Udacity - Full Stack JavaScript Nanodegree Notes

*January 05, 2024 - N/A*

Courtney Downs

[Foundations of Backend Development 2](#_mx2dpysyxnsw)

[Getting Started with Node.JS 4](#_uhaxtl1nrcko)

[Developing with TypeScript 17](#_jnmsn37wd2rg)

[Unit Testing with Jasmine 30](#_9qj9g7elvu7a)

[Building a Server 43](#_ookummwv4dpn)

[Introduction to Building APIs with Postgres and Express 46](#_fv3pheigo5u6)

[Databases and SQL 46](#_kcu1xqyxa84i)

[Create an API with a PostgreSQL connection 46](#_uzyv3843jswk)

[Create an API with Express 46](#_o4jnbg4tt1p1)

[Authentication and Authorization in a Node API 46](#_sjr5f09ji3i7)

[SQL for advanced API functionality 46](#_q9duqvkmj8vh)

[Foundations of Angular 46](#_eupco0rjunm7)

[Angular Overview 46](#_q20yoky3z36m)

[Components 46](#_9w8zex9ll3vy)

[Libraries and Services 46](#_3qiiq5wppxu9)

[Data 46](#_4wrzx0fhkpwp)

[Foundations of Deployment Process 46](#_14hloeuy2dsr)

[Setting up a Production Environment 46](#_4kqzqrqtb346)

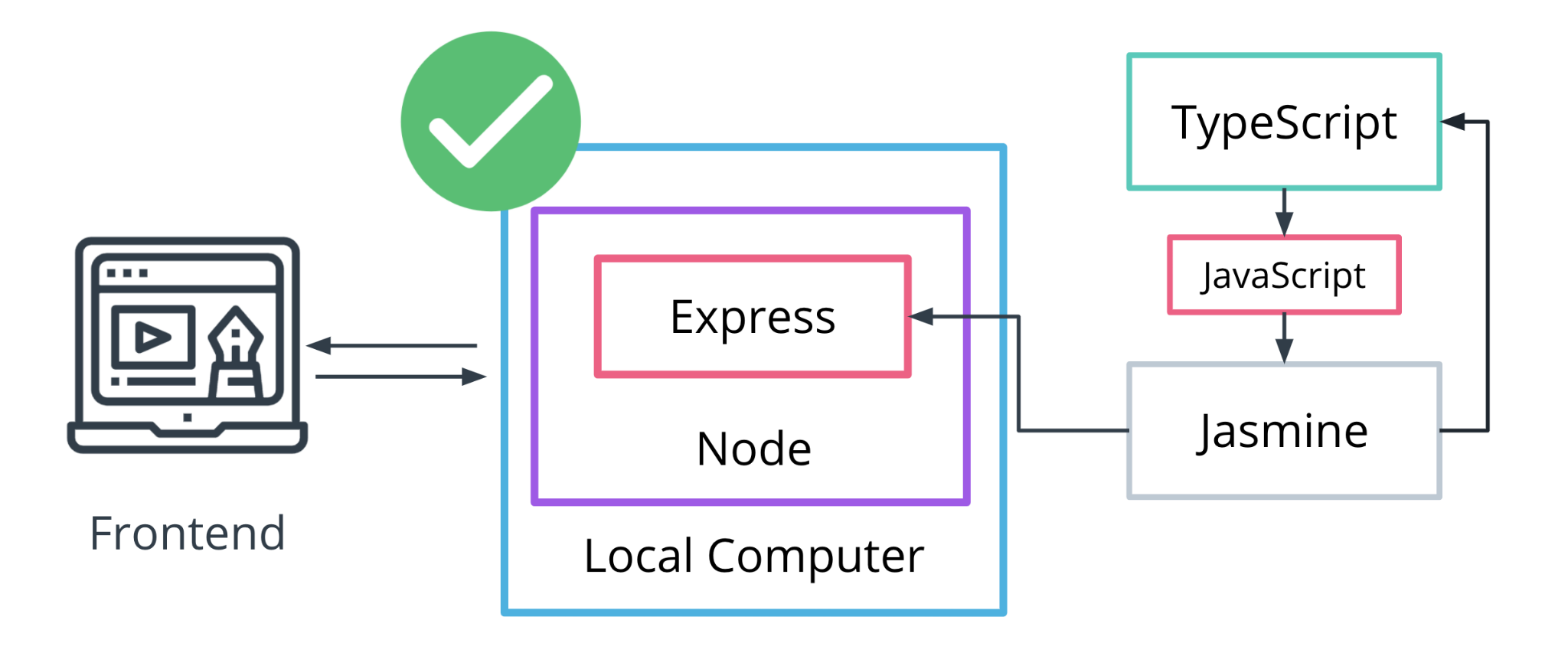
[Interact with Cloud Services via a CLI 46](#_tyamrmjgi8ti)

[Write Scripts for Web Applications 47](#_jkb12dnlw73w)

[Configure and Document a Pipeline 47](#_bbeg0o5odioc)

