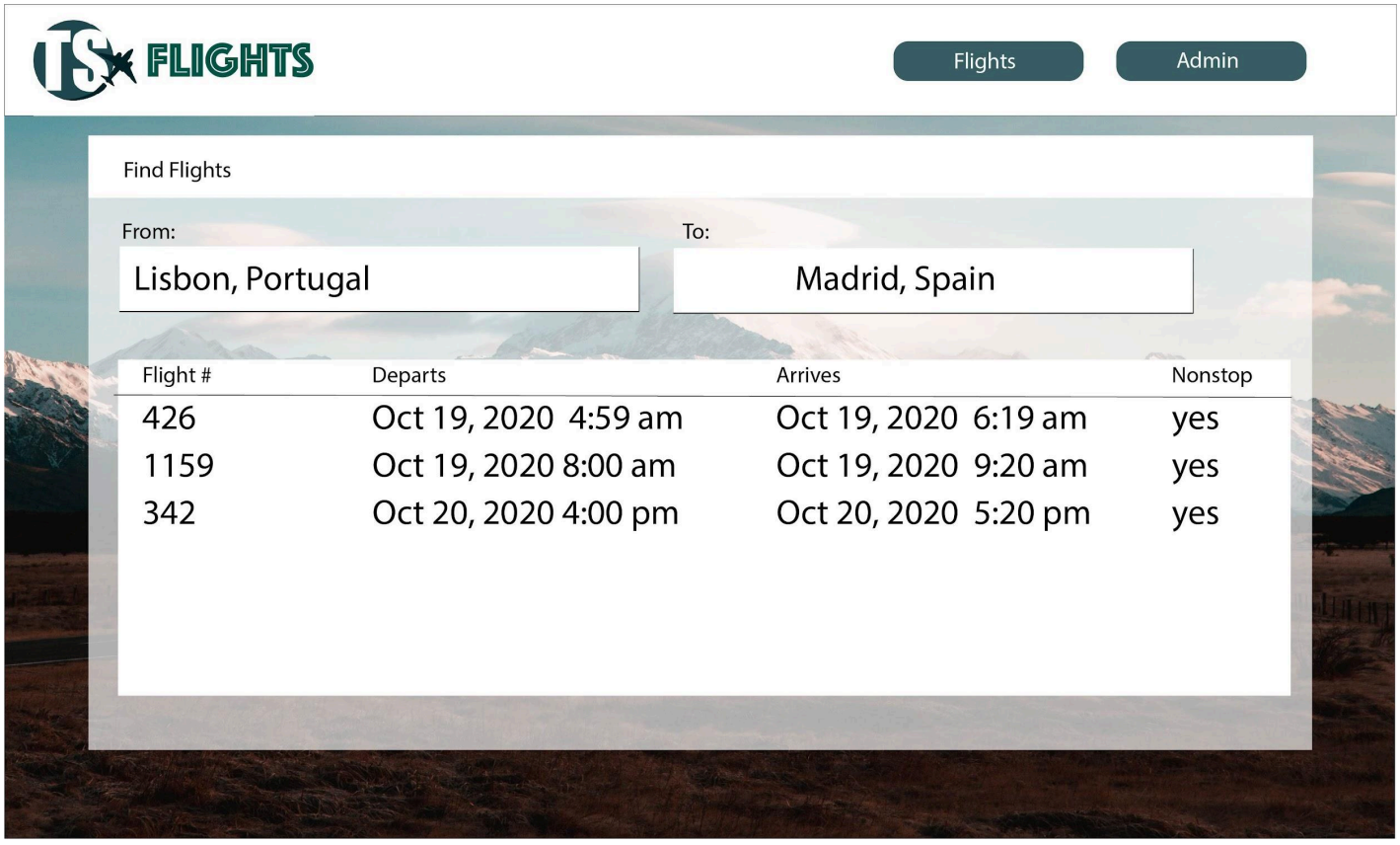
MODULE ONE :
THE TYPESCRIPT STACK

Lesson 1: INTRODUCTION TO THE TYPESCRIPT STACK

Module 1, Lesson one, PART ONE: **Overview**

TS Flights

Welcome to the FullStack Typescript Course ! In this course you will be building dynamic front end web and web applications that work together, and you'll be building everything using Typescript! In your course project, you will be creating a fictional website called "TS Flights" that has two basic interfaces: one that allows you to enter plane flights into an admin panel, and another that allows you to search for flights. We have built a sneak preview here that will allow you to see all of the code on both the front and back end, as well as a working model of the completed website so that you can see what you will be building.



>>>>>

Note: Don’t be intimidated by the code! We will teach you to fly with TypeScript one step at a time.

Module 1 / lesson 1 / quiz 1

In this course , you will be using Typescript to create

The back end of a web application only
The front end of a web application only
Both a front and a back end of a web application

CHECK

Module 1, Lesson One, Part Two: The concept of web "stacks"

Often you will hear programmers referring to various 'stacks' in web development. In the last several years we have seen the proliferation of many stacks, including the **MEAN** stack (Mongo, Express, Angular and Node), the **LAMP** stack (Linux, Apache, MySQL, and PHP), the **Python-Django** stack, **Ruby on Rails**, **.NET**, and so on.

It's important to understand that these stacks are constantly being invented, reinvented, iterated upon, and mixed with other stacks. The MEAN stack evolved into the "PEAN" or "NEAP" stack when some developers started switching back to using Postgresql instead of Mongo for the database resource related to that particular stack.

'Stack' is not a technical term. It's a loosely defined term that can be very fluid and sometimes even subjective. It is important, however, to be able to articulate which stacks you feel most comfortable with when you become a full stack developer.

This course will focus on teaching you the overall principles of full stack web development using a relatively new stack: a **TypeScript** based stack using **Angular** and **NEST JS**. These skills will be transferable to other stacks that you learn in the future.

The TypeScript Stack using Angular and NEST JS is one of the newer stacks in web development.

Module 1 / Lesson 1 / PART Two / Quiz

Which of the following statements is the best definition of a full stack app?

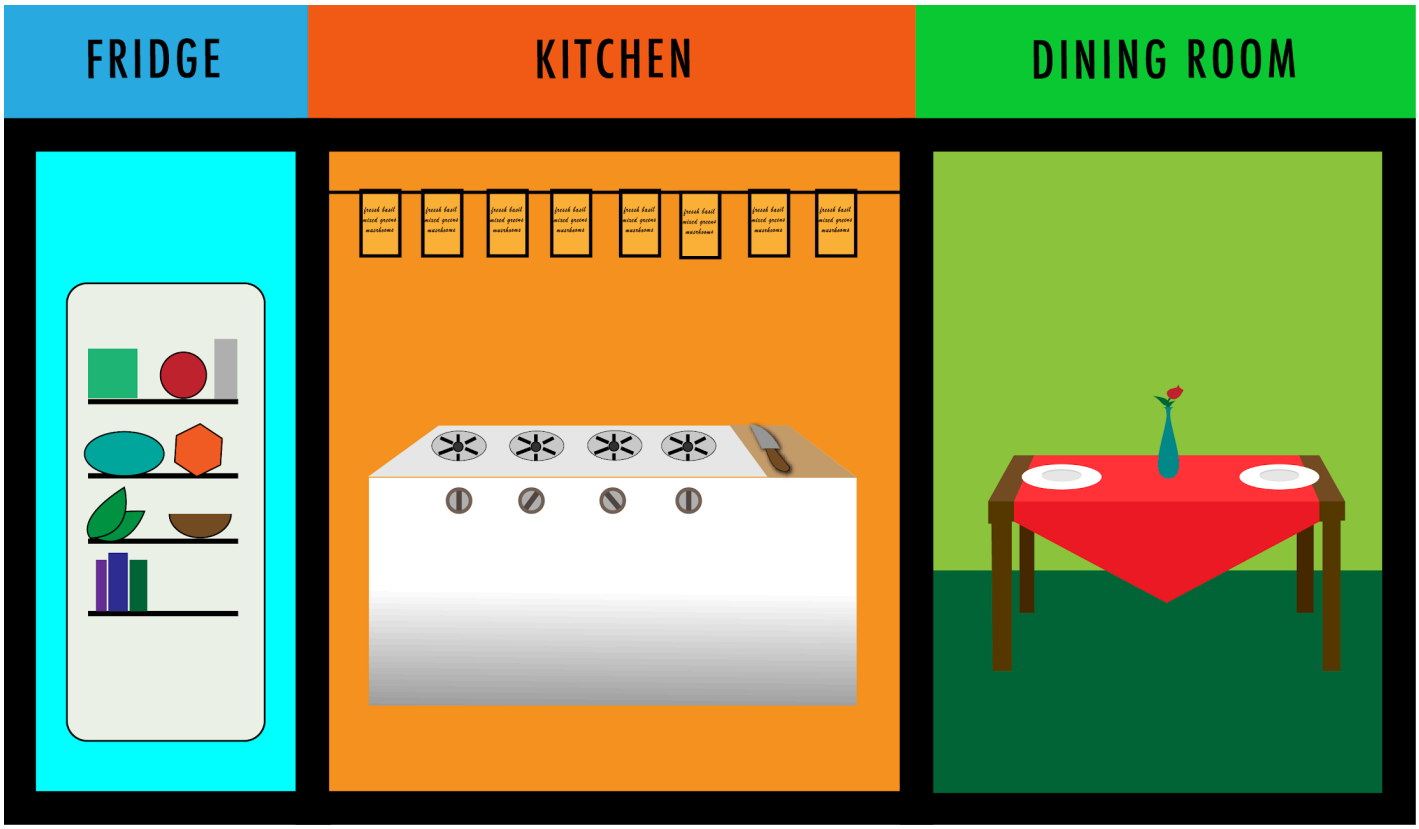
-- "Full stack" refers primarily to front end applications
-- "Full stack" refers primarily to back end applications
-- "Full stack" refers to front end and back end applications working together
-- "Full stack" refers to most applications built with TypeScript

CHECK

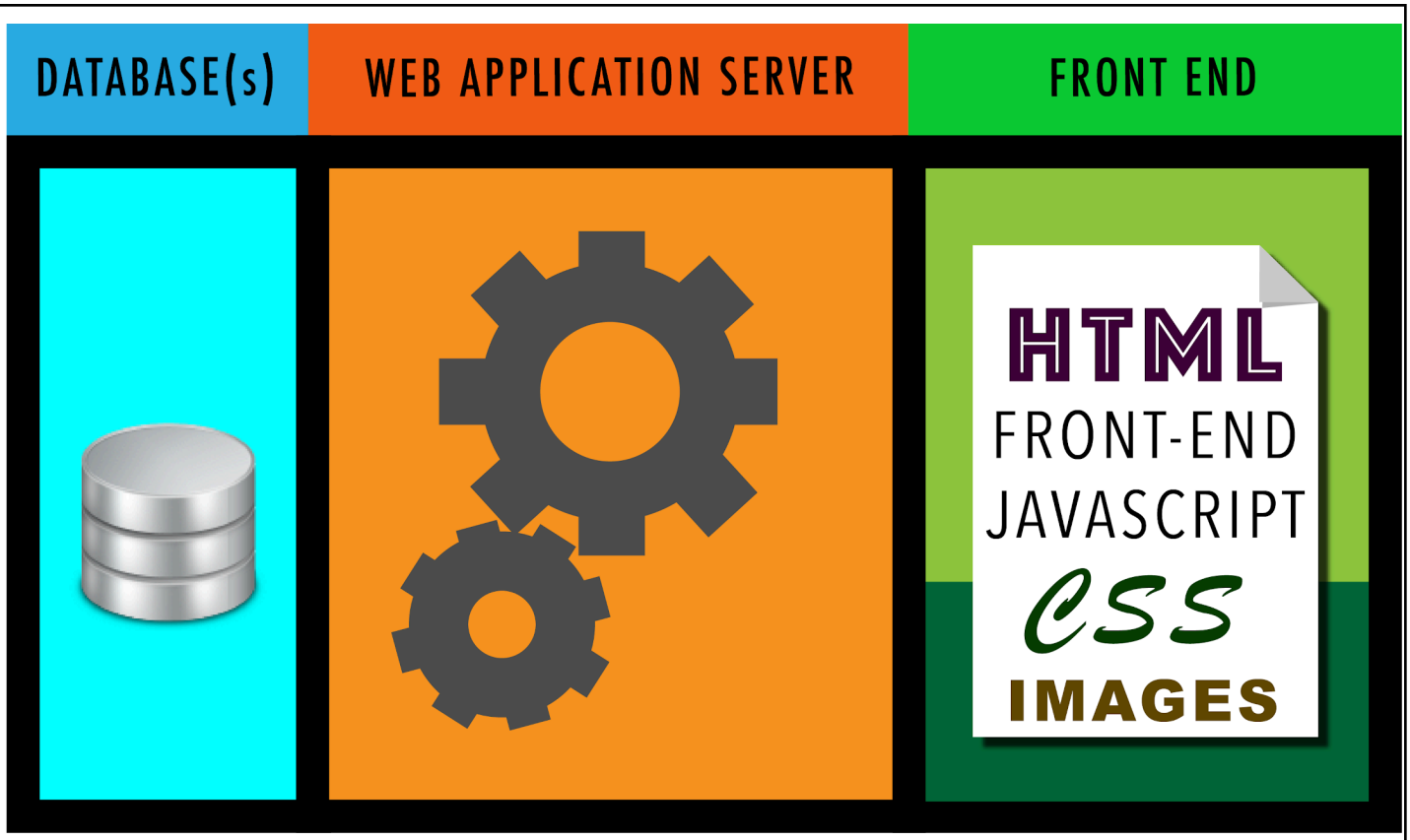
Module 1 / Lesson 2 : PARTS OF A FULL STACK APP

Module 1 / Lesson 2 / Part one: FRONT END vs BACK END

We will be using the analogy of a kitchen to illustrate how the front and the back end communicate with each other. Examine this image:



In full stack web development, the refrigerated room could be seen as the **DATABASE** or a series of data resources, the kitchen is the **WEB APPLICATION SERVER**, and the dining room is a bundle of tech that contains primarily **HTML**, **FRONT-END JAVASCRIPT**, **CSS**, and images.



For the sake of our analogy, the databases and the web application server are part of the **back end**. HTML, CSS, and front-end Javascript are a part of the **front end**.

Javascript and Typescript can be used in BOTH the front and the back end of the stack.

..

Module 1 / Lesson 2 / Part one / Quiz 1

Drag n' drop :

Our back end will contain a web application server and _____

A database
CSS
HTML
A load balancer
images

CHECK

Module 1 / Lesson 2 / Part two: The FRONT END

In web development, the "Front end" is what the user sees. The screen that you are looking at right now is the front end of a web application, whether it's on a phone, a tablet, or a computer.

The front end primarily consists of images, text, and widgets that a user interacts with.

If these elements require a database, behind the scenes logic, a login / authentication process, analytics tracking, communication with other users, and other things of this nature, then the front end needs a back end. However, not all front end applications require a back end.

When the world wide web first became popular in the mid 1990s, many websites did not have a back end. A simple HTML page with images, text, contact information and an email link did not necessarily require a back end!

If you create a simple HTML file on your computer right now, you will have created what we call a ***static*** web application. This static app would be a front end. There are resources online like Firebase where you can host your simple, static html pages almost instantly and for free.

The tech world has become very specialized. Many developers specialize in **only** front end or back end web development.

Module 1 / Lesson 2 / Part two: Quiz

A static web application

Has no back end
Is any application that includes HTML
Is an application without widgets
Is an application that does not use Javascript

CHECK

Module 1 / Lesson 2 / Part three: The BACK END

The back end is where both the web application server and the database live. There are scores of other elements that live within the back end space, like load balancers, service workers, email servers, and so on. In this course, however, we will focus primarily on the web application server and the database.

Our **TS Flights** back end will have these main responsibilities:

- Handle requests from the front end for information
- Creating, Reading, Updating, and Destroying plane flights in the system
- Send info back to the front end in JSON format (Javascript Object Notation)

The back end's primary responsibility is the processing of data, where the front end's main responsibility is the displaying of data and handling user interaction. Front ends process data as well, but heavy data manipulation is generally within the domain of the back end.

We use the acronym **CRUD** to describe the **C**reate, **R**ead, **U**pdate, and **D**elete process with data.

Module 1 / Lesson 2 / Part three: Quiz

TYPE IN

The primary responsibility of the _____ end of a web application is usually processing and storing data

Answer:
back

CHECK

Module 1 / Lesson 2 / Part four: Javascript and TypeScript Throughout the Stack

When JavaScript became a popular full stack tool in 2009 after Node JS was released, it opened people's eyes to the possibility of using Javascript for the back end as well as the front end. Prior to the release of Node, Javascript was seen as primarily a front-end only tool used for creating widgets, user activity on a page, DOM (web page structure) manipulation, and other front-end only tasks. After Node was released, Javascript was seen as a great tool for creating back ends and front ends alike.

Typescript was released in 2012 by Microsoft. It was created because many Javascript developers have expressed the desire for more structure and very clear data types, especially for use in large scale applications. Typescript has some features like static variable typing and interfaces that are present in well-respected legacy languages like Java and C#.

TypeScript is a superset of Javascript. What this means is that TypeScript is *transpiled* into JavaScript. It's literally translated into Javascript code when it is compiled and before it is run. TYPESCRIPT IS TRANSLATED INTO JAVASCRIPT. That's an important concept to remember!

It therefore made logical sense, upon the release of NestJS, that TypeScript could be used throughout the entire web application stack just like Javascript had been.

One of the creators of Typescript was Microsoft employee Anders Hejlsberg, the creator of C#.

Module 1 / Lesson 2 / Part four Quiz

Node JS was groundbreaking for Javascript because

It prompted developers to see Javascript as a full stack language
It prompted developers to see TypeScript as a full stack language
It prompted developers to focus more on the back end
It prompted developers to be happier

CHECK

Module 1 / Lesson 3: HTTP

Module 1 / Lesson 3 / Part 1: XHR

The next concept to understand is the request / response nature of full stack applications.

In a restaurant, the food request comes from the dining room, or the **front end**.

The customer makes a request to the kitchen. In web applications, this typically comes in the form of an **XML/HTTP Request** or an "**XHR**".

Let's break down that term: **"XML/HTTP" request**. Although the term itself is already showing its age (we don't use XML as much to send data over the web, JSON is used instead in most instances now), the second part of that term, **HTTP**, is critically important.

HTTP stands for "**HYPERTEXT TRANSFER PROTOCOL**", and is the foundation of all communication on the

web. Think of HTTP as the *way* in which all web based technologies must communicate with each other. It's massive and the very foundation of internet communication.

You can analyze XHR for any website in the browser's network tab in Chrome developer tools.

Module 1 / Lesson 3 / Part 1 Quiz

Drag n' Drop

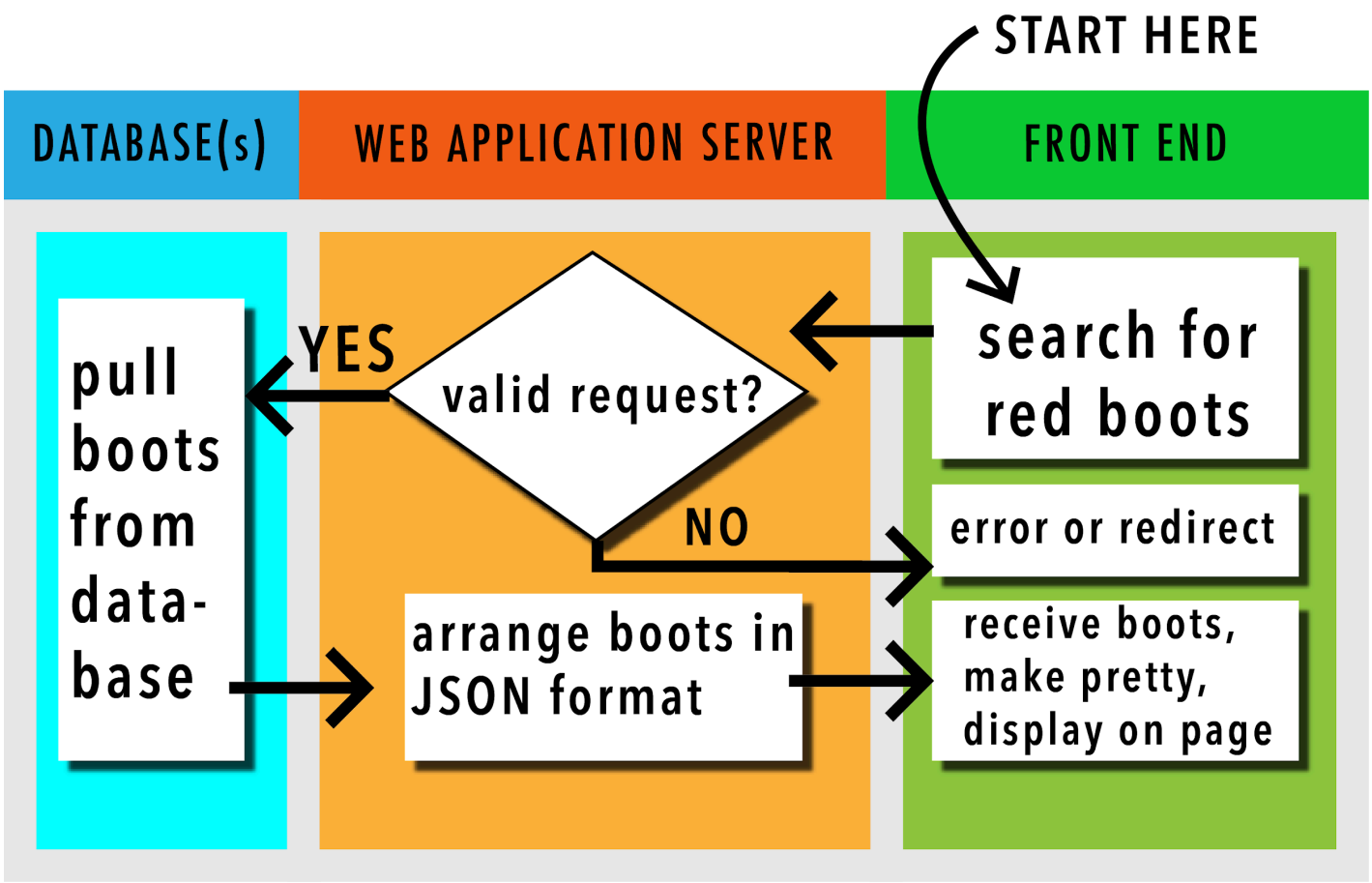
The front end makes a request to the back end by way of an _____ request

XML/HTTP
HTML
XMP
XHTML

CHECK

Module 1 / Lesson 3 / Part 2: THE REQUEST / RESPONSE CYCLE

In our analogy, after the customer makes a request, it is received by the kitchen, or the web application server. The web application server then analyzes the request and decides whether or not it can fulfill that order. If the order is able to be fulfilled, then the kitchen staff will often reach into the database, (the refrigerator) , pull the data (the food), prepare the data, and then send the data back to the customer.



This represents a very simple request / response cycle. A request is made to the web application server, analyzed, and either rejected or accepted.

Module 1 / Lesson 3 / Part 2: Quiz

Drag n drop

In a web application, an _____ sent by the front end can be rejected by the web application server

XHR
XML / CSS request
HTML request

Application server

CHECK

Module 1 / Lesson 3 / Part 3: HTTP ERROR CODES

Yes, XML/HTTP requests can be rejected by the server!

If rejected, the web application server sends an HTTP response back to the client (the front end). Possible HTTP *error* status codes are 404 (not found), 403 (forbidden), 401 (not authorized), 500 (internal server error), and 503 (service unavailable). There are many more HTTP status codes, and they are generally categorized by the first digit in the status code.

"**Server**" and "**client**" are very common terms in computing. The "server" in full stack development typically refers to web application server, and the "client" refers to any device that is interacting with the server on the front end. Common clients include laptop and desktop computers, mobile phones, GPS systems in cars, voice driven devices like Amazon Alexa, Siri, and Google Home, video game consoles, smart watches, and anything else that connects to a back end.

Module 1 / Lesson 3 / Part 3 Quiz

An HTTP error code will be sent

From the client to the server in the case of an error
From the server to the client in the case of an error
From XML to HTTP
From HTTP to XML

CHECK

Module 1 / Lesson 3 / Part 4: HTTP SUCCESS CODES

All of the 200 level codes are generally positive status codes : 200 (ok) , 201 (created), 202 (accepted), etc. By contrast, 400 level status codes generally indicate an issue with the request itself, and 500 level status codes generally indicate an error on the back end.

If the initial request has been accepted by the web application server, and there is a need to retrieve data related to the request, the web application server then queries a database or databases to get the information needed to send back to the user.

The database sends the result of the query back to the web application server, and then the web application server usually formats the data into a format to be sent back to the front end. **JSON** (JavaScript Object Notation) is a very common format currently used to send data to clients (the front end).

Request / response cycles can be incredibly complex, but this very simple flow is the beginning of understanding the volleys between the front and the back end.

Module 1 / Lesson 3 / Part 4: Quiz

Type-in:

HTTP Success codes always begin with a _____

Answer:
2

CHECK

Module 1 / Lesson 4 : SUMMARIZING OUR STACK

Module 1 / Lesson 4 / Part One: The Angular / NEST flow

In summary, we will be using **TypeScript**, a superset of Javascript, to be building a full stack app called **TS Flights** that allows users to search for plane flights. The app will also have a page that allows you as an admin to enter new flights to show up in future searches.

On the front end, we will be using **Angular**. On the back end, we will be using **NEST JS**.

For something like searching for a flight or adding a new flight into the database, our Angular front end app will make requests to our NEST back end using **XHR**. If the request is rejected, it will return a negative **HTTP** status code back to Angular. If the request is accepted, it will then proceed to do the work that it was sent to do...which may include getting all of the flights from New York to Paris or something of that nature.

If the request was accepted and includes a database call, NEST will make a database query to our database, get the result, and if that database query is a success, return the **JSON** response to our Angular app with a 200 level status code. Our Angular app will then use HTML and CSS to make the response pretty so that our users can see all of the pretty flights to Paris on their phone or computer.

You should be able to see with this flow that the user initiates the process on the client, the request goes back to the server, does its work, and then ends up returning back to the client for the end of the process.

Request / response cycles can be incredibly complex, but this very simple flow is the beginning of understanding the volleys between the front and the back end.

Module 1 / Lesson 4 / Part One: The Angular : Quiz

The XML/HTTP request / response cycle between NEST JS and Angular

Begins with Angular and ends with Angular
Begins with NEST and ends with Angular
Begins with NEST and ends with NEST
Begins with Angular and ends with NEST

CHECK

MODULE ONE EXAM

1.

Which of the following is typically used ONLY on the front end ?

Javascript
TypeScript
CSS
JSON

2.

Drag n drop

_____is the way through which web technologies communicate with each other

HTTP
XHR
XML
HTML

3.

TYPE- IN
What is the role of JSON in a typical request / response cycle ?

Servers typically send data (like an array of users and their info) to the client in the form of _____

Answer:
JSON

4.

An XML/HTTP request (XHR) is rejected by the web application server because a client was not logged in. What would be a typical response from the server back to the client?

A database query asking for the correct username and password
Another XHR request asking the client to log in
An HTTP error code
An HTML error code

5.

ORDERING

Scenario:

A user clicks on a button in the front end to get a list of movie showtimes in his area based on her zip code.

Order the responses below, from top to bottom, in terms of the order of the most likely request / response flow. The item at the top should be the first thing that happens and the item at the bottom is the last thing that happens in the flow.

-- User clicks on button in the front end
-- server accepts request as valid
-- server goes into the database to get showtimes
-- server returns the list of movies as JSON
-- Client translates JSON into pretty display

MODULE TWO: THE ANGULAR FRAMEWORK

MODULE 2 / LESSON ONE: Introduction to Angular and Typescript

Module 2 / LESSON ONE / Part 1: INTRODUCTION TO ANGULAR

Angular is a **front end javascript framework.** A **'framework'** is a set of connected libraries designed around a specific programming language. Frameworks provide conventions that support doing things a certain way. A front end framework typically has specific conventions around how HTML is handled, how forms are created, how interactive elements on a web page are handled, how variables are interpolated inside of HTML views, etc.

Angular JS or "**Angular 1.x**" was the first iteration of the Angular framework. With the release of **Angular 2**, Angular was completely redesigned using TypeScript and component based architecture. Although component based architecture was available in the later releases of Angular 1.x, components are now the very foundation of Angular 2 and all subsequent releases after version 2. After 2, **Angular 4** was released and versions counted upwards from there (there was no version 3).

Angular 8 was released in 2019 and that is the focus of this course.

It is important to know that when developers say "**Angular**", then mean **Angular** versions **2 and higher.** "Angular JS" refers to version 1 of Angular only.

Module 2 / lesson 1/ part 1 quiz

Reorder

Order the Angular releases from top to bottom in the order of their release, starting with the oldest.

Angular JS
Angular 2
Angular 4
Angular 5
Angular 8

CHECK

Strings in TypeScript

Angular JS used Javascript as its primary language. Angular began using TypeScript starting in Angular 2. Since then, TypeScript has become the primary language of Angular.

Let's look into some basics of TypeScript. Variables in TypeScript can be **statically typed**. Statically typed variables must always retain the data type that they start out with. Once a variable is statically typed as a string, for example, that variable cannot hold any other data type.

Typescript uses Javascript's **var**, **let**, and **const** keywords to initialize variables.
It also uses a colon after the variable name to designate the type.

Javascript:
let name = 'Fred'.

TypeScript:
let name: string = 'Fred'.

Based on this convention, we could initialize a shoeColor string in TypeScript as such:

```
let shoeColor: string = 'blue' ;
console.log(shoeColor);
```

Try It Yourself

It is possible to change variable values over time:

```
shoeColor = 'red' ;
console.log(shoeColor);
// Now "shoeColor" has a value of red. Note that once a variable has been typed, you don't have to declare a type again when the value of the variable changes.
```

Try It Yourself

Note: you can test this out in our Playground using the “Try It Yourself” buttons above.

> Typescript does not require you to statically type variables. Plain Javascript syntax WILL work in Typescript. However, by convention, most programmers statically type variables in Typescript because it helps to provide structure and prevent programmer errors.

Module 2 / LESSON ONE / Part 2: Strings quiz

Fill in the blank

Declare a string typed variable named user with a value of Samson

let _____ : _____ = 'Samson'.

Answer:

user , string

Module 2 / LESSON ONE / Part 3: Numbers

Numbers in TypeScript

Another TypeScript data type is the **number** data type.

We can initialize a seatCount number variable in TypeScript like this:

```
let seatCount: number = 42;
```

Try It Yourself

TypeScript does not designate between integers and floating point or decimal numbers like other languages when it comes to typing. Initializing a decimal number is just like initializing an integer:

```
let registrationPercentage: number = 92.87;
```

Try It Yourself

Note that registrationPercentage will always have to be of a number type. If you were to try to reassign registrationPercentage to a string value in another line of that program, the program would not compile, and your text editor would most likely throw an error.

```
registrationPercentage = 'Ninety Two' ;

// WILL CAUSE AN ERROR because registrationPercentage is statically typed as a number.
```

Try It Yourself

Typescript includes all methods that you are used to using in Javascript. For example, string methods like parseInt() and parseFloat() that return numbers will also work in TypeScript.

Module 2 / LESSON ONE / Part 3: Numbers quiz

Type in

Complete the TypeScript expression below to declare two number variables named a and b that are later added together to make a third number variable that is the sum of the two variables a and b.

```
const  a : _____  = 6 ;
const ____  : number  = 4;

const result : ______ = a+b ;
```

Answers:
number, b, number

Module 2 / LESSON ONE / Part 4: Booleans

Booleans in TypeScript

Boolean values are either true or false. In TypeScript, you can declare and statically type a variable without initializing it with a value. For example:

```
let isRegistered: boolean;
```

If you try to use isRegistered at this point in the program, however, you will get an error. Some programmers declare variables and type them without initializing them because the value of the variable is completely unknown at the start of the program. Later, you can assign it.

```
 isRegistered = false;
console.log(isRegistered);
// outputs  false
```

Try It Yourself

> Booleans always return **true** or **false** values. These values are not strings, they are actually their own data type.

Module 2 / LESSON ONE / Part 4: Booleans quiz

Reorder

Reorder the following TypeScript lines to declare a boolean, assign it a value, and then output its value.

```
let isOpen: boolean;
isOpen = false;
console.log(isOpen);
```

Module 2/ Lesson One/ Part 5: Enums

Enums in TypeScript

Enum values in Typescript are collections of constants. Think of an Enum as a type of fixed array of things that you use a reference, like days of the week, states in the USA, etc. The number of elements never changes during the run of the program and the order of elements doesn't change either.

Let's create an enum called **spiceLevel**:

```
enum spiceLevel {
  NONE = "no spice",
```

```
  LOW = "barely spicy",
  MEDIUM = "medium spicy",
  HIGH = 'hot'
}

console.log(spiceLevel.MEDIUM);
// outputs "medium spicy"
```

Try It Yourself

Modern Integrated development environments (IDEs) like Visual Studio Code help you complete enum values and other coding constructs by using autocomplete. Enums help to ensure that commonly used constants in your program are all consistent. This helps to avoid typos and general errors.

Module 2 / LESSON ONE / Part 5: Enums quiz

Multiple choice

What will be the output of this program?

```
enum colors {
 RED = "#FF0000",
 GREEN = "#00FF00",
 BLUE = "0000FF",
}

let userChoice = "#FF0000";
let isRed: boolean = false;

if (userChoice == colors.RED) {
isRed = true;
}
```

```
console.log(isRed);
-----------------------------------------
```

The program will throw an error
true
false
undefined

The 'any' type in TypeScript

Finally, we will examine the **'any'** data type.
The 'any' type is essentially a wild card. It can hold anything.

We can declare an **'any'** ~~number~~ data type in TypeScript like this:

```
let userData: any;
```

userData will now be a container that can hold any data type. Once userData is initialized, however, it will infer the type from the value given to it.

```
let userData: any;

userData = 22;
console.log(userData + 2);
// logs 24

userData = "free";
console.log(userData + 'man');
```

```
// logs "freeman";
```

Try It Yourself

Note that with an 'any' data type, even though the type is inferred upon assignment, you can re-assign the variable to another data type and it still works.

Why in the world would you ever need an 'any' type ? Well, in certain situations, like getting user data from a server, you might be accepting variable data that could conceivably be of any data type.

> 'any' types not only accommodate data from any source, but they also signal to other developers the fact that this particular variable is of an unknown type and that the data should be handled with care.

Module 2 / LESSON ONE / Part 6: 'any' data type quiz

Multiple choice

What will be the output of this program?

```
let receivedData: any;

receivedData = '5';

if (typeof(receivedData) === 'number') {
 receivedData += 5;
} else {
 receivedData += 10;
}

console.log(receivedData);
```

```
5
10
15
510
```


Custom Types

If you thought for a minute that TypeScript was limited to number, string, boolean, and other familiar types, then we have a pleasant surprise for you! In TypeScript, you can create your OWN types and use them the same way that you would primitive types like numbers and strings.

One way to do this is by creating an **interface**. An interface in TypeScript is a data structure that defines the shape of data. Let's see this in action:

```
interface Order {
  customerName: string,
  itemNumbers: number[],
  isComplete: boolean
}
```

Try It Yourself

The **interface** keyword is used to initialize an interface, which shows us the SHAPE of the data that's coming. Think of an interface like a factory mold. This interface is used to stamp out Order types for a store. Now let's actually use the Order interface to type a variable:

```
let order1: Order;

order1 = {
  customerName: "Abiye",
```

```
  itemNumbers: [123,44,232],
  isComplete: false
}
```

Try It Yourself

Let's analyze the order1 variable. It is of an "Order" type, so it must have 3 fields: the first field is a string, the second field is an array of integers, and the third field is a boolean. It MUST have each of those fields in order to fulfill the **contract** of the interface. Try omitting one of the fields in order1 (for example, remove the customerName). You will receive an error because the contract has not been fulfilled.

> An interface **contract** is simply the list of fields in that interface that any variable needs if it wants to use that type. All of the normal fields within an interface must be implemented in any variable that uses that type.

Module 2 / LESSON TWO / Part 1: interfaces quiz

Type in

Complete this code so that the Student interface's contract is fulfilled and the code will compile.

```
interface Student {
  id: number;
  name: string;
}

let tewodros: Student;
```

```
tewodros = {
  _____ : 42,
  _____ : "Tewodros"
}
```

Answer:
id, name

Module 2 / LESSON Two / Part 2: optionals

Optional fields in interfaces

We can use **optional** fields in an interface in TypeScript. Optional fields are not part of the strict interface contract. You can omit them when creating an instance of that interface.

Example:

```
interface Order {
  customerName: string;
  itemNumbers: number[];
  isComplete?: boolean;
}
```

Try It Yourself

Notice the **question mark** after isComplete. **isComplete?** means that we can omit that value and the code will still compile. This is useful for fields within an interface that are not mandatory.

```
let order1: Order;

order1 = {
  customerName: "Abiye",
  itemNumbers: [123,44,232],
```

```
}
```

Try It Yourself

order1 only has 2 fields now and it still compiles because isComplete is an optional field.

> Optional fields are helpful when getting data from a database or an api call where some fields may be missing or incomplete.

