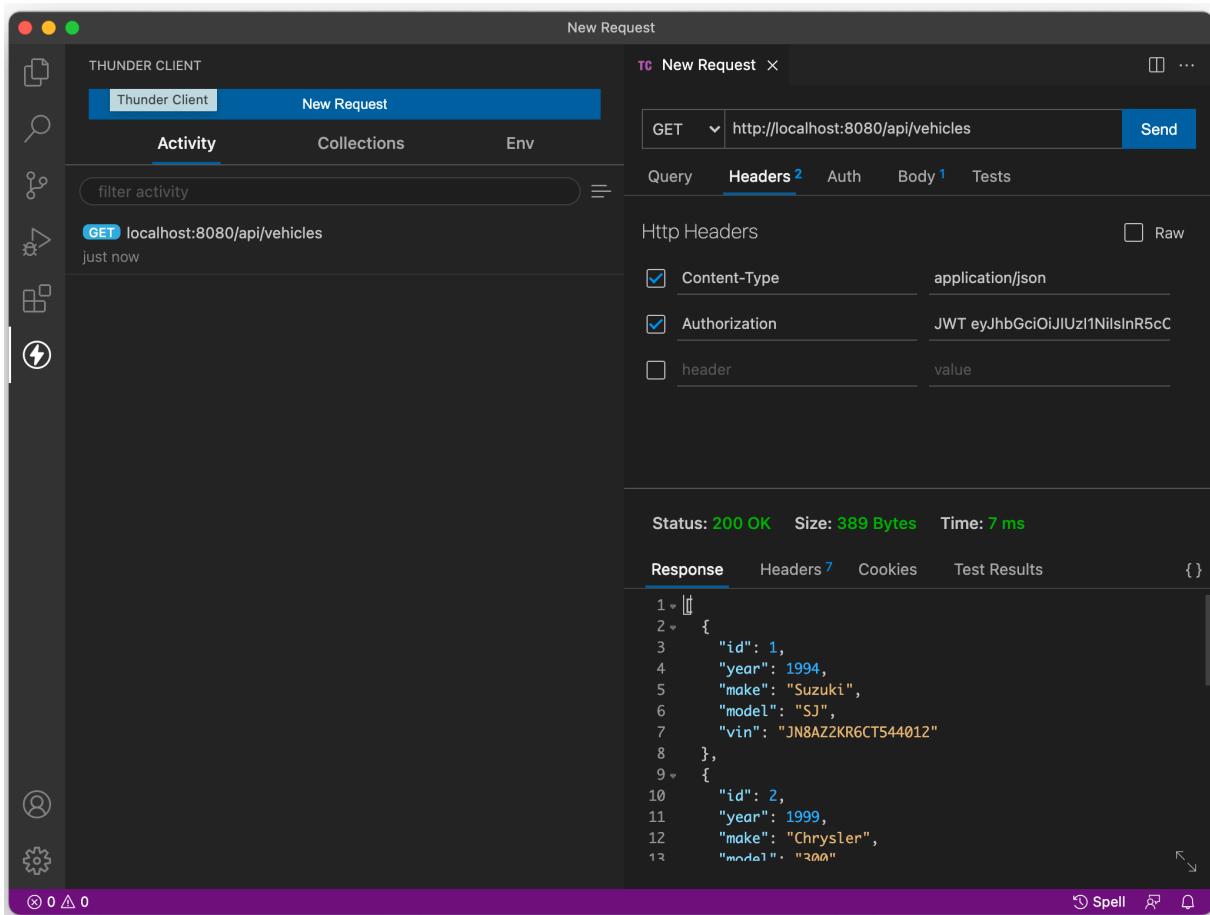
- **Expected Result:** Server returns a 401 error code and the text "unauthorized".



- **Action:** Log in as user "bob" (as above) and copy the value of the returned "token" property.



- **Action:** Attempt to access the route /api/vehicles as before, only this time add the header "Authorization" with the value "JWT" followed by a space, followed by the returned "token" that was sent when "bob" logged in (above)
- **Expected Result:** Vehicle data is returned



- **Action:** Attempt to access the route /api/vehicles again, only this time slightly modify the JWT (ie: remove/add a character).
- **Expected Result:** Server returns a 401 error code and the text "unauthorized".

THUNDER CLIENT

New Request

Activity Collections Env

filter activity

GET localhost:8080/api/vehicles just now

Headers 2 Auth Body 1 Tests

Http Headers

Content-Type application/json

Authorization JWT fyJhbGciOiJIUzI1NiIsInR5cC

header value

Status: 401 Unauthorized Size: 12 Bytes Time: 3 ms

Response Headers 5 Cookies Test Results

1 Unauthorized

Spell ⌂ ⌂

The screenshot shows the Thunder Client interface. On the left is a sidebar with icons for file operations, search, collections, environment, activity (which is selected), filters, and other tools. The main area is titled 'New Request' and shows a request for 'localhost:8080/api/vehicles'. The 'Headers' tab is active, displaying two headers: 'Content-Type: application/json' and 'Authorization: JWT fyJhbGciOiJIUzI1NiIsInR5cC'. Below the headers, there's a section for adding more headers with fields for 'header' and 'value'. The response section shows a status of '401 Unauthorized', a size of '12 Bytes', and a time of '3 ms'. The 'Response' tab is selected, showing the single line '1 Unauthorized'. At the bottom, there are buttons for 'Spell', a refresh icon, and a notifications icon.

Example Code

You may download the sample code for this topic here:

| [Introduction-JWT](#)

Authentication (Logging In)

Authentication can be a very complex topic, especially when working with Next.js. This is largely due to the amount of freedom and options that Next.js makes available to developers.

For example, some pages may be pre-rendered using [Server Side Rendering \(SSR\)](#) while others may be rendered on the client side (CSR). Even [Incremental Static Regeneration](#) is possible, allowing static pages to be created / updated *after* the site is built. Not to mention the ability for developers to write their own [API's within Next.js](#), as well utilize the (now stable) [Middleware](#) functionality to execute code before a request is completed.

For our examples however, we will try to keep things as straightforward as possible, primarily using code that we have seen already.

As an exercise, we will and attempt to write code for a site that connects to the secure API that we created during the discussion on "[Introduction to JWT](#)".

Obtaining & Running the “vehicles-UI” Example

As a starting point download the [Example Code](#), extract the files and open the "vehicles-UI" folder in Visual Studio Code. You will notice that this folder contains a my-app folder with the code for a Next.js app with two pages: "Home" (index.js) and "Vehicles" (vehicles.js).

Before we can run this app however, we must first:

- Ensure that the completed example from [Introduction to JWT](#) (ie: "simple-API-complete") is currently running on port 8080.
- Open the "vehicles-UI/my-app" folder in the integrated terminal for Visual Studio code and execute the command "npm install" to obtain the dependencies

With the dependencies installed and node_modules rebuilt, we can now start up our app with "npm run dev". You will see that we only have two routes available to the user: "Home" and "Vehicles". If we try to access the "Vehicles" route, we will not see any data due to a 401 - Unauthorized error returned from our "simple-API-complete" (this can be confirmed in the browser console).

NOTE: This sample app makes use of the UI components from [React Bootstrap](#). This was accomplished by installing "[react-bootstrap](#)" and "[bootstrap](#)" from NPM and adding the following "import" statement in "_app.js":

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

Building an "Authentication" Library

Since we will be handling authentication and working with JWT, it makes the most sense to have all of our "Authentication" related code in one place. For this example, we will be creating a new folder under "/my-app" called "**lib**". Within the "lib" folder, create a file called "**authenticate.js**". Within this file, we will place code that is responsible for:

- Executing a "POST" request using "[fetch\(\)](#)" to the "api/login" route of our

server with a given user / password.

- Storing / Removing the obtained JWT locally
- Reading the contents of the JWT
- Determining whether or not the user is “authenticated” after logging in

Function: authenticateUser()

The first function that we will create in "authenticate.js" is called "authenticateUser" and will attempt to obtain a JWT from our "simple-API-complete" server at the route "/api/login", given a specific user and password. This function must be "async" since it's making use of the asynchronous "fetch()" function. Additionally, it must only store the token locally if the status code from "/api/login" is **200**, otherwise the function must throw a new Error, with the error message sent from the API:

```
export async function authenticateUser(user, password) {  
  const res = await  
  fetch(`${process.env.NEXT_PUBLIC_API_URL}/login`, {  
    method: 'POST',  
    body: JSON.stringify({ userName: user, password: password }),  
    headers: {  
      'content-type': 'application/json',  
    },  
  });  
  
  const data = await res.json();  
  
  if (res.status === 200) {  
    setToken(data.token);  
    return true;  
  }  
}
```

Notice the url used in the "fetch" call references "process.env.NEXT_PUBLIC_API_URL". The value for this constant is provided in the "/my-app/.env" file:

```
NEXT_PUBLIC_API_URL="http://localhost:8080/api"
```

NOTE: Environment variables in Next.js use the following naming convention:

- **.env:** Defines environment variables, to be loaded in code using "process.env" (this is true for all .env files)
- **.env.local:** Defines "secrets" (tokens, etc) and is meant to be excluded from your code repository (ie: added to .gitignore)
- **.env.development:** Defines environment variables to be used in the *development* environment
- **.env.production:** Defines environment variables to be used in the *production* environment
- **.env.test:** Defines environment variables to be used in the *test* environment

Additionally, in order to make your environment variables available in the browser, they must be prefixed with the text **NEXT_PUBLIC_** as in our example: "NEXT_PUBLIC_API_URL".

Function: setToken()

This function is designed explicitly to store the token. It is used by the above "authenticateUser" as well as elsewhere through our application:

```
function setToken(token) {  
    localStorage.setItem('access_token', token);  
}
```

You will notice that in this case, we have chosen to persist the token using "localStorage" with the key "access_token"

Local Storage

From [MDN](#):

The localStorage read-only property of the window interface allows you to access a Storage object for the Document's origin; the stored data is saved across browser sessions.

Essentially, we are storing the value of the token in the browser for retrieval at a later date / time. This is in contrast to keeping it stored in memory, as we do not wish for the user to be logged out if the page is refreshed and the app is reloaded.

To view all values currently stored in "localStorage" (as well as "sessionStorage" "cookies", etc.) using Chrome, you can access the dev tools and open the "Application" tab. You will find the values under "Storage":

The screenshot shows the Chrome DevTools Application tab. On the left, there's a sidebar with sections for Application (Manifest, Service Workers, Storage), Storage (Local Storage, Session Storage, Cookies), and Network (Localhost:3000). Under Local Storage, 'http://localhost:3000' is selected. In the main pane, a table titled 'Filter' shows one item: 'access_token' with a value of 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2MmQwND...'. There are also other items listed in the table but are mostly cut off.

Function: getToken()

Similar to "setToken()" above, this function is designed explicitly to retrieve the token from "localStorage" using `getItem()`. If the token does not exist, this function returns null:

```
export function getToken() {
  try {
    return localStorage.getItem('access_token');
  } catch (err) {
    return null;
  }
}
```

In this case, we place the "getItem()" call within a try / catch block. This helps us deal with the possibility of encountering "ReferenceError: localStorage is not defined" when a page / pages using getToken() are pre-rendered by Next.js.

Function: removeToken()

This is the final function that works directly with "localStorage" - it simply removes the token from localStorage using `removeItem()`.

```
export function removeToken() {
  localStorage.removeItem('access_token');
}
```

Function: readToken()

The purpose of the `readToken()` function is to obtain the *payload* from the JWT (This is the data that was digitally signed on our server, ie: "`_id`" and "`userName`"). This is accomplished by first retrieving the token from `localStorage` (using `getToken()`), followed by reading the token using "`jwtDecode`" ([available from npm](#) and installed using "`npm i jwt-decode`"):

```
import { jwtDecode } from 'jwt-decode';

// ...

export function readToken() {
  try {
    const token = getToken();
    return token ? jwtDecode(token) : null;
  } catch (err) {
    return null;
  }
}
```

Function: isAuthenticated()

The final function defined within our "authenticate.js" lib, serves to determine whether or not the current user is "authenticated". In this case, we simply attempt to read the token (readToken()). If a value is returned, return *true* otherwise, return *false*. This will be used primarily to determine whether or not a user is allowed to proceed to a specific route / page.

```
export function isAuthenticated() {  
  const token = readToken();  
  return token ? true : false;  
}
```

Creating a "Login" Page

With our "authenticate.js" lib complete, we can now concentrate on implementing a User Interface which enables users to enter their credentials and attempt to "log in" (acquire the JWT).

To begin, we will create a **login.js** file within the "pages" directory.

Form Components

Since we're using "[React Bootstrap](#)", we can leverage some of their components to make a login form that matches the rest of the site:

```
import { Card, Form, Button } from "react-bootstrap";  
  
export default function Login(props){
```

Capturing User Input

During the initial discussion on [Forms in React](#), the concept of "Controlled Components" was first introduced, followed closely by ["React Hook Form"](#). While it was established that React Hook Form is an excellent alternative to "Controlled Components" (in terms of flexibility, ease of use, etc.), it is not required in this case. This is because we only have two simple input fields and all error messages related to logging in come from the API, so client-side validation is not strictly necessary.