# Foundations of Backend Development

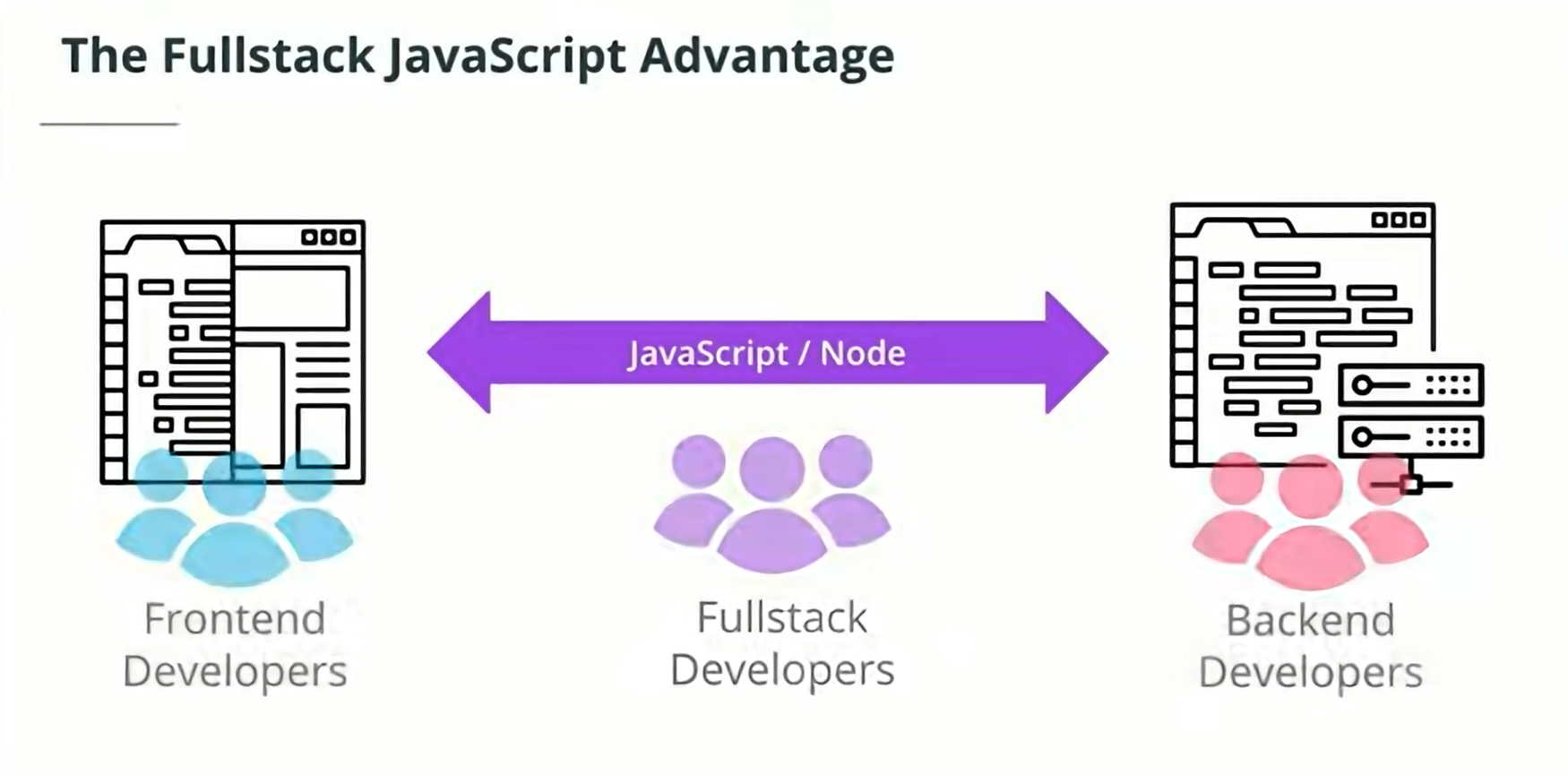
1. Introduction to Backend Development with Node.js
   1. The backend has three parts
      1. **The server**: the computing resource that listens to requests from the frontend.
      2. **The application**: code that runs on the server to process requests and return responses
      3. **The database**: the part of the backend that is responsible for storing and organizing the data.
   2. The backend is responsible for processing the requests that come into the app and managing its data. That can mean different things for different apps. In a simple single-page application, the backend may only be needed to host the website. In other cases, the backend is also used to store, organize, and serve data. The backend also plays an important role in authentication, security, and scalability to ensure that the system has the capacity to handle all of the incoming requests.
   3. The image below shows the architecture of our application. The local computer will host the server and the right side shows that we will be using Jasmine, to ensure that we are writing performant code and catching errors and edge cases before making it to production.