Module 2 / LESSON TWO / Part 2: optional fields quiz

Type in

Complete this code so that the Student interface's contract is fulfilled and the code will compile.

```
interface Student {
  id: number;
  name: string;
  favoriteFood__: string;
  isAlumni__ : boolean;
}

let ana: Student;

ana = {
  id : 25,
  name : "ana",
}
```

Answers:

?, ?
(there are two fields, both of them should accept one character and the correct answer for each is a question mark)

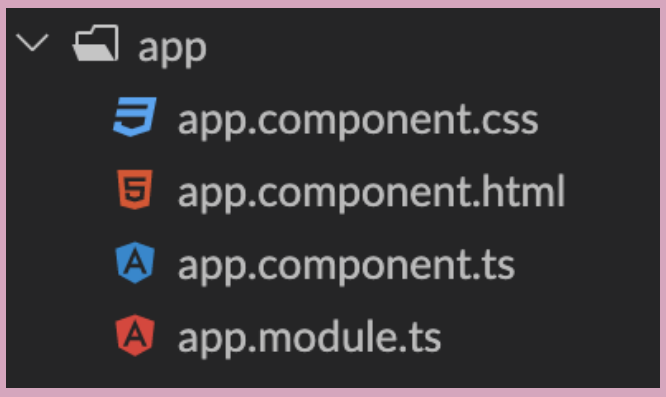
Module 2 / LESSON THREE / Using TypeScript within Angular

Module 2 / LESSON THREE / Part 1: Angular files

Angular files

Up until this point, we have been creating simple Typescript files that run in any environment that runs TypeScript. From this point on, however, we will be writing ALL of our TypeScript in Angular !

In Angular, each component has at least 4 files that work in harmony together. Those files look something like this within a folder:



The only two files that we will concern ourselves with here are the **app.component.ts file**, which we will call ***the component***, and the **app.component.html** file which we will call ***the view.***

The **component** is where variables are declared and modified.

The **view** is an HTML file lives that receives variables from the component and displays them.

We have built a 'hello world' example to show you how these files communicate with each other.

COMPONENT

```
name = "Star student";
```

VIEW

```
Hello, {{name}} !
```

RESULT:

Hello, Star student !

We have set up this code on StackBlitz, which is an online Angular editor that allows you to run the code on your phone! In StackBlitz, the files are on the far left, the code is in the middle, and a small browser is on the right. You can view and or edit the code and see the result within seconds.

View on StackBlitz

https://stackblitz.com/edit/angular-starter-example?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-starter-example

> The double curly braces around {{ name }} demonstrate what we call **interpolation**.
> ***Interpolation*** is the insertion of variable content from the component into the view.

Module 2 / LESSON THREE / Part 1: Angular files quiz

Type-in

Type in an example of variable interpolation to make the title variable display in the view.

component

```
title = "Star Wars";
```

view

```
<div>
   The title of this movie is  _____
</div>
```

---

answer

{{title}}

Module 2 / LESSON THREE / Part 2: Parts of a component

In our diagram below, we have 4 parts of a component that we want to cover in this part of our lesson, represented by arrows of four different colors. The red section is the "imports section". The orange arrow points to code automatically generated by Angular, we don't have to touch this part unless we want to change it. The blue arrow points to where **class variables** are declared. Finally, the green arrow points to where **method variables** are declared. Class variables belong to the whole class, but method variables only apply within each method (or 'function') that they live in.

View on StackBlitz:
https://stackblitz.com/edit/angular-class-vs-method-variables-intro?file=src/app/app.component.ts

View on GitHub:
https://github.com/Nmuta/angular-class-vs-method-variables-intro

> Pro tip: In object oriented programming, a "method" refers to a function that is part of a class. Components are classes, so we refer to "functions" inside of them as "methods".

Module 2 / LESSON THREE / Part 2: new part : Quiz

Multiple choice

Method variables apply to

The method that they are declared in
The entire file
The entire class
The entire view (html) file

Module 2 / LESSON THREE / Part 3: Class vs Method variables

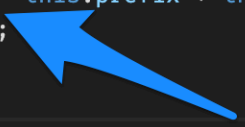
Class vs method variables

~~Let's look at an Angular component that has a method in it.~~
Let's put some code into our speak() method to flesh out what instance variables vs method variables look like in practice:

app.component.ts × app.component.html

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})

export class AppComponent  {

  prefix = "I am a ";
  name = "star student";

  speak() {
    const sentence = this.prefix + this.name;
    return sentence;
  }
}

```

class variables



method variable

The variables **prefix** and **name** are class variables. They belong to the class.
Inside of the speak( ) method, if we want to invoke *class* members, we have to use *this* keyword.

Also, note that inside of the speak( ) method, we are using the ***const*** keyword to declare the sentence variable. INSIDE OF METHODS, you must use a keyword like var, let or const to declare variables. We chose **const** because const keywords declare a value that is a **constant**. It will not change.

This is something you will get used to as you get accustomed to the flow of Angular.

View on StackBlitz

https://stackblitzhttps://github.com/Nmuta/angular-class-vs-method-variables.com/edit/angular-class-vs-method-variables?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-class-vs-method-variables

We did not use static typing for the prefix, name, and sentence variables because the type was inferred when we initialized these variables with strings.

Module 2 / LESSON THREE / Part 3: class vs method variables: Quiz

Multiple choice

Which of the following is true about class and method variables in components?
Check all that apply

if you refer to a class variable inside of a method, you must use 'this'
if you refer to a method variable inside of a class, you must use 'this'
class variables must be initialized with a keyword like var, let, or const
method variables must be initialized with a keyword like var, let, or const

Module 2 / LESSON FOUR / Arrays And Iteration

Module 2 / LESSON FOUR / Part 1: Iterating through string arrays

Iterating through string arrays

With Angular, we can use the *ngFor command in the HTML file (the view) to loop through arrays declared in the component.

Given this array in the Angular component:

```
colors:string[] = ['red', 'blue', 'green', 'purple'];
```

We can loop through the colors in the accompanying HTML file in Angular like this

```
<div *ngFor='let color of colors'>
  {{color}}
</div>
```

The result in your browser will look like this :

red
blue
green
purple

where each of the colors, red, blue, green, etc. are wrapped within a div. ***ngFor** essentially creates a loop where the html tag that it's declared within loops for as many times as there are elements in the array that it's invoking.

View on StackBlitz
https://stackblitz.com/edit/angular-ngfor-looping?file=src/app/app.component.ts

View on GitHub
https://github.com/Nmuta/angular-ngfor-looping

*ngFor is a **structural directive**. Structural directives modify HTML according to variable data that they are associated with.

Module 2 / LESSON FOUR / Part 1: Iterating through string arrays quiz

Drag n' Drop

Complete this Angular HTML code so that it will loop through a string array of fruits coming from the associated Angular component.

<div _____ = ' let _____ of fruits'>
  {{ fruit }}

Answers: (correct answers are bolded first, in order. The rest of the answers are decoys.

***ngFor** , **fruit** , fruits, string, ngFor

Iterating with indices

Sometimes it's helpful, when iterating through an array, to have access to an index within each iteration of the loop.

Here's how we do that in Angular:

```
<div *ngFor='let fruit of fruits; let i=index'>
  Fruit {{ i }} is {{ fruit }}
</div>
```

The result in your browser will look like this :

Fruit 0 is apple
Fruit 1 is orange
Fruit 2 is pear
Fruit 3 is peach

note that our 'iterator' variable is named **i**. This variable can be named anything; we just used the letter i out of convention. The 'index' keyword cannot be changed. Angular knows that the 'index' is the index of each iteration as we loop through the array.

We have created a StackBlitz account that you can use to play with the live code in Angular right from your computer or phone without needing to clone a repo.

View on StackBlitz

(StackBlitz link: https://stackblitz.com/edit/angular-indices?file=src/app/app.component.ts)

View on Github

(GitHub link: https://github.com/Nmuta/angular-indices)

indices during iteration are helpful when iterating through lists of things, like shopping cart items, that are associated with database ids from the back end. We will get to that later in the course! Hang on !

Module 2 / LESSON FOUR / Part 2: Iterating with indices quiz

Type in

Complete the Angular code below in the html file to use indices in the *ngFor loop

```
<div *ngFor='let student of students;  let ____ = ___' >
  Student {{ j }} , whose name is {{ student }}, is in class.
</div>
```

answers:

j , index

Module 2 / LESSON THREE / Part 3: Arrays of custom types

Arrays of custom types

Module 2 / LESSON FOUR / Part 3: Iterating through custom types

Iterating through custom types

We will now connect what you learned about interfaces with our ***ngFor** looping mechanism. Let's make an interface for a Car and put it in its own file called car.ts (.ts is the extension for TypeScript files)

```
export interface Car {
make: string;
model: string;
miles: number;
}
```

Now that we have a Car interface, we can import that interface into our component as such

```
import { Car } from './car';
```

Now we can create 3 entities of the Car type:

```
subaru: Car = {make: 'Subaru', model: 'Outback', miles: 58232};
honda: Car = {make: 'Honda', model: 'Accord', miles: 39393};
bmw: Car = {make: 'BMW', model: 'X3', miles: 4400};

cars:Car[ ] = [this.subaru, this.honda, this.bmw];
```

And finally we can loop through all of our cars in the HTML file:

```
<div *ngFor="let car of cars">
 {{car.make}} {{car.model}}  with a mileage of {{car.miles}}
</div>
```

Finally, here is our output:

Subaru Outback with a mileage of 58232
Honda Accord with a mileage of 39393
BMW X3 with a mileage of 4400

Try it on StackBlitz

(StackBlitz link: https://stackblitz.com/edit/angular-loop-thru-custom-type?file=src%2Fapp%2Fapp.component.html)

> When you iterate over a custom type, like a Car in this instance, you must use dot notation (car.make, car.model, car.miles, etc.) to access the members of that entity because interfaces are made up of various fields, similar to a Javascript object.

Module 2 / LESSON FOUR / Part 3: Arrays of custom types quiz

Drag N Drop

Complete the code where countries is an array of Country type, and each Country has a name and a capital.

<div *ngFor="let country of countries">
  The capital of {{_____}} is {{_____}}

answers:
country.name **country.capital** country name countries.name countries.capital

Module 2 / LESSON FIVE: Reading Data

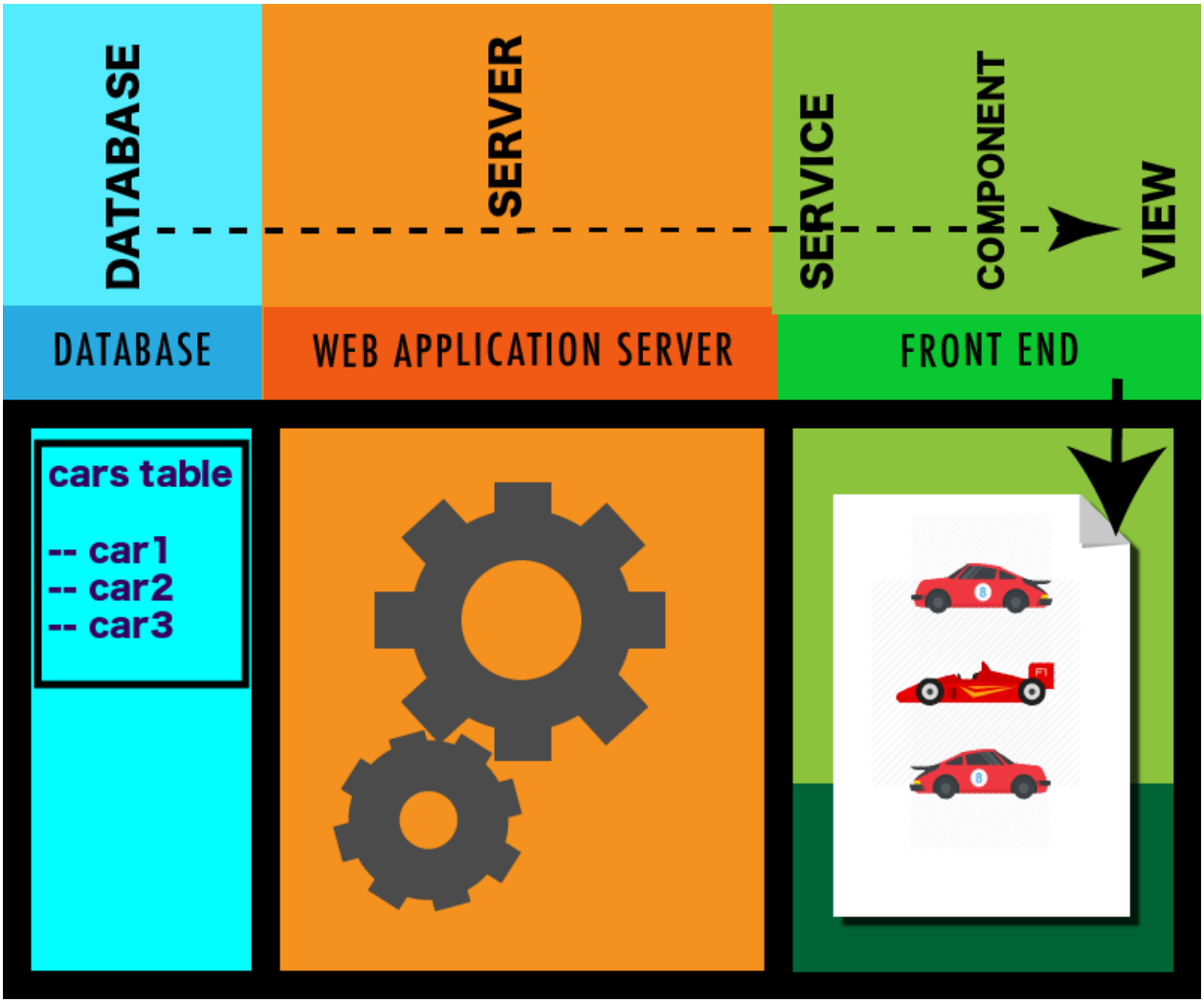
Module 2 / LESSON FIVE/ Part One: Data's long journey

Data's Long Journey

In our previous lesson, we created a list of car types in the component to loop over in the view.

This is fine for local data that's hard coded into the component, but in real world business examples, data makes a very long journey before it finally reaches our view.

Invoking our previous analogy of a restaurant from Module 1, let's look at the long journey that data makes in the real world before it reaches our view:



As we go deeper into the stack in this course, we're going to keep moving the data back further and further in the stack until we are finally getting it from the database. Our next step is to move the data from the component into something called a **service**, which we will cover very soon in upcoming parts of this lesson.

A simplified typical data flow, then, from back to front would be

1. Data retrieved from database query
2. Data routed through server
3. Data reaches **service**
4. Data reaches component
5. Data reaches view

> Databases are most often the place where data is stored for web applications. We will eventually get there. Stay engaged !

Module 2 / Lesson FIVE/ Part One: Data's long journey: quiz

REORDER

A user clicks on a button and receives a list of evening gowns. Arrange the journey of data of evening gowns with the top of the list being the beginning of the journey of the data

Database query is performed
Database delivers query results to server
Server processes and formats gown data
Server sends JSON of gowns to front end
Angular service & then component receive gowns
Angular view receives gowns

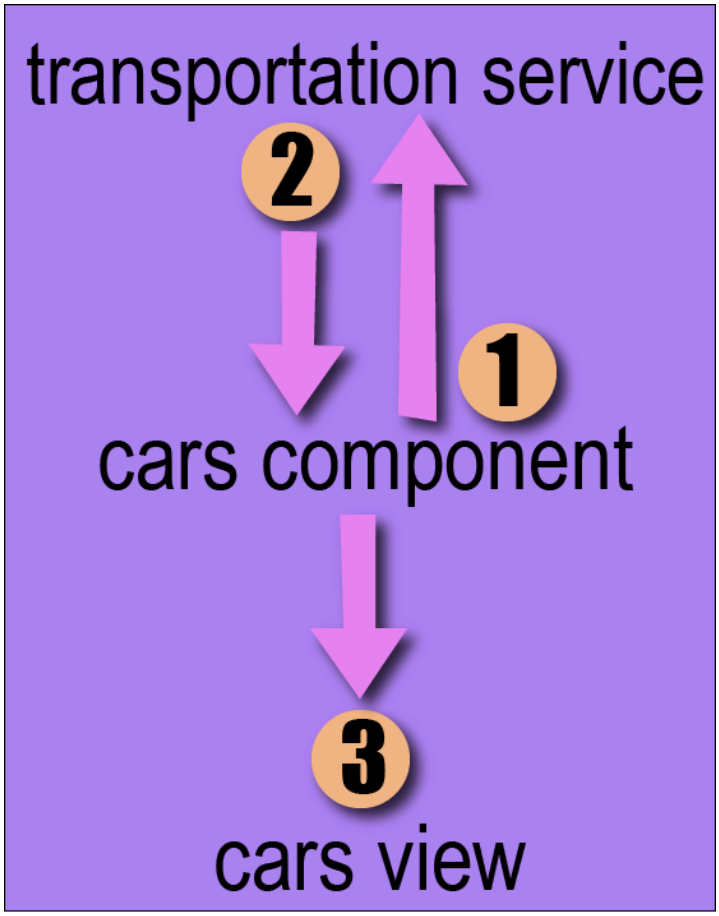
Module 2 / LESSON FIVE/ Part 2: Introduction to Angular Services

Introduction to Angular Services

Services are special files in Angular that are used to manage data. They usually pull data from XHR (remember XML/HTTP Requests from Module 1 ?), but they can also store data on their own.

Remember our diagram regarding the long journey of data ? For now, we're not going all the way to the database to get the data. We are going to simply push the data back into a service and let it live there for now.

This diagram shows a simple flow between a component, a service with data, and an HTML view.



1. The cars component makes a call to the transportation service, which holds a list of cars
2. The transportation service returns a list of cars to the cars component
3. The cars view receives the list of cars and loops over them using *ngFor

> A service is like a 'brain' in an Angular app that either returns data from the service itself or data retrieved from an external source. It can be viewed as a 'data manager'.

Module 2 / LESSON FIVE / Part 2: Introduction to Angular services quiz

Reorder

Order the process through which data makes its way from a service to an HTML view

component makes a call to a service requesting data
service receives call from component and collects data
service sends data to component
component passes data to view
view receives data
view iterates through the data using *ngFor

Module 2 / LESSON FIVE / Part 3: Generating an Angular service

Generating an Angular service

Let's make an Angular service.
Angular can **generate** a service for you with a simple command so you don't have to remember how to build the basic structure.

In order to do this, we use this command in the computer terminal:

```
ng generate service transportation
```

This generates an empty service within the Angular application.

```
import { Injectable } from '@angular/core';
@Injectable({
 providedIn: 'root'
})
export class TransportationService {
 constructor() { }
}
```

We will only concern ourselves with two parts of this service

1. The top of the file where the imports live. This is where we will import files that the service will use

2. the class body where it says export class TransporationService. The class body is contained within the curly brackets after the words **TransporationService**. This is where we will write methods to export data out of the service. Right now, the only method in the class body is the constructor, which we are not using right now. We will add more methods soon.

> Methods within the class body of the service are used to export data out of the service.

Module 2 / LESSON FIVE / Part 3: Generating a service quiz

Multiple Choice

The class body of a service is where

we write the HTML body
we write methods that export data out of the service
we import files that the service will use
we generate the service from the terminal
we write more Angular classes

Module 2 / LESSON FIVE / Part 4: Import interface into a service

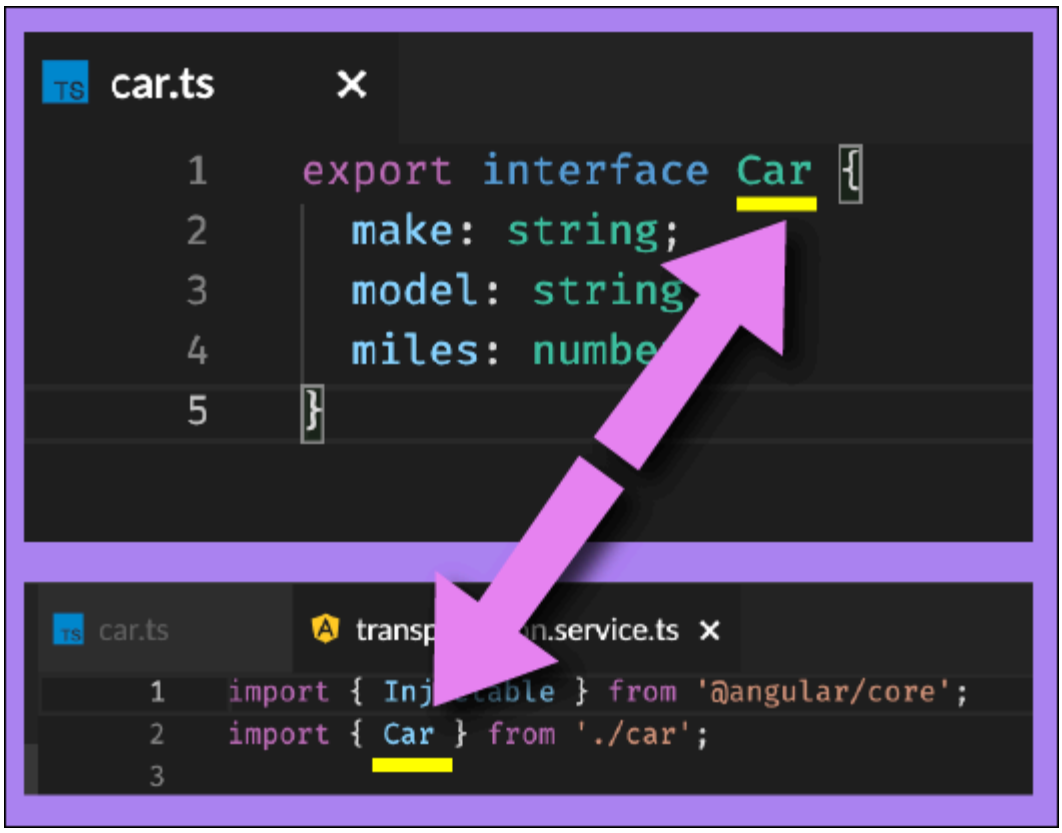
Import interface into a service

Our Transportation Service will contain an array of Car types. To do that, we need to import our Car interface that we created earlier. We learned that the top of the file is where we import things.

We can add this line to the top section of the transportation service:

```
import { Car } from './car';
```

This allows us to use the Car type to make an array of Cars. Our tranportaiton service has now IMPORTED the interface from the car.ts file. Here's the relationship between those two files:



The car.ts file EXPORTS the interface and the transportation service IMPORTS that same interface for use. That will now allow us to create an array of Car types in our service.

View it on StackBlitz

https://stackblitz.com/edit/angular-service-imports-interface?file=src%2Fapp%2Ftransportation.service.ts

View it on GitHub

https://github.com/Nmuta/angular-service-imports-interface

> The import / export functionality we see in Angular is derived from Node.js. This same import / export system is seen in almost every front end Javascript framework, as most of these frameworks are built using Node.

Module 2 / LESSON FIVE / Part 4: Import interface into service

Drag N Drop

Given the GroceryItem interface below, complete the code in a service to import the interface

groceryItem.ts

```
export interface GroceryItem {
  name: string;
  sku: string;
  price: number;
}
```

market.service.ts

import _____ from _____ ;

answer:

{GroceryItem} , **'./groceryItem',** groceryitem, {groceryitem}, groceries, string

Module 2 / LESSON FIVE / Part 5: recreating a Car array in service

Recreating Car array in a service

Now that we've imported the car resource, we can create an array of Car types in our transportation service. The last thing we need to do after we do this is to create a getCars( ) method to export the car data out of the service. Here is the completed service:

```
import { Injectable } from '@angular/core';
import { Car } from './car';

@Injectable({
  providedIn: 'root'
})
export class TransportationService {
    // NEW CODE
    subaru: Car = {make: 'Subaru', model: 'Outback', miles: 58232};
    honda: Car = {make: 'Honda', model: 'Accord', miles: 39393};
    bmw: Car = {make: 'BMW', model: 'X3', miles: 4400};

    cars:Car[] = [this.subaru, this.honda, this.bmw];

  constructor() { }

  // NEW CODE
  getCars() {
   return this.cars;
  }
}
```

This service is now ready to use by our component.

Try it on StackBlitz

https://stackblitz.com/edit/angular-service-completed?file=src%2Fapp%2Ftransportation.service.ts

View on GitHub

https://github.com/Nmuta/angular-service-completed

Services export methods that will later be invoked by Angular components that use the service. One service can conceivably be used by multiple components.

Module 2 / LESSON FIVE / Part 5: Import interface into service

Multiple Choice

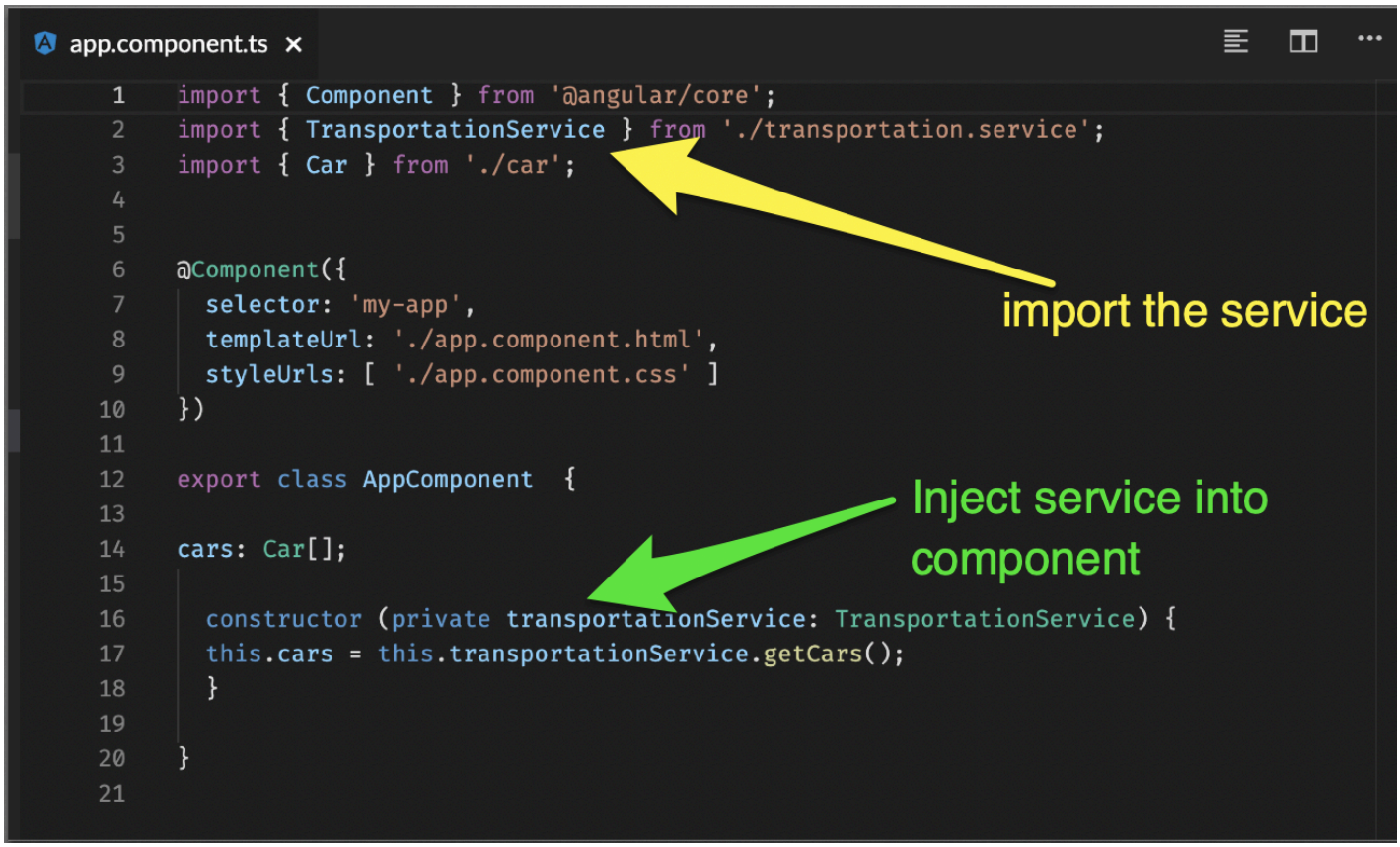
Data is exported from services via

interfaces
methods
strings
objects

Module 2 / LESSON FIVE / Part 6: dependency injection

Dependency Injection

Now that our service is actually exporting data, our component needs to pull it in .
In Angular we use something called **dependency injection** to pull a service into a component.
This service is now ready to use by our component.
Here's what this looks like:



Dependency injection is a common design pattern in object oriented programming.

Think of *dependency injection* like installing a weather app on your phone. Every phone that installs the weather app gets the same app. It can be installed on multiple phones, and every phone that clicks a "get weather" button for a city will get the same weather. The weather service is a central service that provides data for everyone who subscribes to it. That's what a service does through dependency injection. It usually provides data to all of its subscribers, along with other functionality.

Try it on StackBlitz

https://stackblitz.com/edit/angular-service-dependency-injection?file=src%2Fapp%2Fapp.component.ts

View on GitHub

https://github.com/Nmuta/angular-service-dependency-injection

> Note how we create a private variable called **transportationService** that is of the type

> *TransportationService*. **transportationService** (with the lowercase **t**) is the variable and *TransportationService (with an upperCase T)* is the type. Those two naming conventions are called **camelCase** and **PascalCase**, respectively. Using **camelCase** for class variables and **PascalCase** for class and interface names is a convention seen throughout Angular.

Module 2 / LESSON FIVE / Part 6: dependency injection quiz

Type in

Implement dependency injection by injecting the users service into the school component

```
import { Component } from '@angular/core';
import { UserService } from './users.service';
import { User } from './user';

export class AppComponent  {
users: User[ ];

  constructor (private userService : _________ ) {
   this.users = this._____________.getUsers();
  }

}
```

answers:

UserService, userService

Module 2 / LESSON SIX / ANGULAR EVENTS

Module 2 / LESSON SIX /PART 1: Event binding

Event binding

Up until this point, we have mainly been reading data in Angular and outputting it to the screen. What if, however, we wanted to actually change some data or respond to user events? That's where Angular event binding comes into play.

Browser based events in Angular are typically tied to methods within their corresponding components. Here is an example of a button that triggers an alert.

app.component.html

```
<button (click)="saySomething()"> Say Something </button>
```

app.component.ts

```
saySomething() {
  alert('good day.');
}
```

Let's break this code down.

<button **(click)** = "saySomething()" > Say Something </button>

Here we have inserted a click handler on a button that fires whenever a click event occurs. This triggers the “saySomething()” method in the component. In Angular, the parentheses around an event handler within an html tag signify what we call “event binding” . In this instance, “click” is bound with the parentheses to be (click) , which binds that event to the method passed in the quotes, which is “saySomething()” .

There are many other types of events that happen on the web: click, change, mouseover, hover, dragover, etc. are all examples of events that are the result of user actions.

Try it on StackBlitz

https://stackblitz.com/edit/angular-simple-event?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-simple-event

> Note that you must use the **()** after “**saySomething()**” to trigger the method in the corresponding app.component.ts file. This is because Angular not only binds the action to the event, it also fires the event when clicked, similar in execution context to Javascript’s native “call” method. The () after the method symbolizes the triggering of the event once that button is clicked.

Module 2 / LESSON SIX / Part 1: Event binding quiz

Type In

Given the following code in the component:

app.component.ts

```
register() {
  this.userService.registerUser(this.idCache);
}
```

Finish this Angular view code to trigger the “register” method from the component above.

<button (click)=_____> Register new user </button>

Answer: **register()**

Module 2 / LESSON SIX /PART 2: Making a counter

Making a counter

Our next exercise is to example using click events to make a counter increment. When you click on a button, the counter increments. This is what we will be trying to achieve:

As you can imagine, we need a button with a click event in the view which is tied to a corresponding method in the component:

app.component.html

```
<div class="container">
  <div class="number-display">
    {{counter}}
  </div>
  <button (click)="increment()"> increment </button>
</div>
```

app.component.ts

```
counter = 0;


increment() {
  this.counter++;
}
```

In the HTML view, the <button> tag has a (click) method inside of it. This sends a command to the component to fire the increment() method.

The center of the functionality here is the counter variable. When the button is clicked, the counter is incremented. The result is shown in the view because {{ counter }} will always hold the updated value of the counter variable as it's being incremented with clicks.

Try it on StackBlitz
https://stackblitz.com/edit/angular-counter-increment?file=src%2Fapp%2Fapp.component.ts

View on GitHub
https://github.com/Nmuta/angular-counter-increment

> Events further demonstrate the close relationship between *view* and *component* files.

Module 2 / LESSON SIX / Part 2: Making a counter quiz

Rearrange

Arrange the following elements in the order they occur

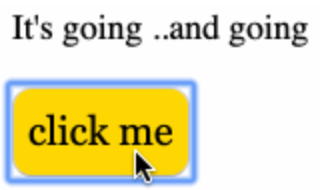
A user clicks on a button
The button triggers an event

The event invokes a method in the component
The component responds by updating a value
The updated value is displayed in the view

Module 2 / LESSON SIX /PART 3: Event updating a string

Event updating a string

We will try another exercise: we want to update a string with an event. In this example, every time we click on a button, an existing string will be appended with a few more words. The string will start off with the words "It's going" , and every time we click on the button, the words " .. and going" , will be added to the end of the string:



app.component.html

```
 <div class="display">
   {{phrase}}
</div>
<button (click)="update()"> click me </button>
```

app.component.ts

```
phrase = "It's going";


update() {
  this.phrase+=" ..and going";
```

```
  }
```

Again, we have a button that's wired to a method that fires in the component. The component has a variable called "phrase" that starts out as just two words: "It's going". However, every time you click the button, the **update()** method is fired and the += sign in that expression forces the componen to add ".. and going" to the end of whatever the former phrase was. The result is that the phrase keeps getting longer and longer and longer ever time we click the button.

It's better seen in action! Either visit the StackBlitz code live online or view on GitHub to see this working.

Try it on StackBlitz
https://stackblitz.com/edit/angular-event-update-string

View on GitHub
https://github.com/Nmuta/angular-event-update-string

> Events are a core part of any Angular project. In fact, events are really key to all of the Typescript and Javascript frameworks and most applications that use Javascript in general.

Module 2 / LESSON SIX / Part 3: quiz

Drag N Drop

This button will double an integer every time a button is pressed. Complete the code.

component