Recall, to capture form data using "Controlled Components", we must include:

- Form field values within the "state":

```
import { useState } from 'react';

// ...

const [user, setUser] = useState('');
const [password, setPassword] = useState('');
```

- A Function to handle form submissions:

```
function handleSubmit(e) {
  e.preventDefault();
  console.log('TODO: Submit Form');
}
```

- Updated form fields to synchronize with the "state" values (implemented by adding a "value" property and an "onChange" event):

```
<Form.Control type="text" value={user} id="userName"
name="userName" onChange={e => setUser(e.target.value)} />
<Form.Control type="password" value={password} id="password"
name="password" onChange={e => setPassword(e.target.value)} />
```

- Update the "Form" component to use the "onSubmit" event:

```
<Form onSubmit={handleSubmit}>
</form>
```

If we apply all of these changes to our form, we should have the following code. When the form is submitted, the values for user and password are available in the state:

```
import { Card, Form, Button } from "react-bootstrap";
import { useState } from 'react';

export default function Login(props){

  const [user, setUser] = useState("");
  const [password, setPassword] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    console.log(`TODO: Submit Form with: ${user} / ${password}`)
  }
  return (
    <>
      <Card bg="light">
        <Card.Body><h2>Login</h2>Enter your login information
below:</Card.Body>
      </Card>
      <br />
    </>
  )
}
```

Authenticating the User / Showing Errors

The final piece for our "Login" component is to make use of "authenticate.js" to actually authenticate the user with the data entered in the form. If the user enters correct credentials, we will redirect them to the "/vehicles" route, however if invalid credentials have been entered, we must show an error message to the user. This will involve:

- Including the "authenticateUser" function from our "authenticate.js" lib as well as the "useRouter" hook from "next/router":

```
import { authenticateUser } from '@/lib/authenticate';
import { useRouter } from 'next/router';
```

- Adding a "warning" string in the "state" to display a login error to the user (if applicable) as well as invoking the "useRouter" hook to get an instance of "router"

```
const [warning, setWarning] = useState('');
const router = useRouter();
```

- Updating "handleSubmit" to use "authenticateUser" and update "warning" if it fails or redirect to "/vehicles" if it succeeds:

```
async function handleSubmit(e) {
  e.preventDefault();
  try {
```

- Adding an "Alert" Component and conditionally showing the warning message:

```
import { Card, Form, Alert, Button } from 'react-bootstrap';

// ...

{ warning && ( <><br /><Alert
variant="danger">{warning}</Alert></> )}
```

Adding an "Authorization" Header to SWR

If we test the app at this point, we should see that our "/login" page correctly shows errors, as well as redirects to "/vehicles" when appropriate (credentials have been accepted). You can also verify that the token has been correctly added as "access_token" in local storage.

The only piece missing is ensuring that the "access_token" (JWT) is correctly added to an "Authorization" header, when making a request for vehicles from our API.

At the moment, the code to make a request in "pages/vehicles.js" currently looks like:

```
const fetcher = (url) => fetch(url).then((res) => res.json());

//...

const { data, error } =
useSWR(`process.env.NEXT_PUBLIC_API_URL}/vehicles`, fetcher);
```

We do not have any code to specify a header, nor do we have any way of accessing the token from `local_storage` within this component.

This can be easily fixed however, by updating the "fetcher" function:

```
import { getToken } from "@/lib/authenticate";

// ...

const fetcher = (url) => fetch(url, { headers: { Authorization:
`JWT ${getToken()}` }}).then((res) => res.json());
```

If we test the route now, we should see that the correct header has been added to our request and we can indeed see the vehicles rendered on the page.

UI Considerations

Continuing with the example for this topic ("simple-UI"); we have seen that our "vehicles" page is finally capable of rendering the data after acquiring the JWT from our "simple-API-complete" server. However, there still remain a few usability concerns that must be addressed. For example, the user should not be permitted to access the "vehicles" page without being authenticated. What if there is sensitive static information visible on this page? Also, does it make sense to allow an unauthenticated user to visit a view, if they're guaranteed to not see any data? It would be better for usability and security, if we did not enable the user to navigate to that route, unless they are authenticated.

Additionally, it's common practice for the UI to show some indication that the user has logged in. This may include showing their user name / avatar somewhere prominent on the site (ie: the navbar) as well as allowing them to "log out".

Creating a "Route Guard" Component

To address the first issue (unauthorized access to the "vehicles" page), we will create a component that functions in a similar way to "Layout", in that it will be placed in App (_app.js) and "wrap" `<Component {...pageProps} />`. The purpose of this component is to only render "props.children" if the user has been authenticated and is allowed to view the requested route.

To begin, create a new component in the "components" folder called: "RouteGuard" and add it to _app.js

File: "/components/RouteGuard.js"

```
export default function RouteGuard(props) {
  return <>{props.children}</>
}
```

File: "/pages/_app.js"

```
import 'bootstrap/dist/css/bootstrap.min.css';
import Layout from '@/components/Layout';
import RouteGuard from '@/components/RouteGuard';

export default function App({ Component, pageProps }) {
  return <RouteGuard><Layout><Component {...pageProps}>
/></Layout></RouteGuard>
}
```

At the moment, this will not have any effect. However, with the component correctly positioned within our app, we can discuss how we can correctly add the desired functionality to the guard.

Redirecting to "/login" if unauthenticated

If we wish to send the user to the "/login" route if they try to access a route without being authenticated first, we need to ensure that the following logic is in place for our route guard:

- Maintains a list of "public" routes, ie: "/login", "/" and "/_error" (Next.js uses the path "/_error" internally when rendering the "404 | This page could not be found" error).
- Ability to check the url of the current (requested) route and compare it against the above list

- Check to see if the user is currently authenticated
- Redirect to "/login" if unauthenticated / render props.children if the user is authenticated

authCheck() Function

To implement these requirements, we should first construct a function called "authCheck" that checks the requested route and compares it against the "public" routes.

```
const PUBLIC_PATHS = ['/login', '/', '/_error'];

// ...

function authCheck(url) {
  const path = url.split('?')[0];
  if (!PUBLIC_PATHS.includes(path)) {
    console.log(`trying to request a secure path: ${path}`);
  }
}
```

Here, we define constant list of "PUBLIC_PATHS" as '/login', '/' and '/_error'. We also remove any query parameters from the url by **splitting** the string at "?" and taking the first half.

If the PUBLIC_PATHS array does not **include** the requested route, output a message to the console indicating that a secure path is being accessed.

invoking authCheck()

To correctly invoke authCheck() we must execute it:

- When the component is first mounted: using the useEffect() hook

- When a client-side route change has completed: using the 'routeChangeComplete' **router event**

NOTE: Router Events in Next.js can be "subscribed" to by using the "events.on" properties of the "router" object (obtained from the useRouter() hook), for example:

```
router.events.on('routeChangeComplete', (url) => {
  console.log(`route change to ${url} complete!`);
});
```

When no longer needed, the event can be "unsubscribed" to by using:

```
router.evnts.off()
```

To ensure that authCheck() is correctly invoked in both of the above scenarios, we can update our RouteGuard component to use the following code:

```
import { useRouter } from 'next/router';
import { useState, useEffect } from 'react';

const PUBLIC_PATHS = ['/login', '/', '/_error'];

export default function RouteGuard(props) {
  const router = useRouter();

  useEffect(() => {
    // on initial load - run auth check
    authCheck(router.pathname);

    // on route change complete - run auth check
    router.events.on('routeChangeComplete', authCheck);

    // unsubscribe from events in useEffect return function
  }, [router.events]);
}
```

With this code in place for RouteGuard, we should see "trying to request a secure path /vehicles" if we try to refresh the "/vehicles" page **or** navigate to it using the navigation bar.

Adding Authentication & Redirection

To complete the RouteGuard functionality, we must implement logic to check whether or not the user is logged in. We can use this to either allow access to the requested route (by rendering "props.children"), or redirect the user back to "/login" (using `router.push("/login");`).

First, we should add a value in the state to store whether or not the user has been authorized to view the protected routes. We can use this to conditionally render "props.children", as described above:

```
const [authorized, setAuthorized] = useState(false);

// ...

return <>{authorized && props.children}</>
```

Next, we must add the ability to check if a user has been authenticated and if so, set "authorized" to true, so that the route may be rendered. However, if the user has not been authenticated, we can set "authorized" to false and redirect the user back to "/login". This can be achieved by importing the "isAuthenticated()" function from our "authenticate" lib, as well as updating our "authCheck" function:

```
import { isAuthenticated } from '@/lib/authenticate';

// ...
```

Here, we make sure to only redirect to "/login" if the user has not been authenticated **and** they are trying to access a restricted route. If they are not authenticated and try to access an unrestricted route (ie: a route defined in "PUBLIC_PATHS"), then they should be allowed to proceed.

Updating the Navigation Component

With our Route Guard in place and successfully preventing unauthorized users from viewing the "/vehicles" page, we can concentrate on the last piece of our UI: Updating the "Navigation" Component. Here, we will show a welcome message, ie "Welcome **userName**" as well as only show the "Vehicles" link if the user has logged in. Additionally, we will replace the "login" link with a "logout" link.

The first thing we must do is add the "readToken" and "removeToken" functions from our "authenticate" lib as well as "useRouter" from "next/router":

```
import { readToken, removeToken } from '@/lib/authenticate';
import { useRouter } from 'next/router';
```

We will use this within the component to:

- Store the current value of the token:

```
let token = readToken();
```

- Implement a "logout" function that removes the token and redirects the user back to "/":

```
const router = useRouter();

function logout() {
    removeToken();
    router.push('/');
}
```

Updating the JSX

Finally, with our token in place and our logout function implemented, we can make the following changes to our JSX code for the component to conditionally render text / elements using the "token" value:

```
return (
    <Navbar bg="light" expand="lg">
        <Container>
            <Link href="/" passHref legacyBehavior><Navbar.Brand>
                Vehicles UI {token && <>- Welcome
                {token.userName}</>}</Navbar.Brand></Link>
            <Navbar.Toggle aria-controls="basic-navbar-nav" />
            <Navbar.Collapse id="basic-navbar-nav">
                <Nav className="me-auto">
                    <Link href="/" passHref legacyBehavior
                        ><Nav.Link>Home</Nav.Link></Link>
                    {token && <Link href="/vehicles" passHref
                        legacyBehavior><Nav.Link>Vehicles</Nav.Link></Link>}
                </Nav>
                <Nav className="ml-auto">
                    {!token && <Link href="/login" passHref
                        legacyBehavior><Nav.Link>Login</Nav.Link></Link>}
                    {token && <Nav.Link onClick={logout}>Logout</Nav.Link>}
                </Nav>
            </Navbar.Collapse>
    </Container>
```

Using the above code, we can ensure that once the user is logged in, they will see their user name as well as the "vehicles" and "logout" links in the navigation bar!

Example Code

You may download the sample code for this topic here:

| [Authentication-In-Next](#)

Unit Testing

Software testing is a vital part of any development effort. There are **many strategies**, from automated tests with Unit Testing, End to End (E2E) testing, performance tests, etc. to manual tests like Acceptance Testing. Software development practices such as **Continuous Integration** rely heavily on testing to help ensure that bugs are not introduced when merging / integrating code from multiple developers. A major piece of this effort takes the form of Unit Testing:

Unit testing finds problems early in the development cycle. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit (one or more computer program modules together with associated control data, usage procedures, and operating procedures). The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behavior. The cost of finding a bug before coding begins or when the code is first written is considerably lower than the cost of detecting, identifying, and correcting the bug later.

https://en.wikipedia.org/wiki/Unit_testing

Jest Introduction

When unit testing our Next.js code we will be using the popular "**Jest**" testing framework.

Jest is a delightful JavaScript Testing Framework with a focus on simplicity. It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!

Getting Started

Before we begin learning Jest, we should create a new Next.js app using the familiar command (ensuring that we use version 14):

```
npx create-next-app@15 my-app --use-npm
```

Once this is complete, we will install Jest as a **"development dependency"** using npm, ie:

```
npm install --save-dev jest
```

Next, we should update the "scripts" section of our **package.json** file to create a new "test" script that runs Jest using the **--watchAll** flag (to run all tests):

```
"scripts": {  
  ...  
  "test": "jest --watchAll"  
}
```

Finally, create a new folder under "my-app" called "tests" (ie: my-app/tests) and add a new file within this folder called **"practice.test.js"**. This is where we will be practicing writing our first tests using Jest.