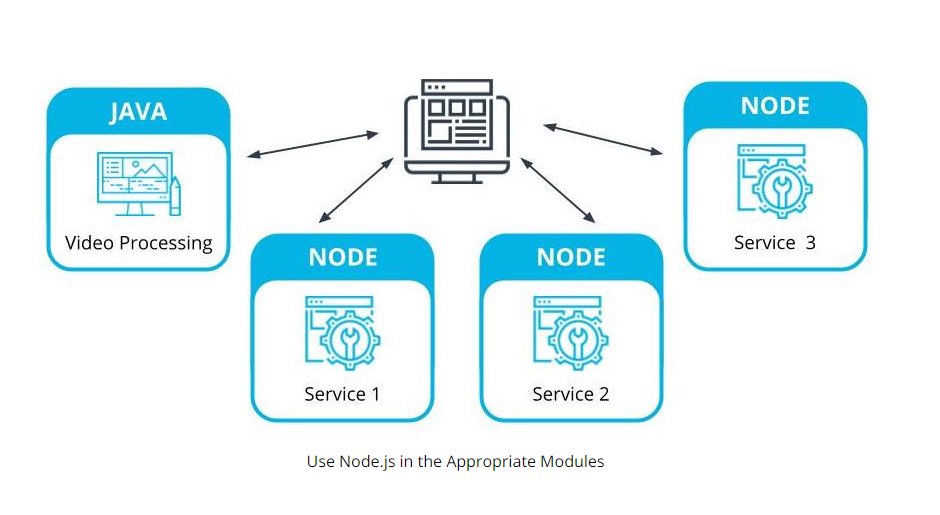
1. Jasmine
   1. Jasmine is a framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It runs in browsers and in Node.js. And it has a clean, obvious syntax so that you can easily write tests.
      1. <https://jasmine.github.io/>
2. Containers
   1. Containers include the runtime, all configurations, and files needed to ensure that all individuals working on a project have the same environment regardless of the operating system or software installed globally. There are several choices to use to create a container for your application. Typically when working on an enterprise project, you will install a container software and run your project within that container which will also include a version of node.js that won't interfere with the version installed globally on your system.
   2. Docker containers
      1. <https://www.docker.com/resources/what-container/>
3. How to Structure a Development Team
   1. A development team’s structure plays a significant role in a project’s success. Surely, other factors like developers’ expertise, experience, and talent are extremely important. Yet, proper management and teams’ structure allow profiting from these factors as well as making the whole development process easier and faster.
      1. <https://stormotion.io/blog/6-tips-on-how-to-structure-a-development-team/>
   2. Generalize Structure
      1. This approach implies building a development team of people with a highly diverse set of skills. Great results are reached thanks to the face-to-face communication and the cooperative effort of all members.
      2. For instance, a front-end developer can also have some knowledge of back-end Java. Or a Project Manager can be familiar with UI design and help with this development part.
   3. Specialist Structure
      1. This arrangement approach means that each team member is an expert in a certain programming language, framework, or technology, and thus, fully responsible for their part of development. You can create teams with their own hierarchy and structure to complete one part of the project. It all depends on the scope of work.
   4. Hybrid
      1. Hybrid project teams imply exactly what the name says. They have both people who focus on a product as a whole and can narrow their focus down when needed.

# Getting Started with Node.JS

* 1. Node.js advantages
     1. Node.js allows for JavaScript to be used on the frontend and backend.
     2. Node.js allows for easy application scaling and maintenance.
     3. Node.js is easy to learn.
  2. Documentation
     1. <https://nodejs.org/api/documentation.html>