```
counter = 0;

 double() {

  _____  *=2;
```

```
}
```

view

```
   {{counter}}
<button _______ ="_________"> double the number </button>
```

this.counter , **(click)** , **double()** , click , double , increment

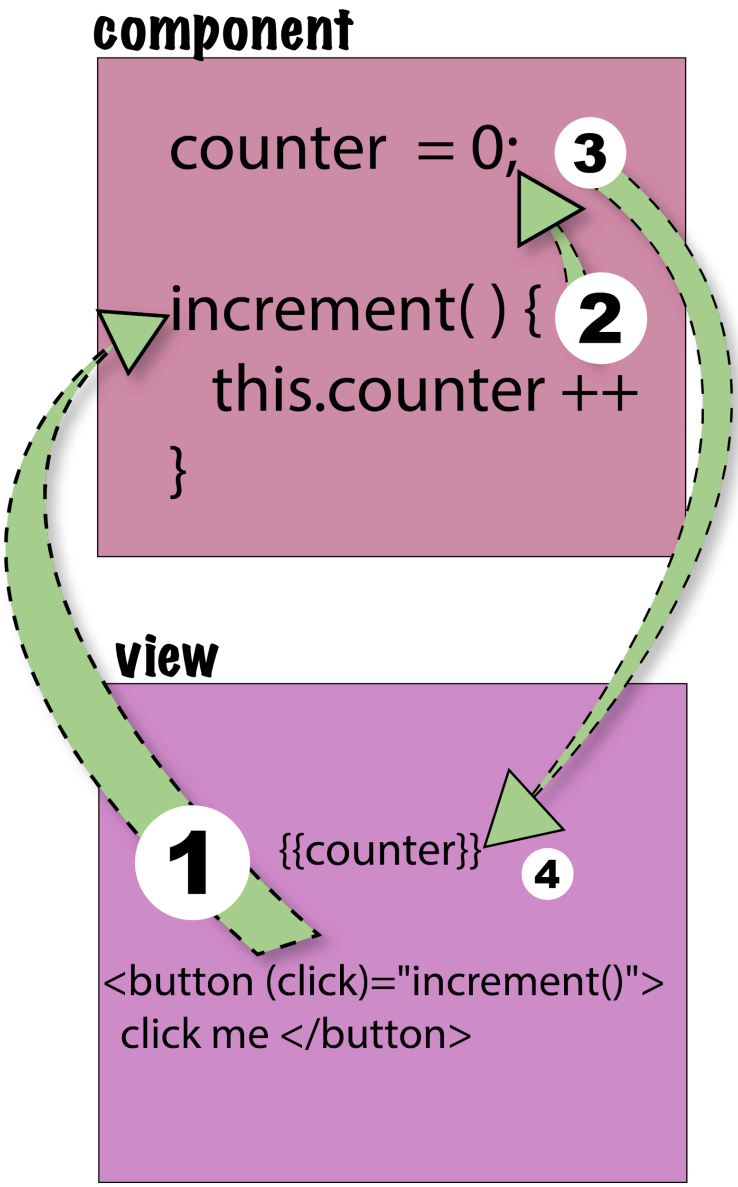
Module 2 / LESSON SEVEN /PART 1: Data binding

Module 2 / LESSON SEVEN /PART 1: One way data binding

One way data binding

You will often hear developers talk about “**one way data binding**” and “**two way data binding**” as it relates to front end javascript frameworks. What is data binding ? Data binding describes the flow of data from a component to a view.

All of the event binding examples in the previous lesson used one-way data binding.
Let's look at the data flow of the counter to analyze how one-way data binding works:



1. User clicks on a button in the view. The (click) event is wired to an “increment” method, asking the component to fire a method with the same name.
2. The component receives the click event and fires the **increment**( ) method, which updates the **counter**
3. The **counter** variable in the component is updated and then sends that updated value to the view
4. The view receives the updated counter variable and then updates the display so that the user sees a new number appear.

With one-way data binding, data flows in one direction. Imagine a ceiling fan. Even though the blades are flowing in a circular fashion, the movement is still only going in one direction.

Module 2 / LESSON Seven / Part 1: quiz

Multiple choice

Data binding describes the flow of data

between component and view
between service and view
between component and service
between module and component
within the view

Module 2 / LESSON SEVEN /PART 2: Two way data binding

Two way data binding

Two way data binding is when the flow of data between the view and the component goes both ways. There's a central "model" or variable container. If its updated in the view, then the component gets the change. If its updated in the component, then the view gets the change. The flow of data can go either way.

When Angular 1 (Angular JS) was released, one of it's key selling points was it's innate ability to do two-way data binding.

Since then, developers have universally agreed that two-way data binding should be used sparingly because of its intense demands on system resources. With that said, two-way data binding is still possible in Angular 8. There are certain situations where it can be useful when used judiciously.

In this lesson, we will implement a simple widget that allows you to see what you're typing being reflected in the view as you are typing it in real time.

The end goal is that you can type into a text field and see the result of what you have typed being stored in a **model**, which is displayed in the view.

> A **model** is a container that holds a variable's value. A model is accessible in a component and its associated view.

MODULE 2 / LESSON Seven / PART 2: Quiz

Type In

Two way data binding stores data inside of a _____

model
controller
view
stylesheet

MODULE 2 / LESSON Seven / part 3 / Banana in a box

The construct that allows two way data binding looks like this:

<input [(ngModel)] = "username">

The [( )] construct in Angular enables two-way data binding. Some developers call this the “banana in a box”.

It’s actually a combination of the square brackets [ ] which indicate data binding from the component into the view, (*the “box”*) and the parentheses ( ) which enable data binding from the view back up to the component (the “*banana*”) .

We’ve seen the parentheses in the view earlier :

button (click) = “increment()”

In our <button (click) = “increment()”> example, the (click) was an example of one-way data binding from the view to the component.

The square brackets, on the other hand, represent the data flow from the component back down into the view. By putting them together, Angular gives us a bi-directional flow of data. The “banana in a box”.

> Angular models as implemented via ngModel have nothing to do with databases or storage on the back end. A **model** in Angular temporarily holds data within the front end application. Models represent data.

MODULE 2 / LESSON Seven / part 3: Banana in a box Quiz

Multiple choice

In an Angular view , square brackets in a construct like this :
<input [(ngModel)] = "nationality">

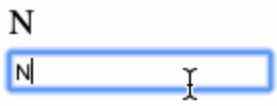
represent

data flow from the component to the view
data flow from the view to the component
three-way data binding
template binding
immutability

MODULE 2 / LESSON Seven / PART 4: ngModel

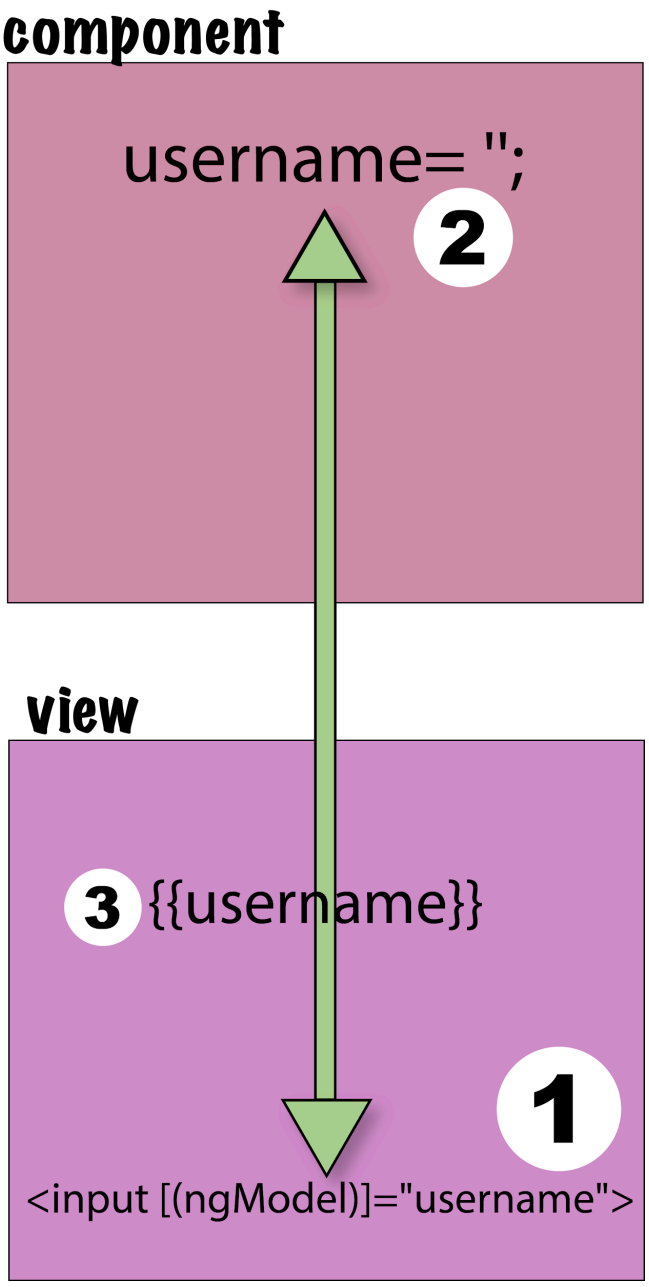
Let's look at a diagram of two way data binding. Data goes in two directions. When you type in something in the input field in the VIEW, it updates the model in the COMPONENT, which then sends the updated model value back to the VIEW. All of this is happening almost instantly. This is possible due to a special type of model called **ngModel**.

ngModel allows you to type in a field and watch the value of that variable be updated instantly as you type.



Here's the code that makes ngModel work in the view:

```
<input [(ngModel)]="username" placeholder="enter a username">
```



In a way, this data flow is a tad simpler than the diagram for one-way data binding one. Play with this two-way data binding example in the StackBlitz link provided.

Try it on StackBlitz

https://stackblitz.com/edit/angular-two-way-data?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-two-way-data

> Angular uses the **ngModel** construct to store variables that are going to be used in a two way binding context.

MODULE 2 / LESSON Seven / PART 4: Quiz

Multiple choice

Two way data binding in Angular refers to 2 way data flow between

The component and the view
An input field and the view
The module and the component
The module and the view

MODULE 2 / LESSON Seven / PART 5: ngModel deconstructed

Let's deconstruct how ngModel is actually working.

1 The user types in a field and then the contents of that field are stored DIRECTLY into the ngModel called "username".

2. The variable "username", which is stored in an ngModel, is stored in the component.

3. The view, which is constantly listening for changes to the ngModel, updates the username to be what the user typed in.

All of this happens instantly.

A ***model*** can be viewed as simply a container that stores information. There is a significant amount of theory about what this entails. A lot has been written on MVC (model-view-controller) and MVVM (Model-View ViewModel) patterns. Without getting too lost in the theory, we can simply say that a model is a **representation of domain data**. Another way of saying this, in the front end, the model is a ***container that stores variables***.

When we typed into the input field, the contents of the input field were immediately stored into the username variable.

Review again on StackBlitz

https://stackblitz.com/edit/angular-two-way-data?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-two-way-data

> An ngModel holds information that can be changed and accessed in both the component and the view.

MODULE 2 / LESSON Seven / part 5 / quiz

Reorder

Order the sequence of two way data binding in Angular

User types word into field
Word is stored in component

Component sends updated value back to view

MODULE 2 / LESSON Seven / part 6: Two way data bind setup

This section is for people who want to set up two-way data binding on a computer.
You can skip this section if you are simply learning online by skipping to the very easy quiz at the end of this section. The quiz does not require you to have read this part of the lesson.

[(ngModel)] will not work out of the box in an Angular component.
In order for this to work, we have to provide **support** to do it in the component.

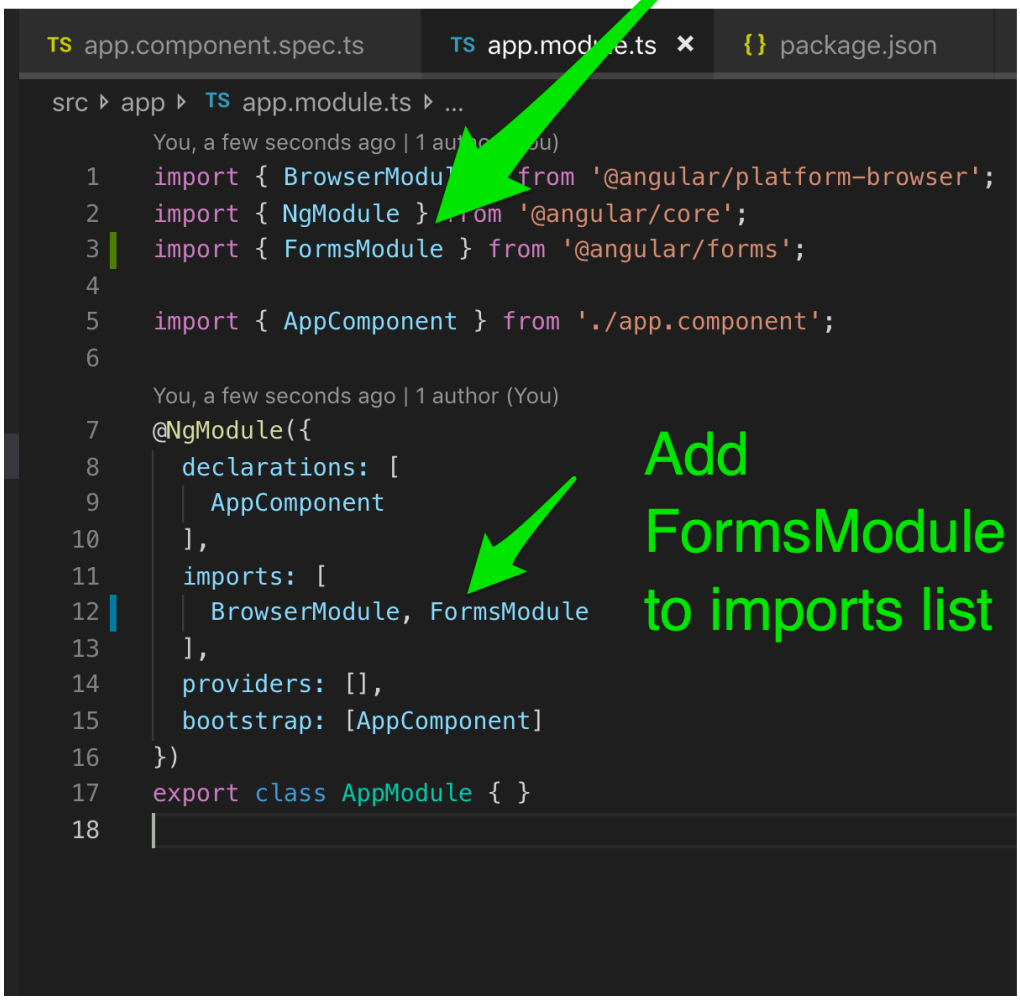
By default, applications are relatively slim. They only include the bare bones functionality needed for most apps. Two-way data binding will not be needed in every single app, so it's not in your project by default. You can give it that functionality by importing the forms module into your app.module.ts file (or whichever module you're working in).

In your app.module.ts file, you would import :

```
import { FormsModule } from "@angular/forms";
```

And then add the FormsModule to your imports in app.module.ts. The illustration below shows how those two tasks were achieved:

Import FormsModule



In our Sandbox and GitHub examples, we have already imported the FormsModule for you, so this is already set up. This section has just been included so that you will have the background knowledge to support two way data binding if you are implementing it on your own.

Two way data bind setup QUIZ

~~choose one (easy)~~

~~Two way data binding features~~

~~**Data that flows two ways**~~
~~Data that flows 345 ways~~

Revised quiz:

From which of the following would you import the FormsModule to your imports in your app.module.ts file?

@angular/forms
@angular/core
./app.component

MODULE 2 / LESSON Eight : Adding new data to our app

MODULE 2 / LESSON Eight / Part One : Sending hard coded data to a service

Now that we have an idea of how we can type data into a text field and have it stored in a model, we are almost ready to send new data to our cars array. In order to do that, we first need to explore how a service can receive data instead of just exporting data. Let's do that now.

In our component, let's add a method called "addCar()" that adds a new car to the transportation service. That method could look like this:

transportation.service.ts

```
addCar(car: Car){
```

```
  this.cars.push(car);
}
```

Here, our addCar method accepts a car of type Car as its only argument. Then it adds it to it's existing array of cars.

In our component, we will make a method that sends a pre-built car to the array every time the method is triggered:

app.component.ts

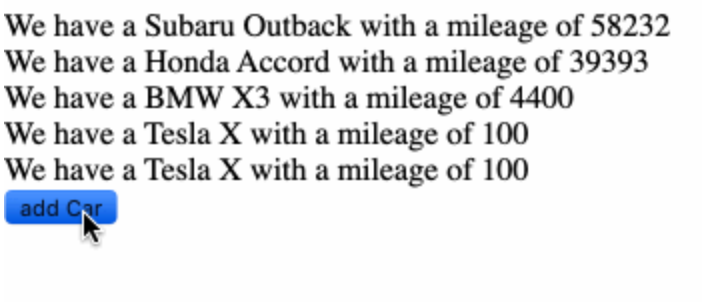
```
addCar( ) {
  const newCar: Car = {make: "Tesla", model: "X", miles: 100 };
  this.transportationService.addCar(newCar);
}
```

lastly, we will make a button in the view trigger the addCar( ) method:

app.component.html

```
  <button (click)="addCar()">add Car</button>
```

If you've ever wanted a new Tesla, now's your chance ! We've written the code to deliver a new one to your door instantly ! You get a car ! And ***YOU*** get a car !!!.... etc. Every time we click the button, a new Tesla appears.



Try it on StackBlitz

https://stackblitz.com/edit/angular-send-static-data-to-service?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-send-static-data-to-service

> Services can send ***and receive*** data from components.

MODULE 2 / LESSON Eight / Part One Quiz : Sending hard coded data to a service

Reorder

Rearrange the events in the order that they appear

user clicks a button to add new data to an array
event is triggered in component that sends data to the service
service receives message and adds an item to its array
service returns array to component
component displays updated data in the view

MODULE 2 / LESSON Eight / Part Two : Sending dynamic data to a service

We are now going to take one last step to tie everything together in terms of adding cars to our data service. In the real world, we rarely ever send "hard coded" data to a data resource ! We need a mechanism that will allow us to type in ANY car details and have them update in our cars array. We will achieve that now.

We can keep the respective **addCar** methods in both the transportation service and the component. The only change would be to collect the make, model, and miles of the new car in 3 new ngModels, then replace the hard coded data with our real data. That's it !

Let's start with the component:

app.component.ts

```
// we would add these variables to our variable declarations:
make: string;
model: string;
miles: number;
```

And then change the newCar variable assignment in our **addCar** method in the component so that it uses our new dynamic variables (make, model, and miles) …

app.component.ts

```
const newCar: Car = {make: this.make, model: this.model, miles: this.miles };
```

lastly, we will add input fields to the view for make, model, and miles

app.component.html

```
<input [(ngModel)]="make" placeholder="make">
<input [(ngModel)]="model" placeholder="model">
<input type="number" [(ngModel)]="miles" placeholder="miles">
```

And now, the code should just work ! When you enter the make, model, and miles of a new car, and click the button, it's sent into our data flow and gets updated in the service, which then refreshes the view and all of our new cars appear as we create them.

Try it yourself ! Play around with the code and even delete some of it and try to rebuild it on your own to gain an understanding of this data flow. Our next step is to learn how to connect the Angular app to a real BACK END, so that the data can persist in a database! Stay tuned.

Try it on StackBlitz

https://stackblitz.com/edit/angular-send-dynamic-data-to-svs?file=src/app/app.component.ts

View on GitHub

https://github.com/Nmuta/angular-send-dynamic-data-to-svs

> A page can have as many ngModels as you wish. They are a quick and easy way to collect data to send to our back end.

MODULE 2 / LESSON Eight / Part Two : Sending dynamic data to a service : quiz

Given the following code:

const newCar: Car = {make: this.make, model: this.model, miles: this.miles };

What is the difference between newCar and Car ?

newCar is a variable that is a Car type
Car is a variable that is a newCar type
they are identical
this cannot be determined out of context
Car is a static type and newCar is a dynamic type

MODULE TWO EXAM

1. **Drag n Drop**
   Complete the following code so that the type is correctly set as a string.

   const name: _____ = "Fred"

   answers:
   string boolean number text varchar

2. **Drag N Drop**
   Complete the following Address interface

```
_________ Address {
  street1: string;
  street2: _____;
  city: string;
  state: string;
  postal_code: _____;
  country: string;
}
```

   answers:
   interface , **string**, **number** , boolean , integer, Interface

3. **Multiple choice**

In the following Angular code living inside of a component :

```
fuel = 0;

addFuel( ) {
   fuel += 10;
}
```

there is an error in the code. What is the error ?

lack of a semicolon after the curly brace
code inside method should read *this.fuel* += *10*;
fuel outside the method should be prefixed by var, let, or const
addFuel method has no return value

4. **Type in**

Complete the following code to create a string array of fruit names:

```
const fruits: _______ = ['apple', 'banana', 'pear', 'plum'];
```

answer: string[]

5. **Type in**

Complete the following code to complete the array:

```
const secruityCodes: _______ = [12451, 77811, 59382, 23432];
```

answer: number[]

6. **Drag n Drop**

Complete the following VIEW code in HTML to iterate over the array of fruits and also provide an index to be used inside of the loop.

```
<div ______='let _____ of fruits; let _____ = ______'>
   {{ fruit }}}
</div>
```

***ngFor** , **fruit** , **i** , **index** , Index , ngFor, Fruit , indices, const

7. **Multiple choice**

A primary role of an Angular service is to

Directly interact with the view (HTML) page
Be a central data manager that components can use
Create databases that the application can use
Inject component instances in its constructor
Organize the rest of the Angular files

8. Type in

Complete the view (HTML) code so that a method named registerUser is triggered in the component

<a (_____)='_____'> register user

answers: **click**, **registerUser()**

9. Drag N Drop

Complete the view (HTML) code so that two way data binding will occur for a variable named **spice**.

<input _____ ='_____' placeholder="name of your seasoning">

answers:

[(ngModel)] , **spice** , model , variable , value

10. **Multiple choice**

What is the purpose of the [(ngModel)] in Angular ?

It enables 2 way data binding
It facilitates dependency injection
It initializes variables
It binds to click events
It stores the value of interfaces

Module 3: Making a Nest JS RESTful API

MODULE 3 / LESSON One : Back end overview
MODULE 3 / LESSON One / Part One : Reviewing data's long journey

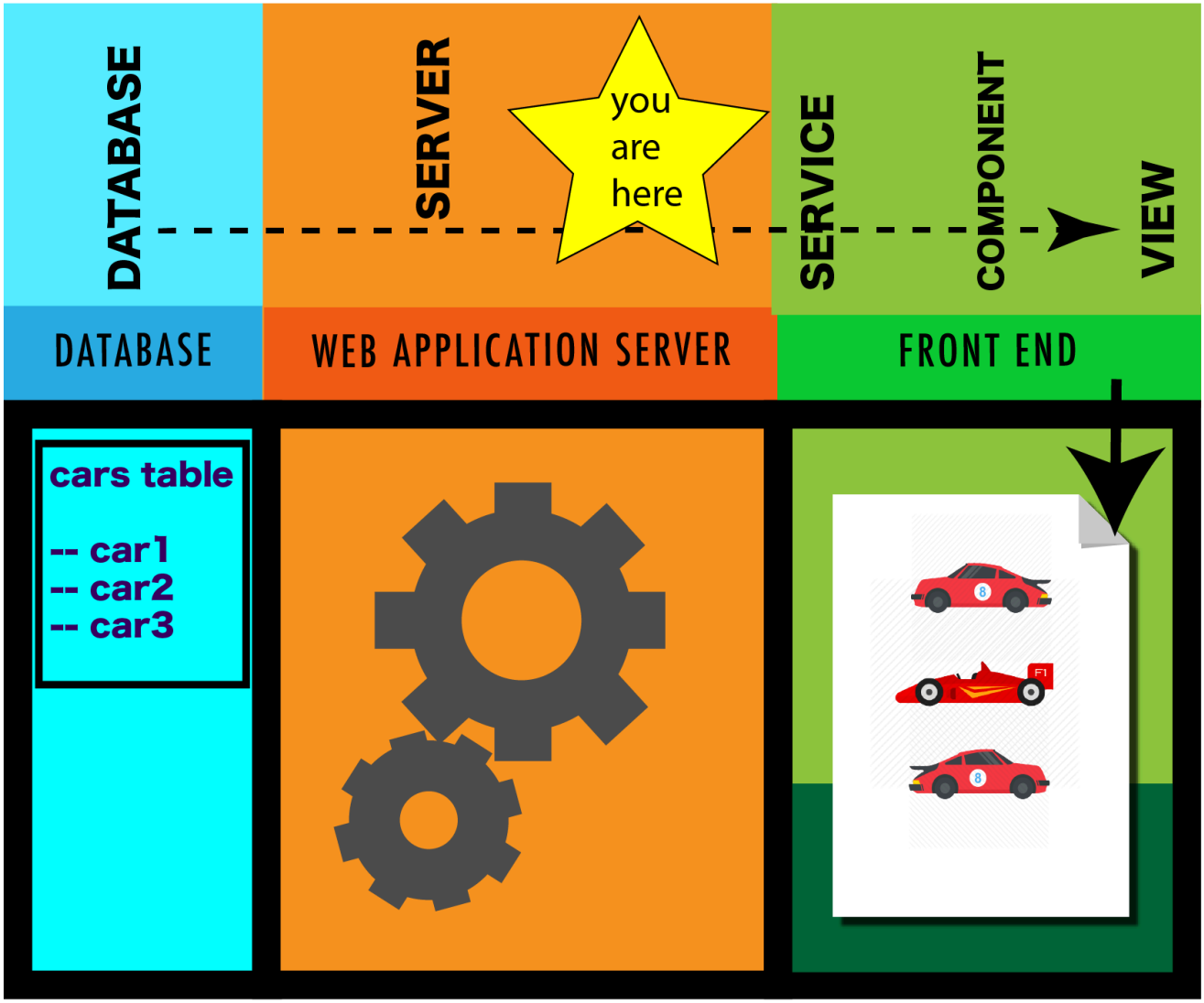
Reviewing Data's Long Journey

NestJS is a type of web application server. This module will teach you what NestJS is and how it works. When you're done, you will understand how web application servers work and how you can build one in the comfort of your own home.

In Module 2 we talked about the long journey that data goes through from the back-end to reach the front-end. In real life, this data will travel from the back-end often within seconds or even milliseconds. However, in our study of data, we are slowly moving to the back-end, which will be the focus of this module. By the time you are finished with this module, you will understand how data flows from a database through the NestJS server and is then pushed out to our Angular front-end.

In the diagram below, the **back-end** consists of the **web application server** and the **database**.

Our database in this course will be a PostgreSQL database and our web application server will be **NestJS**. We will explain how NestJS works as the lessons in this module unfold.



NestJS is a web application server. Its job is to handle requests from the front-end and deliver responses based on whichever tools it has available.

MODULE 3 / LESSON One / Part One : quiz

Multiple choice

Which of the following statements is false ?

NestJS functions as a web application server
Web application servers handle requests and send responses
PostgreSQL is a type of database
Databases serve data directly to front-end applications

MODULE 3 / LESSON One / Part Two : Handling requests

In Module One, we used the analogy of a kitchen to symbolize how full stack applications work.

Let's expand upon that analogy and visualize what's happening inside of the web application controller. In our kitchen analogy, a request is made from the front-end for some food. The web application server receives the request. If the request is valid, the server sends a **response** back to the front-end.

Let's say that inside of the server, there are short order cooks who are receiving requests for the food. These short order cooks, in the context of a server, are called **controllers**.

A **controller**'s job is to get a request and send back a response. Controllers have the ability to reach back into the refrigerator to get data, (the database), but on a fundamental level, the simplest of controllers can send a basic response back from right inside of the controller without touching a database. We will learn the simple controller first and then slowly build controllers that do more than just spit out text.

For now, we want to hit a web address that's received by our server and output HELLO WORLD! Once we do that, we will have built our first server response. This is exciting, so buckle up. We're diving into the back-end now.

> A controller inside of a web application server is like a cook in the kitchen. When it receives a request, it's job is to deliver a response to that request, like a cook responding with "spaghetti" when asked for pasta.

MODULE 3 / LESSON One / Part Two : quiz

Select all that apply

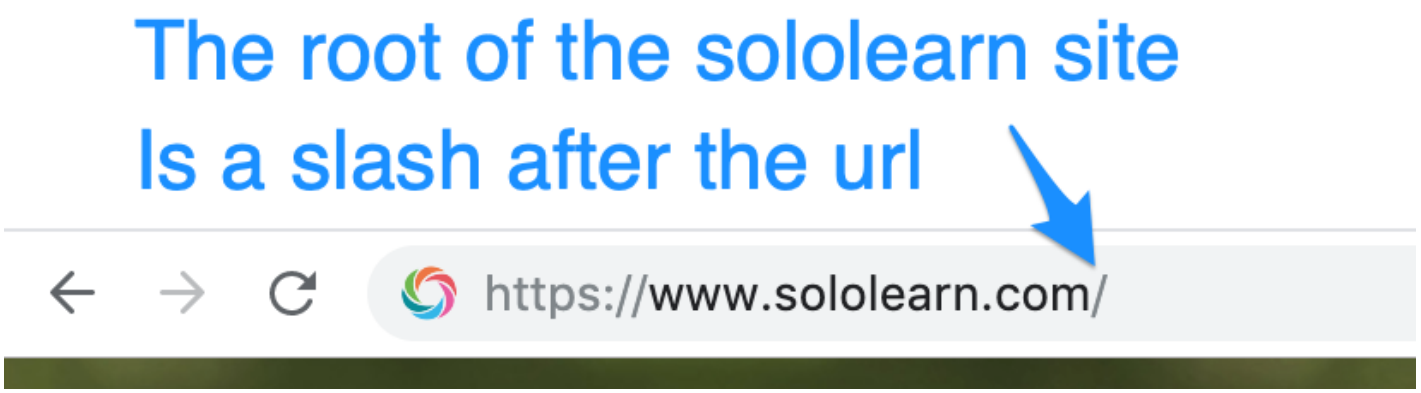
If a controller receives a valid request

It should not return a response
It should return an appropriate response
It may return a response without touching a database
It may hit a database before sending a response
It always hits a database before sending a response
It must return a JSON response

MODULE 3 / LESSON Two: Routes, Requests, and Responses
MODULE 3 / LESSON Two / Part One : Routes

To better understand how our controllers do their job, we will look at routes, requests, and responses.

When you visit https://www.sololearn.com , you are making a request to the **ROOT route** of that website back-end. What is the root? The root is essentially the home page. It's represented by a simple slash in the url.



Destination urls on web sites are called **ROUTES**. A route is simply a url that you are visiting at a particular **domain**. Here, sololearn is the **domain** and the route we are visiting is the root route.

Think of the domain as the entire kitchen, and each route takes you to a different cook. You go to different cooks for different things. In this case, the **root** route will serve as the home page. But a route like https://www.sololearn.com/fries would deliver us french fries. That's a different route. https://www.sololearn.com/soup would deliver us soup. We go to different routes for different purposes. Please, however, do not try to get an order of french fries or soup fulfilled at either of those addresses. This information is simply for demonstration purposes. 😛

> The root route of a website is generally the home page.

MODULE 3 / LESSON Two / part one:: Quiz

Type in

complete the url below to ensure that we are displaying the ROOT of a website called flowers.com

https://www.flowers.com_

answer: /

MODULE 3 / LESSON Two / Part Two : Routes continued

Websites typically have several routes that are for different pages and user actions. Some websites have dozens and even hundreds of routes. Routing can be complex, but for these examples, we will be using one route for handling each different type of request.

When we speak of routes on any given website, we are generally talking about *the part that follows the domain*. Examples:

`https://www.sololearn.com/cats`
….. would be called the "**/cats**" route.

`https://www.sololearn.com/dogs`
…..would be called the "**/dogs**" route.

`https://www.sololearn.com/stores/manage`
…..would be called the "**/stores/manage**" route.

> a route is a pattern of words, often separated by slashes, that points to a specific controller within a web application server. That controller will handle that request appropriately.

MODULE 3 / LESSON Two / Part Two : Routes continued quiz

A "route" generally refers to

the part of the url that's sending a request
the response to a web request
the part of the url after the domain
only the root of any website

MODULE 3 / LESSON Two / Part Three : GET vs POST

GET vs POST

In this module we will be learning about two different types of web routes :
GET routes and POST routes.

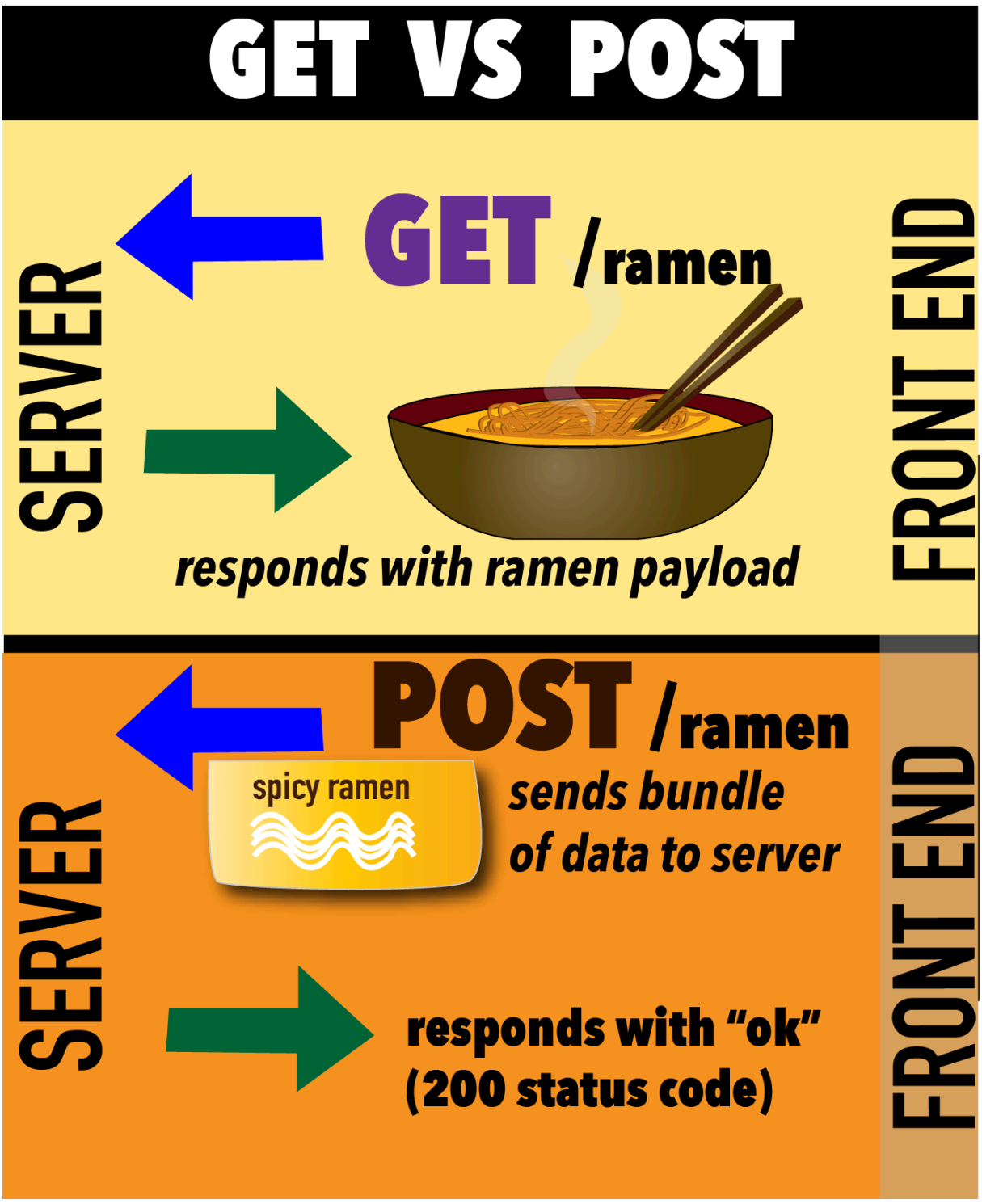
GET routes are generally to GET data from a server.
POST routes are generally to SEND data to a server.

In our kitchen analogy, **GETTING** ramen is different than **POSTING** a shipment of ramen to the kitchen so that they can expand their menu.

In extremely simple terms, a GET route gets data for you and a POST route sends data for you.

This diagram illustrates the concept.

GET VS POST



Essentially, think of GET as "fetching" something and POST as "sending" something.

MODULE 3 / LESSON Two / Part Three : get vs post routes : Quiz

~~Multiple choice~~

~~In simple terms, a _____ route is for sending data to the server, but a _____ route retrieves data from a server~~

~~**POST**, **GET**, SERVER , CLIENT, DATA~~

Revised quiz:
T / **F**
A get route is for sending data to the server and a post route retrieves data from a server.

MODULE 3 / LESSON Two / Part Four : get vs post requests

The front-end can send a **GET** request to the server, which connects to a GET route.

The front-end can also send a **POST** request to the server, which connects to a POST route.

On the server, we are going to set up various routes. Some routes will be GET routes, and other routes will be POST routes.

GET and POST are different types of routes, so they can have the same name but they will be recognized as different by the server because they are different types.

GET /ramen

POST /ramen

both routes are named /ramen but one is a **GET** route and the other one is a **POST** route

The http verbs "GET" and "POST" are derived from the associated English verbs that they were derived from. "Get" is associated with retrieving something from a resource, where "Post" is associated with putting something into a place. When you "post" a parcel, you are sending it to a recipient, where "getting" a parcel means that you are receiving a package.

MODULE 3 / LESSON Two / Part Four : get vs post requests : Quiz

Question: Which statements are true about routes ?

SELECT ALL THAT APPLY

GET requests from a front-end connect to GET routes on back-end
GET request from front-end connects to POST route
POST request from front-end connects to GET route
You can have two routes with the same name but different types
POST requests send bundles of data

MODULE 3 / LESSON Three : Intro to NEST JS
MODULE 3 / LESSON Three / Part One : What is NEST JS ?

Now that we understand the basics of GET, POST and routes, we can talk about the application that allows us to apply all of this knowledge. Ladies and gentlemen, introducing **NestJS**.

NestJS is a type of web application server built with **Node.js** that uses TypeScript as its primary language. Node.js is a technology that allows JavaScript to run outside of a browser giving it the ability to create servers and command line interfaces.

Node.js can be installed by visiting this website: https://nodejs.org/en/ . Run the installers to get Node.js installed on your machine.

Now we need to install NestJS. Visit https://docs.nestjs.com/ and follow the instructions on the page.

Now that Node.js and NestJS are installed, you can run a **scaffolding** command like the following in your terminal.