Writing Tests using Jest

Defining tests using Jest typically involves the following functions:

describe(name, fn): Optional - Creates a block that groups several related tests together:

```
describe('some tests', () => {
  // test definitions here
});
```

test(name, fn, timeout) (also under the alias: `it(name, fn, timeout)`) - This is the function that defines a test, identified by "name"

```
test('test name', () => {
  // test "expectations" here
});
```

expect: This is typically used in the form `expect(value)` and is used to test the value with **matcher functions** such as `".toBe()"`:

```
let x = 5;
expect(x).toBe(5);
```

Putting these concepts together, we can write our first test in **"practice.test.js"** as follows:

```
let sum = (num1, num2) => num1 + num2;

describe('Practice Tests', () => {
  test('sum function adds 1 + 2 to equal 3', () => {
```

Here, we have defined a function called "sum" and written a test within our "Practice Tests" group to ensure that it functions correctly. To run this test, open the integrated terminal for your app and execute the command:

```
npm run test
```

This should run Jest in "watch" mode and execute your practice.test.js file, showing the following output:

```
PASS  tests/practice.test.js
  Practice Tests
    ✓ sum function adds 1 + 2 to equal 3 (2 ms)
```

To ensure that this is working correctly, try modifying the test so that it fails, ie: `expect(sum(1, 2)).toBe(4);` and save the file. The test should run again and you will see the output:

```
FAIL  tests/practice.test.js
  Practice Tests
    ✗ sum function adds 1 + 2 to equal 3 (3 ms)
```

Introduction to "Matchers"

As stated above, Jest uses matcher functions ("matchers") to define a complete "expectation" for a value. These were designed to be as human-readable as possible and typically feature names like "toBe", "toHaveReturned", "toBeCloseTo", etc. By using matchers with `expect`, we can create 1 or more "expectations" for a test. If the test meets all of the expectations, then it passes.

The following is a list of the most common matchers from the official [Jest documentation](#), placed here for **reference**:

NOTE: For the full list, see the [expect API doc](#).

Truthiness

In tests, you sometimes need to distinguish between undefined, null, and false, but you sometimes do not want to treat these differently. Jest contains helpers that let you be explicit about what you want.

- `toBeNull` matches only `null`
- `toBeUndefined` matches only `undefined`
- `toBeDefined` is the opposite of `toBeUndefined`
- `toBeTruthy` matches anything that an `if` statement treats as true
- `toBeFalsy` matches anything that an `if` statement treats as false

For example:

```
test('null', () => {
  const n = null;
  expect(n).toBeNull();
  expect(n).toBeDefined();
  expect(n).not.toBeUndefined();
  expect(n).nottoBeTruthy();
  expect(n).toBeFalsy();
});

test('zero', () => {
  const z = 0;
  expect(z).not.toBeNull();
  expect(z).toBeDefined();
  expect(z).not.toBeUndefined();
```

You should use the matcher that most precisely corresponds to what you want your code to be doing.

Numbers

Most ways of comparing numbers have matcher equivalents.

```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```

For floating point equality, use `toBeCloseTo` instead of `toEqual`, because you don't want a test to depend on a tiny rounding error.

```
test('adding floating point numbers', () => {
  const value = 0.1 + 0.2;
  //expect(value).toBe(0.3); This won't work because of rounding
  //error
  expect(value).toBeCloseTo(0.3); // This works.
});
```

Strings

You can check strings against regular expressions with `toMatch`:

```
test('there is no I in team', () => {
  expect('team').not.toMatch(/I/);
});

test('but there is a "stop" in Christoph', () => {
  expect('Christoph').toMatch(/stop/);
});
```

Arrays and iterables

You can check if an array or iterable contains a particular item using `toContain`:

```
const shoppingList = ['diapers', 'kleenex', 'trash bags', 'paper
towels', 'milk'];

test('the shopping list has milk on it', () => {
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('milk');
});
```

Exceptions

If you want to test whether a particular function throws an error when it's called, use `toThrow`.

```
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK');
}

test('compiling android goes as expected', () => {
  expect(() => compileAndroidCode()).toThrow();
  expect(() => compileAndroidCode()).toThrow(Error);
```

NOTE: the function that throws an exception needs to be invoked within a wrapping function otherwise the `toThrow` assertion will fail.

Testing Components and Pages

If we want to use these testing techniques to test more than functions, arrays, strings, etc. within our Next.js app, we will need to install a few additional dependencies, ie:

- **jest-environment-jsdom:** - Used to define our "Test Environment" (Note: This was removed from Jest as of **version 28**, however it is still actively maintained)
- **@testing-library/react:** - A "very light-weight solution for testing React components. It provides light utility functions on top of react-dom and react-dom/test-utils, in a way that encourages better testing practices."

```
npm install --save-dev jest-environment-jsdom @testing-library/react
```

Before we can begin writing tests, we must create a **jest.config.mjs** file in "my-app" to configure the testing environment, ie:

File: "my-app/jest.config.mjs"

```
import nextJest from 'next/jest';
const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js
  // and .env files in your test environment
  dir: './',
```

This should complete the testing set up - now we can begin writing tests for components and pages. For now, we will start with a simple test that will examine the output of the default "index" page, included when an app is created using "create-next-app":

Test 1: "Vercel" Link rendered the within the "main" element

For this first test, we must ensure that a link to "<https://vercel.com>" is rendered within the within the "main" element (section). We know this to be true by visually expecting the output, but how can we test this programmatically?

1. First, create a new file called: **index.test.js** in the "tests" folder
2. Add the following dependencies:

```
import Home from '@/pages/index';
import { render } from '@testing-library/react';
```

3. Add the "Home Page" group by using the "describe" function:

```
describe('Home Page', () => {
  // ...
});
```

4. Finally, add the "test" function to define the test:

```
test("renders at least one link to https://vercel.com within the
'main' element", () => {
  const {container} = render(<Home />);
```

For this test, we have a number of "expectations" for our test to be true:

- Ensure the existence of the first child element of "main"
- The child element must contain one or more links
- One of the links within the child element must contain the text "<https://vercel.com>".

To test that this is indeed the case, we must render the component: "<Home />" using the `render` function. We store the `container` property of the `result`, which is the DOM node containing the rendered component. Using this, we can use familiar DOM functions such as `querySelector()` to get the child element.

Once we are sure that there is a child element, we use `querySelectorAll()` to grab all of the "a" elements within it. To determine if there are any links to "<https://vercel.com>" in that list, we iterate over all of the links and use the "includes" function to check for the substring.

Test 2: Component with User Event(s)

For this next test, we will re-create our familiar "ClickCounter" component and write a test to ensure that when the user clicks the button, the counter increases. To begin, let's first create the component:

1. Create a new "components" folder
2. Inside the "components" folder, create a new file: "ClickCounter.js"
3. Add the following code to define the "<ClickCounter />" component:

```
import {useState} from 'react'

export default function ClickCounter(){
```

4. (Optional) Place the component somewhere on the "Home" page and confirm that it functions correctly by clicking the button

With the button defined and functioning correctly, we can now proceed to write the corresponding test:

1. Within the "tests" folder, create a new file: "**ClickCounter.test.js**"
2. use npm to install "@testing-library/user-event":

```
npm install --save-dev @testing-library/user-event
```

3. Add the following dependencies:

```
import ClickCounter from '@/components/ClickCounter';
import userEvent from '@testing-library/user-event';
import { render } from '@testing-library/react';
```

4. Add the "ClickCounter Component" group by using the "describe" function:

```
describe('ClickCounter Component', () => {
  // ...
});
```

5. Add the "test" function to define the test:

```
test('increase count by 1 when clicked', async () => {
  const user = userEvent.setup();
  const { container } = render(<ClickCounter />);

  // attempt to fetch the "button" element
  const button = container.querySelector('button');
```

For this test, we have included another external "companion" library for Testing Library: ["user-event"](#)

user-event tries to simulate the real events that would happen in the browser as the user interacts with it. For example `userEvent.click(checkbox)` would change the state of the checkbox.

The more your tests resemble the way your software is used, the more confidence they can give you.

In the code above, we have invoked the ["setup\(\)"](#) method *before rendering our component*, as [recommended in the documentation](#). We then use the familiar `querySelector()` function to get a reference to the button and ensure that it contains the text "0" before the click event has been triggered.

To trigger the event itself, we use the ["click\(\)"](#) method. It's important that we execute this code using "await" as we cannot immediately execute the final "expect" without the event being triggered and the component updated as a result.

Test 3: API Route with Route Parameter

For our final test, we will implement an API route for a subset of our familiar "vehicles" static dataset from our ["simple-API" example](#):

1. Within the "pages/api" folder, create a new folder: **vehicles**
2. Within the newly created "pages/api/vehicles" folder, create a new file: **[id].js**
3. Enter the following code to define our dynamic "vehicles" api route:

```
let vehicleData = [
  {
    id: 1,
    year: 1994,
    make: 'Suzuki',
    model: 'SJ',
    vin: 'JN8AZ2KR6CT544012',
  },
  {
    id: 2,
    year: 1999,
    make: 'Chrysler',
    model: '300',
    vin: '1B3CC5FB5AN648885',
  },
  {
    id: 3,
    year: 2005,
    make: 'BMW',
    model: 'X3',
    vin: 'JTHBP5C29E5152916',
  },
];
export default function handler(req, res) {
  const { method } = req;
  const { id } = req.query;

  switch (method) {
    case 'GET':
      let vehicleIndex = vehicleData.findIndex((v) => v.id == id);
      // if a vehicleIndex was found, return the corresponding
      vehicle, else send a 404 error
      vehicleIndex != -1 ?
        res.status(200).json(vehicleData[vehicleIndex]) :
        res.status(404).end();
  }
}
```

The structure of the above should look somewhat familiar, as it was discussed when we [first introduced API routes](#) (please have a quick review if required). Basically, all we are doing here is allowing for a simple "GET" request with a single route parameter "id", ie: "api/vehicles/3" should return:

```
{  
  "id": 3,  
  "year": 2005,  
  "make": "BMW",  
  "model": "X3",  
  "vin": "JTHBP5C29E5152916"  
}
```

This route also sends a **404** status code if the requested "id" is not found, ie: "api/vehicles/4" should return an empty body with the status: 404.

To test this functionality, we will be using another 3rd party module to help make the request from within our test: "[node-mocks-http](#)". Once you have installed it using npm:

```
npm install --save-dev node-mocks-http
```

You can commence writing the test:

1. First, create a new file called: **vehicles.test.js** in the "tests" folder
2. Add the following dependencies:

```
import { createMocks } from 'node-mocks-http';  
import handler from '@/pages/api/vehicles/[id]';
```

3. Add the "/api/vehicles/[id] Route" group by using the "describe" function:

```
describe('/api/vehicles/[id] Route', () => {
  // ...
});
```

4. Add the first (of two) "test" functions to define the test for a known vehicle:

```
test('returns a vehicle for a specified ID', async () => {
  const { req, res } = createMocks({
    method: 'GET',
    query: {
      id: '1',
    },
  });

  await handler(req, res);

  expect(res._getStatusCode()).toBe(200);
  expect(JSON.parse(res._getData())).toEqual(
    expect.objectContaining({
      id: 1,
    })
  );
});
```

5. Add the second (of two) "test" functions to define the test for an unknown vehicle:

```
test('returns 404 when vehicle id not found', async () => {
  const { req, res } = createMocks({
    method: 'GET',
    query: {
      id: 'abc',
    },
  });
});
```

If we examine the first test ("returns a vehicle for a specified ID"), we can see that we use "createMocks" to create a mock "GET" request / response with the "**id**" route parameter value: **1**. We then provide the mocked req and res objects to the asynchronous API "handler" function. After this has completed, we can use the **_getStatusCode()** function to pull the resultant status code from the response and examine it in our test:

```
expect(res._getStatusCode()).toBe(200);
```

Similarly, we use the **_getData()** function to get the returned data from the response. However, rather than testing every value in the returned object, we **only** make sure that the id value matches route parameter (in this case, **1**):

```
expect(JSON.parse(res._getData())).toEqual(  
  expect.objectContaining({  
    id: 1,  
  })  
)
```

To achieve this, we use the **expect.objectContaining(object)** function with the "toEqual()" matcher.

The second test ("returns 404 when vehicle id not found") functions in almost exactly the same way as the first test, however instead of passing the route parameter: **1**, we pass **abc**. We also only test for the status code "404" as no object is returned.

E2E (End to End) Testing

When it comes to E2E or "End to End" testing, the first testing tool recommended in the [Next.js documentation](#) is "[Cypress](#)", a "next generation front end testing tool built for the modern web. [It addresses] the key pain points developers and QA engineers face when testing modern applications."

E2E Testing is a technique that tests your app from the web browser through to the back end of your application, as well as testing integrations with third-party APIs and services. These types of tests are great at making sure your entire app is functioning as a cohesive whole.

Cypress runs end-to-end tests the same way users interact with your app by using a real browser, visiting URLs, viewing content, clicking on links and buttons, etc. Testing this way helps ensure your tests and the user's experience are the same.

Writing end-to-end tests in Cypress can be done by developers building the application, specialized testing engineers, or a quality assurance team responsible for verifying an app is ready for release. Tests are written in code with an API that simulates the steps a that a real user would take.

Essentially, we will be using Cypress to test how multiple pieces of the application work together to enable the user to perform a series of tasks (ie: logging in, performing an action with multiple steps, logging out, etc.). This is a different approach to "Unit Testing", which focused on testing a specific "unit" of code (ie: a component, module, function, etc).

Installing / Configuring Cypress

To begin using Cypress for E2E testing, we first must install and configure it. For this example, we will be writing some tests on a sample application that implements ["Iron Session"](#).

To get started, obtain the "iron-session" example from the [Example Code](#).

Open this folder in Visual Studio Code and execute the following command in the Integrated Terminal:

```
npm install
```

You can then test that the example is functioning correctly by running the sample using the familiar command:

```
npm run dev
```

Once you are satisfied that it is working as expected, we can begin to install Cypress:

1. Install Cypress using NPM:

```
npm install --save-dev cypress
```

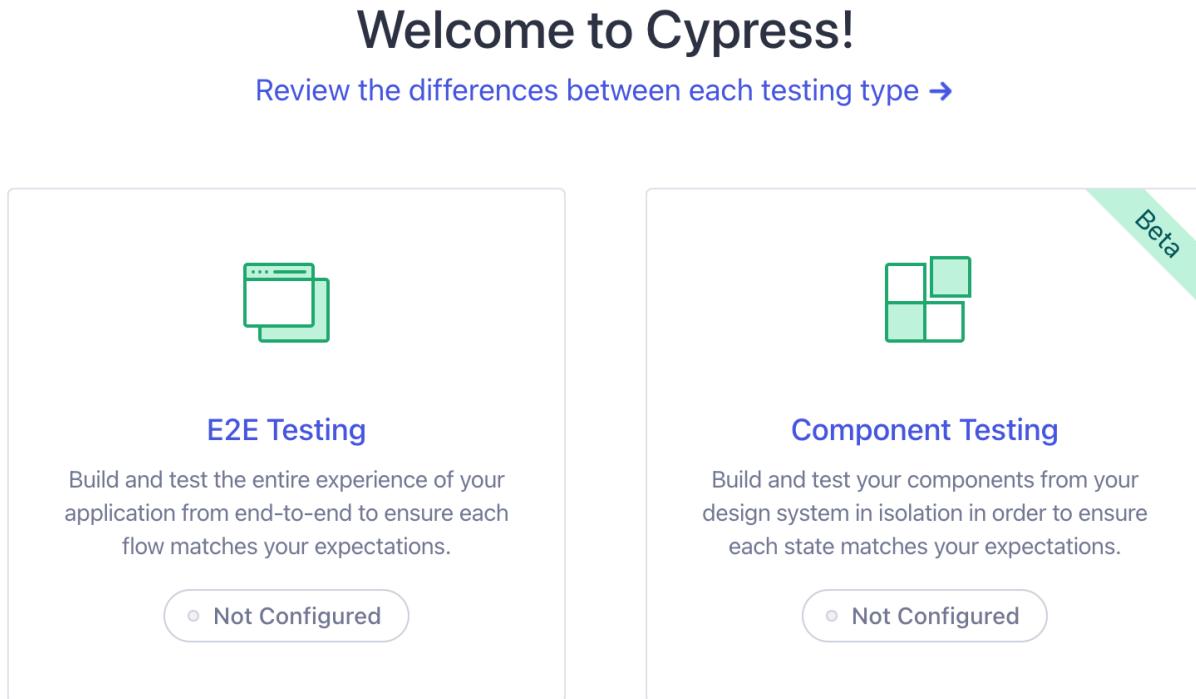
2. Add the following entry to "**scripts**" in **package.json**

```
"cypress": "cypress open"
```

3. Execute the command:

```
npm run cypress
```

This will open the "Cypress" app, which will provide a visual interface to help us configure the application for testing:



3. Click on the "**E2E Testing**" Box to configure the "Iron Session" example.

This performs the following actions:

- Adds a "cypress.config.js" file at the root of the folder
- Adds a "cypress/fixtures" folder containing the file: "example.json"
- Adds a "cypress/support" folder containing the files: "commands.js" and "e2e.js"

You can click the "**Continue**" button at the bottom to proceed to the next step

4. At the "Choose a Browser" prompt, click the green button to use the default option. This will likely be: "Start E2E testing in Chrome"

Choose a Browser

Choose your preferred browser for E2E testing.



Chrome

v103



Electron

v102



Firefox

v99

Start E2E Testing in Chrome

← Switch testing type

This will open a new Chrome window with the Cypress UI

5. At the next prompt: "Create your first spec", you can create your first spec file by clicking the "Create new empty spec" button on the right.

Create your first spec

Since this project looks new, we recommend that you use the specs and tests that we've written for you to get started.



Scaffold example specs

We'll generate several example specs to help guide you on how to write tests in Cypress.



Create new empty spec

We'll generate an empty spec file which can be used to start testing your application.

If you feel that you're seeing this screen in error, and there should be specs listed here, you likely need to update the spec pattern.

 [View spec pattern](#)

This will create a new folder in the "cypress" directory called "e2e" which will contain the first spec file: **spec.cy.js**

6. Once the spec is successfully added, ie:

```
describe('empty spec', () => {
  it('passes', () => {
    cy.visit('https://example.cypress.io');
  });
});
```

You can click the "Okay, run the spec" button to test it. You should see that the test runner successfully navigates to "<https://example.cypress.io>" and the spec passes.

Once this is done, you may close Cypress in the Integrated Terminal in Visual Studio Code with "**Ctrl + C**"

Testing the "iron-session" example

With Cypress correctly configured and executing a simple, boilerplate test, we can now focus on writing meaningful tests that test an example application that leverages "Iron Session" for authentication.

Before we write our first tests however, we must make one important configuration change: adding a **baseUrl** for our application:

By adding a **baseUrl** in your configuration Cypress will attempt to prefix the baseUrl any URL provided to commands like `cy.visit()` and `cy.request()` that are not fully qualified domain name (FQDN) URLs.

This allows you to omit hard-coding fully qualified domain name (FQDN) URLs in commands. For example,

```
cy.visit('http://localhost:3000/login')
```

can be shortened to

```
cy.visit('/login')
```

To achieve this in our application, we must open the **cypress.config.js** file and modify it to include a **baseUrl** property under "e2e", ie:

```
const { defineConfig } = require('cypress');

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      // implement node event listeners here
    },
    baseUrl: 'http://localhost:3000',
  },
});
```

Cypress Test Syntax

You have probably noticed that the syntax for writing tests looks very similar to what was discussed when we wrote our first tests using "Jest". There exists a "describe" function as well as an "it" function that works the same way as the "test" function in Jest (to identify a test).

NOTE: Recall, you can use the function "it()" in Jest as well, instead of "test()", as "it()" is an alias for "test()" - see: <https://jestjs.io/docs/api#testname-fn-timeout>

The common functions and commands that we will be using to write our tests are as follows. For a full list of commands, see "Commands" in the [official Cypress documentation](#):

describe(name, fn): Creates a block that groups several related tests together:

```
describe('some tests', () => {
  // test definitions here
```

it(name, fn) - This is the function that defines a test, identified by "name"

```
it('test name', () => {
  // test "expectations" here
});
```

cy.visit() - Visit (navigate to) a remote URL

```
cy.visit('/'); // visits the baseUrl
cy.visit({
  url: '/pages/hello.html',
  method: 'GET',
});
```

cy.url() - Get the current URL of the page that is currently active.

```
cy.url(); // Yields the current URL as a string
```

cy.should() - Create an assertion. Assertions are automatically retried until they pass or time out. These typically take the form of `.should(chainer, value)`, where "chainer" is one of the available assertions [listed here](#), such as "include", "match", etc. and are chained (cannot be called directly from "cy").

```
cy.url().should('include', '/login');
cy.url().should('match', /.*/(\login)/);
```

cy.get() - Get one or more DOM elements by selector or alias

```
cy.get('.list > li'); // Yield the <li>'s in .list
```

cy.contains() - Get the DOM element containing the text. DOM elements can contain more than the desired text and still match. Additionally, Cypress prefers some DOM elements over the deepest element found.

```
cy.get('.nav').contains('About'); // Yield element in .nav  
containing 'About'  
cy.contains('Hello'); // Yield first element in document  
containing 'Hello'
```

cy.click() - Click a DOM element.

```
cy.get('.btn').click(); // Click on button  
cy.contains('Welcome').click(); // Click on first element  
containing 'Welcome'
```

cy.type() - Type into a DOM element. Curly braces {} may be used to type a key such as "enter", "esc", "backspace", etc.

```
cy.get('input').type('Hello, World'); // Type 'Hello, World' into  
the 'input'  
cy.get('input').type('{enter}'); // Press the "enter" key while on  
the 'input'
```

Test 1 Protected Route /profile-sg

For this first test, we will assert that the "/profile-sg" route cannot be accessed without first logging in. To create this test, we will be using the "spec.cy.js" file, so go ahead and comment out the existing test that was created for us:

```
// describe('empty spec', () => {
//   it('passes', () => {
//     cy.visit('https://example.cypress.io')
//   })
// })
```

Instead, we will be defining a new block of tests, ie:

```
describe('login / logout flow specification', () => {});
```

Within the callback, we will write the first test. The steps we need to verify are

1. User attempts to navigate (visit) the route "/profile-sg"
2. User is redirected to "/login" route

To test the above flow, we can use the following test:

```
it('cannot navigate to /profile-sg without being logged in', () => {
  cy.visit("/profile-sg")
    .url().should('include', "/login");
});
```

Notice how we can "chain" the operations, ie `cy.visit().url().should()`. In the above code, we first attempt to visit the route "/profile-sg" and once this is

complete, we examine the url to ensure that we are indeed at "login".

Test 2 Rejecting Invalid Github Users

To verify the login functionality of the app, we should make sure that an unknown GitHub user is not accepted past the "Login" process, ie:

1. Navigate (visit) the route "/login"
2. Type in an unknown GitHub User (ie: "!!!" into the "userName" input element)
3. Hit the "enter" key to submit the form
4. User remains on the route "/login".

To test this flow, we can use the following test:

```
it('rejects a login attempt by an invalid github user: !!!', () => {
  cy.visit("/login")
    .get('input[name="username"]').type("!!!").type("{enter}")
    .url().should('include', "/login");
});
```

Here, we first navigate to the "/login" route before "getting" the "input" element for username. We then instruct the test to type the invalid username and hit enter. Once this is complete we assert that the url does indeed remain at "/login".

Test 3 Granting Access to Valid Github Users

In an effort to further verify the login functionality of our app, we should also write another test that successfully authenticates a known GitHub user. Additionally, once the user has been authenticated, we must ensure that they

can access the protected route (/profile-sg), which was denied in our first test. Finally, we should ensure that once they have logged in, they can log out.

Essentially, we must verify the following flow:

1. Navigate (visit) the route "/login"
2. Type in an unknown GitHub User (ie: "test-account" into the "userName" input element)
3. Hit the "enter" key to submit the form
4. User should be directed to /profile-sg
5. Click the "Logout" button
6. User should be directed to /login

This can be accomplished using the following test:

```
it('successfully authenticates a valid github user: test-account  
and logs out', () => {  
  cy.visit("/login")  
  .get('input[name="username"]').type("test-  
account").type("{enter}")  
  .url().should('include', '/profile-sg')  
  .get("nav").contains("Logout").click()  
  .url().should('include', "/login");  
});
```

This is very similar to the previous test, however this time we assert that the url includes "/profile-sg" instead of "/login" after the login attempt. Additionally, we get the "Logout" button within the "nav" element and click it. If the user was directed back to "/login" then we know that this flow is functioning correctly .

NOTE: For more examples of how to run tests, including different commands such as working with cookies, files, network requests, the

global window object and much more see the [official documentation](#) as well as the excellent "[Kitchen Sink](#)" example app, provided by Cypress.

Running in "Headless" Mode

If you do not wish to run your tests using the GUI tool, it is also possible to run the tests strictly from the command prompt (ie: "[Headlessly](#)"). All that is required is that we add the "cypress run" command to "scripts" in **package.json**, ie:

```
"cypress:headless": "cypress run"
```

To start testing, we can run:

```
npm run cypress:headless
```

Example Code

You may download the sample code for this topic here:

| Testing-Introduction

Continuous Integration

You have likely seen the acronym "CI/CD" used when describing modern application development. In this context, it stands for "Continuous Integration / Continuous Deployment" and is a vital technique in ensuring software quality and quick delivery. To begin, we will first discuss "Continuous Integration", which [Atlassian](#) defines as:

The practice of automating the integration of code changes from multiple contributors into a single software project. It's a primary [DevOps best practice](#), allowing developers to frequently merge code changes into a central repository where builds and tests then run. Automated tools are used to assert the new code's correctness before integration.

A source code version control system is the crux of the CI process. The version control system is also supplemented with other checks like automated code quality tests, syntax style review tools, and more.

Notice how a "source code version control system" was mentioned as a vital part (the crux) of the process. Therefore, to get started using Continuous Integration, we must be familiar with such a system. Fortunately, there is a widely used version control system that is free to use: [GitHub](#).

Git / GitHub Review

If you have ever "pushed" code to [GitHub](#), you are likely familiar with some of the common "git" commands:

NOTE: If you do not currently have "git" installed on your system (verified using the command: `git --version`), you can install it using the [instructions here](#).

git init: Initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.