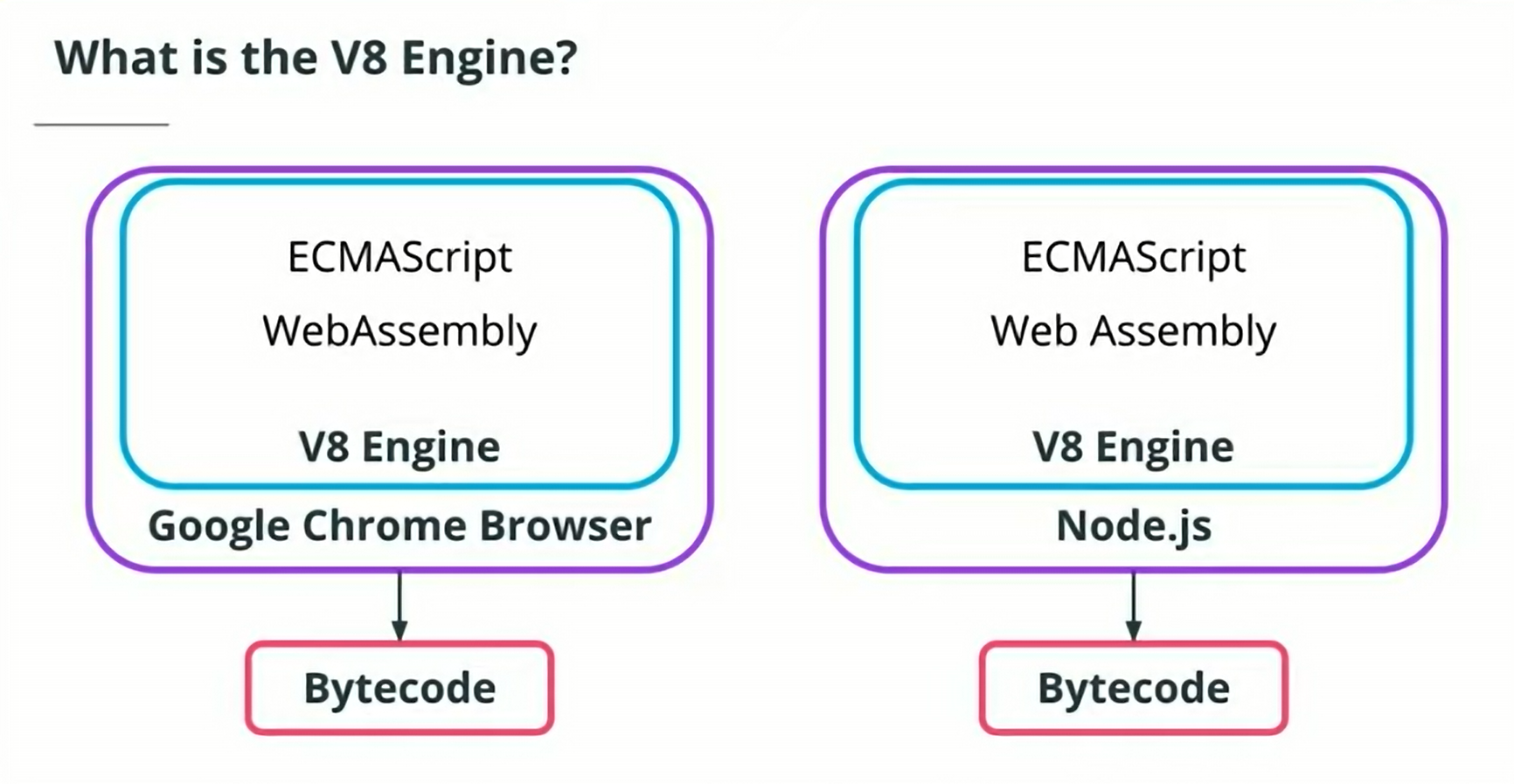


* 1. Node.js disadvantages
     1. Node.js is not well-suited for applications that require heavy processing and computation, like video processing, 3D games, and traffic mapping. In these cases, you can use a **microservice structure** to use a different language for the services that require heavy compute power and use Node.js for the rest.
        1. Note:
           1. Why not just use **Java** for the backend because it's not single-threaded like JavaScript and can handle heavy processing and computation??? Java has frameworks like **Spring** and **Akka** that can handle the event-driven capabilities of JavaScript.

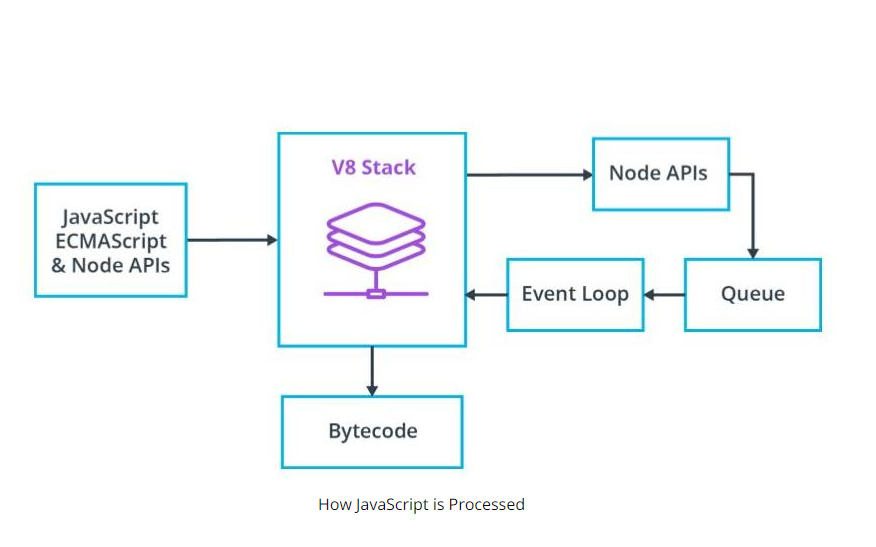


* + 1. **Microservices** are a piece of a larger application. In Microservices applications, the app is broken down into encapsulated parts that can be maintained individually.

1. How to run JavaScript code
   1. Run JS code in REPL(Read, evaluate, print, loop) terminal
      1. Open powershell
      2. Type node
      3. Ctrl+d ( to exit )
   2. Run JS code file
      1. node /path/to/file.js
2. Node.js globals
   1. Node.js and the V8 Engine
      1. The google chrome browser is built on top of an open source V8 engine. The V8 engine processes the following:
         1. **ECMAScript** (which is the JavaScript standard)
         2. **WebAssembly** (runs along JavaScript and allows us to run other languages, like C++/C# on a web platform)
      2. The V8 engine translates the web assembly into byte code.