```
nest new cars
```

~~in your terminal.~~ Scaffolding in software development is the process of running a command that builds a whole bunch of stuff for you very quickly that's ready to use right away.

The command above creates a brand new web application server named 'cars' and then by typing yarn start or npm run start in the terminal, you will have a server running in port localhost:3000 on your computer. We will cover localhost in the next part of this lesson.

We have gone through this scaffolding work of creating a server for you and committing the code to GitHub. If you view this code, you can see the basic project set up of the simplest NestJS server possible, and it works.

View project on GitHub: https://github.com/Nmuta/nest_init

> **Scaffolding** is a term used to describe the process where you can type one line of code and a command line interface will create an entire boilerplate project or feature for you. It's like "just add water" pancakes. You don't need to know how it works, and the command gets you up and running quickly. Angular uses scaffolding as well.

~~Drag n Drop~~

~~In order to create a NestJS app, we use a process called _____ that allows us to type one line of code in the terminal that creates the whole project for us within a matter of seconds.~~

~~**scaffolding**~~ ~~static typing~~ ~~yarn~~ ~~npm~~ ~~localhost~~

Revised quiz:
Which of the following processes allow us to type one line of code in the terminal that creates the whole project for us within a matter of seconds?

- **Scaffolding**
- Static typing
- NPM
- Localhost

MODULE 3 / LESSON Three / Part Two : Running a server locally

When you install a web application server locally like NestJS, it will be running on port 3000 on your computer. What this means is that there's a place on your computer called a **port** that you can visit to see the server running.

When you go to a website like www.sololearn.com, you are visiting the **SoloLearn** server. You can run a "server" directly on your own laptop that is similar in some ways. The server you run on your laptop is only for you to test your applications. In order to be viewed by the outside world, special software needs to be installed on your computer to make it a public server. That doesn't apply to you; as a beginner, a server that you build on *your* computer will be *just* for you.

When you visit your local NestJS server on your computer, you will be visiting an address that looks like this:

http://localhost:3000/

Don't try to hit that address from your phone; it only applies to when you're on your computer and you have a local server running.

Developers use local servers like this to test out their work before uploading it or deploying it to the internet. This way you can test things and make tweaks as you build without having to do all of that construction work in a public space.

> When you run a server locally, most of the time, you are running that server in a space called "localhost", which simply refers to the address of your own personal computer.

MODULE 3 / LESSON Three / Part Two : Quiz

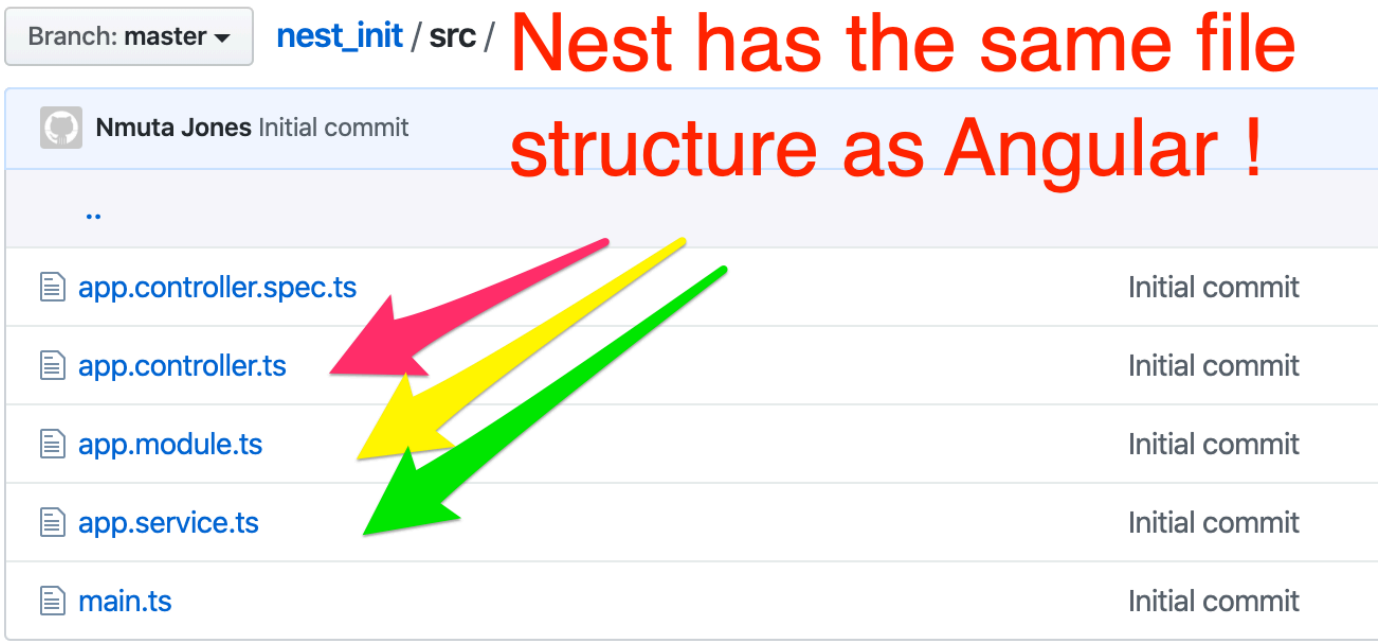
Localhost refers to which of the following:

Your local computer address
A web application server on the internet
A port on the web
A NestJS controller

MODULE 3 / LESSON Three / Part Three : Anatomy of a NEST server

Before we dive into our NestJS app and start making things happen, let's look at our NestJS file structure. When we scaffolded our server, we got a bare bones web application server that's ready for use. It has a **controller**, a **module** and a **service**. Wait a minute….this looks almost exactly like an Angular file structure. Well, this is no coincidence. NestJS uses TypeScript, and the team at NestJS made the decision to make the file structure look like Angular so it could be easily understandable by Angular developers.

This is how "stacks" are born: a set of tools that work easily with each other inevitably will be branded a "stack" by the development community. Hence the TypeScript stack with Angular and Nest now appears to be a thing, even if only by implication.



Looking at this file structure, we can see a controller that handles web traffic, a service to provide data to that controller, and a module that works behind the scenes to make sure that the service and the controller both have the supportive elements that they need in order to operate.

Let's quickly look at what the controller is doing. We see some code that looks like this:

```
@Get()
getHello(): string {
    return this.appService.getHello();
  }
```

Essentially this gives the default "GET" request (when someone visits the website home page) and returns the result of the **appService getHello function**, which simply returns the string "Hello". So we have a controller calling on a service to get data, just like in Angular.

Feel free to peruse the files in the GitHub repo to see how the 'Hello World' NestJS example works:

View project on GitHub: https://github.com/Nmuta/nest_init

> NestJS and Angular use services in almost exactly the same way, and the file structure in general is very similar. The main difference is that in Angular, components call services, but in NestJS, controllers call services.

MODULE 3 / LESSON Three / Part Three : Anatomy of a NEST server : quiz

multiple choice

Services in BOTH Nest JS and Angular do which of the following:

Have nothing in common
Relay data to and from controllers and components respectively
Relay data directly to databases
Relay data directly to views

MODULE 3 / LESSON Three / Part Four : How NEST responds to requests

We are about to see how NEST handles a simple **GET** request made to the **root route**. If you understand that last sentence, you should be proud of yourself… that means you're learning. Let's examine our NestJS "init" code from here:

```
1  import { Controller, Get } from '@nestjs/common';
2  import { AppService } from './app.service';
3
4  @Controller()
5  export class AppController {
6    constructor(private readonly appService: AppService) {}
7
8    @Get()
9    getHello(): string {
10     return this.appService.getHello();
11   }
12 }
```

The AppContoller in NestJS is similar to the app.component.ts file in Angular. It's the default code that is run in the app when we start the app after scaffolding it. It's our starting point.

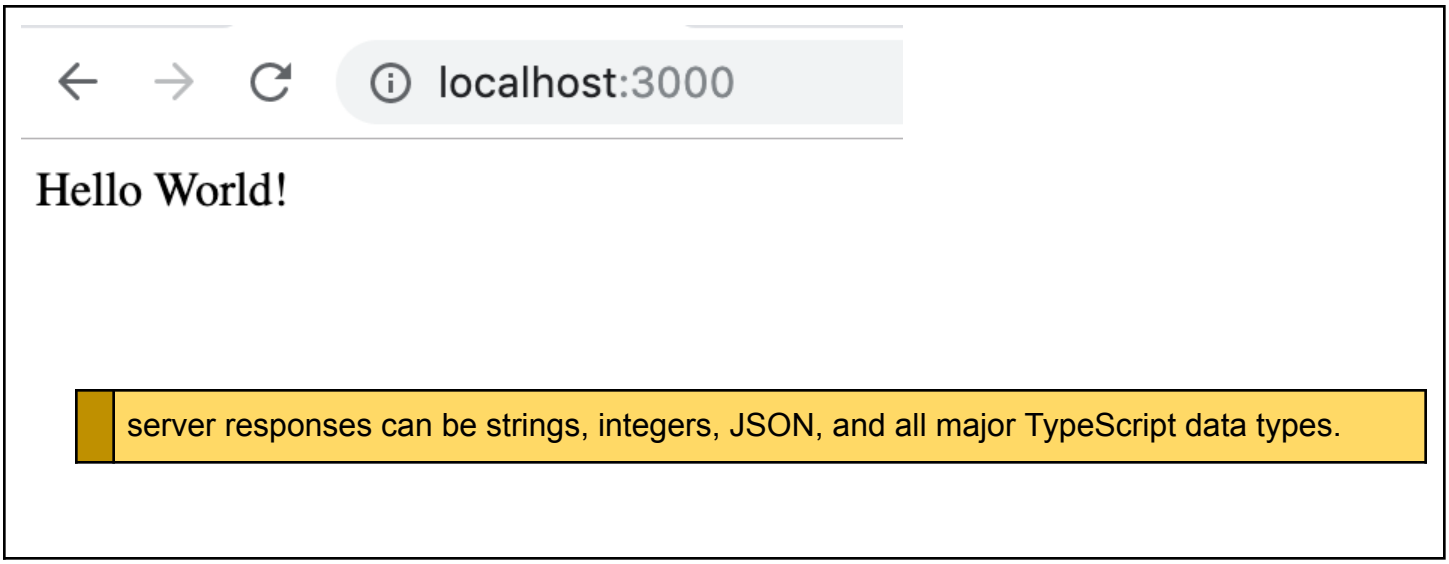
There's a GET route lurking in that code somewhere. Do you see it? Quick, don't read the next line, go back up and look at the code and see if you can find the GET route and its response.

SPOILER ALERT! Ok so here's the answer: The GET route starts on line 8. **@Get( )** is what we call a **decorator**, which just means that this is our way of telling NestJS that the function that follows should return a response to the GET request.

Line 10 is where we dish out this **response**. Notice that the default boilerplate code is using a service. This is a little fancy, we don't need a service for this, in this case, for a simple response, we could just use this as our code for line 10:

```
return "Hello World";
```

The bottom line is that this code runs when we hit our server. When we hit http://localhost:3000, we see the response "Hello World!".



server responses can be strings, integers, JSON, and all major TypeScript data types.

MODULE 3 / LESSON Three / Part Four : Quiz

Drag n Drop

Complete the NestJS server response to respond with the word "rice" when someone visits localhost:3000. Do not use a service, just return a string.

```
@Get()
 getFood(): string {
   return ____________
}
```

"rice" rice rice:string rice: "string" rice: "String"

MODULE 3 / LESSON Three / Part Five : Return types on routes

RETURN TYPES ON ROUTES

Let's look at our code again from our NestJS init code from here:
https://github.com/Nmuta/nest_init/blob/master/src/app.controller.ts

Please observe line 9 :

getHello( ): string {

```
1  import { Controller, Get } from '@nestjs/common';
2  import { AppService } from './app.service';
3
4  @Controller()
5  export class AppController {
6    constructor(private readonly appService: AppService) {}
7
8    @Get()
9    getHello(): string {
10     return this.appService.getHello();
11   }
12 }
```

getHello() is a function. It has a return value. Everything about this looks like a normal JavaScript function except for the colon after the method name and the word string. Behold another TypeScript feature that we will now learn in the process of learning NestJS: **function return types**

Yes, my dear friends, Typescript has the ability to statically type the RETURN TYPE of a function.

getHello():string means that the getHello function MUST return a string value. If it doesn't the code won't compile.

getHello(): number would mean that the getHello method would have to return a number.

getHello(): boolean would mean that a boolean value would have to be returned. You get the picture.

> Adding return types to routes provides structure and consistency to our routes so that you as a programmer and other developers working in this code base can follow uniform patterns in relation to handling data within the app.

MODULE 3 / LESSON Three / Part Five : Return types on routes : quiz

TYPE IN

Complete this code so that it will compile.

```
@Get()
  randomNumber(): _________ {
   _________ Math.floor(Math.random( ) * 10);
 }
```

answers: **number** **return**

MODULE 3 / LESSON Four: Writing Nest routes
MODULE 3 / LESSON Four / Part One : scaffold cars route

Earlier we looked at this back-end code: https://github.com/Nmuta/nest_init/ and we looked at a controller that returned a response to when users visit the root route of the website.

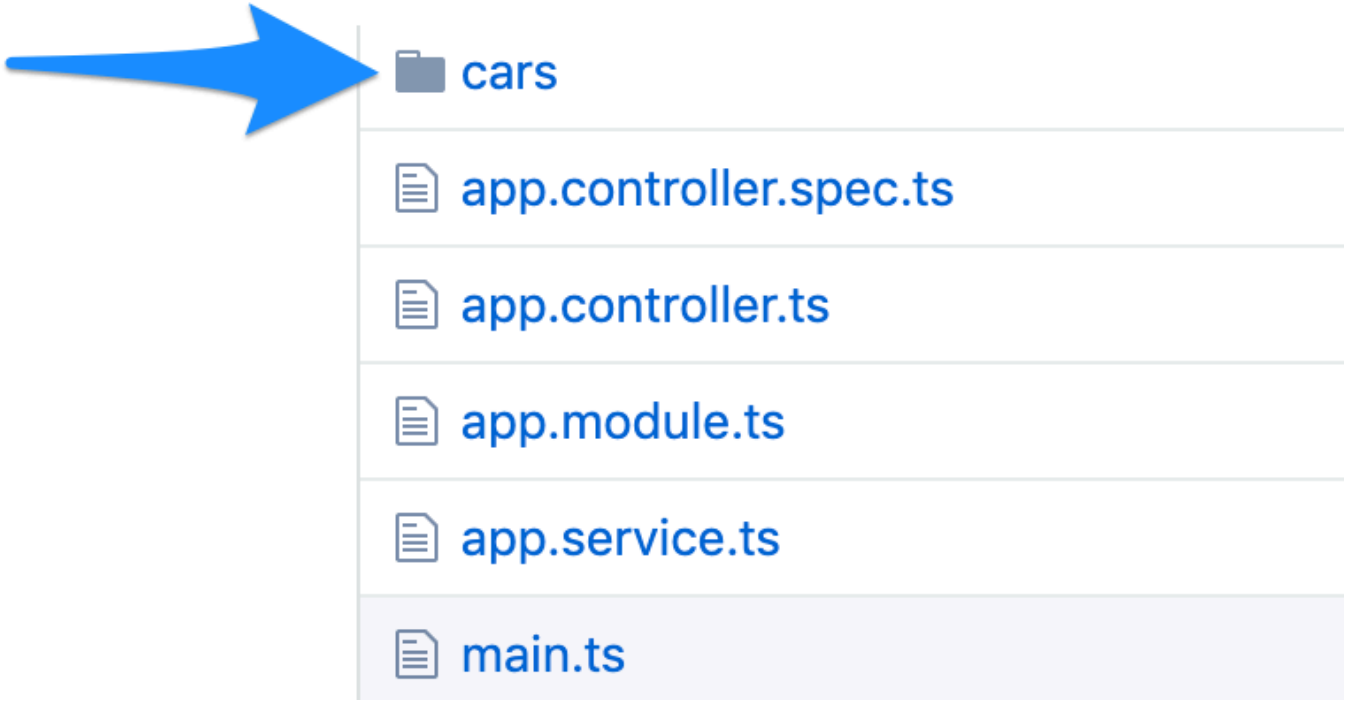
This is great, but what if we wanted to create another route? Ideally we would like to create a route where you could GET cars, which would return an array of cars in JSON format that our front-end would use to display all of the cars coming from the back-end.

We have a new code repository that illustrates just that : https://github.com/Nmuta/nest_controller

In NestJS, we can **scaffold** a new route by simply doing this in the terminal:

```
nest generate route cars
```

This adds a new folder:



And it inserts other support in the code base that allows us to now have access to a new GET route called /cars.

> When you are beginning with NestJS, you can use the scaffolder to create your app and create new routes without fully understanding all of the details of how these things work under the hood. As you grow as a developer, curiosity will most likely lead you to dig under the hood to uncover the myriad of supporting code structures that make everything work. For now, "it just works!", and it's a great place to start. We now have a new route.

MODULE 3 / LESSON Four / Part One : Quiz

Reorder

Order the procedure of setting up a NestJS back-end application with a /hello route that is designed to print out the words "hello everyone!". The route the user will hit will look something like this :

https://www.xyz.com/hello

Scaffold the NestJS application in the terminal
Scaffold the /hello route
Locate the /hello folder within the application
Open the hello controller add a string return type to the hello route
Return "hello everyone!" in the /hello route code block

MODULE 3 / LESSON Four / Part Two : View a Scaffolded Route

In our previous lesson, we scaffolded a new cars route by doing **nest generate route cars**, which created a folder of files to support a /cars **GET** route. Peeking at the cars **GET** route that was created, it looks like this :

```
@Get()
findAll(@Req() request: Request): string {
  return 'this will eventually return a car collection';
}
```

This route, named findAll( ) , is one of the many routes created by the Nest generator.

It is very important to understand the context of how routes work in any Node.js based web application folder. Inside of a "cars" resource a @Get( ) route or a @Get('/') route is a function that accepts an argument. The argument that it accepts becomes the sub-route of whichever route you are in. If there is no sub-route provided, it defaults to the root route of the folder you are in.

Examples:

a plain @Get( ) route in /students will default to http://www.yourserver.com/students
a plain @Get( ) route in /buildings will default to http://www.yourserver.com/buildings
a plain @Get( ) route in /recipes will default to http://www.yourserver.com/recipes

… and so on. The name of the folder that you are in is the prefix. The part inside of the **@Get( )** parenthesis is what to add on to the end of that prefix. So:

@Get("models") within a dedicated cars controller, would take you to
http://www.yourserver.com/cars/models

What is returned in any of these routes is what the browser will display when a user hits that route in a browser. Here the route simply returns a string that says 'this will eventually return a car collection'.

View the default cars GET route on GitHub :
https://github.com/Nmuta/nest_controller/blob/master/src/cars/cars.controller.ts

> A GET route accepts one argument: a string that represents the rest of the GET route after the prefix. The prefix is the name of whichever controller you're in.

MODULE 3 / LESSON Four / Part Two : view a scaffolded route: Quiz

Multiple choice

A bicycles controller in NestJS that has a GET route with no parameters will direct users to which of the following:

a route with the url of /bicycles
a route with the url of /bicycles/root
an error page
a 404 page until a database is set up
the root page of the website

MODULE 3 / LESSON Four / Part Three : get all cars

A /cars route that just returns a simple string is boring and not useful at all. Let's return an actual array of cars in JSON format.

In our previous code sample in the previous lesson part, we had a GET cars route that looked like this:

```
@Get()
findAll(@Req() request: Request): string {
  return 'this will eventually return a car collection';
}
```

The @Req decorator tells NestJS that we are expecting a **request** object that we are calling "request" that is of the **Request** type.

Let's change the return type to this:

```
findAll(@Req() request: Request): {} {
```

Notice that instead of a **string** return type, we have an **object** as the return type. This will accommodate an object or even an array of objects.

Now we can do something like this:

```
@Get()
findAll(@Req() request: Request): {} {
  return [{make: 'honda', model: 'accord'},
          {make: 'subaru', model: 'outback'},
          {make: 'fiat', model: '123 spider'}];
}
}
```

If we hit our /cars route now, we get this in the browser window:


```
[
  {
    "make": "honda",
    "model": "accord"
  },
  {
    "make": "subaru",
    "model": "outback"
  },
  {
    "make": "fiat",
    "model": "123 spider"
  }
]
```


The RESPONSE TYPE of a route is very typically JSON in modern web application server architecture. Here we simply made the response type {}, but it can also be { }[ ] which literally designates "an array of objects". Remember when we created arrays with TypeScript? Since an array of strings is typed like this : string[ ] , it follows that an array of objects could be typed like this : [ ] { }

MODULE 3 / LESSON Four / Part Three : Quiz

Drag n Drop

Finish the code below so that this GET route is set up to return a simple JavaScript object.

```
getCreds(@Req() request: Request): ____ {
  return {uuid: '12x8s', auth: true, tokenize: false}
 }
```

{} object string boolean request

(the correct answer is the set of curly braces)

MODULE 3 / LESSON Four / Part Four : Nested routes

Nested Routes

On the web not all routes have simply one word after the domain. It's common on the web to see **urls** with more than just one section after the domain. For example, you may see a route that looks like this:

https://www.sololearn.com/cars/engines

In this case, **engines** is nested within the cars route. We call this (shockingly)….a **nested route**.

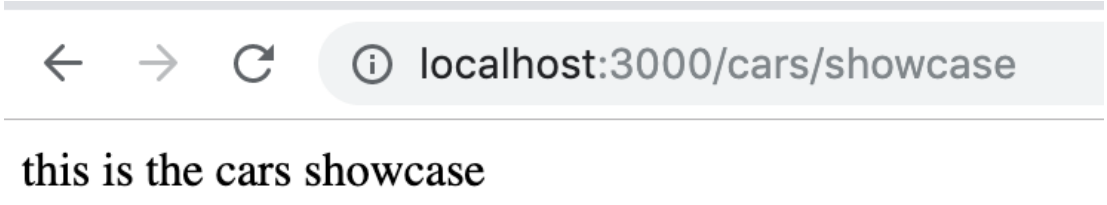
In NestJS, within a controller file like the cars controller, here's how we create a nested route called "showcase" within cars:

~~We simply~~ **Add another section** to the route file and put the name of the nested route within the function declaration for that route:

Nested 'showcase' route within cars

```
@Controller('cars')
export class CarsController {
  @Get()
  findAll(@Req() request: Request): {} {
    return [{make: 'honda', model: 'accord'},
            {make: 'subaru', model: 'outback'},
            {make: 'fiat', model: '123 spider'}];
  }

  @Get('showcase')
  showcase(@Req() request: Request): string {
    return 'this is the cars showcase';
  }
}
```

Then, if we visit in our local server this address: http://localhost:3000/cars/showcase , we see the cars showcase route:



Here's the code on GitHub for you to examine:

https://github.com/Nmuta/cars_sub_route

In short, nested routes are created by chaining routes with a "/" between each word in the route. A route like this:
www.sololearn.com/quizzes/english/typescript
could look like this in a QUIZZES controller: remember, the word 'quizzes' is implicit in this route because this route lives in the quizzes controller:

@Get('/english/javascript')

> Think of a nested route as a 'sub route' that lives underneath a parent. Nested routes can go deep. You can have a route that looks like this : http://www.mydomain.com/countries/usa/states/colorado/cities/denver . Nested routes are very common on the web.

MODULE 3 / LESSON Four / Part Four : Nested routes: Quiz

Type in

Complete this nested route decorator where 'vegan' is a nested route under 'foods'

```
@Get('_____/_____')
     veganfoods(@Req() request: Request): string {
    return 'this is the vegan foods route';
}
```

answers: **foods vegan**

MODULE 3 / LESSON Four / Part Five : Wildcard routes

Sometimes certain routes have a **segment** in them that can change according to data that changes. Let's say we have a database with several cars in it:

id	make	model	name
5	ford	mustang	Spirit
6	chevy	impala	Curtis
12	toyota	corolla	Simpson

To GET a car from the database, we would call a route that looks like this :

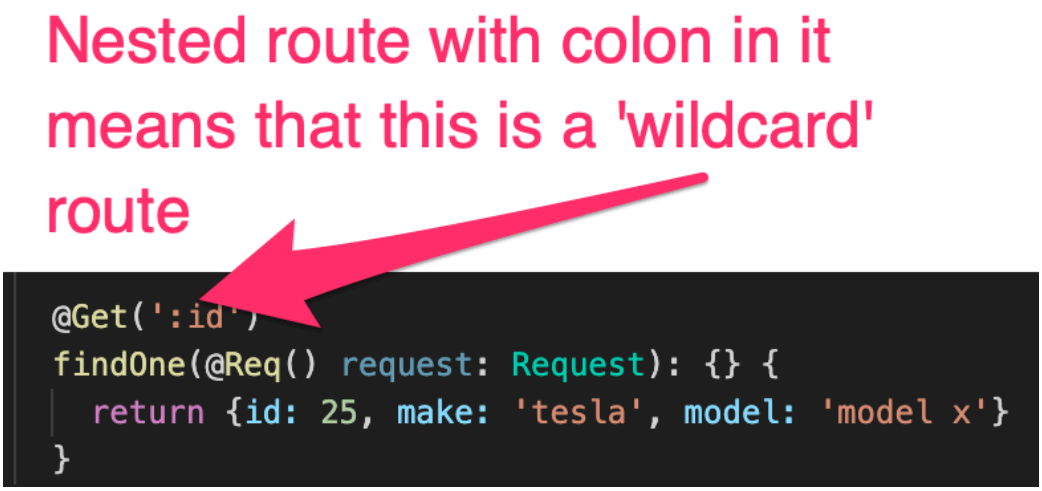
http://www.sololearn.com/cars/6

which would return this data:
{id: 6, make: chevy, model: impala, name: Curtis}

but if we call
http://www.sololearn.com/cars/5

we get
{id: 5, make: ford, model: mustang, name: Spirit}

The LAST SEGMENT of the url is called a "wildcard". Feed it 5, and you get the ford. Feed it 6, and you get the chevy. Feed it 12 and you get the toyota. How does the back-end know how to handle a "variable" in the route? We use this syntax in the route:



Ignore the return value here …later in this course we will not 'hard code" the return value; we will look it up in a database based on the id passed in.

Here we are hard coding the response just to show some sample data returned. **The key thing to notice here is that the "id" passed in is a variable.** Later, we will capture that id passed in and use this value to look up the proper car in the database.

Our route is essentially a method called "**findOne**( )". The name **findOne** is just a name I chose; it's not special. NestJS just needs a name for the function so this name can be anything that appropriately describes what the function is doing.

A wildcard route with several levels of nesting could look like this:

@Get('gamers/alliances/:country/:region/:city')
A route like that would not be very common, but we are providing this example to show you what multiple levels of nesting could look like.

> Wildcard routes are the key to looking up specific entries in a database.

MODULE 3 / LESSON Four / Part Five : Wildcard routes Quiz

Multiple choice
Select all that are true

wildcard routes accept variable segments in the url
wildcard routes can be nested routes
wildcard routes can be GET routes
~~**wildcard routes can accept data that will be used to query a database**~~
the root route to any site is usually a wildcard route

MODULE 3 / LESSON Five : RESTful APIs
MODULE 3 / LESSON Five / Part One : Intro to Web APIs

Introduction to Web APIs

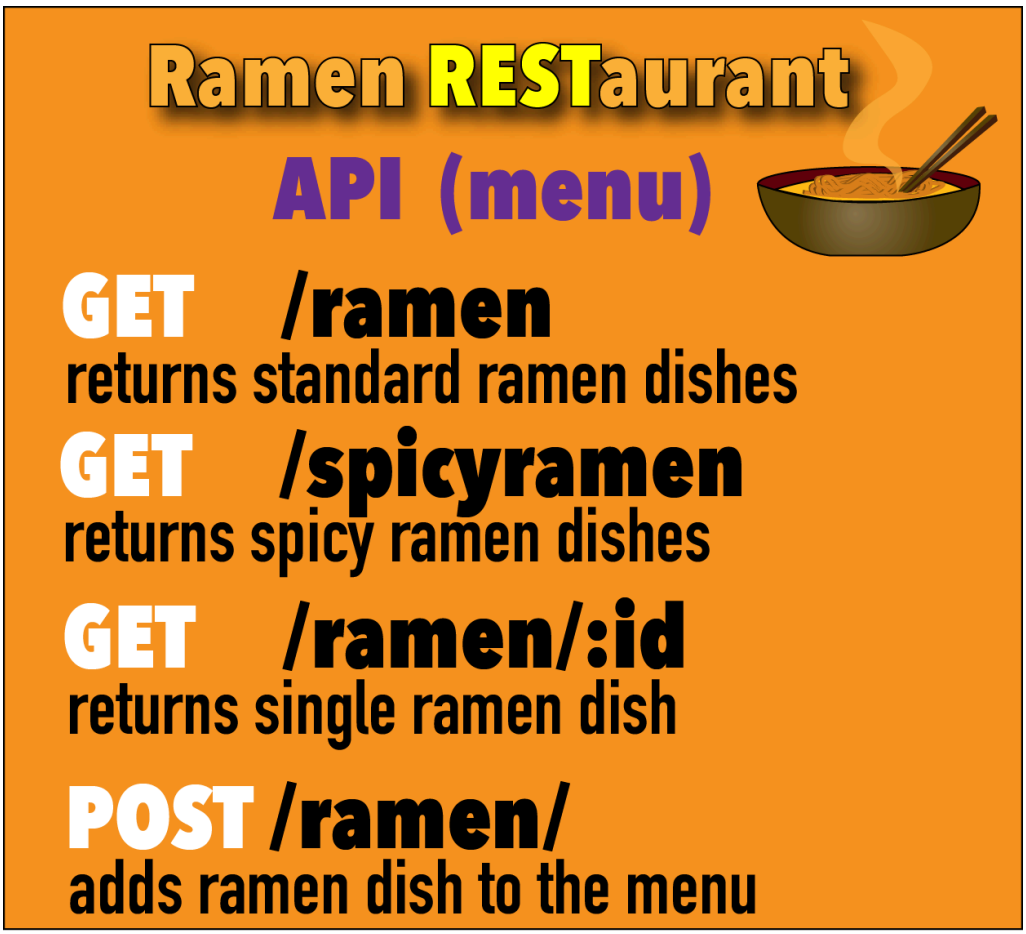
If you want to be cool with all of your developer friends, one great way to do this is to start discussing **RESTful APIs**. At the very least, they will know that you understand something about development and you can probably sit at the cool kids' table at lunch.

Seriously, though, RESTful APIs are at the heart of web communication. **REST** is a deep and complex subject, but what we will do in this lesson is demystify it so that you can know enough to be dangerous.

Let's start with **REST**. *REST is simply a protocol that defines how web technologies should communicate with each other*. Remember when we learned **HTTP**? Well, REST uses HTTP verbs and certain conventions with how urls are written as part of its process.

API stands for **application programming interface**. An API is like a menu in our restaurant. If you want a particular thing, you have to ask for it a certain way . The menu gives you instructions for how to order exactly what you want, and if you ask for it exactly in the right way, you'll get it.

In summary, REST is a convention of how computers talk to each other on the web.
An API is like a menu of specific addresses that you use to get things from a website.



An API is a way to interact with a back-end using URLs that match certain patterns.

MODULE 3 / LESSON Five / Part One : RESTful APIs: Quiz

~~What is the best metaphor for an API ? (choose one)~~
Which one of the following best describes the function of an API for a grocery store? (choose one)

An API describes the color of a store
An API is like a way to arrange products in a store
An API describes how much money a store will make
An API is like a specific way to engage with a store
An API describes store hours

MODULE 3 / LESSON Five / Part Two: RESTful routes

REST defines certain standard ways that we should interact with a back-end. There are specific URL patterns that are generally used to **CRUD** the back-end. CRUD stands for **CREATE**, **READ**, **UPDATE**, and **DELETE** records. In our case, this could be how to create, read, update, and delete cars in our database.

We will display a simplified version of these RESTful API rules here. We will prefix each call with the HTTP verb (GET or POST) for each method.

WARNING: Do not be discouraged by these rules. When I first learned about RESTful APIs and routing, I read a certain chapter in a programming book about 10 times before I understood it. I will attempt to make this easier for you by simplifying it.

YOU DO NOT NEED TO MEMORIZE these rules. This is here to expose you to the fact that these rules exist. We will provide a "cheat sheet" to these rules throughout the rest of the course so that when we're writing the routes, you can use that as a guide.

GET /cars => gets a list of cars
POST /cars => creates a new car based on the bundle of data you sent it
GET /cars/:id => gets one car based on the id you sent it as a wildcard
POST /cars/:id => updates a car based on a bundle of data to update a car
POST /cars/:id/delete => deletes a car

There are other HTTP verbs like PUT and DELETE, but since web pages only understand GET and POST, we are using these verbs for simplicity.

> The GET and POST patterns we're showing above are called **RESTful routes.**

MODULE 3 / LESSON Five / Part Two : RESTful routes : Quiz

Multiple choice

RESTful routes

Are a standard way to communicate with an API
Are generally GET routes
Only relate to the front end
Use GET requests to insert records into a database
Use POST requests to get collections of data

MODULE 3 / LESSON Six: Making our NEST routes
MODULE 3 / LESSON Six / Part One : Intro

Introduction to building our NEST Routes

We are finally ready to start building our server. In this lesson we are going to build our entire server. The only thing we won't do yet is connect it to our database. We are going to "hard code" all of the responses to CRUD our cars resource. Remember, CRUD stands for 'create', 'read', 'update', and 'delete' resources on the back end. When we are done, we will have routes that:

1. Create a new car (*Create*)
2. Retrieve all cars (*Read*)
3. Retrieve one car (also *Read*)
4. Update one car (*Update*)
5. Delete a car (*Delete*)

Let's do it!

> Ability to CRUD a resource is a foundational skill in any full stack application.

MODULE 3 / LESSON Six: Making our NEST routes : Intro: Quiz

Type in

Complete the full name of each step of the CRUD process

C_____
R_____
U_____
D_____

answers: **reate**, **ead**, **pdate**, **elete**

MODULE 3 / LESSON Six / Part Two :Making a GET ALL resource

Making a GET ALL resource

We already made and hard coded our 'GET /cars' route earlier, so we will review it here.

The code for this lives here on GitHub:
https://github.com/Nmuta/nest_cars_json/blob/master/src/cars/cars.controller.ts

And here's the code for review:

```
import { Controller, Get, Req, Request} from '@nestjs/common';