```
git init
```

git clone: Creates a local copy of a project that already exists remotely. The clone includes all the project's files, history, and branches.

```
git clone git://git.kernel.org/pub/scm/.../linux.git my-linux
cd my-linux
```

git add: Stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project's history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.

```
git add .
```

git commit: Saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that's been staged with git add will become a part of the snapshot with git commit.

```
git commit -m "initial commit"
```

git status: Shows the status of changes as untracked, modified, or staged.

```
git status
```

git remote: Manage the set of repositories ("remotes") whose branches you track.

```
git remote -v
```

git checkout: Switch branches or restore working tree files (the -b flag creates a new branch before switching to it)

```
git checkout -b new-feature
```

git branch: Shows the branches being worked on locally.

```
git branch
```

git merge: Merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature

branch into the main branch for deployment.

```
git checkout master  
git merge new-feature
```

git pull: Updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.

```
git pull origin master
```

git push: Updates the remote repository with any commits made locally to a branch.

```
git push origin master
```

For more information, see the [full reference guide to Git commands](#).

To practice some of these commands, grab the [example code](#) for this week and open the "**app-with-tests**" folder in Visual Studio Code. Next, issue the command `npm install` to fetch the dependencies / rebuild the `node_modules` folder.

Hosting Your Code

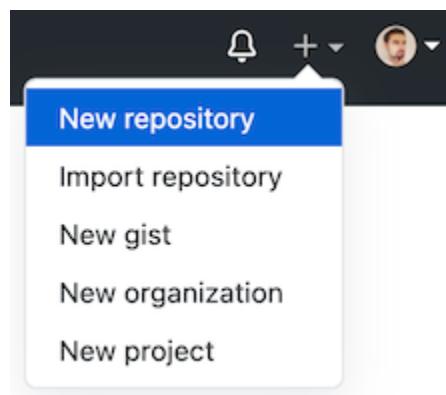
Now that we have reviewed some of the basic "git" commands, it's time to host

our code on GitHub. If you don't have an account on [GitHub](#), create one now.

Create a GitHub Repository

Sign in to your GitHub account.

Find and click a "+" button on the Navigation Bar. Then, choose "New Repository" from the dropdown menu.



Fill in the repository name text field with the name of your project. Also, make sure that the "Private" option is selected:

Create a new repository

Owner *



Repository name *

my-app 

Great repository names are short and memorable. Need inspiration? How about [shiny-palm-tree](#)?

Description (optional)

 Public

Anyone on the internet can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

[Create repository](#)

Once you're happy with the settings, hit the "Create repository" button.

Prepare Our Local Git Repo

Open the terminal and change the current working directory to your app.

You can run `git status` to verify that Git is set up properly. If you see `fatal: not a git repository (or any of the parent directories): .git` error

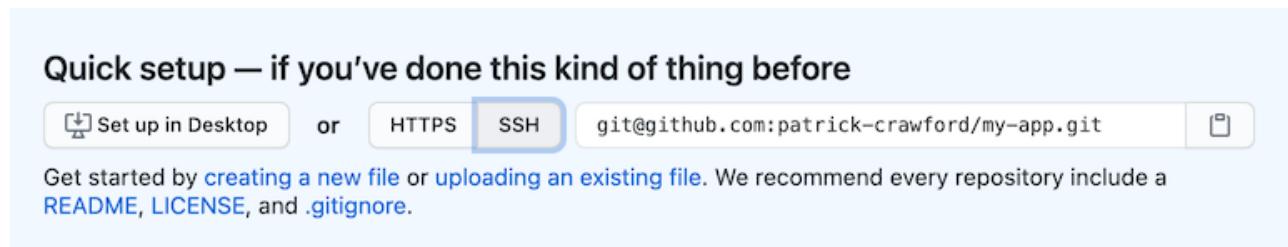
message, then your local Git repository does not exist and you need to initialize it using `git init`.

Now that we're sure that our local git repository is set up, we need to add and commit all of our code changes:

1. Add the files to the local repository by running `git add .`
2. Commit the newly added files: `git commit -m "Initial commit"`.

Connect the Local Git Repository to GitHub

Go to your GitHub repository and click the "copy" button in the "Quick Setup" block:



This will copy the URL of your remote GitHub repository.

Now, go back to your Terminal again and add this remote URL by running the following command:

```
git remote add origin URL
```

where **URL** is the remote repository URL that you have copied in the previous step.

If you run `git remote -v`, you should see something like this:

```
origin      git@github.com:patrick-crawford/my-app.git (fetch)  
origin      git@github.com:patrick-crawford/my-app.git (push)
```

Finally, commit your changes (if you have not yet done so) and push the code from your local repository to the remote one:

```
git push origin master
```

NOTE: Your default branch may be "main" - to confirm which branch you are on, execute the command `git status`

You can verify that the code was pushed by going back to your Browser and opening your GitHub repository.

The screenshot shows a GitHub repository page for 'patrick-crawford / my-app'. The 'Code' tab is selected. At the top, there are buttons for 'Unwatch', 'Fork', 'Star', and settings. Below the tabs, there are buttons for 'master', '1 branch', '0 tags', 'Go to file', 'Add file', and 'Code'. The main area displays a list of commits:

| Author | Commit Message | Time | Commits |
|------------------|----------------|---------------|----------|
| patrick-crawford | initial commit | 4 minutes ago | 1 commit |
| cypress | initial commit | 4 minutes ago | |
| pages | initial commit | 4 minutes ago | |
| public | initial commit | 4 minutes ago | |
| styles | initial commit | 4 minutes ago | |
| tests | initial commit | 4 minutes ago | |
| .eslintrc.json | initial commit | 4 minutes ago | |
| .gitignore | initial commit | 4 minutes ago | |

To the right, there is an 'About' section with the message 'No description, website, or topics provided.' and a 'Readme' link. It also shows statistics: 0 stars, 1 watching, and 0 forks. Below that is a 'Releases' section with a link to 'Create a new release'.

Automating Tasks

GitHub has an amazing automation platform that we can use within our projects called "[GitHub Actions](#)":

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository. For example, you can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in your repository.

GitHub provides Linux, Windows, and macOS virtual machines to run your workflows, or you can host your own self-hosted runners in your own data center or cloud infrastructure.

To get started using GitHub actions, we begin by creating a "workflow" that responds to an event. This involves creating the following starter file:

File: .github/workflows/ci.yaml

```
name: CI  
on: [push]
```

Notice how we have named the file "ci.yaml", ie: "ci" for "continuous integration" and the ".yaml" extension for a "[yaml](#)" file: "a human-friendly data serialization language for all programming languages". You can think of it as an alternative to .json that is typically used for writing configuration files. The

official documentation offers a "[Reference Card](#)" as a reference for the full syntax, however we will only be using a small subset of this - just enough to correctly configure our "ci" workflow. One of the major differences that you will notice right away is that it uses whitespace instead of curly braces.

In the above file, we have two properties:

- The workflow [name](#)
- The event to trigger the workflow, in this case "push" defined in an array for the property [on](#). By using a "push" event, the workflow will run whenever code is "pushed" to the repository using [git push](#).

While this is enough to register the action for our repository on GitHub, it will not run as we have not defined any "jobs". As our first job, let's run a simple "lint" check using ESLint.

NOTE: Since version 11.0.0, Next.js provides an integrated [ESLint](#) experience out of the box (which we opted into when creating a new app). By executing the command "npm run lint" you can "lint" your code, which "statically analyzes your code to quickly find problems". This is an important step in CI, as we do not want any JS errors potentially breaking our build further down in the pipeline.

Add the following code to update the "ci.yaml" file:

```
jobs:  
  run-tests:  
    name: Lint and Test  
    runs-on: ubuntu-latest  
  
    steps:  
      - name: Checkout code  
        uses: actions/checkout@v4
```

We have added quite a lot to the file, so let's discuss the purpose of each property before moving on and testing our workflow:

- **jobs:** A workflow run is made up of one or more jobs, which run in parallel by default.
- **run-tests:** The id of our job - notice it is "run-tests", this is because we will eventually be automating tests as part of our workflow
- **name:** The name of our job, which will be displayed in the Github GUI
- **runs-on:** Used to define the type of machine to run the job on, in this case we will use the extremely common "ubuntu-latest" (ubuntu 22.04 at time of writing) setting, however GitHub offers other [GitHub-hosted runners](#), including: **windows-latest** and **macos-latest**, in addition to specific versions.
- **steps:** A job contains a sequence of tasks called steps. Steps can run commands, run setup tasks, or run an action in your repository, a public repository, or an action published in a Docker registry. Not all steps run actions, but all actions run as a step.

NOTE the following properties are prefixed with a "-" character, to indicate that they are part of a [Block Sequence](#)

- **- name:** Checkout code, **uses:** actions/checkout@v4: This defines the first step of our job, named "Checkout code". It uses the ["Checkout V4" Action](#), which: "checks-out your repository under \$GITHUB_WORKSPACE, so your workflow can access it."
- **- name:** Install packages, **run:** npm ci: This second step, named "Install packages" runs a special version of "npm install"; **npm ci**. The "ci" command is "similar to npm install, except it's meant to be used in automated environments such as test platforms, continuous integration,

and deployment -- or any situation where you want to make sure you're doing a clean install of your dependencies."

- - **name:** Run ESLint, **run:** npm run lint: This is the final "linting" step, as identified above. With the environment set up and the packages obtained, we should be able to successfully execute this command.

Running our First Workflow

Now that we have a complete workflow defined, ie:

File: .github/workflows/ci.yaml

```
name: CI
on: [push]
jobs:
  run-tests:
    name: Lint and Test
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Install packages
        run: npm ci

      - name: Run ESLint
        run: npm run lint
```

Let's **check in our code** using "git" and once again "**push it to GitHub**".

Once this has completed, view your repository on <https://github.com> and

navigate to the "Actions" tab to see the workflow status. Each workflow run is named after the commit message, in this case "added ci.yaml".

NOTE: You may also click on the workflow run from this screen to see the detailed steps:

The screenshot shows the GitHub Actions page for the repository 'patrick-crawford / my-app'. The 'Actions' tab is selected. A single workflow run is listed under 'All workflows', titled 'added ci.yaml'. The run was triggered by CI #2: Commit c4d8723 pushed by patrick-crawford. It is associated with the 'master' branch and completed 1 minute ago, with a duration of 48s. There are filter options for Event, Status, Branch, and Actor.

Next, let's see what happens if we introduce some code that causes ESLint to report an error:

File: "pages/_app.js"

```
import '@/styles/globals.css'

export default function App({ Component, pageProps }) {
  return <br /><Component {...pageProps} />
}
```

This should yield the following ESLint Error:

```
Error: Parsing error: Adjacent JSX elements must be wrapped in an
enclosing tag. Did you want a JSX fragment <>...</>? (4:15)
```

With the broken code in place, push your updated code to GitHub and view the next workflow run in the "Actions" tab - it should appear directly above the previous workflow:

| 2 workflow runs | Event ▾ | Status ▾ | Branch ▾ | Actor ▾ |
|--|---------|----------|----------------|---------|
| ✖ introduced ESLint error CI #3: Commit e31b85f pushed by patrick-crawford | master | | 1 minute ago | ... |
| ✓ added ci.yaml CI #2: Commit c4d8723 pushed by patrick-crawford | master | | 13 minutes ago | ... |

If you examine the detailed output, you should see error repeated as above.

Adding Unit / E2E Testing

You will notice that our sample project "app-with-tests" has been configured to test the "Home" component using both "Jest" and "Cypress" Tests:

File: "tests/index.test.js"

This is simply the first "Jest" test that was covered when "Unit Testing" was **first discussed**, ie: asserting that the Home component renders at least one link to <https://vercel.com> within the first child element of the "main" section.

File: "cypress/e2e/spec.cy.js"

```
describe('check Not Found (404)', () => {
  it('should return status 404 when visiting /unknown', () => {
    // See Request: https://docs.cypress.io/api/commands/request and
```

This Cypress test uses a "command" that we haven't seen before: `"cy.request()"`. Essentially all that we are doing here is making a request to `/unknown`, knowing that we do not have a route defined for that path. The expected behaviour is that the status code for the returned page is "404". The `"its()"` command is used to get a property value ("status") of the returned object.

If we wish to include the execution of these tests as a part of our workflow, we must add the following additional steps to our "Lint and Test" job, defined in **.github/workflows/ci.yaml**:

```
- name: Run Jest Tests
  run: npm run ci:test

- name: Run Cypress Tests
  uses: cypress-io/github-action@v6
  with:
    build: npm run build
    start: npm start
```

Here, the syntax for the "Jest" test is straightforward, as it is the same command that we would run in the integrated terminal in Visual Studio Code. However, setting up the "Cypress" E2E test is less straightforward, since the app must technically be running to perform the tests.

To get this working, we have used a github action [created for Cypress](#). We use `"with"` to provide a map of input parameters to the action, ie the "build" and "start" commands.

Once you have added the above code to your `".github/workflows/ci.yaml"` file, check in the updated code and "push" it to GitHub to confirm that the updated workflow is indeed running the tests.

| 4 workflow runs | Event ▾ | Status ▾ | Branch ▾ | Actor ▾ |
|---|---------|----------|---------------|---------|
| ✓ added testing to ci.yaml CI #5: Commit 469b65a pushed by patrick-crawford | master | | 2 minutes ago | ... |
| ✓ fixed ESLint Error CI #4: Commit 4d62fd0 pushed by patrick-crawford | master | | 1 hour ago | ... |
| ✗ introduced ESLint error CI #3: Commit e31b85f pushed by patrick-crawford | master | | 1 hour ago | ... |
| ✓ added ci.yaml CI #2: Commit c4d8723 pushed by patrick-crawford | master | | 2 hours ago | ... |

Merging Code from Other Branches

Since part of "Continuous Integration" is integrating code changes and merging them into a central repository, we must discuss what this looks like in GitHub and how our Action (workflow) can help ensure code correctness for our main / master branch.

First, let's work on a bug identified by the client; in this case, they have noted that there must be a ":" character after the text "By" before the "Vercel" logo. In addition to fixing this issue, let's break one of our tests to see how we may be alerted of the potential danger of merging this bug fix into the master branch:

To begin, issue the following command to checkout a new "branch" for the fix

```
git checkout -b fix/logo
```

You can verify that we have moved to the new branch by executing the

command:

```
git branch
```

Now that we know we are working on the "fix/logo" branch and not the "master" branch, we can proceed to update the code next to the "Vercel" logo within the "Home" component:

File: "pages/index.js"

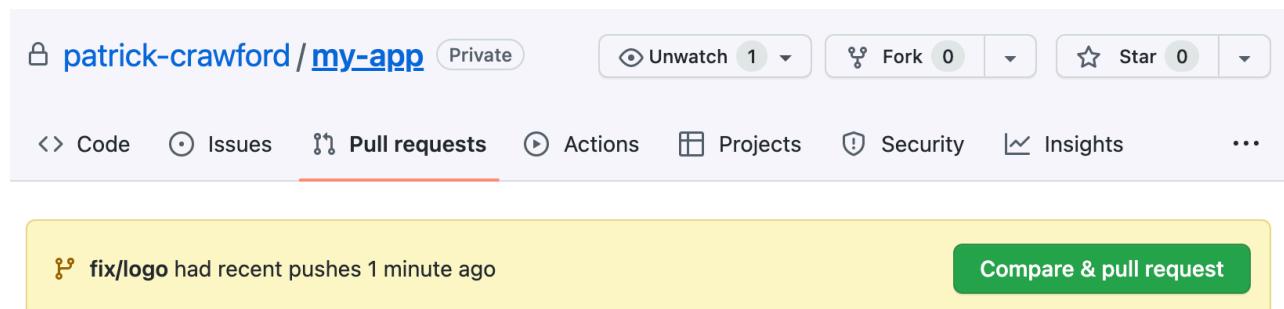
```
<div>
  <a
    href="https://abc.com?utm_source=create-next-app&utm_medium=default-template&utm_campaign=create-next-app"
    target="_blank"
    rel="noopener noreferrer"
  >
    By:{' '}
    <Image
      src="/vercel.svg"
      alt="Vercel Logo"
      className={styles.vercelLogo}
      width={100}
      height={24}
      priority
    />
  </a>
</div>
```

Notice how we have added the required ":" character in the appropriate place in the UI to fix the bug. However, we have also introduced a potential problem by changing the "vercel.com" link to "abc.com", thus breaking our "Jest" unit test and making this bug fix unfit for merging with the "master" branch.

To see how we can be alerted of this, go ahead and execute the following commands to commit the code and push it to GitHub:

```
git add .
git commit -m "Fix - added : after By before logo"
git push origin fix/logo
```

After you have pushed your branch to GitHub using the above code, open your browser to view your code on <https://github.com> and navigate to the "Pull Requests" tab:



You will notice that GitHub has detected a recent push to the branch fix/logo and is suggesting that you make a "pull request"

NOTE: Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch. For more information, see the [GitHub Documentation on Pull Requests](#)

Click this button to open a "pull request". At the next page, use the default values and create the request by pressing the "Create pull request" button.

This should take you to detailed information for the pull request where you can see that there is an issue: "All checks have failed"

Fix - added : after By before logo #1

[Open](#) patrick-crawford wants to merge 1 commit into `master` from `fix/logo`

Conversation 0 Commits 1 Checks 1 Files changed 1

patrick-crawford commented now

No description provided.

Fix - added : after By before logo 2f28f0b

Add more commits by pushing to the `fix/logo` branch on [patrick-crawford/my-app](#).

All checks have failed 1 failing check [Hide all checks](#)

CI / Lint and Test (push) Failing after 30s [Details](#)

This branch has no conflicts with the base branch Merging can be performed automatically.

[Merge pull request](#) You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

It is from this screen that you can comment on the pull request, requesting that the developer fix the breaking code, etc.

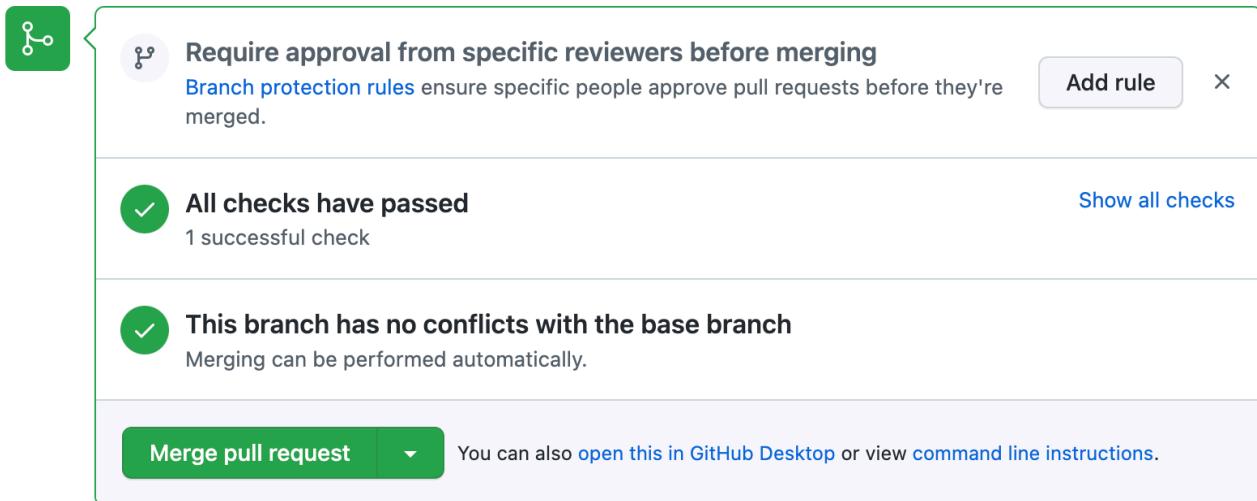
To fix this code and make it fit for merging, change "abc.com" back to "vercel.com" and check in the branch once again using the previous commands, ie:

```
git add .
git commit -m "Fix - restored vercel.com link"
git push origin fix/logo
```

This will trigger the tests to once again run and update the pull request, indicating that "All Checks have passed" and the "Merge pull request" is now

shown in green:

Add more commits by pushing to the **fix/logo** branch on [patrick-crawford/my-app](#).



A screenshot of a GitHub pull request merge interface. On the left, there's a green 'Merge' button with a gear icon. To its right, a box contains several status items: 'Require approval from specific reviewers before merging' (with a note about branch protection rules), 'All checks have passed' (1 successful check), and 'This branch has no conflicts with the base branch' (noting automatic merging). At the bottom, a large green 'Merge pull request' button is visible, along with a note about opening in GitHub Desktop or viewing command line instructions.

Go ahead and merge the request to update your "master" branch on GitHub with the bug fix. If you wish to update your local "master" branch with the fix, you can issue the commands:

```
git checkout master  
git pull origin master
```

Continuous Deployment

The second piece of the "CI/CD" acronym may either stand for "Continuous Delivery" or "Continuous Deployment". From [Atlassian](#):

Continuous Delivery:

Continuous delivery is an extension of continuous integration since it automatically deploys all code changes to a testing and/or production environment after the build stage.

This means that on top of automated testing, you have an automated release process and you can deploy your application any time by clicking a button.

In theory, with continuous delivery, you can decide to release daily, weekly, fortnightly, or whatever suits your business requirements. However, if you truly want to get the benefits of continuous delivery, you should deploy to production as early as possible to make sure that you release small batches that are easy to troubleshoot in case of a problem.

Continuous Deployment:

Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

Continuous deployment is an excellent way to accelerate the feedback loop with your customers and take pressure off the team as there isn't a

"release day" anymore. Developers can focus on building software, and they see their work go live minutes after they've finished working on it.

For our purposes, we will be discussing and practicing "Continuous Deployment" using the same "app-with-tests" **example code** from the discussion on "Continuous Integration" (if you wish to begin with a completed version of that code, you may use the "app-with-CI" folder. However, you will need your code to be pushed to GitHub for the following instructions).

(Re)Introduction to Vercel

To add Continuous Deployment to our pipeline, we will leverage the excellent free services available from **Vercel**, the creators of Next.js.

To begin using Vercel, we must ensure that our code has been pushed to a public Git provider such as: **GitHub**, **GitLab** or **BitBucket**. If you have been following along with the **Continuous Integration** instructions, this should indeed be the case.

1. First, browse to <https://vercel.com> and hit the "Start Deploying" button.
2. Next, press the "Continue with GitHub" button, since our code is located on GitHub.
3. If you are not currently logged in to GitHub, you will need to provide your credentials in a pop-up window before continuing.
4. Once you have logged in to GitHub, you will be taken to the **Let's build something new.** screen in Vercel, which prompts you to "Import Git Repository". From here, you will need to click "+ Add GitHub Account"

Import Git Repository

The screenshot shows a user interface for importing a Git repository. At the top left is a dropdown menu labeled "Select a Git Namespace" with a magnifying glass icon. To its right is a search bar with a magnifying glass icon and the placeholder text "Search...". Below these are two buttons: "+ Add GitHub Account" with a plus sign icon and "Switch Git Provider" with a three-dot icon.

5. This will prompt you to "Install Vercel". From here, we would like to find our specific app, ie "my-app" on GitHub. You will need to start by clicking your account and choosing the "**Only select repositories**" and finding the "my-app" repository.

The screenshot shows the "Select repositories" interface. At the top, it says "Install on your personal account Patrick Crawford" and shows a profile picture of a man. Below are two radio button options: "All repositories" (unchecked) and "Only select repositories" (checked). A note below the checked option says "Select at least one repository." A "Select repositories" button with a dropdown arrow is shown. Below the button, it says "Selected 1 repository." A list item "patrick-crawford/my-app" is shown with a delete "X" icon next to it.

6. Once you have selected the target repository (ie: "my-app"), click the green **Install** button
7. You should now see the **my-app** repository available for import. To proceed, click **Import**

Import Git Repository

patrick-crawford

Search...

my-app · 1d ago

Import

Missing Git repository? [Adjust GitHub App Permissions →](#)

8. At the next page, you are not required to make any changes, as Vercel should detect that we are using Next.js in the "framework preset". If you had any *environment variables*, you could set them here. However, since we aren't currently using any environment variables, click **Deploy**.
9. Once the deploy step has completed, you should be taken to a "Congratulations!" page with a black button labeled **Go To Dashboard**. Click this to see the information about your deployment.

Production Deployment



The deployment that is available to your visitors.

The screenshot shows the Vercel dashboard for a production deployment. On the left, there's a preview of the app with the title "Welcome to Next.js!" and links to documentation, learn, examples, and deploy. On the right, under "DEPLOYMENT", it shows the URL "my-obq11m1fs-patrick-crawford.vercel.app". Below that, under "DOMAINS", is the URL "my-app-lime-eight.vercel.app" with a "+2" badge. Under "STATUS", it says "CREATED" with a "Ready" status and a timestamp of "6m ago". Under "BRANCH", it shows "master". A message at the bottom says "Merge pull request #1 from patrick-crawford/fix/footer Fix - add...".

To update your Production Deployment, push to the "master" branch.

[Learn More](#)

- From the dashboard, we are presented with some important information, primarily the "Domains" section, which shows the production URL of the app (in this case: my-app-lime-eight.vercel.app)

Updating the Production Site

Now that we have a working site deployed on Vercel (ie: in "Production"), let's take a step back and see how we can use this pipeline to make an update:

For example, say we wish to append the text "(Starter Site)" to the `<title>...<title>` element of the "Home" page:

- Create (and switch to) a new branch for this update.