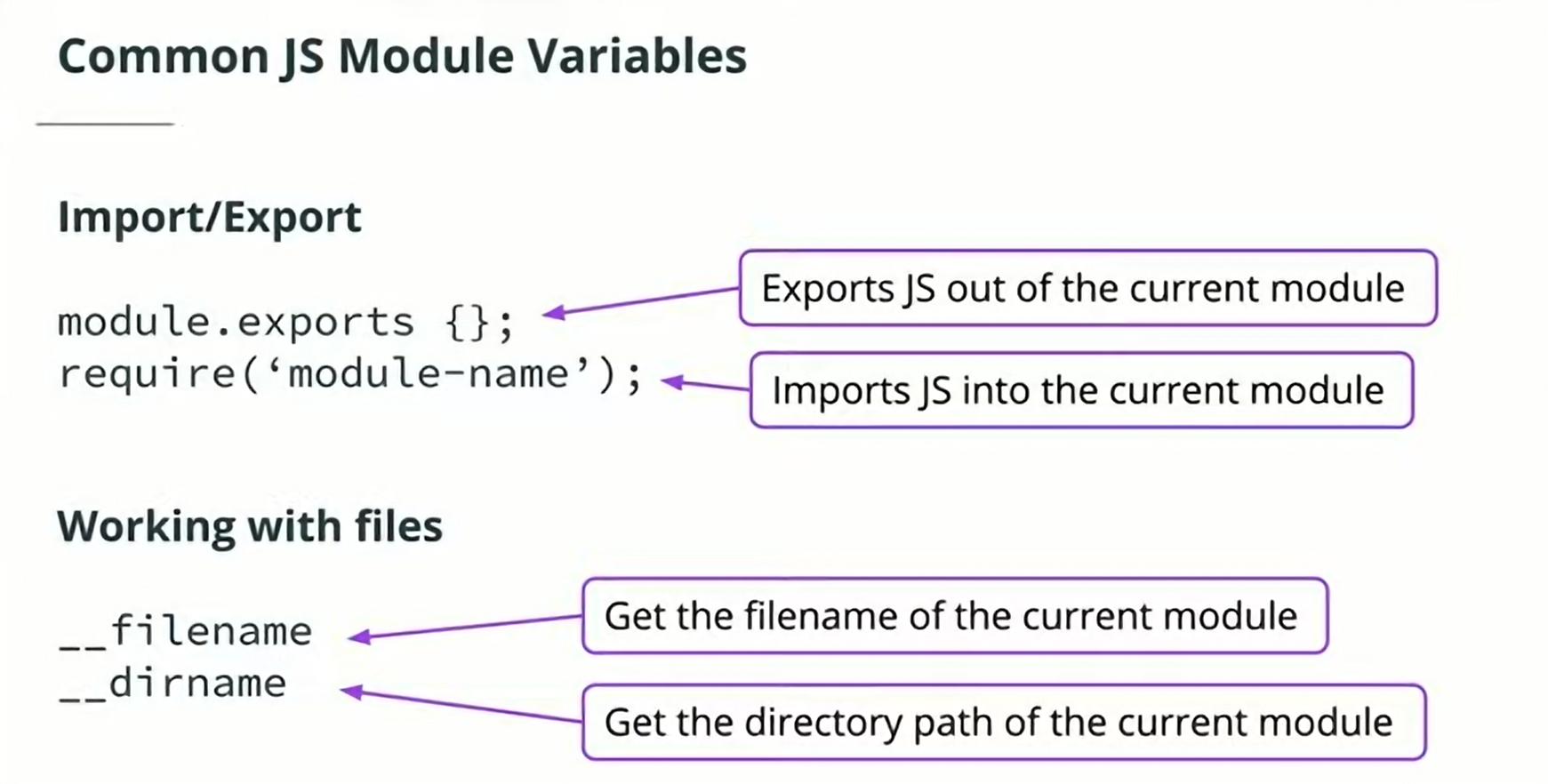


* 1. The google chrome browser is built on top of the open source V8 engine. The V8 engine processes ECMAScript (which is a JavaScript standard) and WebAssembly. WebAssembly runs along side JavaScript and allows us to run code written in other languages (such as C and C# on a web platform) and translates it into byte code.



* 1. Node.js also uses the V8 engine. This is how it converts JavaScript to ByteCode. That bytecode can then be run on a server. This is the magic that allows JavaScript to be used outside of a web platform / browser.

1. Importing and Exporting Modules
   1. Modules are how JavaScript allows you to break your code into focused functional chunks. You can export chunks as modules and import and use them as part of a larger program. Some modules are in node.js by not the browser apparently.
      1. Browser APIs
         1. Timers
         2. Console
         3. Fetch
         4. DOM
         5. Canvas
         6. Etc…
      2. Node APIs
         1. Timers
         2. Console
         3. Fetch - 3rd party
         4. Process
         5. File System
   2. Timers module
      1. Browser APIs
         1. setTimeout()
         2. setInterval()
      2. Node.js APIs
         1. setImmediate()
            1. Allows you to run Asynchronous code within the input/output blocks, without circling back to the start of the event loop.