@Controller('cars')
export class CarsController {
  @Get()
  findAll(@Req() request: Request): {} {
    return [{make: 'honda', model: 'accord'},
            {make: 'subaru', model: 'outback'},
            {make: 'fiat', model: '123 spider'}];
  }
}
```

Remember, this is the root route for cars, and here's our cheat sheet:

GET /cars => gets a list of cars
POST /cars => creates a new car based on the bundle of data you sent it
GET /cars/:id => gets one car based on the id you sent it as a wildcard
POST /cars/:id => updates a car based on a bundle of data to update a car
POST /cars/:id/delete => deletes a car

~~We've made the current route bigger to emphasize it. We just built that.~~

> GET /cars is the root route for the cars folder. It returns a collection of cars.

MODULE 3 / LESSON Six / Part Two :Making a GET ALL resource : Quiz

Multiple choice

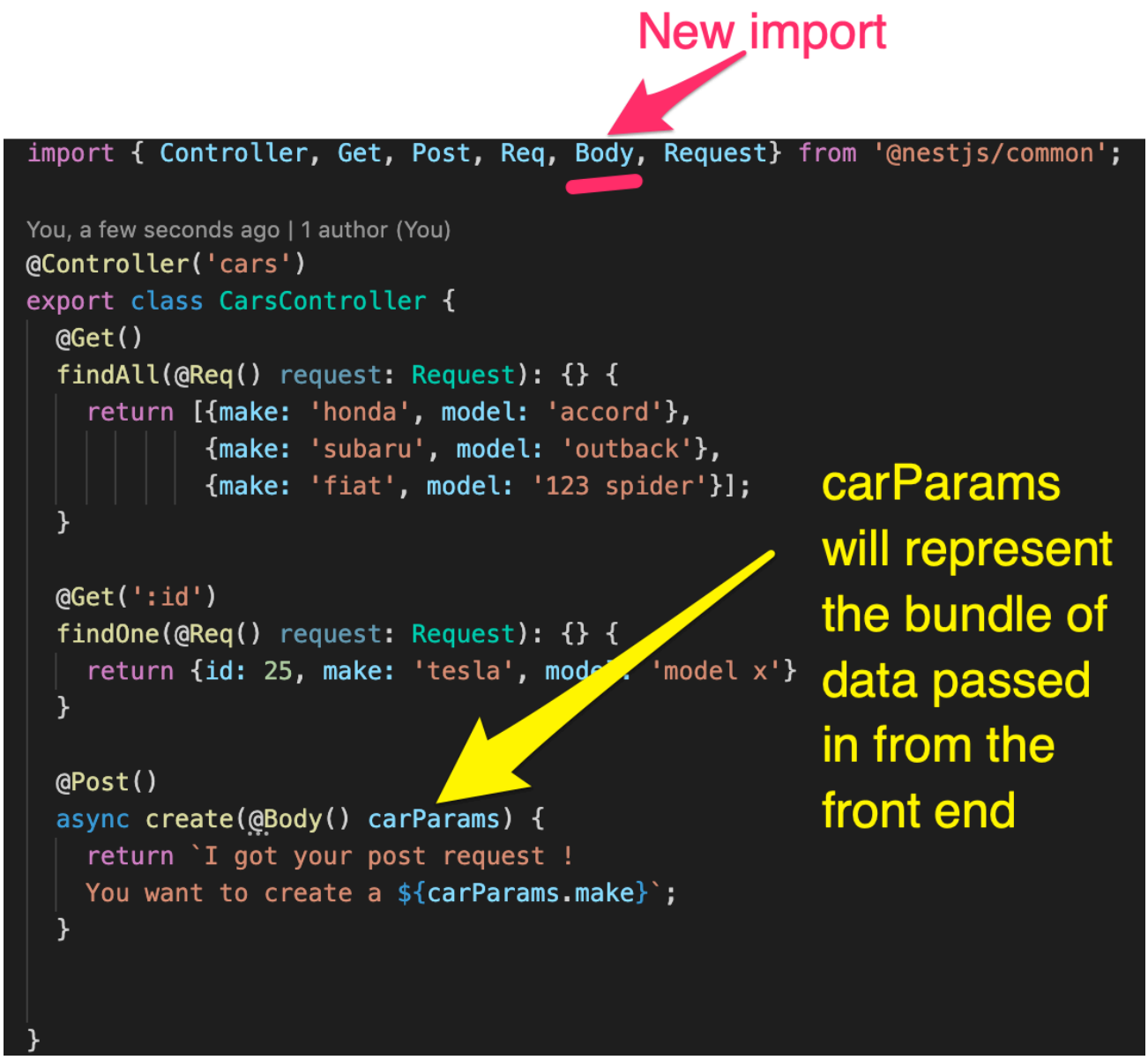
By convention, in RESTful routes, which of the following should return a collection of stamps?

GET /stamps/:id
GET /stamps
POST /stamps
POST /stamps/:id
GET /stamps/:id/delete

MODULE 3 / LESSON Six / Part Three: Making a POST resource

Let's make a route that accepts a bundle of data that will eventually be inserted into a database.

This will be a different kind of route. It needs to accept what we call a **Body** of data. Think about it … if you want to add a new car to a list of cars, you need more than just the make of the car, right? You need at least a make, a model, and in our case, the mileage. Cars have an id but the database will eventually automatically assign an id to the car. So we really just need to accept make, model, and mileage. This **Body** will always come in the form of a **JavaScript object** that has been sent from the front-end. Let's make our route to receive the data:



In the "create" method, we have an **@Body** decorator, which sets us up to declare a variable named whatever we want it to be to serve as a container for the data. We chose carParams to be this container. When building a web application server, the **BODY** of a POST request is the bundle of data coming from the user. This data is captured with the @Body decorator.

The create method is **async**, which means that the method is **asynchronous**. **Async / await** is a JavaScript construct that allows you to create "async" methods (methods that wait for a process to

finish before returning a result). In this case, we don't need to wait for anything because we're returning a string, but when we fully flesh out this method, it will need to wait for a database query to finish before it returns a value. That is why we use **async** on a **post** create method.

The route will accept car params and then return a string that says "I got your post request! You want to create a _____". (it then inserts the make from the **Body** params). This confirms that our route RECEIVED the data and also it confirms that the data has in it what we expect to be in there in terms of a make. If the make is present, then we can assume for now that the object is a valid object.

~~In real life, we would just return a 200 "ok" status code to say the code was received after we received the data and then dumped it into our database for storage.~~

This GitHub repository has the code where you can examine our POST route:
https://github.com/Nmuta/nest_cars_post_route

> A POST request to a resource like /cars will create a new car. Remember, it has to have a bundle of data attached to the POST from the front-end which becomes part of the Body when received by NestJS.

MODULE 3 / LESSON Six / Part Three: Making a POST resource : Quiz

Reorder

Describe the data journey associated with a POST request by rearranging the steps:

Front-end sends a bundle of data to a POST route
Back-end receives POST request
POST request is sent to appropriate POST route
POST route receives bundle of data and encodes it in a temp variable
POST route does something with received data
POST route sends a response

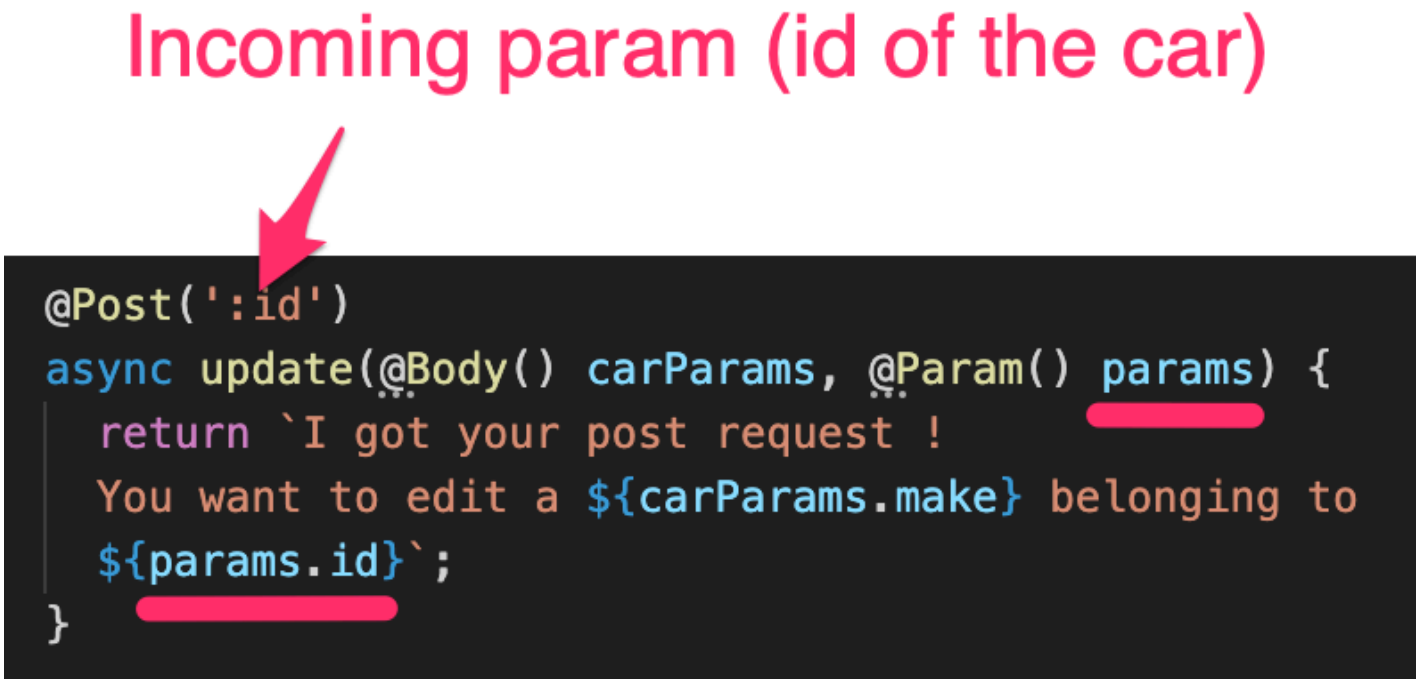
MODULE 3 / LESSON Six / Part Four: Making an EDIT resource

An edit (or "update") resource also involves a POST request because you need a bundle of data to be sent in to tell us which car to update and what the update should entail. Let's look at our cheat sheet to see how to write this route:

GET /cars => gets a list of cars
POST /cars => creates a new car based on the bundle of data you sent it
GET /cars/:id => gets one car based on the id you sent it as a wildcard
POST /cars/:id => updates a car based on a bundle of data to update a car
POST /cars/:id/delete => deletes a car

~~Bingo!~~ We need a POST route that looks like this:
POST /cars/:id

So here it is:



This is almost identical to our create route, except that we've added a new argument to the route function. We use the **@Param() decorator**, which we imported, and we called it "params" but it could really be anything. Examine the red marks in the diagram above starting with the :id in order to follow the flow of the parameter we are passing in through the URL.

We are making a POST request to a route that looks like this:

/cars/5

5 is our :id when passed into this function. Any wildcards passed in through the url are stored in params (the first underlined red value). Then, anywhere within the method, we can call params, a dot, and then the value we want to invoke, which is, in this case, the id. How does it know that it's called an id? Because that was what we named it in @Post(':id').

Finally, we return a string that reveals the name of the make of the car AND the id of the car that we will be updating.

Here's the full GitHub code:
https://github.com/Nmuta/nest_cars_update

> Both **update** and **create** routes need to be POST requests because they have a **BODY**, a bundle of data containing the information needed to **create** and **update** records respectively.

MODULE 3 / LESSON Six / Part Four: Making an EDIT resource : Quiz

Multiple choice

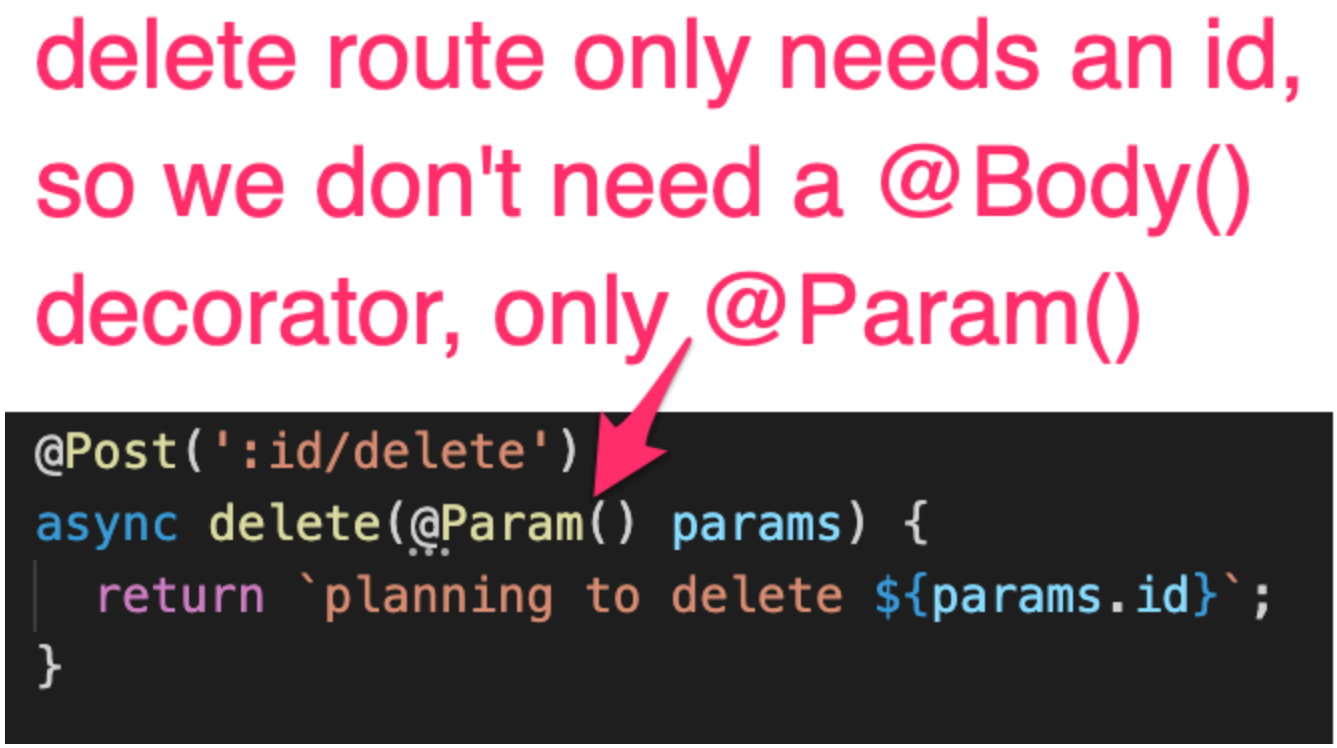
Why do we need to accept an id when updating something ?

We have to do an id check in case the user is under age
We will use the id to look up the record in the database
An id is required for any and all database transactions
The id will be inserted into the database
The id is the only value in the record that will change

MODULE 3 / LESSON Six / Part Five: Finishing all of our routes

We now have seen enough of routing to complete the rest of the routes. We have completed all of the routes from our cheat sheet and put the routes in our NestJS **controller**.

The only remaining route needed is our **DELETE** route. We only need to accept an ID to delete something from the database (no bundle of data is needed) . You don't need any information to delete a car. The make, model and miles don't matter. All the database needs is an id to identify the record that needs to be nixed.



We have added a GitHub repository that has all of the routes that we need, so the infrastructure for all of our routes to CRUD cars is now complete.

https://github.com/Nmuta/nest_cars_all_routes

Specifically, the link below is a direct link to the controller. We put our cheat sheet in the controller for your convenience.

https://github.com/Nmuta/nest_cars_all_routes/blob/master/src/cars/cars.controller.ts

> You will probably not have all of the routing memorized even after reading it a few times. This is normal. The most important thing is understanding the concept of get and post routes and the concept of get and post requests. The rest will slowly sink in with time and usage.

MODULE 3 / LESSON Six / Part Five: Quiz

Multiple choice

Why do we only need to pass an id in a "delete" POST request ?

We use the same id to delete all records
Ids are case sensitive
All we need is the id to locate the record to delete
The id will gather @Body() parameters that are needed

MODULE 3 / LESSON Seven : Setting up our database
MODULE 3 / LESSON Seven / Part one: Creating and populating a database

In "data's long journey", we have finally reached the deepest data layer on our stack: The **database**. ~~This calls for a celebration!~~

In our terminal we will simply create a PostgreSQL database called "transportation". Then, inside of PostgreSQL, we create a cars table and populate it with three cars. Creating tables and populating them was covered in a previous section, so we are simply posting all of the commands here for your convenience.

Here are the commands written out in order:

```
createdb transportation
psql  // gets you into the psql command line
\c transportation
CREATE TABLE cars (id SERIAL, make TEXT, model TEXT, miles INT);
INSERT into cars values (DEFAULT, 'subaru', 'outback', 3420);
INSERT into cars values (DEFAULT, 'honda', 'passport', 27);
INSERT into cars values (DEFAULT, 'volvo', 'XC40', 33000);
select * from cars;
```

And here's what it looks like when we do it in the terminal :

```
➜  cars git:(master) ✗ createdb transportation
➜  cars git:(master) ✗ psql
psql (12.1)
Type "help" for help.

user100 =# \c transportation
You are now connected to database "transportation" as user " user100 ".
transportation=# CREATE TABLE cars (id SERIAL, make TEXT, model TEXT, miles INT);
CREATE TABLE
transportation=# INSERT into cars values (DEFAULT, 'subaru', 'outback', 3420);
INSERT 0 1
transportation=# INSERT into cars values (DEFAULT, 'honda', 'passport', 27);
INSERT 0 1
transportation=# INSERT into cars values (DEFAULT, 'volvo', 'XC40', 33000);
INSERT 0 1
transportation=# select * from cars;
 id |  make  |  model   | miles
----+--------+----------+-------
  1 | subaru | outback  |  3420
  2 | honda  | passport |    27
  3 | volvo  | XC40     | 33000
(3 rows)
```

Eureka! We now have a cars database with 3 cars in it.

> Our database is our deepest level of data storage.

MODULE 3 / LESSON Seven / Part one: Creating and populating a database: Quiz

Drag n Drop

Complete the following SQL statement to insert a book into a PostgreSQL books table in a database that has the following fields in order. Assume that the id field is an auto-incrementing (serial) value.
The fields in the book table, in order, are: **id, title, subtitle, author, isbn**

_____ into _____ _____ (DEFAULT, "Goodnight June", "A quiet book", "April Johnson", " 978-3-16-148410-0")

INSERT **books** **VALUES** INJECT PUT properties

MODULE 3 / LESSON Eight: TYPE ORM
MODULE 3 / LESSON Eight / Part one: TYPE ORM intro

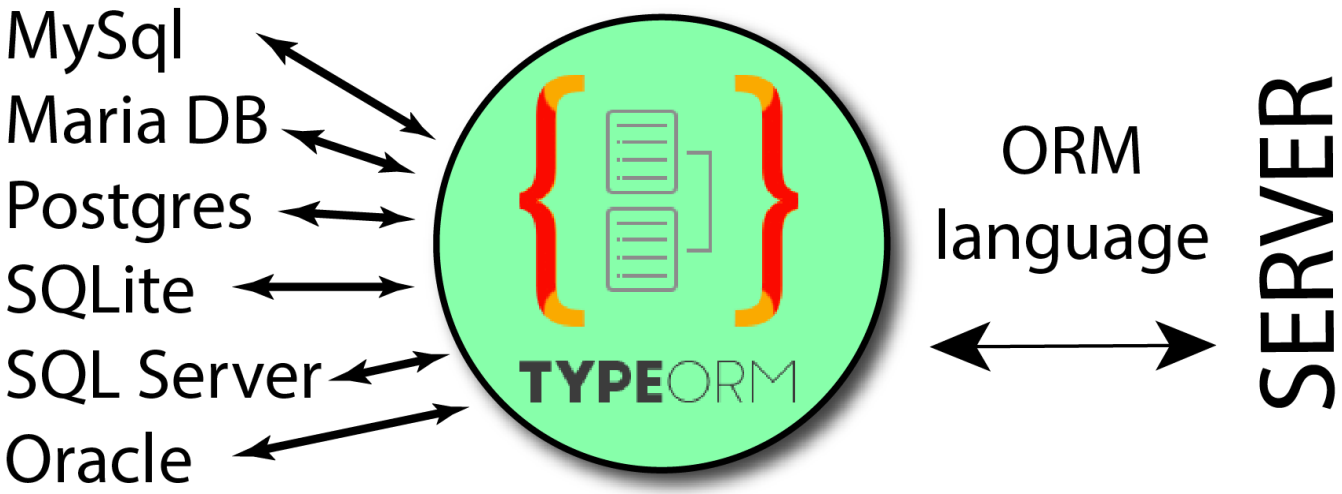
TYPE ORM intro

We're finally to the point in our course where we can start interacting with our database from our NestJS back-end.

In order to interact with our database from NestJS, we will use an ORM named TypeORM. As we learned in a previous module, **ORM** stands for "Object Relational Mapper". It's the glue between the server and the database.

Why do we need an ORM? Can't the web application server talk directly to the database? Well, it can, but the purpose of the ORM is to make back-ends more flexible by serving as a "universal translator" between the server and the database.

ORMs speak one language to the server: the ORM language. But they speak MANY languages to the different databases. This means that you write one language in your back-end, the ORM language, and then the ORM, with its multilingual capabilities, will TRANSLATE the ORM language to all of the database types that it supports.



What this means is that you can switch the database from a PostgreSQL database to a MySQL database on the backend and not have to touch your database queries in your NestJS server, because your ORM will do the translation for you.

> An ORM is essentially a multilingual database communicator.

MODULE 3 / LESSON Eight / Part one: TYPE ORM intro: Quiz

Drag N Drop

The job of an ORM is to provide universal _____ services between a web application server and a _____ .

answers: **translation** **database** server front-end typing typescript

Revised quiz:
T / F

By using an ORM, like TypeORM, we are able to change databases while not having to change database queries to the server.

MODULE 3 / LESSON Eight / Part two: Get All Route

The NestJS team has done a great job of providing **TypeORM** examples ready for you to use in your project. They have a good SQL code example in Github here:

https://github.com/nestjs/nest/tree/master/sample/05-sql-typeorm

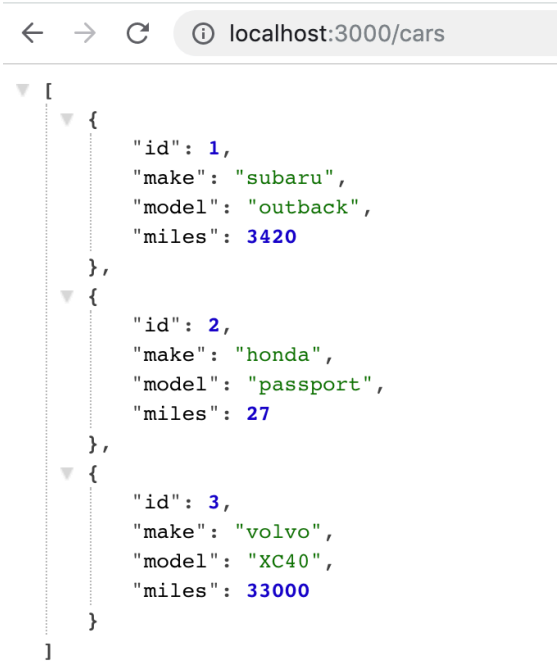
We've taken that example ~~which is built for MySQL and a "photos" database,~~ and adapted it to our PostgreSQL **transportation** database. Our starter file repo is here:

https://github.com/Nmuta/nest_typeorm-postgres

Cloning the above repo (nest_typeorm-postgres), and following the directions in the ReadMe, will lead you to run the server, which will appear in http://localhost:3000

We have no "root" route, since this server is only an **API**. Remember, API stands for "application programming interface", and it's like a menu in a restaurant. What do we want? We want a list of cars. Well let's fire up the route.

If we hit the **/cars** route, we get this :



It works! But why does this work? Well, one thing is that in our TypeORM setup, we have told it:

1. We are looking for a PostgreSQL database
2. Our PostgreSQL database is named transportation
3. Our transportation database has a table named 'cars' with id, make, model, and miles columns

In the next part of this lesson we will analyze HOW we tell TypeORM those three things.

> ORMs usually require just a few key pieces of data like those three points above to lock into your database and start using it. It's like a "just add water" pancake mix.

QUIZ added

MODULE 3 / LESSON Eight / Part two: Get All Route: Quiz

How does an ORM know where to pull data when querying in an underlying database?

TypeORM can be configured to know the name of the database and the table we are targeting
We hard code SQL into all of our controllers
Angular makes the database connection for us
Our API connects to the database for us

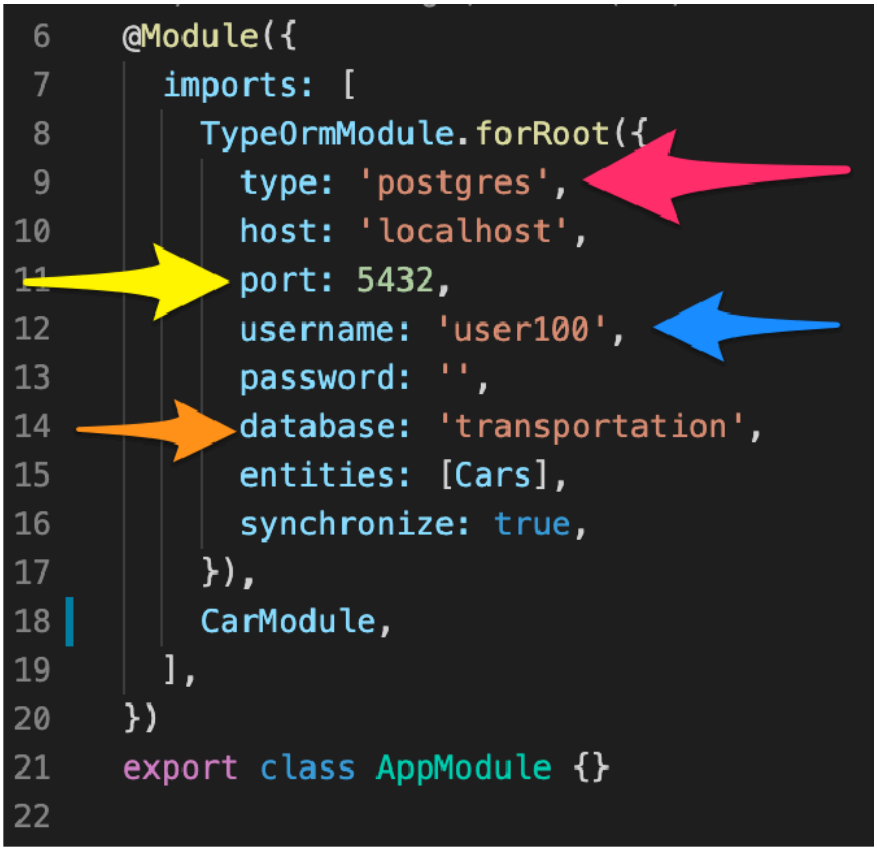
MODULE 3 / LESSON Eight / Part three: Type ORM : connecting with postgres

TypeORM : connecting with PostgreSQL

We pinpointed these three things as the key to helping TypeORM connect to our existing PostgreSQL database. The first step was to "tell TypeORM that we're looking for a PostgreSQL database".

In our app.module.ts file we see these settings:

https://github.com/Nmuta/nest_typeorm-postgres/blob/master/src/app.module.ts



From top to bottom, we have to:

1. tell it that the "type" is 'postgres' around line 9
2. Tell it that the port is 5432 around line 11. The default port for postgres on your machine should be 5432. If not, you can go into your postgres command line by typing psql and enter the following command :

```
\conninfo
```

which will give a response something like this :

```
You are connected to database "transportation" as user "p2873541" via socket in "/tmp" at port "5432".
```

3. Tell it that the username is **whatever your postgres username is**
4. Tell it that the database is named 'transportation'

> The app.module file in NestJS, just like in Angular, has some config properties that affect the entire application.

MODULE 3 / LESSON Eight / Part three: Type ORM : connecting with postgres: Quiz

Multiple Choice

In which of the following files do you tell NestJS where to establish a database connection for TypeORM?

app.module.ts
app.component.ts
main.ts
main.module.ts

MODULE 3 / LESSON Eight / Part four: Type ORM : connect with the cars table

Next we need to tell TypeORM that we are connecting to the cars table within the transportation database. If you look at the top of the app.module.ts file, you'll see that we're importing something called a Cars **entity**:
https://github.com/Nmuta/nest_typeorm-postgres/blob/master/src/app.module.ts

```
import { Cars } from './car/cars.entity';
```

This file maps out all of the database fields that live on our cars table in our database:

```typescript
import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

You, 2 hours ago | 1 author (You)
@Entity()
export class Cars {
  @PrimaryGeneratedColumn()
  id: number;

  @Column('text')
  make: string;

  @Column('text')
  model: string;

  @Column('int')
  miles: number;
}
```

> An "Entity" in TypeORM is like a database schema. It maps out all of the fields that we need to communicate with our underlying tables.

MODULE 3 / LESSON Eight / Part four: Type ORM : connect with the cars table : Quiz

Drag N Drop

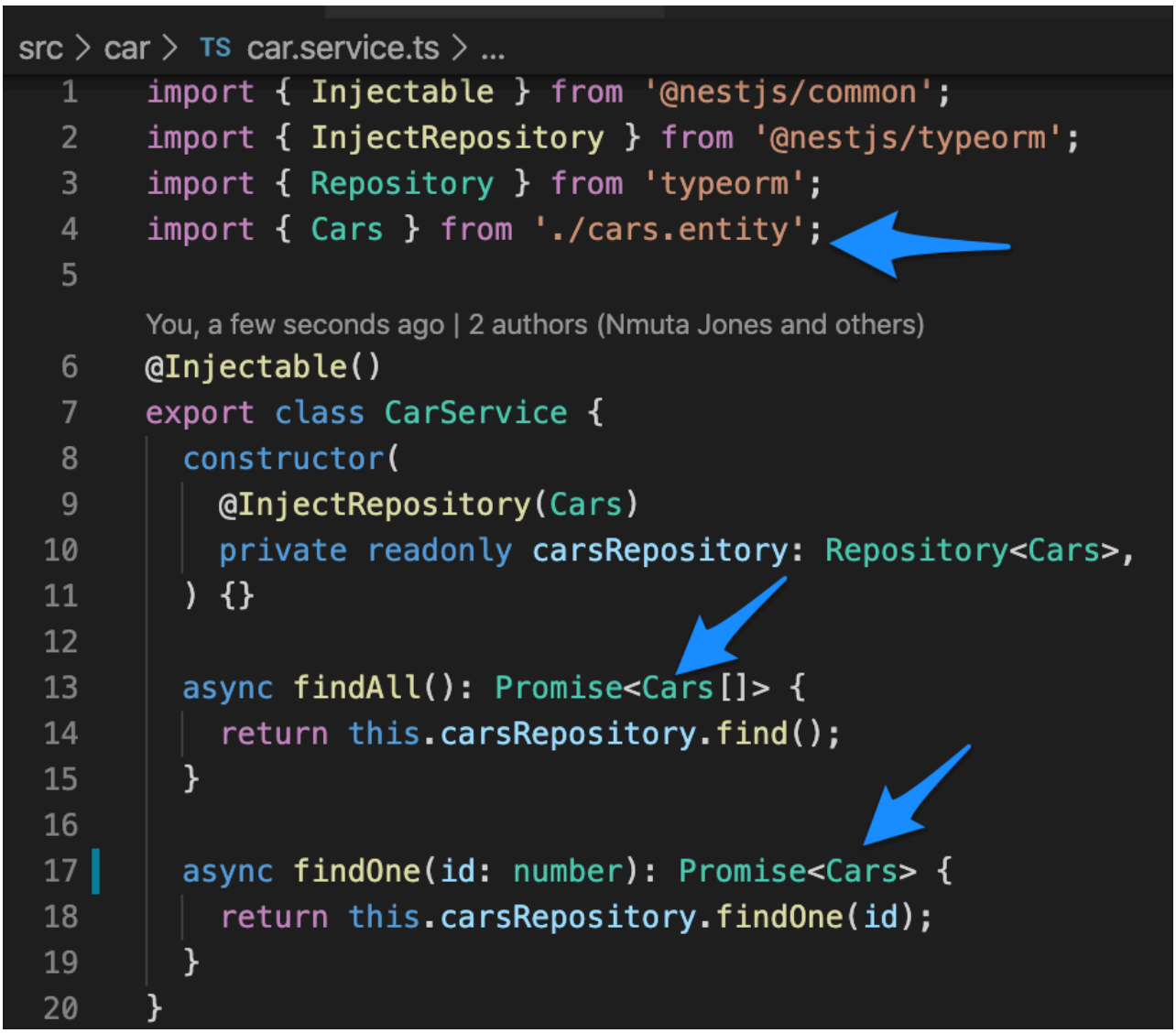
An "Entity" in Type ORM helps us to interact with which of the following?

answers: **database table** controller module

MODULE 3 / LESSON Eight / Part five: The car service

Lastly, in terms of our TypeORM setup, we should look at our car service.

https://github.com/Nmuta/nest_typeorm-postgres/blob/master/src/car/car.service.ts



The car service, like any service in Angular, is a bridge between the **controller** and the **data** sources.

Don't feel intimidated if you don't understand everything that is in this file. We will direct you to the pieces that you need to know in order to navigate through the files and only introduce you to what you need to know at any given time to be productive.

In this case, the service imports something called a **Repository** from the TypeORM source code. The **Repository** is the main brain of the database query machine in NestJS. It has all of the 'magical' methods to query the database. Our repository here is of a **Repository** type, which means that it inherits all of **Repository's** cool methods and we can use them to query a database.

Repository has methods like **find()**, **findOne()**, **create()**, **insert()**, **update()**, etc.

In our **/cars** route, we simply call find() and we get our entire collection of cars. Under the hood, since we told it that we are connected to PostgreSQL, it's constructing a query something like:

"SELECT * from cars;"

and sending that query to PostgreSQL.

Let's focus on what our async methods return here. These methods return a standard Javascript Promise. The return type of the promise is a Cars[ ] type for the findAll method on line 13, but a Cars type on line 17. Here, Cars represents the database response for a single car (it's plural, but it's really the 'model' for one car response from the database. So we use **Cars[]** for the collection and **Cars** for the single car response.

> Services in NestJS, when working with TypeORM, feature methods that can return promises of a type associated with a particular TypeORM entity.

MODULE 3 / LESSON Eight / Part five: The car service: Quiz

Multiple choice

What is the role of a Repository in TypeORM ?

It holds all of the database query methods we will need to use
It translates TypeScript into JavaScript
It stores all of the cars in our database model
It inserts data into our Angular app

MODULE 3 / LESSON Nine : Finishing data's journey
MODULE 3 / LESSON Nine / Part one: controller findOne method

controller findOne method

Our current controller already has a findOne method:
https://github.com/Nmuta/nest_typeorm-postgres/blob/master/src/car/car.controller.ts

Let's review the controller code.

car.controller.ts

```
@Get("/:id")
findOne(@Param() param): Promise<Cars[]> {
  return this.CarService.findOne(param.id);
}
```

The GET /cars/:id method accepts an id. The colon tells us that this is a variable value. Remember that "/cars/" prefix to this method is already included since we're in the cars controller.

The findOne method (which is the method associated with the GET cars/:id route) invokes the @Param() decorator with its associated param local variable. **IMPORTANT**: 'param' is simply what we decided to call the parameter object. This variable is a local method variable and can be called *anything*. If it were named "**scooby**", then on line 16, we would write findOne(**scooby**.id) . @Param(), however, is a decorator and keyword and will always be the same.

We then call the CarService's findOne method with the param.id (the :id variable from the route)

The CarService then does the rest of the work. That will be covered in the next part.

> Remember: the controller is the first point of response when a GET or POST request comes in from the front-end.

MODULE 3 / LESSON Nine / Part one: controller findOne method

TYPE IN

complete the following code so that the id coming in from the GET request is passed on to the FoodService

```
@Get("/:id")
  findOne(@Param() incoming): Promise<FoodItem[]> {
  return this.FoodService.findOne(_________);
}
```

answer: incoming.id

MODULE 3 / LESSON Nine / Part two: service findOne method

service findOne method

The controller has passed the incoming id to the **findOne( ) method** in the service:

https://github.com/Nmuta/nest_typeorm-postgres/blob/master/src/car/car.service.ts

car.service.ts

```
async findOne(id: number): Promise<Cars> {
  return this.carsRepository.findOne(id);
}
```

We accept the id from the controller.

The carsRepository calls its own **findOne** method, which accepts an id and uses its multilingual self to speak to the underlying database (in this case, it's PostgreSQL), retrieve a car, and send it back to the controller.

You've probably noticed that many of these methods return **Promises**. While Promises are an important part of JavaScript, understanding how they work in these instances are not critical to getting this entire application to work.

What IS important, however, is understanding the data flow here. The controller gets the id, passes the id to the service, which calls the repository, calls the database, which gets the item and returns it back to the controller, which returns JSON to the front-end. The long road of data is complete when all of this has occurred.

> Remember: the repository has all of the methods we need to speak to our database. You can view our sample repository here: https://github.com/Nmuta/nest_typeorm-postgres/blob/master/repository.ts. This file contains all of the methods that we would need to fully CRUD any resource, like cars or ramen noodles.

MODULE 3 / LESSON Nine / Part two: service findOne method: Quiz

Multiple choice:

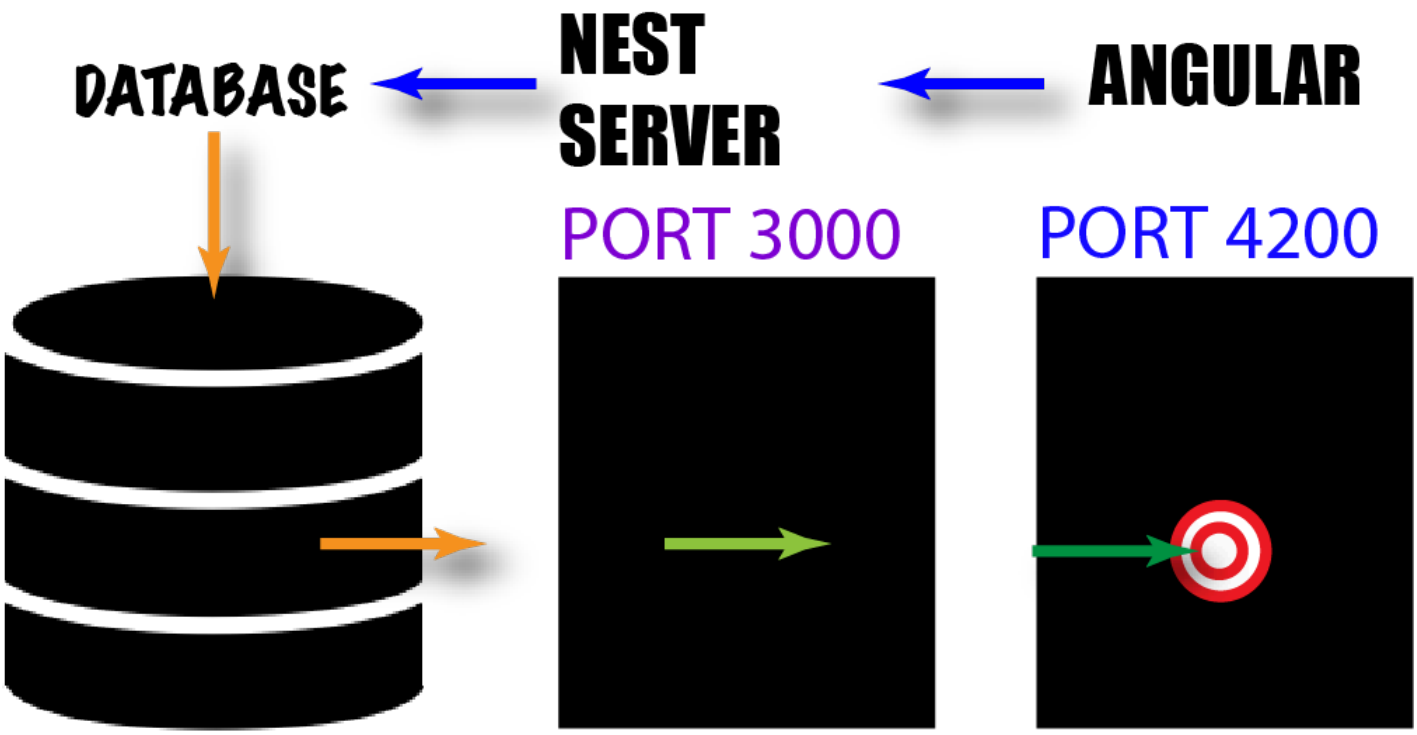
(check all that apply)

In NestJS one key difference between a service and a controller is which of the following:

The controller receives GET requests from the front-end
The service receives GET requests from the front-end
The service interfaces with a repository of database methods
The controller interfaces with a repository of database methods
The controller makes POST requests to the front-end

MODULE 3 / LESSON Ten : Front end meets back end
MODULE 3 / LESSON Ten / Part one: Client and server together

We have reached the moment of truth. All we need to establish is the ability to send a GET request from Angular (on the right side of the diagram), have NestJS receive the request, get the data from the database, return it to NestJS and then send that data back to our Angular front-end.



In order to do this, we need to run both the "**client**" (Angular) and the "**server**" (NestJS) at the same time on our computer: Angular running on its default port, which is port 4200, and NestJS running on its default port, which is 3000.

We can look at our computer as one big "sandbox" or test bed where we simulate what would happen in the real world when these applications are living and being used by consumers. In the "real world", the **server** would be running the cloud somewhere and the **client** would be either a web browser, your mobile phone, a tablet, or any other consumer facing device.

> "client" and "server" are universal computing terms that refer to a place where data is served from and the consumer of that data.

MODULE 3 / LESSON Ten / Part one: Client and server together: Quiz

Drag n Drop

In web applications, a _____ on the front end consumes data that is delivered by the _____ on the back end

client **server** port database sandbox

Which of the following refers to the "server" in our app?

Angular
NestJS
JavaScript
TypeORM

MODULE 3 / LESSON Ten / Part two: Data's journey complete

Data's journey complete

Previously, our client (Angular) was hard coding its own data in the service to get cars. How boring is that? We don't need to hard code our data anymore. We will now connect our Angular front-end to our NestJS back-end by making an **XHR** (XML/HTTP Request) from Angular to our NestJS back-end to get the car data.

Let's modify our Angular service to make a real XHR to our back-end instead of hard coding the data.

We'll resurrect our old StackBlitz code as an example, but this code would normally live on an Angular app on your machine.

Here's our example service, the "old" way of hard coding the data:
https://stackblitz.com/edit/angular-send-dynamic-data-to-svs?file=src/app/transportation.service.ts

Previously in our transportation service we had:

```
getCars() {
  return this.cars;
 }
```

Let's change that to:

```
getCars(): Observable {
  return this.http.get('/cars/');
 }
```

What is this doing? Instead of returning a hard coded value, we are returning an **Observable**, which is a stream of data. If you're familiar with JavaScript promises, an Observable is similar to a promise. It's essentially **asynchronous** data, which means that since it has to travel over a network, it doesn't arrive instantly, so we just "keep the door open" and wait for the data to return after we've requested it.

Lastly, all we have to do is add these two headers to the top of the file:

import { HttpClient} from '@angular/common/http';
import { Observable } from 'rxjs';

Those two lines import our ability to use XHR and Observables, respectively.

Finally, if we refresh our Angular page now, we get :

We have a subaru outback with a mileage of 3420
We have a honda passport with a mileage of 27
We have a volvo XC40 with a mileage of 33000

Our data is now coming all the way from our database to Angular! **Data's full journey is complete.**

> Having server code that's separate from client code is called a "**decoupled app**" and is by far the most common way of delivering data in today's rapidly evolving digital world, since many different "front-ends" exist that consume back-end data. We will put together a real, working flights app in the next module where we will use GET and POST requests to connect a front and a back end.

MODULE 3 / LESSON Ten / Part two: Data's journey complete : Quiz

multiple choice

In order to make a service method use http to call a back-end we have to make the return type on the method be which of the following?

a string
an Observable
http
null
a GET request

MODULE 3 / LESSON Ten / Part THREE: Post Request Example

MODULE THREE EXAM

1.

Which of the following is NOT a function of a controller in NestJS?

communicating with a repository
handling GET requests
handling POST requests
communicating with a service

2.

~~Drag n drop~~

~~**On the front-end, a GET _____ is sent to the server via an XHR, which is then handled by a GET _____ within the back-end.**~~

~~**request route** POST object controller~~

Revised quiz:
On the front-end a GET request is sent to the server via an XHR which is then handled by what within the back-end?
GET route
POST request
POST route
object
controller

3.

drag n drop

When you run a NestJS server locally, it typically runs on _____ 3000.

Answers:
port gateway ORM interface

4.

Which of the following routes should get a collection of flowers?

GET /flowers/:id
POST /flowers/
GET /flowers/
POST /flowers/:id

5.

ORDERING

Scenario:

A user clicks on a button in the front-end to get a list of cats.

Order the responses below, from top to bottom, in terms of the order of the most likely request / response flow. The item at the top should be the first thing that happens and the item at the bottom is the last thing that happens in the flow.

-- User clicks on button in the front end
-- server accepts request as valid
-- server goes into the database to get cats
-- server returns the list of cats as JSON
-- Client translates JSON into pretty display

6.

Which of the following is true about a POST request?
Select all that apply

It sends a bundle of data to the back end
It is always different than a GET request
It connects to a POST route on the back end
It is used when adding an item to a database
It functions just like a GET request with the same name

7.

Drag n Drop

_____ is a multilingual database communicator.
TypeORM NestJS A service Postgres

Which of the following is considered a multilingual database communicator?

TypeORM
NestJS
A service
PostgreSQL

8.

Drag n Drop

In web development, a front-end _____ communicates with a *server* on the back-end.

answers :

client ORM object database entity

9.

Multiple choice

Which of the following constructs helps us WAIT for data until the back-end call for data has returned and is ready to display?

A GET request
A POST request
An Observable
A controller

10.

Type in:
Complete the Entity that maps to an existing student table in a database.