```
git checkout -b fix/home-title
```

2. Update the code and save your changes, ie:

File: "pages/index.js"

```
<title>Create Next App (Starter Site)</title>
```

3. Commit your local changes:

```
git add .  
git commit -m "appended text to title on Home"
```

4. Push the "fix/home-title" branch to GitHub

```
git push origin fix/home-title
```

Preview Deployments

If you navigate back to the dashboard for the Vercel app, you will see that we now have a "Preview Deployment", generated from our push to the "fix/home-title" branch:

Preview Deployments

| | | | |
|---|----------------|--|--|
| my-jk1f8wunw-patrick-crawford.vercel.app Preview | ● Ready 20s | appended text to title on Home ↳ fix/home-title | 3m ago by patrick-crawford  : |
|---|----------------|--|--|

Effectively, this means that we have not altered our production site (my-app-lime-eight.vercel.app), but instead created a "Preview" of what the production

site would look like with the change. In this case, this preview has been assigned the url "my-jk1f8wunw-patrick-crawford.vercel.app".

As before, since we pushed a separate branch to our repository, GitHub gives an option to create a Pull Request. Once we do, we can confirm that Vercel has created the preview deployment and all of our tests passed:

The screenshot shows a GitHub pull request interface. At the top, a comment from the 'vercel' bot is displayed, stating: 'The latest updates on your projects. Learn more about [Vercel for Git](#)'. Below this is a table showing a single deployment:

| Name | Status | Preview | Updated |
|--------|---|-------------------------------|------------------------------|
| my-app | <input checked="" type="checkbox"/> Ready (Inspect) | Visit Preview | Aug 18, 2022 at 3:08PM (UTC) |

Below the table, a message encourages adding more commits by pushing to the `fix/home-title` branch on the `patrick-crawford/my-app` repository.

Further down, a summary states: 'This branch was successfully deployed' with '1 active deployment'. A green 'Merge pull request' button is visible.

A detailed view of merge requirements follows:

- Require approval from specific reviewers before merging**: Ensures specific people or teams approve pull requests before they're merged into your master branch. An 'Add rule' button is present.
- All checks have passed**: 2 successful checks. A 'Show all checks' link is available.
- This branch has no conflicts with the base branch**: Merging can be performed automatically.

At the bottom, there's a note: 'Merge pull request' with a dropdown arrow, and 'You can also open this in GitHub Desktop or view command line instructions.'

If the "Preview" is approved, then we can proceed to merge the pull request. This will trigger a rebuild with Vercel and our production site will be updated to match the Preview.

As before, if you wish to update your local "master" branch with the fix, you can issue the commands:

```
git checkout master  
git pull origin master
```

Example Code

You may download the sample code for this topic here:

| Deployment-Automated-Testing

Analyzing Performance

As we have seen, Next.js applications can be extremely fast out of the box, thanks to features like [automatic static optimization](#) and [route prefetching](#). However, it is still extremely important to ensure that your code is correctly using all of the resources available to achieve the best experience possible for your users.

Core Web Vitals

In May of 2020, Google introduced "Core Web Vitals", a subset of their "[Web Vitals](#)" metrics, designed to focus on distinct facets of the user experience:

Core Web Vitals are the subset of Web Vitals that apply to all web pages, should be measured by all site owners, and will be surfaced across all Google tools. Each of the Core Web Vitals represents a distinct facet of the user experience, is measurable [in the field](#), and reflects the real-world experience of a critical [user-centric](#) outcome.

The metrics that make up Core Web Vitals will [evolve](#) over time. The current set for 2020 focuses on three aspects of the user experience: *loading, interactivity, and visual stability*

By providing a guideline and metrics for creating performant web sites, Google has made it easier for developers to identify and fix potential problems:

Site owners should not have to be performance gurus in order to understand the quality of experience they are delivering to their users. The Web Vitals initiative aims to simplify the landscape, and help sites focus on the metrics that matter most.

Additionally, "Core Web Vitals" have been added to Google's "[page experience signals](#)", which have an impact on how your site is ranked in Google. Therefore it is imperative that we understand Core Web Vitals and how we can measure and improve them.

Next.js has provided an [excellent introduction](#) for these metrics in their documentation:

Largest Contentful Paint (LCP)

The **Largest Contentful Paint (LCP)** metric looks at the loading performance of your page. LCP measures the time it takes to get the largest element on the page visible within the viewport. This could be a large text block, video, or image that takes up the primary real estate on the page.

As the DOM is rendered, the largest element on the page may change. The Largest Contentful Paint doesn't stop counting until the largest image or element is seen on-screen.

[According to Google](#), sites should strive to have Largest Contentful Paint take **2.5 seconds or less**.



First Input Delay (FID)

The **First Input Delay (FID)** metric measures the time from when a user first

interacts with a page (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to begin processing event handlers in response to that interaction.

The First Input Delay (FID) metric helps measure your user's first impression of your site's interactivity and responsiveness.

According to Google, sites should strive to have a First Input Delay of **100 milliseconds or less**.



Cumulative Layout Shift (CLS)

The **Cumulative Layout Shift (CLS)** metric is a measure of your site's overall layout stability. A site that unexpectedly shifts layout as the page loads can lead to accidental user error and distraction.

Cumulative Layout Shift (CLS) occurs when elements have been shifted after initially being rendered by the DOM. For example, a button rendered to the screen after a text block, which then causes the block to shift downward would be considered a layout shift.

A combination of impact and distance is considered when calculating CLS.

According to Google, sites should strive to have a CLS score of **0.1 or less**.



Cumulative Layout Shift



Introduction to Lighthouse

Now that we are aware of what the "Core Web Vitals" are in a nutshell, how do we go about measuring them? Fortunately, Google has created a tool called "[Lighthouse](#)", which not only measures Core Web Vitals, but other important metrics as well:

[Lighthouse](#) is an open-source, automated tool for improving the quality of web pages. You can run it against any web page, public or requiring authentication. It has audits for performance, accessibility, progressive web apps, SEO and more.

You can run Lighthouse in Chrome DevTools, from the command line, or as a Node module. You give Lighthouse a URL to audit, it runs a series of audits against the page, and then it generates a report on how well the page did. From there, use the failing audits as indicators on how to improve the page. Each audit has a reference doc explaining why the audit is important, as well as how to fix it.

While Lighthouse is integrated directly into the Chrome DevTools (available in the "Lighthouse" panel), you may wish to access lighthouse in one of the other methods specified above. For more information, see:

- [Using the Node CLI](#)

- As a Node Module
- Using the online tool: [PageSpeed Insights](#)

NOTE: Lighthouse is also available as a [GitHub action](#), which allows us to use it in our CI pipeline. Using this, It is possible to set minimum acceptable scores for various performance metrics, which result in errors if not met. This can help to reduce performance-related bugs from being introduced into production.

See [the documentation for the CI Action](#) for more information.

Analyzing Page Load

While we can run Lighthouse on *any page*, it would be best to see what kind of results we would obtain from a simple Next.js app without any additional optimizations (discussed in: [Improving / Optimizing Performance](#)). To begin, download the [Example Code](#) and open the folder "app" in Visual Studio Code. Here you will find the familiar "my-app" folder containing a Next.js app.

As usual, before proceeding, you will need to change to the "my-app" folder and execute the commands:

```
npm install  
npm run dev
```

Since we will be using "Lighthouse" to measure the performance of this app, we should ideally be opening it in Google Chrome.

Once it's running, you will notice that it's a fairly simple app that shows a searchable accordion list of films in a collection. To obtain the data, It pulls the films from an "/api/movies" endpoint, specified in our "pages/api/movies.js" file.

Let's go ahead and see how well this app performs in Lighthouse:

1. Open a new "incognito" window in Google Chrome and navigate to the locally running app: <http://localhost:3000>
2. Open the "Developer Tools" and Switch to the "**Lighthouse**" tab
3. For this first run, check the "Performance", "Accessability", "Best Practices" and "SEO" Categories:

The screenshot shows the Lighthouse configuration interface. At the top, there is a button labeled "Generate a Lighthouse report" with a person icon and a blue button labeled "Analyze page load". Below these are three sections: Mode (with "Learn more" link), Device (with "Mobile" and "Desktop" options), and Categories (with checkboxes for Performance, Accessibility, Best practices, SEO, and Progressive Web App). Underneath the categories section is a "Plugins" section with a checkbox for Publisher Ads.

Mode [Learn more](#)

Navigation (Default)
 Timespan
 Snapshot

Device

Mobile
 Desktop

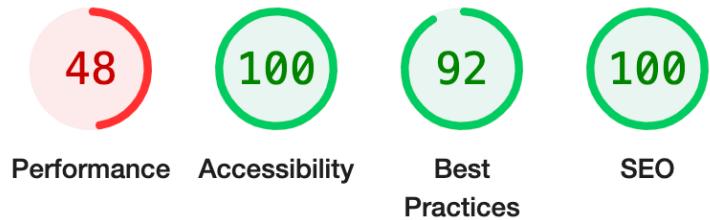
Categories

Performance
 Accessibility
 Best practices
 SEO
 Progressive Web App

Plugins

Publisher Ads

4. Click the **Analyze Page Load** button and wait for the audit to finish.



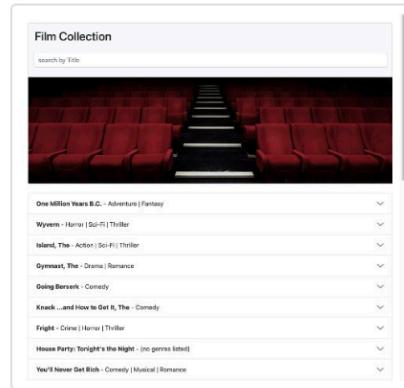
Performance

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

▲ 0–49

■ 50–89

● 90–100



It appears that things are looking pretty good, except for the performance section. If we look at the code for the app, it doesn't seem like there's all that much we can do to improve performance, at first glance. However, Next.js has some built-in techniques and components that we can use to optimize our initial page load and help to improve that result.

Important Note: The performance numbers *will improve* if we do a production build before starting the app (using `npm start`) and testing it with Lighthouse. However, the above report does shine some light on potential areas of improvement that we should explore before going to production.

Improving / Optimizing Performance

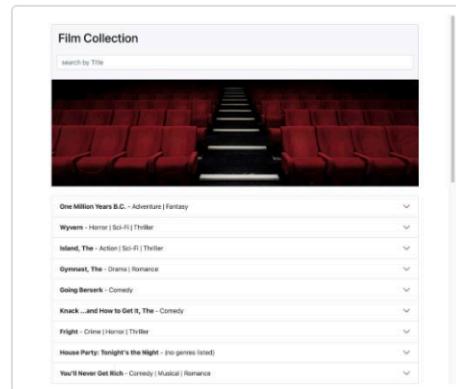
As we have discovered in the [Analyzing Performance](#) section, our web app for this topic may have some room for improvement when it comes to performance:



Performance

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

▲ 0–49 ■ 50–89 ● 90–100



METRICS

[Collapse view](#)

- First Contentful Paint

0.2 s

First Contentful Paint marks the time at which the first text or image is painted. [Learn more about the First Contentful Paint metric.](#)

- ▲ Total Blocking Time

1,200 ms

Sum of all time periods between FCP and Time to Interactive, when task length exceeded 50ms, expressed in milliseconds. [Learn more about the Total Blocking Time metric.](#)

- Speed Index

1.0 s

Speed Index shows how quickly the contents of a page are visibly populated. [Learn more about the Speed Index metric.](#)

- ▲ Largest Contentful Paint

4.6 s

Largest Contentful Paint marks the time at which the largest text or image is painted. [Learn more about the Largest Contentful Paint metric](#)

- Cumulative Layout Shift

0.006

Cumulative Layout Shift measures the movement of visible elements within the viewport. [Learn more about the Cumulative Layout Shift metric.](#)

Fortunately, Next.js has a few techniques that we can use to improve these numbers before doing a production build:

Using the <Image /> Component

The custom **Image component** included with Next.js is an alternative to the native `` element and offers a number of optimizations, including:

- **Improved Performance:** Always serve correctly sized image for each device, using modern image formats
- **Visual Stability:** Prevent Cumulative Layout Shift automatically
- **Faster Page Loads:** Images are only loaded when they enter the viewport, with optional blur-up placeholders
- **Asset Flexibility:** On-demand image resizing, even for images stored on remote servers

Let's update our app to use the Image component and look at some of its main features:

1. Add the correct "import" statement for the Image component

```
import Image from 'next/image';
```

2. Remove the current `` component and **replace it** with the following:

```
<Image  
  src="/theatre-bkrd.jpg"  
  alt="theatre background"  
  className={styles.headerImage}  
  sizes="100vw"  
  width={800}  
  height={232}  
  priority
```

Notice that we have provided a number of additional properties to the "Image" element, specifically:

```
sizes="100vw"
```

- This property provides information on how wide the image should be at different breakpoints, for more information on the "sizes" property, see: ["sizes" in the Image documentation](#)

```
width={800}  
height={232}
```

- These properties represent the *original* width in pixels.