* 1. Export module
     1. *// working file = util/logger.js*

*// exports as object*

module.exports = {

myFirstFunction: myFirstFunction,

mySecondFunction: mySecondFunction

}

*// using ES6 shorthand property names*

module.exports = {

myFirstFunction,

mySecondFunction

}

* 1. Require module
     1. *// working file = index.js*

*// all functions in util/logger.js are available*

**const** logger = **require**('./util/logger.js');

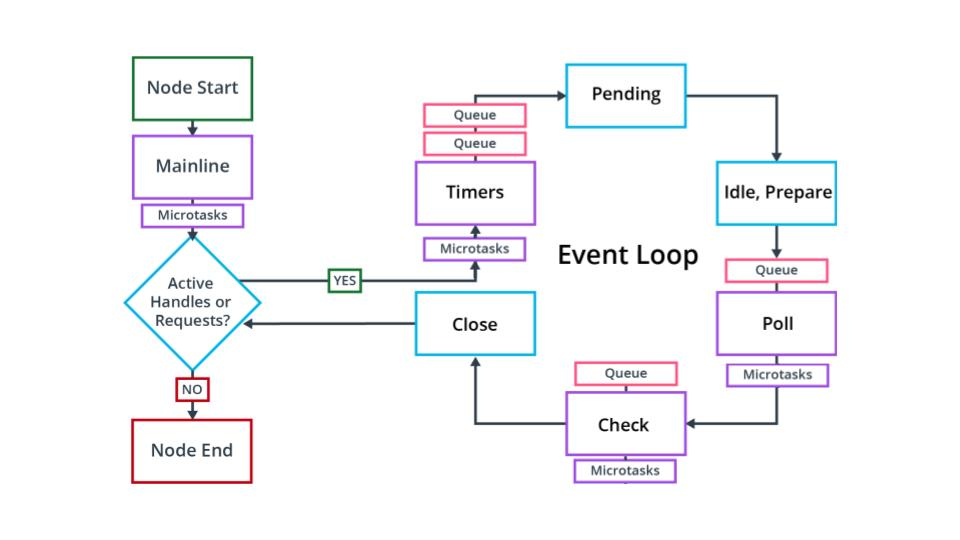
*// using ES6 object destructuring, only myFirstFunction is available*

**const** { myFirstFunction } = **require**('./util/logger.js');

* 1. Path module
     1. **Path.join**: concatenates strings to create a path that works across operating systems.
        1. **console**.**log**(path.**join**('/app', 'src', 'util', '..', '/index.js'));
     2. **Path.resolve:** get the absolute path from a relative path
        1. **console**.**log**(path.**resolve**('index.js'));
        2. *// prints /Users/user/Desktop/app/index.js*
     3. **Path.normalize**: normalizes a path by removing dots and double slashes
        1. **console**.**log**(path.**normalize**('./app//src//util/'));
        2. *// prints app/src/util*

1. The Event Loop
   1. **Code Example:** 
      1. **/ProjectsUdacityFullstackJS/EventLoop**
   2. Nearly every Node.js feature is considered to be asynchronous (non-blocking). This means that we can request an API using **promises** and have our application continue running while that request is being waited for. But how does Node.js process that asynchronous request? Both the Browser and Node.js take advantage of something called the Event Loop.
   3. The Event Loop controls the order in which results (output) of asynchronous tasks (input) are displayed. Think of the Event Loop kind of like the Life Cycle for React.js or Android apps.
      1. Further reading:
         1. <https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c>
   4. Six phases of the event loop
      1. **Timers** - executes callbacks using timers. If there are timers set to 0 ms or setImmediate(), they will run here. Incomplete timers will run in later iterations of the loop.
      2. **Pending** - *internal phase*
      3. **Idle/Prepare** - *internal phase*
      4. **Poll** - process I/O callbacks
      5. **Check** - execute any setImmediate() timers added in the Poll phase
      6. **Close** - loop continues if there are more timers or I/O calls. If all timers and I/O calls are done, the loop closes and the process ends.