```
@Entity()
export class Student {
  @PrimaryGeneratedColumn()
  id: _________;

  @Column('text')
  first_name: _______;

  @Column('text')
  last_name: _______;

  @Column()
  hasRegistered: _______;
}
```

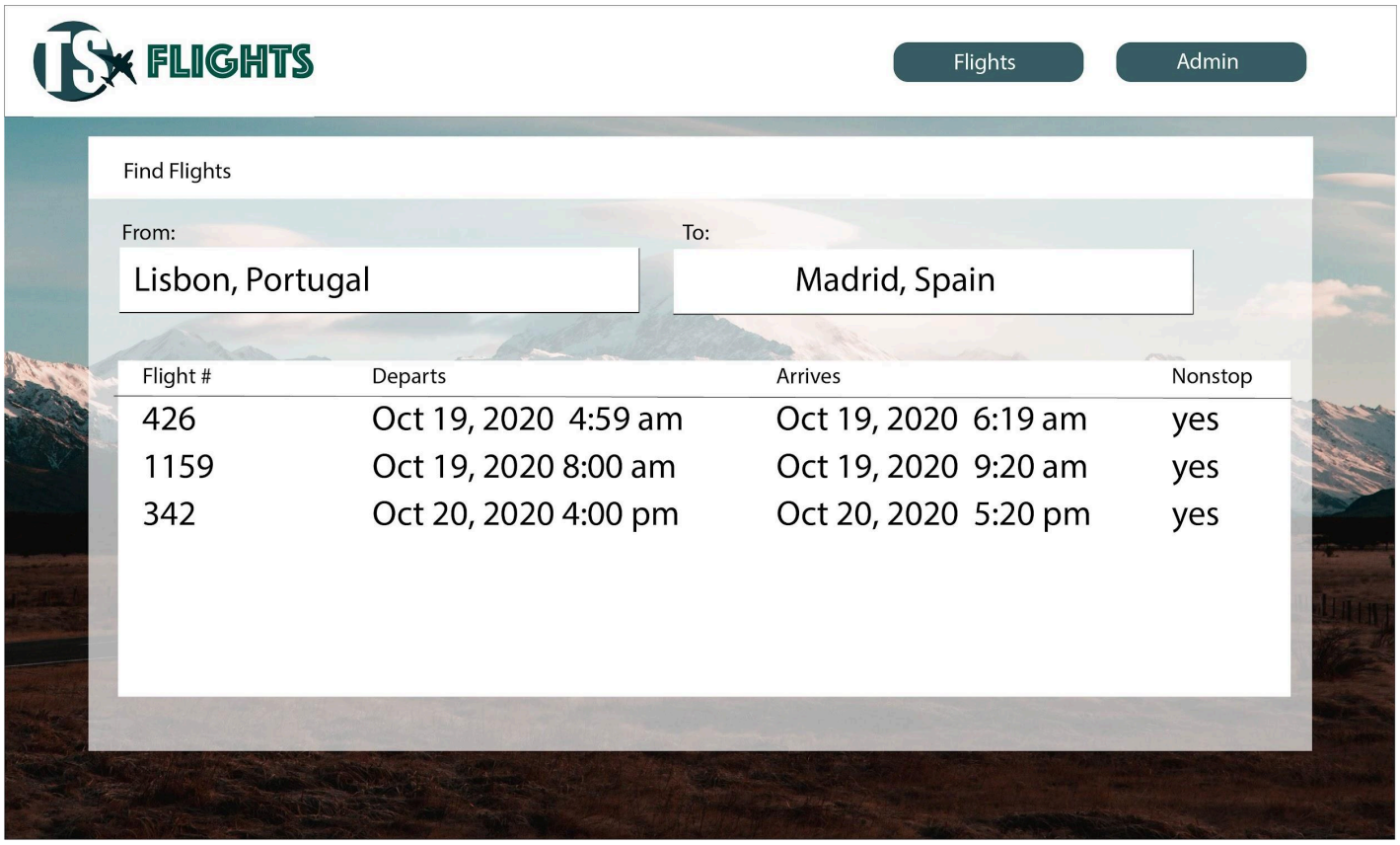
answers: **number string string boolean**

MODULE SIX :
PUTTING IT ALL TOGETHER

MODULE 6 / LESSON One / Part one: Overview

Congratulations, you now have enough of a foundation to build a full stack application using Angular. In this module, we will be building a full stack application called TSFlights.

We will be learning some new concepts along the way, like Angular routing, and we will be exposing you to some peripheral subjects like SCSS stylesheets, so there is still new content to learn. However, most of the foundational skills required to build the app have already been covered.



To summarize our technology stack, we will be using Angular for our front-end, SCSS for our stylesheets, (SCSS is a superset of CSS), NestJS for our back-end, and a PostgreSQL database. What's great about open source full stack development is that many of these elements are interchangeable. Down the road, you could switch out the PostgreSQL database with a MySQL database and the application would still work beautifully. You could even replace the Angular front-end with another JavaScript framework, like Vue.js, and the back-end would not even know the difference. This is what it means to build a **decoupled app**. Each piece is somewhat autonomous.

> You may remember us mentioning in a previous module that a "decoupled app" is an application where the front end and the back end are separate. In this module, we will be building a decoupled app.

MODULE 6 / LESSON One / Part One: Overview : Quiz

~~Drag n Drop~~

~~Our application will feature a front end built with _____ , a _____ web application server, and a _____ database~~

~~**Angular NestJS PostgreSQL** MySQL Node.js Vue.js SCSS~~

Revised quiz: Our application will feature a front-end built with Angular, a NestJS web application server and which type of database?

- **PostgreSQL**
- MySQL
- Node.js
- SCSS

MODULE 6 / LESSON One / Part two: Upgrading Angular

When this course was written, Google was releasing a new Angular version every six months or so. The beginning of the course was written using Angular 8, but Angular 9 was released right as we were building this final module.

Have no fear. Angular versions since Angular 4 or so have all featured very little changes to the core syntax of the framework. Most of the changes from version to version at this point involve extra features, advanced functionality, improvements to the core rendering engine, and the deprecation of a few features along the way.

Since Angular 9 is almost identical to Angular 8 in terms of core syntax, we have built our GitHub repo examples using Angular 9. The functionality and behaviour of everything is pretty much the same.

In order to start using the latest version of Angular in your computer terminal simply type:

ng upgrade

This upgrades Angular to the latest version. In our case, this now became Angular 9. Any new projects started after this point will be using version 9 automatically, but all OLD versions of Angular down to Angular 2 will still work on our system.

> You can always check the version of any Angular project by inspecting the 'package.json' file that is in the repository. In the 'dependencies' category, there is an entry for "@angular/common". In our repo, that entry looks like this : `"@angular/common": "~9.0.2",` which signifies that we are using version 9.0.2

MODULE 6 / LESSON One / Part Two: Upgrading Angular : Quiz

~~Type in~~

~~In our terminal, we use the command _____ to upgrade our system to have access to the latest version of Angular.~~

~~**ng upgrade**~~

Revised quiz: What command would you type in the terminal to ensure your system has access to the latest version of Angular?

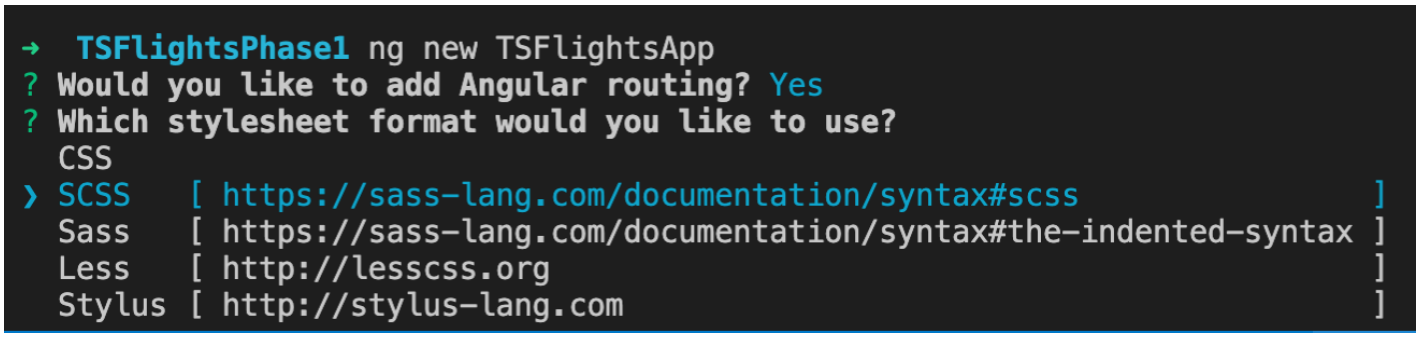
- **ng upgrade**
- upgrade ng
- ang upgrade
- upgrade ang

MODULE 6 / LESSON One / Part three: Scaffolding a new Angular app

Now that we've upgraded Angular, we can scaffold a new Angular app.

```
ng new TSFlightsApp
```

The 'ng new' command creates a new Angular project for us, and then we are given some choices. We have captured these choices in the image below. We said "yes" to Angular routing, and we used the arrow keys on the keyboard to select "SCSS" as our stylesheet format.



Routing allows us to have more than one page in our Angular app and navigate to each of those pages by clicking links that are made in a format that works with the Angular router. We will cover routes in the next lesson.

SCSS is a way of writing CSS that allows you to write nested CSS styles, use variables called mixins, and a bunch of other cool features. We will discuss this in a future lesson as well.

> The ability to create a new Angular app by using the ng new command is one of the most useful features of the Angular **CLI** (command line interface).

MODULE 6 / LESSON One / Part Three: Scaffolding a new Angular app : Quiz

Multiple choice

The ng new command

Upgrades Angular to the newest version
Tells you what the newest version of Angular is

Creates a new Angular application
Tells you what the newest features are of your ng version

MODULE 6 / LESSON One / Part four: Project resources

Let's take a minute to explore the project resources that will support you through this final journey of building the app or examining our files as we build this app.

You can choose to do this project on your own, or you can simply follow along by looking at the code in each iteration.

StackBlitz is the place where you will be able to see each iteration of the front-end in Angular. Since it is an online resource that has security restrictions, we won't be able to tie it to a real back-end, but we will mock out the functionality that the back-end provides us.

If you are following along on your phone, you will be able to view with the resources in StackBlitz and modify the code to experiment with new things.

Github is where we will have the full repositories for both our front-end and our back-end. This is where the full source code for both the front-end and the back-end will live.

> StackBlitz is a great way to run Angular code online and it is now being used by Google's Angular team as the Angular online playground of choice in the official Angular documentation.

MODULE 6 / LESSON One / Part Four: Project Resources : Quiz

Drag n Drop

_____ is a place where we can experiment with live front end code in a browser, where _____ is where we will have access to the full source code for both the front end and the back end .

StackBlitz **GitHub** GitLab Node.js NestJS

Revised quiz: The full source code for both the front-end and back-end will be accessible where?

- StackBlitz
- **GitHub**
- Node.js
- NestJS

MODULE 6 / LESSON Two : Routing
MODULE 6 / LESSON Two / Part one: Single Page Apps

Angular routing is based on the **single page app** concept. When the web was first created, most web sites consisted of several individual html pages. A common pattern was that each page would have it's own header, footer, and many other common elements shared with all other pages on the site. This was incredibly inefficient because there was a LOT of duplicated content. Every time you created a page you would create a clone of a previous page and simply change the "main content" of the page to something else. The 'about us' page was really a clone of the 'contact us' page with the middle contents or core content replaced. This made it easy for search engines to identify different pages on the web, but part of the inefficiency came from the fact that each new page had to 're-load' the same header and footer and other common content over and over.

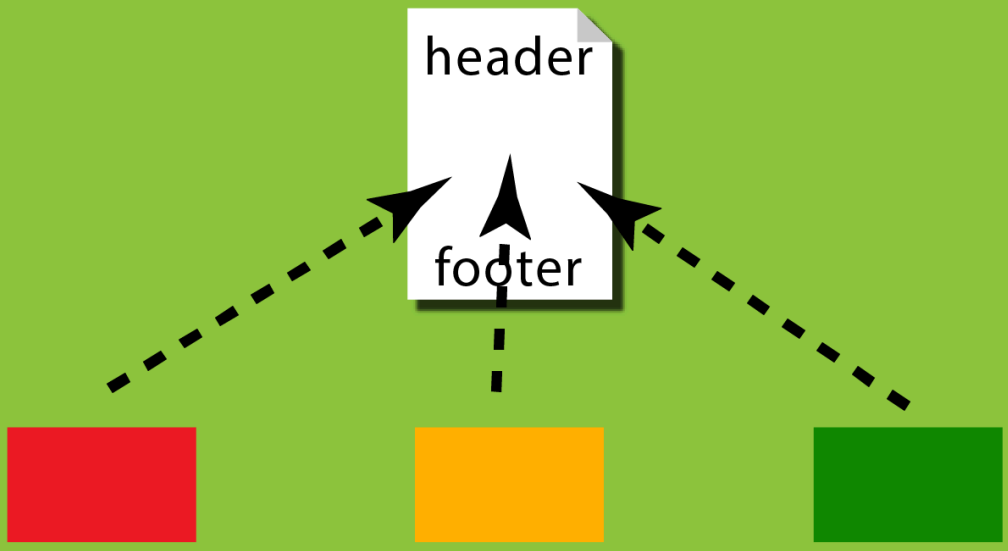
Single page apps fundamentally changed that paradigm. In a single page app, there is typically only ONE page on the website. That's it. The framework then loads customized "sub pages" or "partials" into designated sections on the page that need to have variable content. This makes single page applications really fast.

TRADITIONAL WEB APPLICATION

(MANY UNIQUE PAGES)

SINGLE PAGE APP

(ONE PAGE THAT LOADS DIFFERENT SUB PAGES)



Gmail was the world's first large scale single page app. If you have Gmail, load it up and click on any link within the Gmail app (read an individual email, visit the 'social' or 'updates'

tab, etc.). You will notice that it never actually loads a new page. The content on those sections is swapped out with the new content corresponding to what you clicked on. This is the classic example of a **single page app**.

MODULE 6 / LESSON Two / Part One: Single Page Apps: Quiz

Multiple choice

Which of the following is FALSE

SPAs load sub pages into one main page
Google used the SPA concept to build Gmail
SPAs are typically slower than traditional web apps
SPAs typically build an entire website on one page
SPAs cut down on duplicated content

MODULE 6 / LESSON Two / Part two: Making new routes

Our app is going to be very simple. We will have only two pages: "home" and "admin". Let's create the "home" route.

In Angular, you may recall that we can scaffold a home component by doing

```
ng generate component home
```

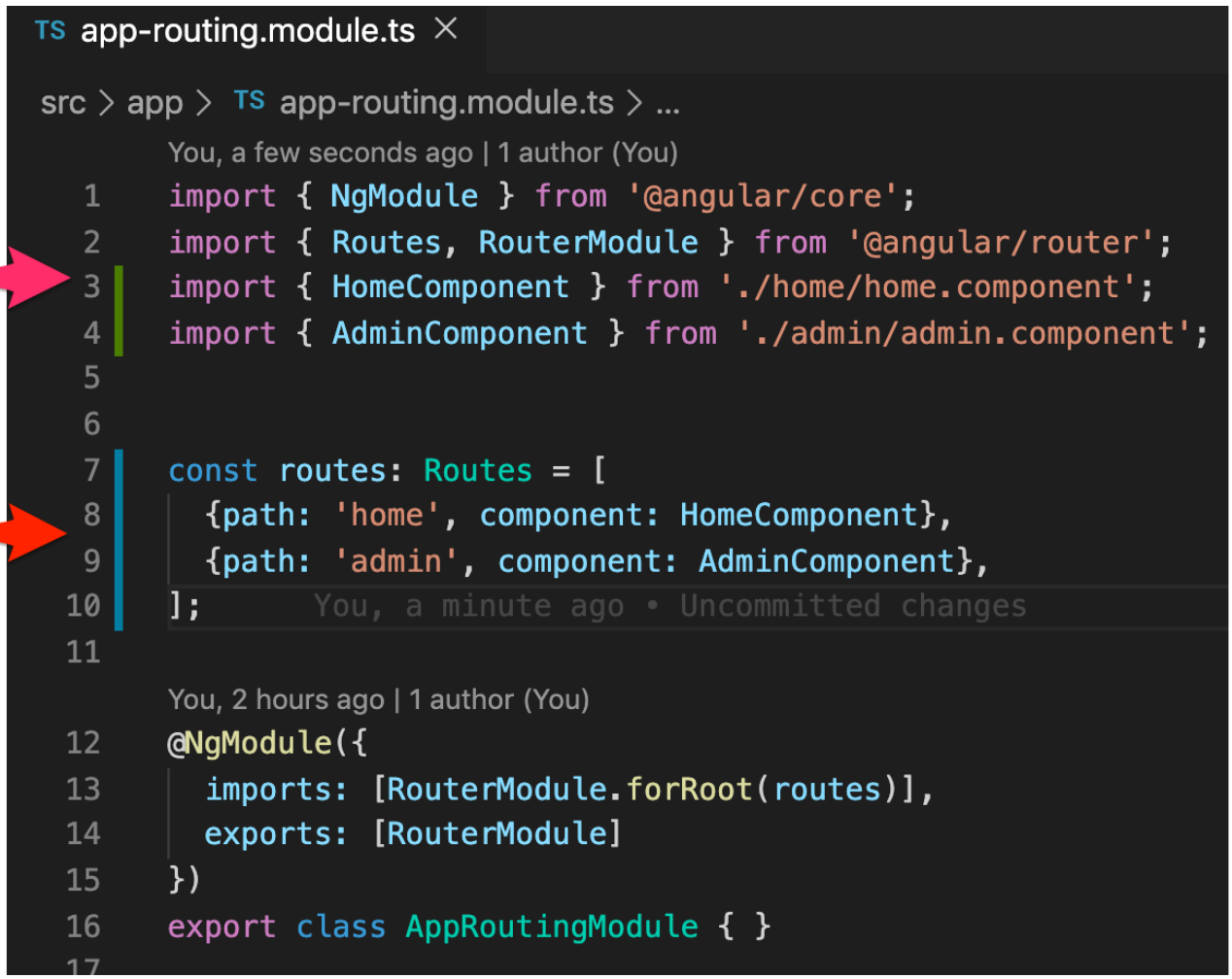
or

```
ng g c home
```

We will do the same for our 'admin' component by doing

ng g c admin

Now we need to go to our src/app/app-routing.module.ts file and add routes for our newly created components.



The routes array needs to be an array of objects where each object has a "path" key whose value is the name of the link, and the "component" key whose value is the actual component that you're linking to.

The code above lets our router know that we have "home" and "admin" links that will go to the HomeCompoent and AdminComponents respectively.

Explore on StackBlitz

https://stackblitz.com/edit/angular-new-routes?file=src%2Fapp%2Fapp-routing.module.ts

When creating routes, the "path" can be anything you want it to be. By convention, most people choose a link name that reflects all or part of the component name. But if you

> wanted to, the "home" link could be named "dashboard" or any other alias that better fits the name or spirit of any given page.

MODULE 6 / LESSON Two / Part Two: Making new routes: Quiz

Type in

Complete the code to create two routes with path names "home" and "admin"

```
import { HomeComponent } from './home/home.component';
import { _________________ } from './admin/admin.component';


const routes: Routes = [
  {______ : '_______' , component: HomeComponent},
  { ______ : '______', component: AdminComponent},
];
```

answers, in order:

AdminComponent path home path admin

MODULE 6 / LESSON Two / Part three: Making a nav bar

The "path" in each of the routes we created in our last lesson is the name of the link that the user will click on to visit each "page". Remember that we really only have one page and Angular is dynamically loading the content of each of our **sub pages** into our single page app.

Now we will go to app.component.html and add a "**nav bar**" above our router outlet so that we can have links that go to our two respective pages.

The "nav bar" below is the simple section within the div tags. It lets our router know that we have "home" and "admin" links that will go to the HomeComponent and AdminComponents respectively.

It is critical that the nav bar does not exist within the router-outlet tags. The links at the top of the page in the nav bar cannot be a part of the page that changes when a link is clicked, otherwise the

menu would go away the minute we click on a link. In other words, the menu must be independent of the changing part of the page.

Let's examine the code below. An anchor tag (<a>), uses the **routerLink** attribute in terms of href. The **routerLink** attribute makes these links work with the Angular router. The **router-outlet** section at the bottom of the code is like a "container" where the sub pages appear when you click on any link that has the **routerLink** attribute. When you click on either 'home' or 'admin', the home and admin components respectively show up as sub-pages in the router-outlet. These pages say "home works" and "admin works" respectively, as these are the default pages that were created when we scaffolded them.

```
<div>
  <a routerLink="home"> home </a>
  <a routerLink="admin"> admin </a>
</div>
<router-outlet></router-outlet>
```

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/angular-navigation-bar?file=src%2Fapp%2Fapp.component.html

View on GitHub

https://github.com/Nmuta/angular-navigation-bar/blob/master/src/app/app.component.html

> Any nav bar that you create in Angular should exist outside of the router-outlet component.

Drag n Drop

Complete the HTML code so that clicking on the link will make the store page appear as a sub page beneath the nav bar:

  <a _____="store">shop now !

<_____><_____>

answers:

routerLink **router-outlet** **/router-outlet** router-link routerOutlet /routerOutlet

MODULE 6 / LESSON Two / Part four: Using routerLink

~~The **path** in each of the routes we created in our last lesson is the name of the link that the user will click on to visit each "page". Remember that we really only have one page and Angular is dynamically loading the content of each of our sub pages into our single page app.~~

Let's look at our **app.component.html** page. At the top, we create links to the sub pages using the **routerLink** attribute on the **a** tag. The **routerLink** attribute is used instead of **href** to make the links work with the Angular router. Finally, at the bottom of the page (or wherever we want the sub pages to appear) we use Angular's **router-outlet** component. This component is kind of like a "box" or an iframe that loads each page into our single page app. It is NOT an iframe, but the functionality may look somewhat similar in terms of the concept.

app.component.html

routerLink instead of href
to make links work
with Angular router

```
1
2   <div>
3       <a routerLink="home"> home </a>
4       <a routerLink="admin"> admin </a>
5   </div>
6
7
8   <router-outlet></router-outlet>
9
10
11
```

Router-outlet is where various 'sub pages' are inserted in the single page app

With the addition of these few lines of code in our html file, we now have a working single page app! Try it out on StackBlitz or view the GitHub code.

Explore on StackBlitz

https://stackblitz.com/edit/angular-navigation-bar?file=src/app/app.component.html

View on GitHub

https://github.com/Nmuta/angular-navigation-bar

> In Angular, the router-outlet is a component that will display the result of whatever the current route is.

Drag n Drop

complete the code to make the following link work with the Angular router

<a _____ = "cars"> view cars

answers:
routerLink href link onClick click route

MODULE 6 / LESSON Three: Making a flights resource
MODULE 6 / LESSON Three / Part one: creating the service

Ok, so now that we have an admin page and a home page, we are free to start putting some actual flight information on the home page.

We will add a flights service

```
ng g service flights
```

which creates a flights.service.ts file for us.

Then we add this line to the top of home.component.ts

```
import { FlightsService } from './flights.service';
```

and alter the home.component.ts file like this :

Inject flightService into constructor

Explore on StackBlitz

https://stackblitz.com/edit/angular-flights-service?file=src%2Fapp%2Fhome%2Fhome.component.ts

View on GitHub

https://github.com/Nmuta/angular-flights-service

> Remember that a service is usually used as a gateway to shared data within an application.

MODULE 6 / LESSON Three / Part One: creating the service : Quiz

drag n drop

Use dependency injection to inject a taco service into the restaurant constructor

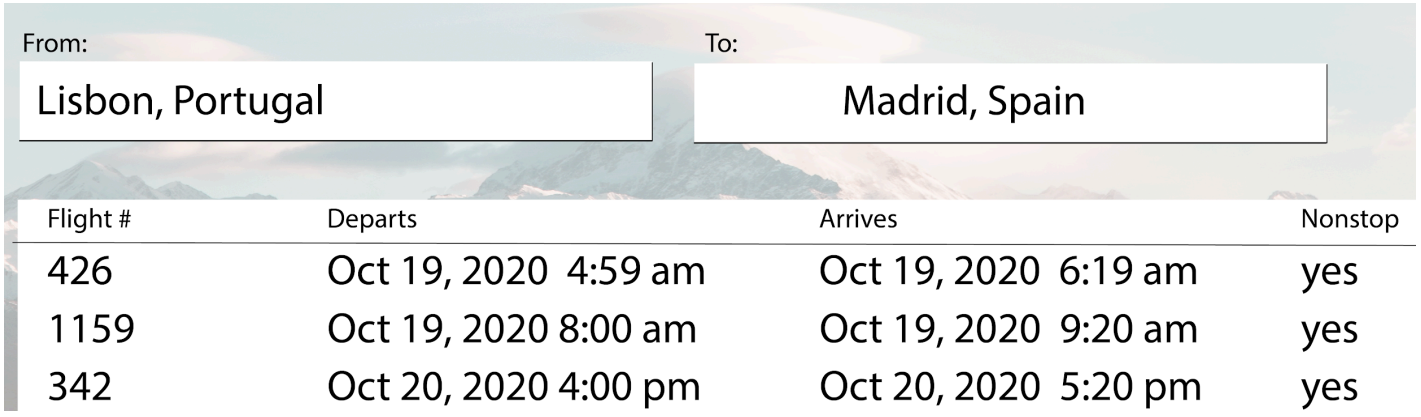
```
import { TacoService } from '../flights.service';

export class FoodComponent {
  constructor(_________  _____________: _________________) { }
}
```

answers:

private **tacoService** **TacoService** service static void @inject

MODULE 6 / LESSON Three / Part two: making a flight model

We are finally at the point where we need to decide what our actual flight data looks like. We have a mockup, so let's look at that.



Based on the mockup, it appears that a flight needs a "from" city, a "to" city, a flight number, a departure time, an arrival time, and a boolean to indicate whether or not it's a nonstop flight.

We can make a model like this and stick it in a flight.model.ts file:

```
export interface Flight
 { origin: string;
  destination: string;
  flightNumber: number;
  depart: Date;
  arrive: Date;
  nonstop: boolean };
```

Notice that we are using a **Date** type for two of our fields. A **Date** type? It exists. It's built into TypeScript. The JavaScript Date object is inherently a built-in TypeScript **type**. Another great feature of TypeScript...the ability for a variable to specifically hold an explicit **Date** value.

Explore on StackBlitz
https://stackblitz.com/edit/angular-flight-model?file=src/app/flight.model.ts

View on GitHub
https://github.com/Nmuta/angular-flight-model

> Another way to view a "model" in TypeScript is that it's like a blueprint. A "model" airplane has the shape of a real airplane. It's not the real thing, but it's a blueprint for the real thing. A "model" for a flight is not a real flight, but it has the shape of a flight: origin, destination, flightNumber, etc.

MODULE 6 / LESSON Three / Part two: making a flight model : Quiz

Multiple choice

~~Our flight model~~ Which of the following is true about our flight model?

It provides structure for what a flight should look like
It can only exist within a service
It is an array of strings

It is an object with only numerical values

MODULE 6 / LESSON Three / Part three: making our service methods

Let's now populate our **service** with all of the methods that we will need. These methods will soon be used to communicate with our back-end:

```
7   export class FlightsService {
8
9     flights: Flight[] = [
10       {origin: "Miami", destination: 'Chicago', flightNumber: 345,
11       depart: '2020-02-25T23:18:21.932Z',
12       arrive: '2020-02-25T23:21:21.932Z', nonstop: true},
13       {origin: "New York", destination: 'Los Angeles', flightNumber: 432,
14       depart: '2020-05-25T23:18:00.932Z',
15       arrive: '2020-05-25T23:23:21.932Z', nonstop: false},
16       ];
17
18    constructor() { }
19
20    getFlights() {
21      return this.flights;
22
23    }
24
25    postFlight(flight: Flight){
26
27    }
28
29    deleteFlight(id: number){
30
31    }
```

Here we are using a mock **flights: Flight[]** array to mock out two simple flights just to give our service something to return for the front-end so we can style it with CSS. The real get, post, and delete flight methods will be communicating with our back-end eventually.

Explore on StackBlitz

https://stackblitz.com/edit/angular-flights-service-methods?file=src%2Fapp%2Fflights.service.ts

View on GitHub

https://github.com/Nmuta/angular-flights-service-methods

> Our service is like a bus or train station. It serves as a hub for data transport.

MODULE 6 / LESSON Three / Part three: making our service methods: Quiz

~~Drag n Drop~~

~~In Angular, we often use _____ to relay or transport data or information to a back-end.~~

~~**services** stylesheets models constructors modules~~

Revised quiz: Angular often uses which of the following to relay data or information to a back-end?

- **services**
- constructors
- modules

MODULE 6 / LESSON Three / Part four: populating the view

Finally, we now are able to loop over our flights coming from our service by implementing a simple ***ngFor** in the view. This gives us a little bit of flight data that we can use to iterate over for now so we can organize the data.

*ngFor loop

CSS class for flight results

home.component.html

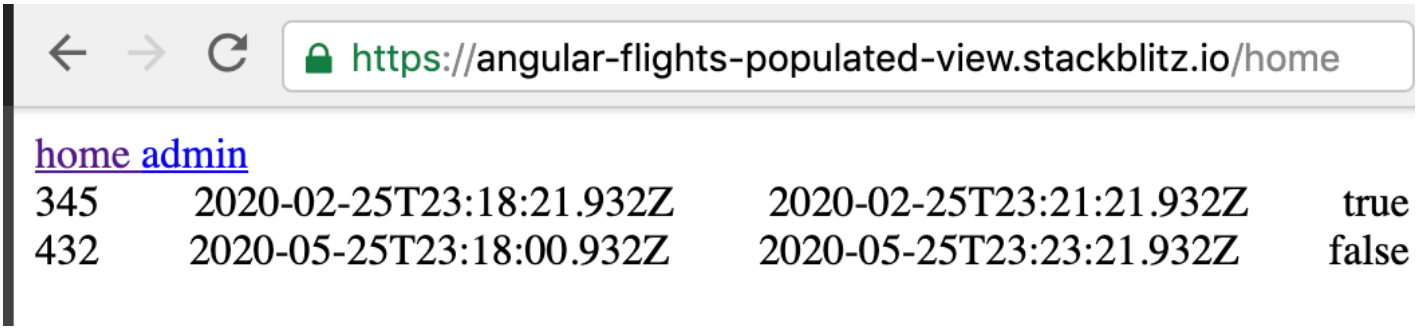
```
1   <div>
2     <div *ngFor="let flight of flights" class="flight_results">
3       <div class="flight_number"> {{flight.flightNumber}}</div>
4       <div class="departs"> {{flight.depart }}</div>
5       <div class="arrives"> {{flight.arrive }}</div>
6       <div class="nonstop"> {{flight.nonstop}}</div>
7     </div>
8   <div>
9
10
```

The CSS class we applied does a very basic job of making the contents of each row line up correctly using **flexbox**. Flexbox is a set of tools built into CSS3 that make layout easy. A full dive into flexbox is beyond the scope of this course, but see if you can figure out what this CSS is doing in the code.

home.component.scss ×

```
1   .flight_results {
2     display: flex;
3     flex-direction: row;
4     justify-content: space-between;
5   }
```

And here is what it looks like in the view:



~~So, we're getting somewhere now.~~ If you're curious about Flexbox, remove the styling in the live code on StackBlitz and see what happens. This is a good hands-on way to see these elements coming together.

Also, keep reading. If you were wondering how we can fix those long dates, there's a quick fix that Angular provides, and it's only about 5 characters! We will cover that in the next lesson.

Explore on StackBlitz

https://stackblitz.com/edit/angular-flights-populated-view?file=src%2Fapp%2Fhome%2Fhome.component.html

View on GitHub

https://github.com/Nmuta/angular-flights-populated-view

> **Flexbox** and **CSS grid** are two popular layout frameworks built into standard CSS and are supported in all browsers. We will not use CSS grid in this project, but it's another tool that exists for your use, if you like exploring visual layout tools.

MODULE 6 / LESSON Three / Part four: populating the view: Quiz

Type in

Complete the following code so that the outer div has a class of "container" and the boxes inside of the container have a class of "box"

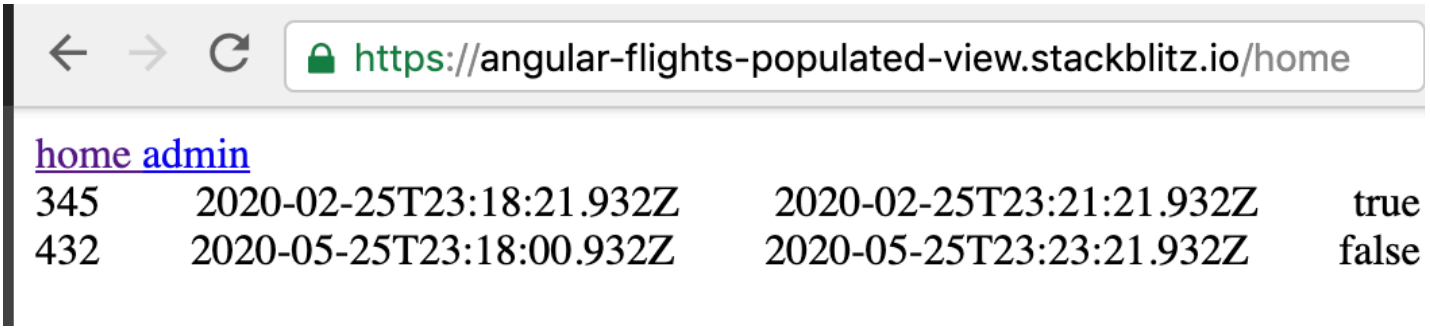
```
<div *ngFor="let color of colors" id="container" _____ = "_____">
  <div _____="_____"> {{ color }} </div>
</div>
```

answers:

class container class box

MODULE 6 / LESSON Three / Part five: pipes

We will now address the fact that, when looking at flight times, nobody can or wants to read "2020-02-25T23:18:21.932Z". This is a raw JavaScript date. We need a process to parse the date into human readable terms. This is our current view:



Luckily, Angular has something called **pipes** that can transform data in a view from one format to another. Pipes, in programming, are processes that take a value and transform that value into something else. Strictly speaking, pipes take the output of one function and then feed it to be the input of another. But in layman's terms, pipes TRANSFORM data. That's all you need to know to use Angular's built-in pipes. You can write a pipe using the | key. ~~above the 'return' or 'enter' key on the keyboard; it looks like this : |~~

Right now, the flight departure time code in the view looks like this:

```
<div class="departs"> {{flight.depart }}</div>
```

We can take the flight.depart value and pipe it to a date format like this:

```
<div class="departs"> {{flight.depart | date  }}</div>
```

And then, magically, our dates now look like this in the view:

Explore on StackBlitz

https://stackblitz.com/edit/angular-flights-pipes?file=src%2Fapp%2Fhome%2Fhome.component.html

View on GitHub

https://github.com/Nmuta/angular-flights-pipes

> Angular has several built in pipes. There's a currency pipe where {{ total | currency }} will output something like this: $4.00 where "total" is the number of 4, a lowercase pipe than converts strings to lowercase, a percent pipe, and so on. Angular even allows you to build custom pipes of your own.

MODULE 6 / LESSON Three / Part five: pipes: Quiz

type in

Complete the basic date pipe in order to transform the **copyright_date** into a readable format. Assume that the **copyright_date** field is a JavaScript Date object.

 All content © {{ copyright_date | _____ }}

answer: **date**

MODULE 6 / LESSON Four : Styling the view
MODULE 6 / LESSON Four / Part One: Structure

We're finally to the point where we will start to provide some visual structure to our app. A picture is worth a thousand words, so we will incrementally put some SCSS structure into the app and talk about it. We are assuming, if you're taking this course, that you have foundational CSS knowledge, so we will not teach the CSS but we will discuss how Angular uses CSS in an SCSS context.

The first thing we are going to do is put in a beautiful mountain background image and a header.

Angular has a master, or "root", SCSS file (**src/styles.scss**) that applies to all components in the app. Since the background and header will be common to all pages, we will put them in this root stylesheet.

Here is a mobile (phone) view of the app so far:

home admin

345	Feb 25, 2020	Feb 25, 2020	true
432	May 25, 2020	May 25, 2020	false

Note: The mountain background image was courtesy of the "Unsplash" website, where you can download free and royalty-free images without having to worry about copyright violations.
Mountains photo credit: Marco Bonomo

```
body {
    margin: 0;
    padding: 0;
    width: 100vw;
    height: 100vh;
    background: url("assets/mountains.jpeg");
    background-repeat: no-repeat;
    background-size: cover;
}
```

assets folder contains images

You, 8 minutes ago • add mountain background

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step1?file=src%2Fstyles.scss

View on GitHub

https://github.com/Nmuta/TSFlightsApp/

💡 ***NOTE***: If you clone the GitHub repo to a computer, make sure to checkout the branch called "step1". ~~If you need help, inquire about this in the comments section.~~

> SCSS is a superset of CSS. It allows for special things like nested styles and mixins, which are essentially variables within CSS.

MODULE 6 / LESSON Four / Part One: Structure: Quiz

Multiple choice

The **src/styles.scss** stylesheet contains styles that will apply to which of the following?

all components in the app
only components in the app named 'styles'
the first component in the hierarchy tree
the back-end

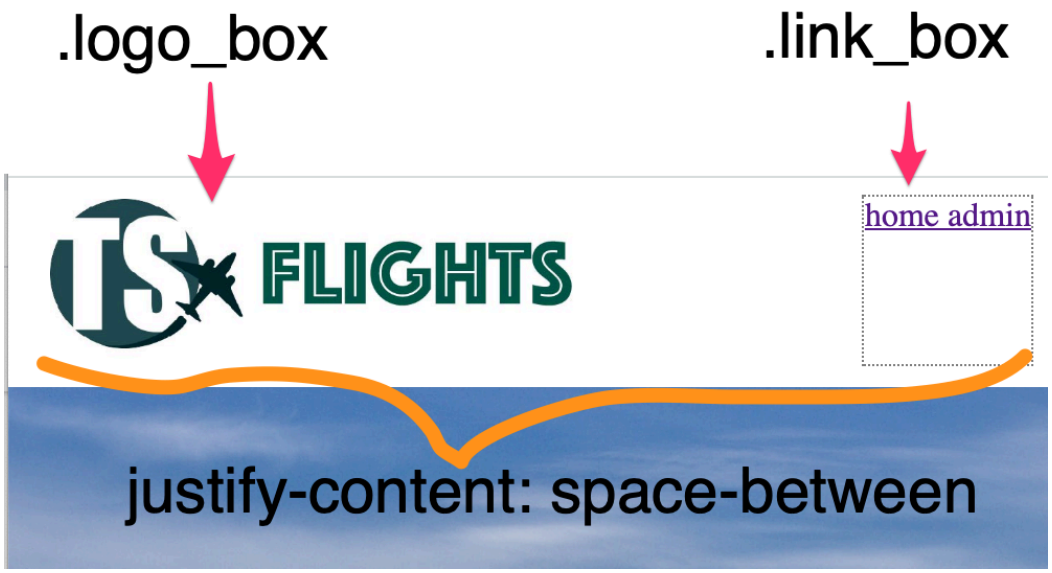
MODULE 6 / LESSON Four / Part Two: Blocking the header

In the header, we will make a div to contain the logo, and then another div to contain the links. Then we will use the **justify-content: space-between** flexbox property to make the logo box and the link box 'float" left and right, respectively.

Since both the logo box and the link box are inside of the header, SCSS allows us to NEST those two styles within the header style. See the image below:

```
13  header {
14      width: 100vw;
15      height: 100px;
16      background-color: white;
17      display: flex;
18      flex-direction: row;
19      align-items: cen
20      justify-co      t: space-bet      ween;
21      .logo_box {
22          background-image:      rl("TsFlightsHorizLogo.jpg");
23          height: 80px;
24          width: 250px;
25          background-re      eat: no-repeat;
26          background-      ize: cover;
27          margin: 1      x 20px;
28          align-s      f: flex-start;
29      }
30
31      .link_box {
32          height: 80px;
33          margin: 10px;
34          align-self: flex-end;
35          border: 1px dotted grey;
36      }
37  }
```


Here is the result:



Note that we have put a dotted line around the link box for now. I sometimes do that when blocking out elements just to see the space that they occupy. Later, when the spacing is right, I delete the border.

We've also used an **align-self: flex-start** and **align-self: flex-end** to align the **logo-box** and the **link-box** to the start and the end respectively. In this case, since the flex direction is row (horizontally), flex-start is almost like a *float: left* and flex-end is almost like a *float:right*. They are not floats, and very different in many ways, but they position things to the respective ends of the container in a similar way.

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step2

View on GitHub

https://github.com/Nmuta/TSFlights1/

> Nested styles are a feature of SCSS. They save room, because they reduce the amount of repetition when declaring CSS styles. This is one of the huge benefits of SCSS. Nested styles also tend to be easier to read due to their hierarchical indentation.

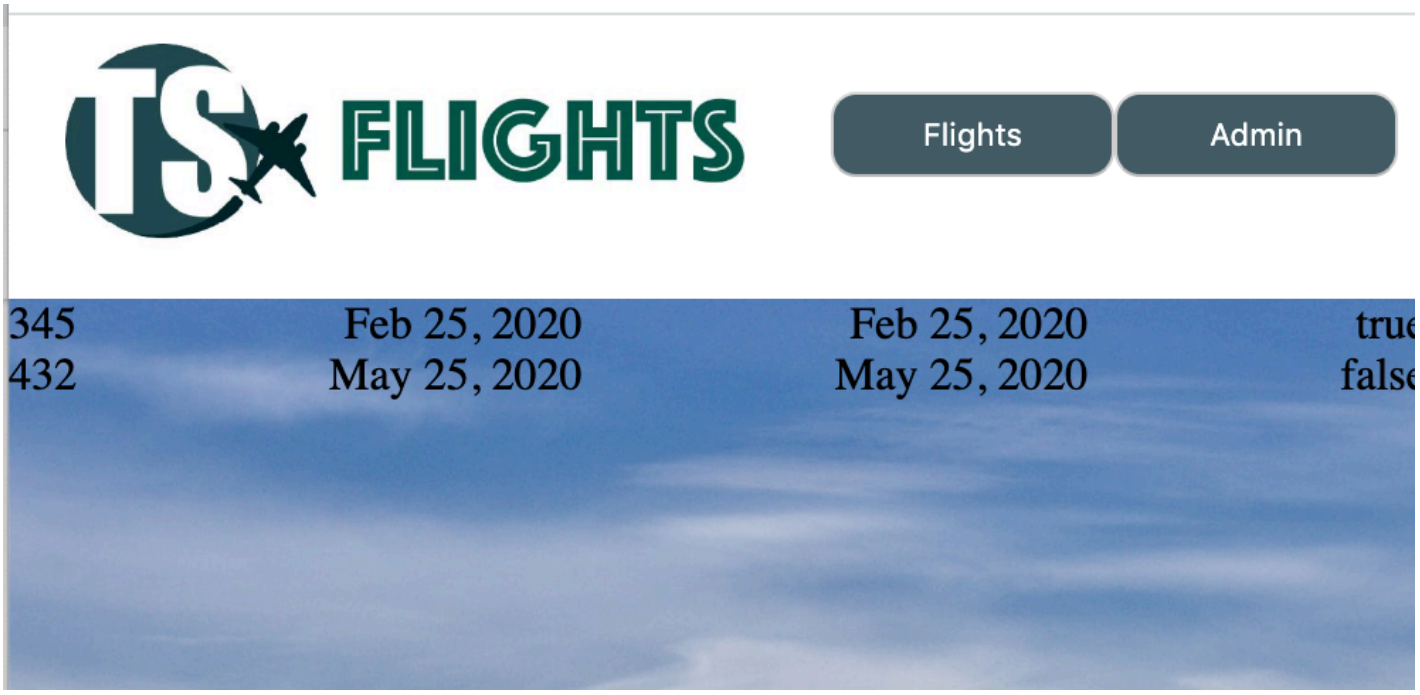
MODULE 6 / LESSON Four / Part Two: Blocking the header: Quiz

True / False

In SCSS, we can use nested styles to hierarchically fit styles inside of their parent elements in the stylesheet.

MODULE 6 / LESSON Four / Part Three: Finishing the header

Here is the finished header.



Note the html structure of the header now:

```
<header>
  <div class="logo_box"></div>
  <div class="link_box">
    <button routerLink="home"> Flights </button>
    <button routerLink="admin"> Admin </button>
  </div>
</header>
```

Please observe the fact that what used to be "a" links are now buttons. **routerLink** works on buttons as well as anchor tags. ('a' stands for anchor in HTML).

~~As always, please feel free to examine the code, as the SCSS files are too long to put the full code in this lesson part.~~

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step3

View on GitHub

https://github.com/Nmuta/TSFlights1/ (step 3 branch)

> The routerLink attribute is usually reserved for anchor tags and buttons, but it can apply to any valid html element if you want to create a link that triggers a page registered with the Angular router.

Drag N Drop

An Angular controller has an array of link objects that looks like this:

links = [{name: "home", class: "red"}, {name: "about", class: "black"}, {name: "contact", class: "green"}] ;

Complete the html view so that all of the links connect to the name of each link and each link's class is set to the class of the object:

```
<div *ngFor="let link of links">
   <a __________={{link.name}}    ________={{____________}}> {{link.name}} </a>
</div>
```

routerLink **class** **link.class** link.name class router-outlet

MODULE 6 / LESSON Four / Part Four: Media Queries

Included in our last StackBlitz code sample in the previous lesson (https://stackblitz.com/github/Nmuta/TSFlights1/tree/step3?file=src%2Fstyles.scss) was another CSS property that we haven't discussed yet: media queries.

A **media query** is a special rule that applies only to specific conditions that a web page is being viewed under. For example, there are media queries to format a page that ONLY apply if you're looking at a print preview. There are also media queries that only apply if your screen is below a certain number of pixels.

Why is this important? Well, the device width of your phone may only be 480px, while a full desktop device width will usually be well over 1200px. We use media queries to tell the browser: "hey, if you're really running out of space, use this special set of CSS rules for when things get tight".

In a full production website, there are usually three or more media queries for different device sizes that define the styles for mobile, tablet, and desktop respectively. In our demo, I noticed that the

logo was getting cut off in the StackBlitz window due to having a small screen width, so I wrote this media query inside of the .logo_box:

```
.logo_box {
    background-image: url("TsFlightsHorizLogo.jpg");
    height: 80px;
    width: 250px;
    background-repeat: no-repeat;
    background-size: contain;
    background-position-y: center;
    margin: 10px 0 0 px;
    align-self: flex-start;
    @media screen and (max-width: 497px) {
        width: 200px;
        margin-left: 4px;
        margin-top: 7px;
    }
}
```

@media screen and max-width: 497px means "If we are on a screen, and the screen is 497 pixels and under, use these nested styles for the logo_box. " Notice that these apply ONLY to the logo_box and not to anything else on the page.

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step3?file=src%2Fstyles.scss
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step3?file=src%2Fstyles.scss

View on GitHub

https://github.com/Nmuta/TSFlights1 (step 3 branch)

> Media queries help define how CSS styles should behave when the screen size changes. This is why you will often see three bars or what we call a "hamburger menu" as a menu on responsive websites when you view them on a phone instead of your desktop computer or laptop.

MODULE 6 / LESSON Four / Part Four: Media Queries: Quiz

Type in

Complete the media query below to remove borders from the .registration_box when the screen width falls below 500 pixels:

```
.registration_box {
border: 1px solid black;
@_________  ________ and (max-width: 499px) {
        border: none;
  }
}
```

answers:

media **screen**

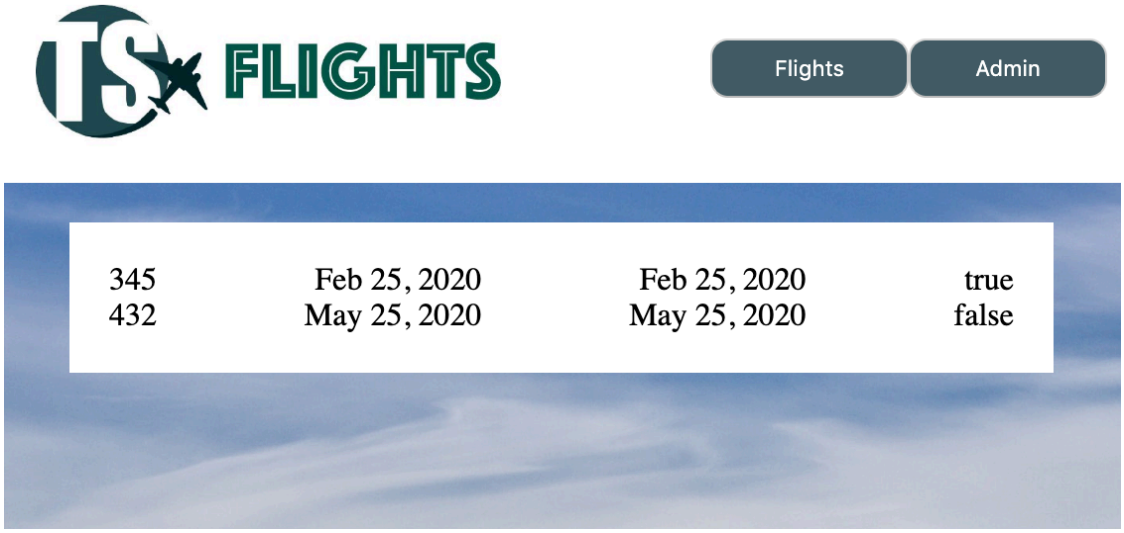
MODULE 6 / LESSON Four / Part Five: Making the flights panel

We will now build styles that are *only* seen on the home component. For this, we use the home.component.scss file.

We added a div that we named "flight_pane". It will be the white box that contains the flight data. We've also added a media query to our flight_pane so that it "snaps" to full screen when the browser screen goes below 481 pixels.

Note that we have been using **max-width** as a rule, but you can also use min-width as a media query rule. **Max-width** applies to everything UP TO that maximum width. **Min-width** applies to everything AT LEAST that minimum width and above.

… and here is the result:



Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step4?file=src%2Fstyles.scss

View on GitHub

https://github.com/Nmuta/TSFlights1/ (step4 branch)

> The rules for any given media query are up to the designer. If you are the main programmer and designer, then it's totally up to you as to what pages should look like on a *phone* vs *tablet* vs *full screen device*.

MODULE 6 / LESSON Four / Part Five: Making the flights panel: Quiz

Multiple choice

A media query that says **@media screen and (min-width: 400px)** would apply to all devices with

a width of 399 pixels or less
a width of 400 pixels or more
a width of 399 pixels or more
a width of 400 pixels or less

MODULE 6 / LESSON Four / Part Six: Finishing the Flights page

I've taken the time to flesh out more of the styling on the flights page. When making the page responsive for mobile, I found that the departure date, even in **date** format using Angular's date pipe, was too long. So I went into the Angular documentation and found a **format** for the **date** pipe called 'shortDate'. The Angular documentation article that goes over all of the DatePipe options is here: https://angular.io/api/common/DatePipe

I'm being very transparent about the creation process here because the skill of having a problem and then solving that problem through reading documentation is a core developer skill.

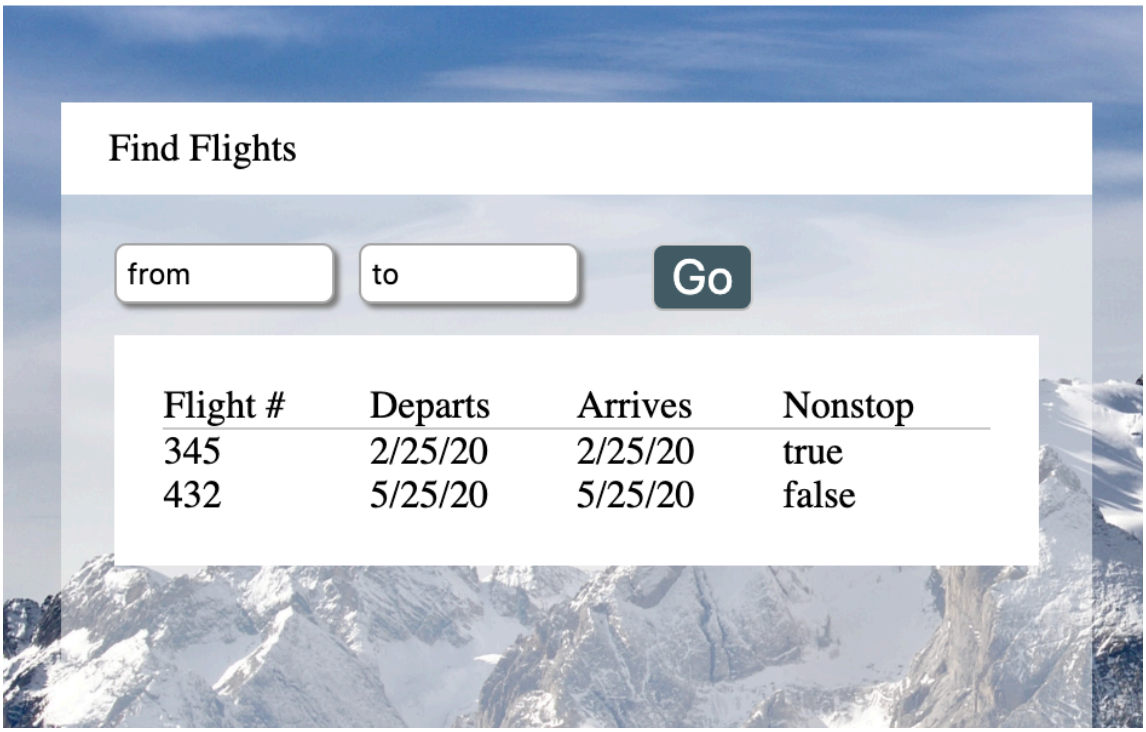
At any rate, here is the change to the date format to allow the date to fit within our newer, fresher, flights page with more styling:

```
<div class="departs cell">
   {{flight.depart | date : 'shortDate' }}
</div>
```

The date pipe all by itself gave us Feb 25, 2020, where the date pipe with the **shortDate** format gave us 2/25/20, which suits our requirements better when using a small screen.

Of course, a real flights page will have a ton of information, including choices between one way and round trip, actual time of day in hours or minutes as it relates to departure and arrival, payment options, and the whole gamut. The purpose of this project is to simply get you far enough along in terms of understanding the basics of TypeScript and Angular so that you can venture out on your own later and build something of your own.

Here is what our page looks like now:



We encourage you to look at the source code in progress on StackBlitz and/or Github if you're curious about the code behind our progress so far.

We have also changed the inputs in the form to use a select menu. The reasons for this are practical; we don't have a database full of cities yet, so giving the test users a few predefined cities will ensure that they only request cities that we have in our system right now.

Explore on StackBlitz
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step5

View on GitHub
https://github.com/Nmuta/TSFlights1/blob/step5/src/app/home/home.component.scss

> The output of certain pipes can be modified by way of 'format' options. Dates can be formatted in a dozen different ways. The currency pipe can be formatted to hundreds of different currencies from around the world. View the Angular pipes documentation for more information: https://angular.io/api/common#pipes

MODULE 6 / LESSON Four / Part Six: Finishing the Flights page: Quiz

~~Drag N Drop~~

~~Given this code in a view:~~

~~ {{ customerTotal | currency : "EUR" }} ~~

~~It would be safe to say that customerTotal is being fed through a _____ pipe with a _____ that outputs the currency into Euros.~~

~~answers:~~
~~**currency** **format** filter modify date~~

Revised quiz: Complete the code to ensure that customerTotal is being fed through a currency pipe with a format that outputs the currency in Euros.

 {{ customerTotal | _____ : "EUR" }}

A: currency

MODULE 6 / LESSON Five : Building our backend
MODULE 6 / LESSON Five / Part One: The database

We have enough of a front-end now that if we were able to pull in data from the back-end, we could actually see "real" flights making data's long journey from the database, through the NestJS server, into our Angular front-end. We will build that pipeline now.

The first thing that we will do is build out our database. We are going to keep our "transportation" database that we started in a previous module. Right now that database has only a table named "cars". We will add a table named "flights". Our flight model from our Angular front-end reminds us of all of the fields that we will need.

Remember by typing **psql** into the terminal on your computer, that would get you into the psql command line. From there, you could follow these steps:

```
\c transportation
CREATE TABLE flights (id SERIAL,
origin VARCHAR(20),
destination VARCHAR(20),
flightNumber INT,
depart TIMESTAMP WITH TIME ZONE,
arrive TIMESTAMP WITH TIME ZONE,
nonstop BOOLEAN);
```

Note: Caps are optional in the create table syntax. I'm just using them here to distinguish between the column names and the column types.

After we've created the table, this is what it may look like :

```
                                  Table "public.flights"
   Column    |           Type           | Collation | Nullable |
-------------+--------------------------+-----------+----------+-
 id          | integer                  |           | not null |
 origin      | character varying(20)    |           |          |
 destination | character varying(20)    |           |          |
 flightNumber | integer                 |           |          |
 depart      | timestamp with time zone |           |          |
 arrive      | timestamp with time zone |           |          |
 nonstop     | boolean                  |           |          |
```

It surprises many new developers when they discover that **time**, especially time on an international scale, is one of the more difficult and complicated areas of programming. For the sake of our app, we are using very simple fictional times for our departure and landing times without taking into account time zones, daylight saving time, states that don't honor daylight saving time, etc.

To create a new date object in JavaScript, you can use the **Date** object. To get the date right now, open up a JavaScript terminal or use the playground link listed below to experiment with dates.

Explore in Playground

https://code.sololearn.com/WRTmH86v4GTd/#js

> In PostgreSQL, TIMESTAMP WITH TIME ZONE corresponds to a date / time object in JavaScript .You can take a Date( ) object in JavaScript (as in const currentDate = new Date( )), which will give you and directly deposit that value into a PostgreSQL database.

MODULE 6 / LESSON Five / Part One: The database : Quiz

Reorder

You need a PostgreSQL database of animals with a birds table with one bird. Order the following commands to make that happen:

createdb animals
psql (logs into postgres)
\c animals
CREATE TABLE birds (id SERIAL, name VARCHAR(20));
INSERT into birds values (DEFAULT, 'Mockingjay');

MODULE 6 / LESSON Five / Part Two: Create a Nest server

Now we will set up our NestJS server with TypeORM to interact with our database.

We will set up the server to CRUD (Create, Read, Update, and Delete) flights in our system.

This is a repeat of the process that we did earlier in this course. Before, we set up a car folder with a controller, module, service, and entity. We will do the same for flights.

To review:
The **controller** is what actually interacts with the outside world… other applications like Angular that want to interact with our NestJS server by way of our API. The controller contains the **API endpoints**.

The **service** contains methods like findAll(), findOne(), delete(), etc that talk to TypeORM (our ORM is what talks to the database).

The **entity** helps define the shape of our database table that we are communicating with; in this case "flights".

The **module** provides support for the service and the controller.

When we set all of that up, add some flights to our database, and we visit localhost:3002/flights on our computer, we get a JSON collection of all of our flights currently in the system:

```
[
    {
        "id": 2,
        "origin": "Phoenix",
        "destination": "Denver",
        "flightNumber": 23,
        "depart": "2020-02-28T16:00:00.000Z",
        "arrive": "2020-02-28T18:00:00.000Z",
        "nonstop": false
    },
    {
        "id": 3,
        "origin": "Jackson",
        "destination": "Chicago",
        "flightNumber": 234,
        "depart": "2020-10-02T15:30:00.000Z",
        "arrive": "2020-10-02T18:30:00.000Z",
        "nonstop": true
    },
    {
        "id": 1,
        "origin": "Atlanta",
        "destination": "New York",
        "flightNumber": 456,
        "depart": "2020-02-28T17:17:20.315Z",
        "arrive": "2020-02-28T20:17:20.315Z",
        "nonstop": true
    }
]
```


Here is a link to the GitHub branch that has all of the code for each of these four files.

View on GitHub

https://github.com/Nmuta/nest_flights/tree/step1

> Typically, in NestJS, we use a service to talk to TypeORM, which then talks to the database. When the service gets its answer, it sends that data to the controller. The controller then is connected to the API endpoints which speak to our front-end.

MODULE 6 / LESSON Five / Part Two: Create a Nest server : QUIZ

~~Type in~~

~~In a NestJS application, the _____ contains the code that defines the API endpoints.~~

~~answer:~~
~~**Controller**~~

Revised quiz: In a NestJS application, which of the following contains the code that defines the API endpoints?

- **Controller**
- Administrator
- Regulator
- Manager

MODULE 6 / LESSON Five / Part Three: Connecting our Angular front end

Now that our NestJS API is correctly retrieving flights from the database through the ORM, we can pull those flights into Angular and, for the first time, data will be flowing from our database all the way into our front-end Angular TS Flights app.

Currently, in Angular, we are mocking a few flights in our flightsContoller:

```
  flights: Flight[] = [
     {origin: "Miami", destination: 'Chicago', flightNumber: 345,
     depart: new Date("2020-02-25T23:18:21.932Z"),
     arrive: new Date('2020-02-25T23:21:21.932Z'),
     nonstop: true},
     {origin: "New York", destination: 'Los Angeles', flightNumber: 432,
     depart: new Date('2020-05-25T23:18:00.932Z'),
     arrive: new Date('2020-05-25T23:23:21.932Z'),
     nonstop: false},
     ];
```

We will convert this to an observable that makes an API call to our NestJS back-end running on port 3002.

Then we will delete that entire code block and instead, pull in the data from the flights service with an observable.

So we will change *THIS*:

```
getFlights() {
  return this.flights;
}
```

to *THIS*:

```
getFlights(): Observable<any> {
  return this.http.get('http://localhost:3002/flights/');
}
```

Note the colon after the getFlights() and before the word Observable. This means that this method returns an **Observable**.

In a nutshell, what we are doing here is replacing a hard coded array of flights with an Observable, which is a stream of data that is flowing from the back-end. The code won't work just yet because we need to adjust the view in order to **subscribe** to the stream of data coming from the observable. We will do that in the next lesson part.

> An **Observable** is a stream of data. We can **subscribe** to an observable in order to tap into the data that it contains.

View on GitHub

https://github.com/Nmuta/TSFlights1/tree/step6

MODULE 6 / LESSON Five / Part Three: Connecting our Angular front end : Quiz

Drag n Drop

In Angular, _____ are things that we can subscribe to in order to obtain a stream of data from an external source like an API coming from a NestJs backend.

Observables Subscriptions Observers Subscribables Models

Revised quiz: In Angular, which of the following are things that we can subscribe to in order to obtain a stream of data from an external source like an API coming from a NestJS back-end?

- **Observables**
- Subscriptions
- Models

MODULE 6 / LESSON Five / Part Four: Subscribing to Observables

Now that our Angular app is pulling in data correctly, we have to fix the **view** so that the view can receive the data coming from the service, which is no longer "hard coded".

In our service, the Flights service getFlights( ) method now looks like this:

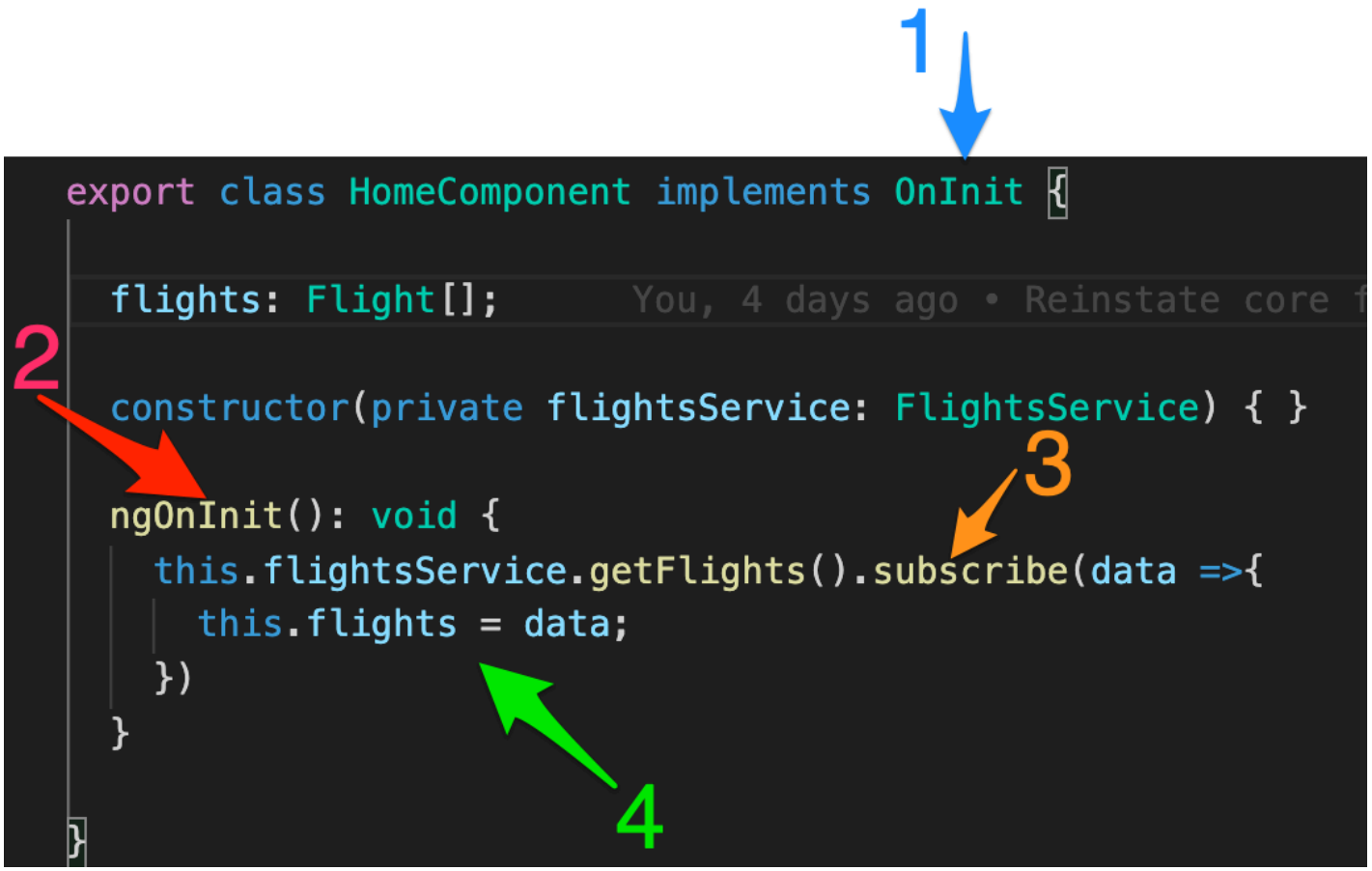
```
getFlights(): Observable<any> {
  return this.http.get('http://localhost:3002/flights/');
}
```

which means that when our view is initialized, we need to start **subscribing** to the stream of data coming in.

For now, let's subscribe to our Observable this way in our home.component.ts file. First, we add "OnInit" to our Angular imports at the top of home.component.ts:

```
import { Component, OnInit } from '@angular/core';
```

then we modify **home.component.ts** as follows:



ngOnInit( ) is a method that is called when the component first starts. Follow the numbers in the diagram to match the flow below:

(1) make the component class implement the OnInit method so that we can use it. Then,
(2) Use the ngOnInit method to call the **getFlights**( ) method from the flightsService. That method returns an Observable, so get the Observable and
(3) subscribe to it. When the subscription is finished (meaning when the data has arrived) , then
(4) assign whatever you get from the Observable to the **this.flights** variable.

After we've done all of this, we don't need to change our view at all…it's looking for an array of flights that are of Flight type, and they've got it now. In the view, our flights are now working. The data looks the same, but we know that our data is all coming from the database, and that's a big win.

Flight #	Departs	Arrives	Nonstop
23	2/28/20	2/28/20	false
456	2/28/20	2/28/20	true
234	10/2/20	10/2/20	true

View on GitHub

https://github.com/Nmuta/TSFlights1/tree/step7

> There are a few different ways to tap into Observables in Angular. Using .subscribe( ) is one of the simplest ways when you are first learning. We will go over another way later.

MODULE 6 / LESSON Five / Part Four: Subscribing to Observables: Quiz

Drag n Drop

In Angular, if we see this inside of a method in a service:

```
return this.http.get('http://localhost:3002/flights/');
```

What will always be returned by this type of return statement?

a string
an http method
an observable
a boolean

MODULE 6 / LESSON Six : Finishing the backend
MODULE 6 / LESSON Six / Part One: Custom Query method

At this point, we will finish the back-end so that we will have a full API ready to use by our front-end.

First, we will add a new route called "query" that will allow us to retrieve all routes that match a query from the user on the front-end (e.g. "give me all the flights from Jackson to Chicago").