```
priority
```

- When set, **priority** will mark the image as "priority" causing it to **preload**. Using "priority" in this case was recommended as this image was detected as the "Largest Contentful Paint (LCP)" element, as seen in the browser console:

"Image with src "/theatre-bkrd.jpg" was detected as the Largest Contentful Paint (LCP). Please add the "priority" property if this image is above the fold"

If you inspect the image in the browser console, you should see that it now has a number of extra properties, including

- **srcset**: This is the "source set", which identifies different urls for images to be served at different viewport widths (breakpoints). By default the following **device sizes** are used: 640, 750, 828, 1080, 1200, 1920, 2048,

3840.

You can see how Next.js has associated each device size with a url based on our original url, ie: the 640 width is set to serve: `"/_next/image?url=%2Ftheatre-bkrd.jpg&w=640&q=75"`, whereas the 750 width is set to serve: `"_next/image?url=%2Ftheatre-bkrd.jpg&w=750&q=75"`. If you try opening each of these images, you will see that Next.js has correctly scaled them to match the widths.

NOTE: Next.js will only scale images *down* in size (not up), therefore the image for the 2048 width: `"_next/image?url=%2Ftheatre-bkrd.jpg&w=2048&q=75"`, simply renders our original image (800px x 232px).

You will also notice that the source images have additional query parameter: "q". This represents the "quality" of the image, as Next.js will automatically optimize the original image to provide varying levels of quality. By default the quality setting is set to 75, however it can be changed using the `quality` property.

- **decoding:** Next.js sets the `decoding` value to "async", which is done to "reduce delay in presenting other content".

Remote Images

If you attempt to use a remote image with the Image component, ie:

```
<Image  
  src="https://www.senecapolytechnic.ca/content/dam/projects/  
seneca/campus-photos/magna-hall_tile.jpg"  
  className={styles.headerImage}  
  width={600}  
  height={386}
```

You will see the following error:

```
Error: Invalid src prop (https://www.senecapolytechnic.ca/content/dam/projects/seneca/campus-photos/magna-hall_tile.jpg) on `next/image`, hostname "www.senecapolytechnic.ca" is not configured under images in your `next.config.mjs`
See more info: https://nextjs.org/docs/messages/next-image-unconfigured-host
```

If we navigate to the link in "[more info](#)", we will see that the error occurred because the "src" value uses a hostname in the URL that isn't defined in the images.remotePatterns in next.config.mjs. This is done to ensure that only images from approved domains are able to use the Next.js image optimization API.

To solve this problem, open the "next.config.mjs" file, and update the **nextConfig** object to include an "images" property with "remotePatterns":

File: "next.config.mjs"

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // ...

  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'www.senecapolytechnic.ca',
        pathname: '**',
      },
    ],
  },
};

export default nextConfig;
```

Dynamically Importing Libraries

Next.js supports "Lazy Loading" for external libraries with "import" as well as [images](#). In larger apps, this can have an impact on metrics such as Largest Contentful Paint (LCP) and First Input Delay (FID) due to the smaller bundle size that is required on the first page load.

You will notice that our example includes the library "[lodash](#)" near the top of the index.js file as:

```
import _ from 'lodash';
```

However, the only lodash function that is used is within the "filterResults(data, searchText)" function:

```
function filterResults(data, searchText) {
  setFilteredResults(
    _.filter(data, (movie) =>
      movie.title.toLowerCase().includes(searchText.toLowerCase()))
  );
}
```

This function only gets invoked once the user starts typing in the "search" field. As such, we can potentially improve our performance by only loading the "lodash" library when it's required (ie: once the user starts typing).

NOTE: Technically, lodash is not required in this case, as the native "["filter\(\)](#)" method would also work here. However, this example highlights the syntax for dynamic imports, so we'll keep it in.

To dynamically import "_" from lodash we first **remove** it from the top of the

file:

```
// import _ from 'lodash';
```

and insert it in our "filterResults" function using "await" and "default", ie:

```
async function filterResults(data, searchText) {
  const _ = (await import('lodash')).default;
  setFilteredResults(
    _.filter(data, (movie) =>
      movie.title.toLowerCase().includes(searchText.toLowerCase())))
  );
}
```

Notice how we updated our filterResults to use "async" - this was required as we must use "await" to wait for "lodash" to finish importing before we can use it in the "setFilterResults" function.

You can confirm that this is working if you open the "network" tab in the Developer Tools and refresh the app. You should see "node_modules_lodash_lodash_js.js" appear in the list once you start typing:

| Name | Status | Type | Initiator | Size | Time |
|----------------------------------|--------|--------|-----------------------|--------|-------|
| node_modules_lodash_lodash_js.js | 200 | script | webpack.js?ts=1661... | 269 kB | 38 ms |

Dynamically Importing Components

Components can also be dynamically imported to reduce the initial bundle size and improve your performance.

In our application we use a number of components to render the UI, primarily from 'react-bootstrap': "Container", "Row", "Col", "Card" and "Accordion". In

addition to these, we also include a custom component: "**StarRating**" that is only visible once a user clicks on an accordion header to view the content (Rating and Plot Summary). Like our above "lodash" example, this is a perfect candidate for dynamic loading, as it is not visible / required until a user initiates the an action.

If we wish to dynamically load the "StarRating" component, we must update our code as follows:

1. As before, remove the initial import:

```
// import StarRating from '@/components/StarRating';
```

2. Import the "dynamic" function from 'next/js'

```
import dynamic from 'next/dynamic';
```

3. Import the "StarRating" component using the "dynamic" function (included above), making sure to set the **loading** property:

```
const StarRating = dynamic(() => import('@/components/StarRating'), {  
  loading: () => <>Loading...</>,  
});
```

4. Create a flag in the "state" to track when the accordion has been opened:

```
const [accordionOpened, setAccordionOpened] = useState(false);
```

5. Add an "**onSelect**" event to the <Accordion
className="mt-4">...</Accordion> element so that we can execute code

once the user opens the accordion.

```
<Accordion className="mt-4" onSelect={accordionSelected}>
  ...
</Accordion>
```

6. Write the "accordionSelected" function to set the "accordionOpened" flag to true (after 200 seconds, to ensure that the animation is completed)

```
function accordionSelected(eventKey, e) {
  setTimeout(() => {
    setAccordionOpened(true);
  }, 200); // allow for the accordion animation to complete
}
```

7. Ensure that the "StarRating" component is only shown once the "accordionOpened" flag has been set:

```
<strong>Rating:</strong> {accordionOpened && <StarRating
rating={movie.rating} />}
```

Once again, you can confirm that this is working if you open the "network" tab in the Developer Tools and refresh the app. You should see "components_StarRating_js.js" appear in the list once you open the first accordion section:

| Name | Status | Type | Initiator | Size | Time |
|-----------------------------|--------|--------|---------------------|--------|--------|
| components_StarRating_js.js | 200 | script | webpack.js?ts=16... | 1.4 MB | 184 ms |

Additionally, you should temporarily see the text "**Loading...**" in place of the star rating the first time this component is loaded.

Refactoring to use SSR

If at all possible, we would ideally like to **pre-render** as much of the page as we can. This can help reduce the time to first render and improve application performance. The home page for our practice "app" is a good candidate for SSR, since this is simply a static list of movies that isn't likely to change frequently. We have seen how this works when [discussing Handling Events & Rendering Data](#), so let's take what we have learned there and apply it to our Film Collection app as a final optimization before going to production:

1. Remove the import for useSWR as we will no longer need it:

```
//import useSWR from 'swr';
```

2. Remove the 'fetcher' definition used by SWR:

```
// const fetcher = (url) => fetch(url).then((res) => res.json());
```

3. Import the same "getMovieData()" function that your API uses to fetch the movie data. This will be used by "getStaticProps", since this function is also **executed on the server**.

```
import getMovieData from '@/lib/movieData';
```

4. Add a "getStaticProps" function above the "Home" component definition:

```
export function getStaticProps() {
  const data = getMovieData();
  return { props: { staticMovies: data } };
}
```

5. Update the "Home" component function definition to accept "props" (specifically, the "staticMovies" prop)

```
export default function Home({staticMovies})
```

6. Remove the "useSWR" function call (since we will no longer be needing it to obtain the data):

```
//const { data, error } = useSWR(`api/movies`, fetcher);
```

7. Update the "useState()" hook for "filteredResults" to use "staticMovies" as the default value:

```
const [filteredResults, setFilteredResults] =  
  useState(staticMovies);
```

8. Update our "useEffect()" hook to only watch for changes in "searchText" (since we no longer have "data" from SWR)

```
useEffect(() => {  
  if (searchText) filterResults(staticMovies, searchText);  
}, [searchText]);
```

To confirm this is working, once again refresh the page. You can either view the Page Source directly to see all of the movie details in HTML, or view the "localhost" entry in the "network tab" of the Developer Tools. If you "Preview" the results, you will see an unstyled version of the page with the details for each movie visible.

Final Lighthouse Run

As a final check before our production build, let's re-run Lighthouse to confirm that our optimizations have helped to improve the Core Web Vitals of our app:



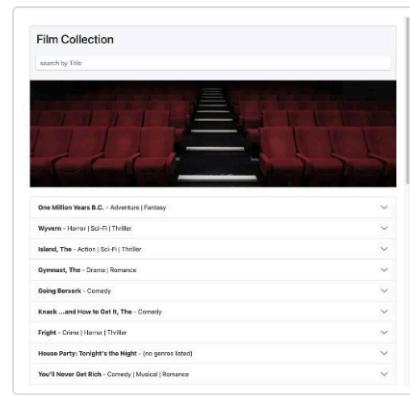
Performance

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

▲ 0–49

■ 50–89

● 90–100



METRICS

[Expand view](#)

● First Contentful Paint

0.2 s

▲ Largest Contentful Paint

2.6 s

■ Total Blocking Time

330 ms

● Cumulative Layout Shift

0

● Speed Index

0.5 s

Example Code

You may download the sample code for this topic here:

| Performance-Optimizations

Learning resources

Here you will find information about and links to learning resources that you will use in this course.

Developer tools

- [Visual Studio Code](#) (info, download)
 - [Getting started docs](#)
- Browsers (current versions of Chrome, Firefox, Safari, Opera, Edge)
- Browser dev tools
- HTTP inspector (e.g. [Thunder Client](#))
- [Plunker](#)
- [jsFiddle](#)
- [jsBin](#)
- Other dev tools will be introduced as we make progress through the weekly topics

You will need one or more devices. A College or personal desktop or laptop, and (likely) a personal mobile device (e.g. a smartphone).

Required textbook

None. This is a web programming course therefore, the best source for content is on the web!

Required online resources

There are several required online resources:

- The Mozilla Developer Network **MDN Web Docs** web site is vast, with thousands of documents. It is a trusted and authoritative source for web developer information. Of interest:
 - [HTML5 Guide](#), including [HTML Forms Guide](#), and the [HTML5 Element List](#)
 - [CSS Reference](#)
 - [JavaScript Reference](#)
 - [JavaScript Reference](#)
 - (Also suggested) [You Don't Know JavaScript](#)
 - [DOM Reference](#)

In October 2017, Microsoft, Google, and the W3C committed to making *MDN Web Docs* the single authoritative source for web developer documentation.

Read more about this in an [article by Ali Spivak](#).

Others:

- [Bootstrap \(version 3\) CSS documentation](#)
Links to other topics (e.g. Getting started, etc.) are on that page
- Official [Next.js Documentation](#)
- [Node Reference](#)

- [Node.js Documentation](#)
- (Suggested) [nodejitsu](#)
- [Express.js Documentation](#)
- [TypeScript Reference](#)

Oh, and you should must know (and love!) the series of RFCs that describe HTTP, [7230](#) through 7235. If you want a friendlier introduction to [HTTP](#), read its Wikipedia article.

Other resources

- [W3C Standards](#)
- Google [web developers](#) content

Front end frameworks

- [React](#)
- [Next.js](#)

Alternatives:

- [Angular](#) (which is NOT AngularJS, a legacy technology)
- [Vue](#)
- and many others...

State management

- [JSON data API - jsonplaceholder](#)

HTML and CSS

- [CSS Selectors](#) - MDN
- [CSS Attribute Selectors](#) - MDN

Visual Studio Code tips and info

Start VS Code from the command line

Make sure that you're in your project folder.

Then type this command: `code .`

NOTE: This assumes that your computer is configured to run this command. See the [Running VS Code on Mac](#) to configure that feature.

Useful keyboard shortcuts

Trigger IntelliSense: `Control + Space`

Toggle comments on/off: `Command + /`

Reformat document: `Shift + Option + Command + F`

Show/hide left-side bar: `Command + B`

Show/hide terminal: `Control + (back-tick)`

Markdown preview pane toggle on/off: `Command + K, V`

Useful Emmet snippets

For most elements, just begin typing the element name, without the angle bracket.

- . Declare class(es), e.g. `div.row`
- # Declare unique identifier, e.g. `table#customers`
- > Child, e.g. `div>p`
- * Multiplier, e.g. `ul>li*5`
- () Grouping, often used with multiplier
- + Sibling, e.g. `div>h3+p*3`
- {blahblah} Text content for an element, e.g. `h3{Hello, world!}`
- [] Custom attribute, e.g. `span[data-bind]`

See the [Emmet cheat sheet](#) for full coverage.