**NOTE:** process.nextTick(); will always run at the end of whichever phase is called and before the next phase.



1. Best Practices for Server Side Development
   1. In a professional environment, form becomes as critical to your application’s success as the function. There are often many developers working on the same codebase. For this reason, there are some steps you can take at the start of a project to integrate quickly into a professional environment.
      1. Code Quality
         1. Use VS code extension **‘Prettier’** for syntactic consistency.
         2. Use a linting tool like **‘ESLint’** for making sure that your code follows style guides and that you aren’t calling functions before they are declared.
      2. Use ES6+ and Async/Await
         1. Our goal is always to make our code more maintainable and more readable. ES6 has become the standard and should be used.
         2. You may find yourself using promise chains if a module provides poor documentation on using Async/Await, or your team prefers it, but this is becoming less common.
      3. Keep code small
         1. Applications should be scalable. Node.js is built for scalability. Keep services separate. Node.js encourages the use of modules. Take advantage. Don't make every function its own module, but it is reasonable to group similar functions as individual modules. If you create a module that can be used across your organization, NPM allows for creating [**private npm packages**](https://docs.npmjs.com/creating-and-publishing-private-packages) (discussed later).
         2. With Node.js, there is no reason for your project to turn into an unmaintainable monolith. If your project is small and you never intend to grow it, perhaps monolith architecture works for you. Otherwise, it's worthwhile to learn more about[**microservice architecture**](https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/) and how it can improve an enterprise project.
            1. <https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/>
      4. Handle Errors
         1. With respect to a server, the user should be presented with feedback about what has happened and a solution to continue using the application. The developer should be writing and presenting relevant error messages to locate edge cases, improve application reliability and debug.
      5. Node.js best practices
         1. <https://github.com/goldbergyoni/nodebestpractices?tab=readme-ov-file>
2. What is NPM ?
   1. **npm** is both a tool for managing project dependencies via command line and a website hosting more than 1M third-party packages that can be used for your project.



* 1. Initializing npm and creating a package.json file
     1. Initializing npm will create a package.json within the root of your application folder containing general information about the project.
     2. To initialize npm and go through all of the settings use:
        1. **npm** init
     3. To automatically select all default use:
        1. **npm** init -y
  2. Applications will either include both dependencies and **devDependencies** or just **dependencies**. It is dependent on the team setting up the project. devDependencies are thought of as dependencies that are only necessary for development whereas dependencies are those dependencies used in both development and production.
  3. An example would be needing TypeScript added as a dependency for development, but since it compiles to standard JavaScript to be used in production, TypeScript is not needed for production and therefore could be just a devDependency. Many teams find little use in separating but when learning, it can be a helpful practice to determine which dependencies are only being used in development vs which are also needed for production.
     1. **npm** i module-name

// **install** module to dependencies

* + 1. **npm** i --save-dev module-name

// **install** to dev dependencies

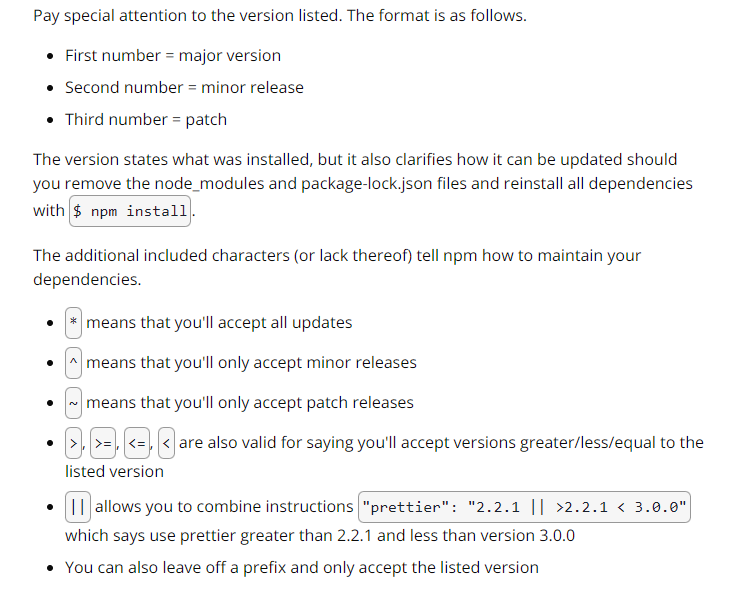
* + 1. **npm** i --save-dev module-name@1.19

// **install** a specific version (1.19 here) of module

* + 1. Installing dependencies adds the dependency to your package.json file in the format:
       1. "devDependencies": {

"prettier": "^2.2.1"

}



* 1. Prettier is a code formatter that will ensure you're keeping your code consistent. It's commonly added to projects to ensure all members on a team are formatting in a consistent way such as always using semicolons, trailing commas, and single quotes. It can be configured to the preferred settings of the team and works well with additional tools like linting.