```
// QUERY
@Get("query/:orig/:dest")
async query(@Param('orig') orig, @Param('dest') dest): Promise<any> {
  return this.flightService.query(orig,dest);
}
```

Let's break this down. It's a **Get method**, with the url shape of query/CITY1/CITY2 where 'city1' and 'city2' represent origin and destination cities respectively. Then it's an async query, meaning that JavaScript has to wait for the results to be ready. We then declare our two incoming city params as orig and dest respectively. Finally, we return the query from our flight service which actually speaks to TypeORM to do the query.

Now, let's look at the corresponding flight service method:

```
async query(orig: string, dest: string): Promise<any> {
  return await this.flightRepository.find({origin: orig, destination: dest});
}
```

In the flight service, we call the repository's built in "**find**" method of the ORM that takes in two parameters that we are looking for, in the form of key / value pairs, and does the query for us. The result when we make a url call like this:

http://localhost:3002/flights/query/Jackson/Chicago

We get this :


```
[
  {
    "id": 3,
    "origin": "Jackson",
    "destination": "Chicago",
    "flightNumber": 234,
    "depart": "2020-10-02T15:30:00.000Z",
    "arrive": "2020-10-02T18:30:00.000Z",
    "nonstop": true
  }
]
```


We encourage you to peek at the full code on GitHub to see how the flights controller and the flights service work together to make these beautiful queries.

View on GitHub

~~https://github.com/Nmuta/TSFlightsApp/tree/step2~~

https://github.com/Nmuta/nest_flights/ (on the GitHub repo, checkout or view the 'step2' branch)

> TypeORM has a built in "query" method that can take raw SQL that allows you to build your own custom queries.

MODULE 6 / LESSON Six / Part One: Custom Query method: Quiz

Type in

Complete the query to find all flowers whose color is the parameter passed in by the user

```
async query(color: string): Promise<any> {
  return await this.flowerRepository.query(
    ` Select * from flowers where _______ = '${_____}'
```

```
    `);
  }
```

answers:

color color

MODULE 6 / LESSON Six / Part Two: adding a flight

If we want to add a flight to the system, in RESTful APIs, we make a **POST** request to a collection. In this case, we would make a **POST** request to the flights route. Here's what that looks like:

```
// CREATE
@Post()
async create(@Body() flight: Flight): Promise<Flights[]> {
  return this.flightService.create(flight);
}
```

Let's break this down. The http verb being used here, instead of GET is **POST**. The method is an async method called "create", and it uses the **@Body** decorator, and takes in one parameter, which is a bundle of information that we are nicknaming "flight".

The body (@Body) of a POST request is the bundle of data coming in. Every back-end has a different way of accessing the body of a post request, but it's essentially the package of data coming in from the front-end in the form of a JavaScript object or a similar data structure, depending on which language your back-end is written in.

Now, let's look at the corresponding flight service method:

```
async create(flight: Flight): Promise<any> {
  return await this.flightRepository.save(flight);
}
```

Those two methods together help us to create a new flight coming in from the front-end.

View on GitHub

~~https://github.com/Nmuta/TSFlightsApp/tree/step2~~
https://github.com/Nmuta/nest_flights/tree/step3

> The BODY of a post request contains all of the information the back-end needs to insert a new item into the database.

MODULE 6 / LESSON Six / Part Two: adding a flight: Quiz

drag n drop

POST requests have a bundle of data that we call the _____ , where GET requests take in incoming segments of information in the url that we call _____

ANSWERS:

body **params** decorators services async

Revised quiz: POST requests have a bundle of data that we call the body, where GET requests take in incoming segments of information in the url that we call what?

- **params**
- decorators
- services

MODULE 6 / LESSON Six / Part Three: updating a flight

If we want to update a flight to the system, in RESTful APIs, we make a **PATCH** request. Here's what that looks like:

```
// UPDATE
@Patch(":id/update")
async update(@Param('id') id, @Body() flight: Flight): Promise<any> {
  flight.id = Number(id);
  return this.flightService.update(flight);
}
```

Let's break this down. The http verb being used here instead of POST is **PATCH**.

Think of "patch" like patching a hole in a sweater. You're not replacing the whole sweater, you're just patching one part of it. Let's say you want to update the flight time from 9am to 9:45 am due to a delay. You would patch it. A patch is similar to a POST in the sense that it's sending a bundle of data to the server, but it's also sending an id in the URL. So you may send a patch request to:

http://localhost:3002/flights/8/update

This would patch flight number 8 with the bundle of data that you're sending that contains the keys of the fields that you want to patch.

Now, let's look at the corresponding flight service method:

```
async update(flight: Flight): Promise<UpdateResult> {
  return await this.flightRepository.update(flight.id, flight);
}
```

Those two methods together help us to update, or 'patch', an existing flight in the system.

View on GitHub

https://github.com/Nmuta/nest_flights/blob/step3/src/flights/flights.controller.ts

> Edit, update, and PATCH are all words that describe pretty much the same thing. PATCH is an http verb.

MODULE 6 / LESSON Six / Part Three: updating a flight: / Quiz

Multiple Choice

The difference between a PATCH and a POST in HTTP is :

PATCH is for creating and POST is for updating
POST is for creating and PATCH is for updating
POST takes parameters and PATCH doesn't
POST is for updating and PATCH is for deleting
They are mostly the same

MODULE 6 / LESSON Six / Part Four: deleting a flight

If we want to delete a flight in the system, in RESTful APIs, we make a **DELETE** request. Here's what that looks like:

```
// DELETE
@Delete(":id/delete")
async delete(@Param('id') id): Promise<any> {
  return this.flightService.delete(id);
}
```

Let's break this down. The http verb being used here is **DELETE**.

The delete method only requires one thing: an id. No bundle of information is needed. Why? Well, in order to delete a resource, all we need to know is its database id. That's it….no other questions asked.

Now, let's look at the corresponding flight service method:

```
async delete(id: number): Promise<any> {
  return this.flightRepository.delete(id);
}
```

Those two methods together help us to delete an existing flight in the system.

View on GitHub

https://github.com/Nmuta/nest_flights/blob/step3/src/flights/flights.controller.ts

> Deleting a record in most RESTful APIs only requires an id.

multiple choice

Delete methods in apis typically only require ids because:

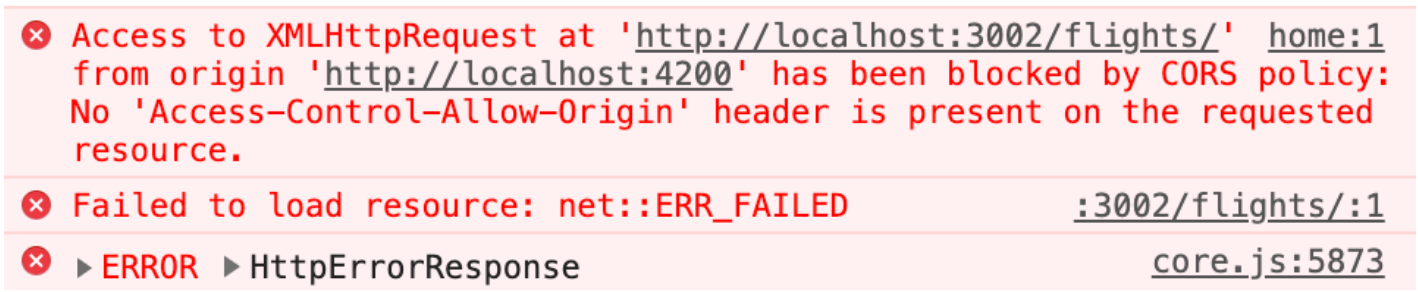
They use get requests which can pass the id as a query param
POST requests are only able to contain one variable in the body
An id is all that's needed to locate a record to delete
TypeORM requires that delete methods in apis accept numbers

MODULE 6 / LESSON Seven / Finishing the front end
MODULE 6 / LESSON Seven / Part one: CORS

Before we go any further, we should discuss an important concept in web security: Cross-origin resource sharing or **CORS**.

When running our full stack locally, NestJS by default runs on port 3000 (we're running our on port 3002), and Angular, by default, runs on port 4200.

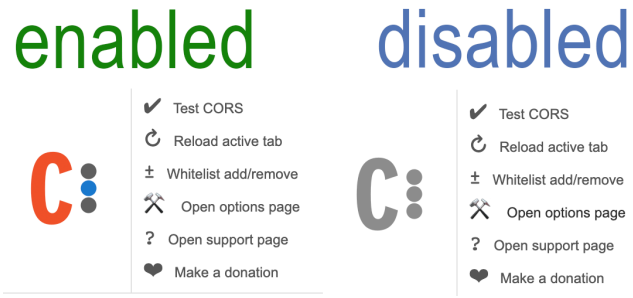
When I try to make API calls from my Angular service to my NestJS back-end, I get an error:



This is because there are default rules in place to make sure that a computer is not trying to access a server on another domain without permission (we're running these two resources on different ports, so they are separate).

Handling **CORS** is a huge topic, but for our purposes of just testing locally, there's a quick fix. We can use Chrome for all of our testing, and there's a plugin in Chrome called Allow CORS: Access-Control-Allow-Origin. When this plugin is installed and activated in your browser, then Chrome will allow the two resources (NestJS and ANGULAR) on their different ports to talk to each other.

This particular plugin has changed appearance over the years, but as of 2020, here is what it looks like when its enabled vs disabled:



WARNING! If you use the Chrome Allow CORS: Access-Control-Allow-Origin plugin, be sure to disable it once you are finished. If that plugin is enabled, it can interfere with authentication / authorization on sites like Gmail, Facebook, YouTube, and several other online resources. Turn it off when you're done with it, otherwise it will seem like parts of the internet are broken.

> Cross-origin resource sharing or **CORS** is not allowed in most browsers by default, but there are browser plugins that you could use to get around this issue for testing purposes on your computer.

MODULE 6 / LESSON Seven / Part one: CORS: Quiz

multiple choice

You might get a CORS error if you are

running two different servers on the same port
running more than two servers on the same port
trying to access one port from another port
trying to run Angular and Nest at the same time
running Angular on the wrong port

MODULE 6 / LESSON Seven / Part two: data on demand

Now that our servers are running, and we have a fully functional back-end, all we need to do is to make the front-end fully communicate with our back-end.

The very first thing we will do is make it so that flights don't load on ngOnInit anymore in our home.component.ts file. We want to load flights only when we hit the go button. So we will make a new method in our home.component.ts file called query() and hook up the "go" button to the query method in our component.

home.component.html

```
<div class="search_box">
  <button (click)="query()">Go</button>
</div>
```

home.component.ts

```
query(): void {
  this.flightsService.getFlights().subscribe(data =>{
    this.flights = data;
  })
}
```

Now, flights will not load unless we click a button that TELLS them to load.

In Angular, components and views work closely together to manage the user experience.

MODULE 6 / LESSON Seven / Part two: Quiz

Type in
Complete the Angular code to connect this button to a component method named "calculate( )"

<button (_____)="_____()">Calculate fare</button>

answers:

click **calculate**

MODULE 6 / LESSON Seven / Part three: flight queries

Now, let's adjust our flight query to, instead of getting ALL flights, to only get the flights that we requested. We will add an **ngModel** to our origin select and our destination select menus respectively. Each one will look something like this:

```
<select [(ngModel)]="selectedOrigin">
```

And then in our component, we will send the ngModel data from our select menus right to our back-end NestJS query route (our route that looks like this):

/flights/query/city1/city2

We will declare our models **selectedOrigin** and **selectedDestination** in the home.component.ts file, and then use those models to store the data of the two cities the user has selected.

In our home.component.ts file, we will send the two cities to the service:

```
query(): void {
  const origin = this.selectedOrigin;
  const destination = this.selectedDestination;

  this.flightsService.getFlights(origin, destination).subscribe(data =>{
    this.flights = data;
  })
}
```

Then, our service needs to be updated to make the API call to the query method in our back-end:

```
getFlights(orig: string, dest: string): Observable<any> {
  return this.http.get(`http://localhost:3002/flights/query/${orig}/${dest}`);
}
```

Now we have a complete query that pulls flights from our back-end into our front-end, based on user selections. **We have a full stack app working from back to front**, and data making its journey all the way from the database, through the NestJS server, into our front-end based on our user query.

Explore on StackBlitz
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step8

View on GitHub
https://github.com/Nmuta/TSFlights1/blob/step8/src/app/flights.service.ts

> Remember that we can use ngModels to store data in the view that we can later grab in the component.

MODULE 6 / LESSON Seven / Part three: flight queries: quiz

Multiple choice

We use ngModels to do which of the following:

store variable data
make http requests
define CSS styles
communicate with the NestJS back-end

MODULE 6 / LESSON Seven / Part four: admin panel

We've come a long way in our Angular and NestJS journey. Our final task now will be to send data to our back-end. Once we have sent data and retrieved data from our back-end, then we have a full cycle, full stack application.

The very first thing that we need to do is construct a form on the Angular side to send a new flight back to the server. This is achieved by way of a simple html form on the front-end that uses [(ngModel)] to keep track of each value in the form:
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9?file=src%2Fapp%2Fadmin%2Fadmin.component.html

We will now send a POST request to send a new flight to our server. The key piece to get this working includes storing all of our flight variables in ngModels and then sending them to our Angular service. Here's what our **admin.component.ts file** looks like:

```
sendFlight(){
  const flight: Flight = {
    origin: this.origin,
    destination: this.destination,
    flightNumber: this.flightNumber,
    depart: this.depart,
    arrive: this.arrive,
    nonstop: this.nonstop
  }
  this.flightService.postFlight(flight);

}
```

Here we essentially gather all of the data that we need to construct a flight object, and then send it to our service, which looks like this:

```
postFlight(flight: Flight) {
  return this.http.post(`http://localhost:3002/flights`,flight).subscribe(data =>{
    console.log("data posted to server!")
  })
}
```

That essentially sends the newly constructed flight object to our back-end, and voila! We now have full ability to create and search for flights in our application.

POST requests, unlike GET requests, send our flight data behind the scenes, without being visible in the browser. This is critical because we don't want to expose data that's going into our database to casual observers over the internet.

CONGRATULATIONS! ~~If you've made it this far, you have gotten~~ You have now been exposed to quite a few tools that you can use to make a full stack TypeScript application from scratch.

You can view the admin sections in the StackBlitz and/or GitHub examples below:

Explore on StackBlitz

https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9

View on GitHub

https://github.com/Nmuta/TSFlights1/tree/step9

As an extension to this course, another lesson follows for people who want to keep going and construct the remaining CRUD operations (UPDATE AND DELETE). Much of it is 'more of the same'. Once you get a handle on how information flows from front-end to back-end, your greatest asset is your ability to pattern match other examples of working code to get something working. We hope you've had a positive experience learning Angular and NestJS. ~~Feel free to ask any questions in the comments below.~~ Happy coding!

> A POST request is similar to a GET request, but remember that POST requests usually send a bundle of data to the server that is hidden in the request body which is more secure.

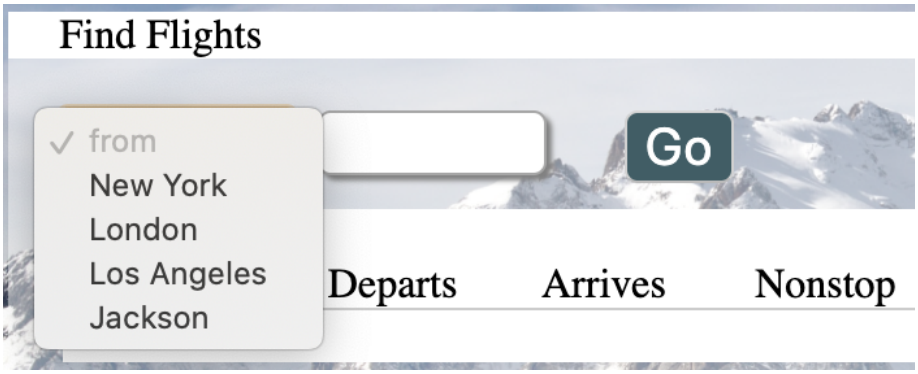
MODULE 6 / LESSON Seven / Part four: admin panel quiz

What is the primary reason that we must use a POST request instead of a GET request to send a flight object to the back-end?

POST requests do not expose data the way GET requests do
We can only use POST requests to make API calls
GET requests only work within Angular
POST requests are faster than GET requests

MODULE 6 / LESSON Seven / Part five: dynamically loading cities

Up until this point, we have hard coded the city choices in our query drop down on the home page:



Here is the corresponding html in our view for that drop down menu:

```
<div class="search_box">
  <select [(ngModel)]="selectedOrigin">
    <option value="" disabled selected>from</option>
    <option value="New York" >New York</option>
    <option value="London" >London</option>
    <option value="Los Angeles" >Los Angeles</option>
    <option value="Jackson" >Jackson</option>
  </select>
</div>
```

The problem to solve here is that we have a drop down list of cities that needs to come from the **database**, so that it's not hard coded into the select menu. That way, when city names are added, updated, or deleted, the drop down menu will accurately show an updated list of cities everytime this page loads.

Our strategy will be that we will write new queries to only get the city names of origin and destination flights. Then we will iterate over those cities and populate our drop down menus in the front end with these dynamic city names from the database.

> *ngFor can iterate over literally any html element, not just div tags. In this case, we will use *ngFor to iterate over options in our select menu to dynamically populate cities.

MODULE 6 / LESSON Seven / Part five: dynamically loading cities : quiz

Reorder

Order the events required to dynamically populate a view with a list of destination cities from our database.

make an XHR to retrieve data from back end
map through the back end data to extract all destinations
filter the destinations list to make it unique
store the list of unique cities
use *ngFor to iterate over unique cities in the view

MODULE 6 / LESSON Seven / Part six: get dynamic cities from the database

We will now implement the strategy that was laid out in the last part.

First, we add two methods to our **flight.service.ts** file to have methods that get our flight origin cities and flight destination cities respectively:

flight.service.ts in Angular:

```
getAllOrigins(): Observable<any> {
  return this.http.get(`${this.backEndURL}/flights/cities/origins`);
}

getAllDestinations(): Observable<any> {
  return this.http.get(`${this.backEndURL}/flights/cities/destinations`);
}
```

Now , in our back end code on NEST, we need to add a controller method to match what our front end is calling: flights/cities/origins and flights/cities/destinations respectively:

in flights.controller.ts in NEST:

```
@Get("cities/origins")
 getOrigins(): Promise<String[]> {
   return this.flightService.getFlightOrigins();
 }


 @Get("cities/destinations")
 getDestinations(): Promise<String[]> {
   return this.flightService.getFlightDestinations();
 }
```

Next, we make new methods in the flightService in our Nest server to handle these new queries:
flights.server.ts in NEST:

```
async getFlightOrigins(): Promise<String[]> {
  return this.flightRepository.query("SELECT DISTINCT origin FROM flights");
}


async getFlightDestinations(): Promise<String[]> {
  return this.flightRepository.query("SELECT DISTINCT destination FROM flights");
}
```

Here we are using TypeORM's "query" method to write a custom query that gets DISTINCT cities from our flights table which will remove all duplicates when getting origins and destinations respectively.

Now, when we hit our http://localhost:3002/flights/cities/origins route, we get an array of objects with the key "origin" and the value being each of the unique cities in our database.

Example of our data now from the flights/cities/origins api call:

[{"origin":"Phoenix"},{"origin":"London"},{"origin":"Lima"},{"origin":"Bogota"}]

In our next lesson part, we will finish the front end view to bring in this new list of dynamic, unique cities from the back end.

View on StackBlitz
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9a

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9a)

View back end on GitHub:
https://github.com/Nmuta/nest_flights/tree/step4

> SQL's DISTINCT keyword can be used to return all unique instances of a value within a database column while removing duplicates.

MODULE 6 / LESSON Seven / Part six: get dynamic cities from the database :quiz

Drag n Drop

Complete the TypeORM method to write a custom query that gets all unique first_name values from a "students" table without returning any duplicates.

```
async getUniqueFirstNames(): Promise<String[]> {
   return this.studentRepository._______("SELECT _________ first_name FROM students");
 }
```

query **DISTINCT** UNIQUE find findOne getOne ORIGINAL

MODULE 6 / LESSON Seven / Part seven: pull in dynamic cities

The last thing that we need to do is pull the dynamic cities from the back end into our Angular front end view. We're already making the call to the database in flight.service.ts in Angular, which is

called by home.component.ts , but our view is not showing the new city data yet. Let's build that now.

In our home.component.ts file, we have built lists called filteredOriginList and filteredDestinationList that each look something like this (with their own unique cities pulled from the back end database) :

[{"origin":"Phoenix"},{"origin":"London"},{"origin":"Lima"},{"origin":"Bogota"}]

In our view, then, we can change our select menu option to look like this:

```
<option *ngFor ="let result of filteredOriginList; let i=index"
value={{result.origin}} >{{result.origin}}</option>
```

Essentially, the list of cities is coming as an array of objects, where the key is the word origin and the value is the unique city in every entry from the database. This will populate the dropdown menu on our home page with all of the unique cities in the database so we don't have to hard code them.

We repeat this process with the destination cities, and now, finally, our origin and destination cities are being dynamically populated in our flights page where we search for flights !

It's easier to see the full code in our live example, so feel free to explore the code on StackBlitz or Github:

View on StackBlitz
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9b

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9b)

> Always know your data. Sometimes it helps to make an api call in the browser just to see how the data is coming back so that you can properly iterate through it on the front end.

MODULE 6 / LESSON Seven / Part seven: pull in dynamic cities : quiz

Drag n Drop

You have a controller with an array of animal names called **animals**. Code a drop down in the view to put all of the animal names in a select menu. Note that the value of each option in the select menus is being set to the index of the animal but the display name should be the actual animal name.

<select>
  <option _____ = "let animal of _____; let i=index" value={{i}}> {{_____}} </option>
</select>

answers:

***ngFor** **animals** **animal** animalName forEach

MODULE 6 / LESSON Eight: update and delete
MODULE 6 / LESSON Eight / Part one: admin flight table

In order to provide the ability to update flights, we need to see a list of all flights ~~in order~~ to decide which flights to delete. In the admin.component.ts file, we will get all flights in the **ngOnInit method** just like we did in the home component. This time we don't need to just grab the origin; we will get all of the complete flights with all of their data. In admin.component.ts we can simply do :

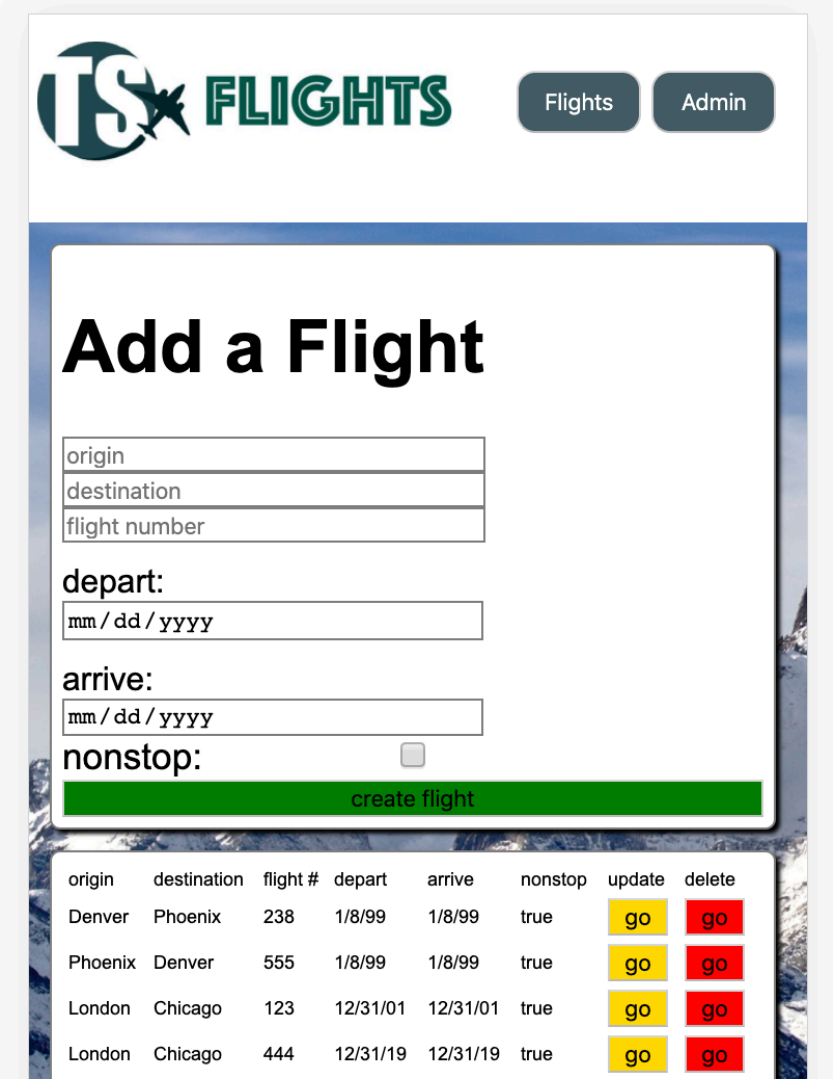
```
ngOnInit(): void {
  this.flightService.getAllFlights().subscribe(data =>{
    this.flightList = data;
  })
}
```

Then, in our view, we can loop through all of the flights in a table in the view:

```
<tr *ngFor="let flight of flightList; let i=index">
        <td>{{flight.origin}}</td>
        <td>{{flight.destination}}</td>
        <td>{{flight.flightNumber}}</td>
        <td>{{flight.depart | date : 'shortDate'}}</td>
        <td>{{flight.arrive | date : 'shortDate'}}</td>
        <td>{{flight.nonstop}}</td>
        <td><button class="update">go</button></td>
        <td><button class="delete">go</button></td>
</tr>
```

Note the **let i=index** part of the *ngFor loop. This is important; we are not using indices in this step of the code, but we will use it later. These indices are like indices in a for loop; they start with zero and count upwards.

For the sake of space, we've abbreviated as much as possible so that all of these columns will fit on a mobile screen. Here's what our view looks like now:



For security reasons, we had to disable updating flights on the live server, but you can view the code live on StackBlitz or download the front and back-end together on your computer to test everything locally:

View on StackBlitz:
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9b

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9b)

View back-end on GitHub:
https://github.com/Nmuta/nest_flights/tree/step4

When iterating over an array in an Angular view, if you are using a table, the best method is usually to implement *ngFor on each table row (tr) as you step through the data. This way each table row represents one row of data from your database.

MODULE 6 / LESSON Eight / Part one: admin flight table: quiz

Drag n Drop

You have an array of fruits in a component file. In the corresponding view, complete this ngFor loop so that each fruit in the array is listed with its number but using natural numbers, so that the counting starts with 1 instead of 0. In essence, the list should look like this: "apple is 1, pear is 2, orange is 3… ", etc.

```
<div *ngFor="let fruit of fruits; let i=index">
   {{__________}} is {{_____}}
</div>
```

fruit **i+1** fruits[i] index+1 fruit[i]

MODULE 6 / LESSON Eight / Part two: update flight strategy

Hold onto your hats, things are about to get fun! Now that we have a list of flights, it's time to update a flight. Angular has so many tools that the actual process of updating a flight could be (and is) done several different ways. The way we will be demonstrating here uses very simple tools that you've already learned.

Let's talk through our process. What we will do is: we will make every single element in the flight list we are getting an **ngModel**, so that it is dynamic. Then we will, instead of simply listing out the flights, ~~we will~~ make each field in the flights editable by making it a text input. Then, when we update the flight, we will pass the entire flight object to the update method. The update method will know which flight to update in the database because "id" is part of the data we are getting from our api call.

It's critical to note here that we have TWO **ids** …. there's an id that relates to the sequential id of each flight in our flights array, and then there's a database id that is one of the fields of each flight. Make sure you keep those straight. This is a common scenario in front-end web development: array based ids are used for iterating over things in the view and keeping track of which elements we are interacting with, and database ids are used later when we send an item to the database to get updated or deleted.

CHANGES TO DATA FIELDS:
in our front-end, we are changing **each flight data field** from something like *this*:

```
  <td>{{flight.origin}}</td>
```

to something like *this*:

```
  <td><input [(ngModel)]='flight.origin'></td>
```

CHANGES TO UPDATE BUTTON:
and then we change our **update button** from *this*:

```
  <td><button class="update">go</button></td>
```

to *this*:

```
<td><button class="update" (click)="update(flight)">go</button></td>
```

Note: due to space issues in the view, for mobile devices, we will not make all fields editable. We will make three editable fields: origin, destination, and flightNumber. Now we can "edit in place" those first three fields, simply type in the new values we want, and hit "go" in the update column. Our update method receives the new values that we need to send to the database in our next lesson part.

View on StackBlitz:
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9c

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9c)

View back-end on GitHub:
https://github.com/Nmuta/nest_flights/tree/step4

> Any variable with a unique name in the component can be made into an ngModel. Although there are many ways in Angular to send form data from the view to a component, ngModels are an easy way to achieve this task.

MODULE 6 / LESSON Eight / Part two: update flight strategy: quiz

Multiple choice

How would we add a line under the <tr> row to show the database id of each flight?

```
<tr *ngFor="let flight of flightList; let i=index">
        <td><input [(ngModel)]='flight.origin'></td>
        <td><button class="update" (click)="update(flight)">go</button></td>
        <td><button class="delete">go</button></td>
</tr>
```

```
<td>{{ i }}</td>
<td>{{ flightList[i] }}</td>
<td>{{ flight[i].id }}</td>
<td>{{ flight.id }}</td>
<td>{{ id }}</td>
```

MODULE 6 / LESSON Eight / Part three: finish updating flights

Thinking ahead, if we've updated all flights in the system, we want the page to refresh in some way to show the changes. Users usually like some visual change to signal that a change occurred. In fact, when we delete a flight (which is coming in the next lesson part), we will also want all flights to be updated in the system. Let's plan for this now.

Originally, we loaded all flights into our admin panel in the **ngOnInit** method. Let's instead have a master update method that can be called whenever we need it. This will be when the page loads, when we update a flight, and when we delete a flight. We can move the flight loading code from the ngOnInit method into the refresh method as such. In **admin.controller.ts**, we can do this:

```
refresh(){
  this.flightService.getAllFlights().subscribe(data =>{
    this.flightList = data;
  })
}
```

Same functionality, different place. Now we can go back to our ngOnInit method and call our new multi-purpose update method :

```
ngOnInit(): void {
  this.refresh();
}
```

Splendid. Now all we need to do is create an update method that sends a flight to our flight service:

```
 update(flight:Flight){
  this.flightService.updateFlight(flight).subscribe(data =>{
    if(data && data['affected']){
      this.refresh();
    }
  });
 }
```

The 'if' statement is important. If we got data **and** the data is an object with an "affected" non-zero value, then that's a pretty good sign that the update worked based on how our form is built. We know that it will have an "affected" key in the api response from examining the api response coming back from NestJS.

With that, all we need to do is design the update method in our flightService.ts :

```
 updateFlight(flight: Flight) {
   return this.http.post(`${this.backEndURL}/flights/${flight.id}/update`,flight);
 }
```

That completes our update functionality. You can view this new code on StackBlitz, but remember that for security reasons, we can't allow live code on the internet to be updated by users, so that functionality is not present. However, you can view both the front-end Angular code and back-end NestJS code, download them and run them together if you want to see it working live:

View on StackBlitz:
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9d

View frontend Angular code on GitHub:
https://github.com/Nmuta/TSFlights1 (branch: step9d)

View backend Nest code on Github:
https://github.com/Nmuta/nest_flights/tree/step4

> Remember that our update flight method passes the id of the flight to be updated in the url, along with the bundle of data that represents what the new flight should look like. In essence, we hit a url like this: https://backend/flights/4/update, and we send it a flight object in the post request as part of the request body.

MODULE 6 / LESSON Eight / Part three: finish updating flights: quiz

type-in

Complete this method in the Angular service to call a back-end /students/:id/update method. Remember that the second parameter sent to the back-end needs to be a student object.

```
updateStudent(student: Student) {
   return this.http.post(`https://olduniversity.com/students/${_______}/update`,_________);
 }
```

student.id student

MODULE 6 / LESSON Eight / Part four: delete flights method

Finally, we need to code a flight deletion process. You may remember that deleting is easy. All we need to do is pass an id to a back-end delete route. This will be very similar to our update method, but we don't have to pass any bundles of data around.

Here is our update button with an updated 'delete' button below it in admin.component.html:

```
          <td><button class="update" (click)="update(flight)">go</button></td>
          <td><button class="delete" (click)="delete(flight)">go</button></td>
```

And our delete method in admin.component.ts:

```
delete(flight:Flight){
  this.flightService.deleteFlight(flight.id).subscribe(data =>{
    console.log('data is', data);
    if(data && data['affected']){
      this.refresh();
    }
  });
}
```

And finally our delete method in the flight.server.ts that sends this to the back end:

```
  deleteFlight(id: number) {
   return this.http.post(`${this.backEndURL}/flights/${id}/delete`, null);
  }
```

All post requests require **at least two parameters**: the first one is the url that you are posting to, and our second parameter is the associated bundle of data. Since there's no data to send, we have to put null as a placeholder to satisfy the 2 param minimum for post requests.

Note that we are using POST requests for both **update** and **delete** actions, even though these two methods are associated with patch and delete http verbs respectively. The reason for this is that, at the time of this writing, some browsers only understand POST and GET requests coming from browsers. To make this work in everyone's browser, we use POST and GET requests only to talk to our back-end.

Optional: If you want, you can wrap all of the code in the admin.component.ts file inside of a confirm dialog conditional statement like this:

```
if (window.confirm('are you sure you want to delete this flight? ')){
  /// your delete method code
}
```

We have included that optional line in the repo below.

View on StackBlitz: (remember that CRUD operations on live server are disabled)
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9e

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9e)

View backend Nest code on Github:
https://github.com/Nmuta/nest_flights/tree/step4

> Since delete methods are fast and easy, it is sometimes helpful to wrap destructive actions like delete inside of a confirm statement, just as a courtesy to your users so people don't accidentally fat finger the deletion of an important resource.

MODULE 6 / LESSON Eight / Part four: delete flights method : quiz

Multiple choice

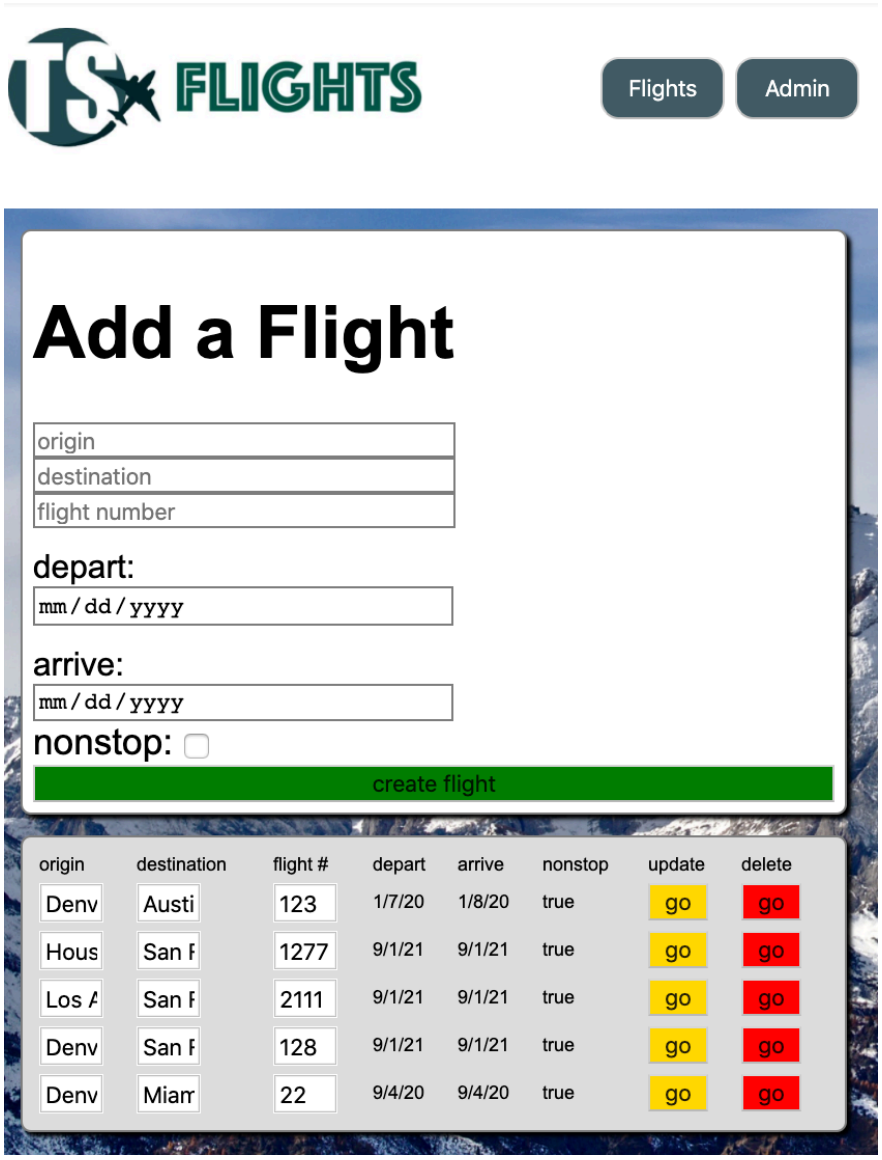
When sending a delete method to a NestJS back-end from Angular using http.post,

1 parameter is required, which is null
at least 2 parameters are required, the 2nd one being null
1 parameter is required, which is the id
at least 2 parameters are required, the 2nd one being the id

MODULE 6 / LESSON Eight / Part five: summary

We now have a full stack application written entirely with TypeScript with all CRUD operations working.

In a nutshell, we have a flights page that dynamically loads all of the flights from our back-end, maps and filters out a list of possible origins and destinations, and puts those options in a select menu to search for flights. Then, we have an admin menu that allows you to create, read, and update all flights. We are using an ngModel to keep track of each row of flights, so you can "edit in place" any given flight and press "go" to update. We have a delete method that uses a confirm pop up to confirm flight deletion. Finally, we are using media queries to resize the page so that it behaves differently depending on the screen size. (In StackBlitz, on a mobile phone, it may take a while for all of the data to load, so be patient). Here's what the admin panel should look like on a mobile device:



Feel free to download the front and back-ends respectively (make sure you're on the correct branches) and play with the code, extend it, and see ways to improve what we have so far. There are a ton of future considerations here. We could add some sort of authentication to have a login for users before they start tinkering with the backend. One way to do this is using JSON web tokens or "JWTs" (pronounced "jawts"). JWTs create a secure connection between the front-end and the back-end to make sure that only people with a valid token are able to access restricted areas. Another quick extension would be to add a simple "loading" notice that tells the user to wait while the flights from the back-end are loading. We have gone ahead and done this; view the StackBlitz live site to see the loader in action.

IMPORTANT: If running the app locally, remember that you need the local CORS plugin for Chrome in order for the front and the back-end domains to communicate with each other. Do a web search for 'control allow access origin chrome' to find it. ALSO: make sure you disable it before resuming normal web browsing, or else key websites on the internet will no longer work.

Thank you for participating in this course! Happy coding.

View on StackBlitz: (remember that CRUD operations on live server are disabled)
https://stackblitz.com/github/Nmuta/TSFlights1/tree/step9e

View on GitHub:
https://github.com/Nmuta/TSFlights1 (branch step9e)

View backend Nest code on Github:
https://github.com/Nmuta/nest_flights/tree/step4 (branch step 4)

> Our decoupled app, when we run it locally, requires cross site scripting because we have two different ports (port 3002 and port 4200 respectively), communicating with each other, which are considered separate domains by the browser. The CORS plugin allows us to make this work by relaxing the security restriction.

MODULE 6 / LESSON Eight / Part five: summary: quiz

Multiple choice (select all that apply)

Which of the following processes do we use POST requests for in our final app?

delete
get all flights
create a flight
update a flight
get one flight

MODULE 6 END OF MODULE EXAM

1. ~~(drag n drop) Angular places sub-pages in a _____ within an Angular view.~~

~~**router-outlet** ngModel router-component page-outlet~~

Revised quiz: Angular places subpages in what part within an Angular view?

- **router-outlet**
- ngModel
- router-component
- page-outlet

2. (multiple choice) In Angular, which of the following constructs ties an HTML element in the view to the Angular router ?

href
~~a~~
routerLink
router
route

3. (drag n drop) Complete the html Angular code so that it reformats raw data data into a user readable date.

 {{flight.depart | _____ }}

date dateTime dateFormat dateObject time

4. In SCSS , if you nest **.box** under **.container**, then the .box styles under .container will…..
(check all that apply)

apply to all divs on the page
apply to .box elements within .container elements
apply to all .box elements on the page
override any general styles listed directly under .container
apply to all .container elements within .box elements also

5. Multiple Choice
   @media screen and (max-width : 499px) { } is a style that IGNORES all elements that are:

   - **500 pixels and higher**
   - 499 pixels and higher
   - 500 pixels and lower
   - 499 pixels and lower

6. (multiple choice)
   Observe the following NEST code. What is the difference here between @Body and @Param?

```
@Patch(":id/update")
async update(@Param('id') id, @Body() flight: Flight): Promise<any> {
  flight.id = Number(id);
  return this.flightService.update(flight);
}
```

   Param is the url parameter and Body is the incoming bundle of data
   Body is the url parameter and Body is the incoming bundle of data
   They are the same
   Body is the flight to delete and param is the data bundle

7. (type in)
   Assume the apiResult$ is an observable. Complete the subscription so that *this.results* is populated with *payload*.

```
apiResult$._____(payload => {
    this.results = payload;
});
```

   answer: **subscribe**