We are able to add it to a project with NPM by doing the following:

* Locate prettier on npmjs.com to get the install script and other information.
* Run the install script npm i --save-dev prettier.
* Add a prettier script to your package.json file. The script you choose can vary dramatically depending on the project. The one below will only overwrite files located in the src directory that are js files. You may need a [**different script**](https://prettier.io/docs/en/cli.html) depending on the project.
  + 1. *// example config file, path structure to check, and write fixes*

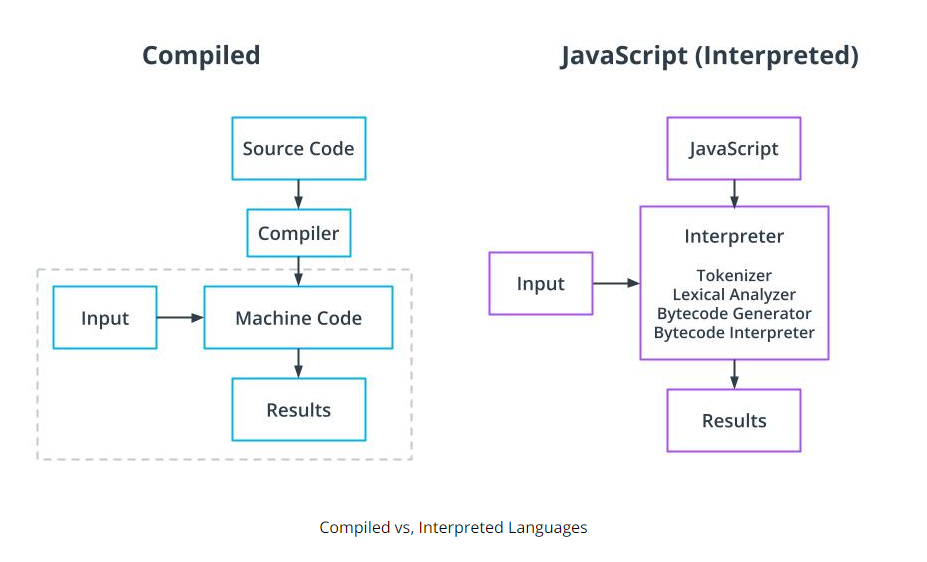
"prettier": "prettier --config .prettierrc 'src/\*\*/\*.js' --write"

*// or*

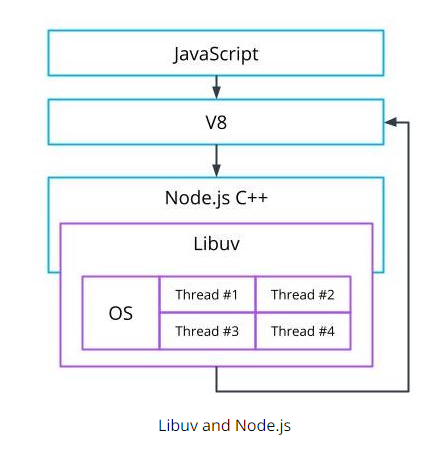
"prettier": "prettier --config .prettierrc src/\*\*/\*.js --write"

* Create a .prettierrc file for any custom configurations.
* Run npm run prettier to run prettier (or whatever you named your script).

1. Is Node.js single threaded
   1. JavaScript is single threaded, is Node.js?
      1. Compiled vs. Interpreted language
         1. Compiled language
            1. Language is written and compiled to machine code inside of an application
            2. Errors are detected during compiling
            3. The code won’t compile until it’s error-free
            4. Examples: C, Java, C++, Erlang, Go
         2. JavaScript is an interpreted language
            1. Errors are found when the code is run
            2. The interpreter translates and runs code one statement at a time
            3. Interpreted code runs more slowly
            4. JavaScript runs the same on different systems because it is run in the browser not the CPU.



* 1. Node.js is an interpreter
     1. Node.js is mostly written in C++.
     2. Many node modules also include some C++ code
     3. Other modules include python or C
  2. C Libuv library gives Node.js access to a thread pool
     1. Main thread runs async JavaScript
     2. Libuv takes advantage of the operating system's asynchronous interfaces before engaging the thread pool
     3. The thread pool is engaged for events that require more processing power including compression and encryption tasks
     4. The default thread pool includes four threads



# Developing with TypeScript

1. **Code Example:** 
   1. **/ProjectsUdacityFullstackJS/TypeScriptInit**
   2. **/ProjectsUdacityFullstackJS/TemplateProject ← *for creating new .ts projects***
2. Why TypeScript is Important
   1. JavaScript is loosely typed and all variable’s types are inferred. The type inference can lead to programmer errors, like trying to pass a string to a function that performs math operations.
      1. **sum(2,’2’) => ‘22’**
   2. TypeScript is a compiled language, it compiles (transpiles) from TypeScript to JavaScript. In short, TypeScript is a static and string typed superset of JavaScript. *(JavaScript with types)*
   3. How to run .ts?
      1. **Option 1:** 
         1. npm install -g typescript
         2. tsc .\<filename>.ts
         3. node .\<filename>.js
      2. **Option 2:** 
         1. npm i typescript
         2. Create a folder for your new project
         3. npm init -y
         4. Update package.json

“scripts”:{

“build”:”npx tsc”

}

* + - 1. npm run build
      2. npx tsc - -init *(installs the tsconfig.json file)*
  1. Update the default Typescript configuration file **tsconfig.json**
     1. You should always check your compiler options to note what you are transpiling to as well as your output directory. Common output directory names include ***dist****,* ***build****,* ***prod****, and* ***server***

{

"compilerOptions": {

"target": "es5",

"module": "commonjs",

"lib": ["ES2018", "DOM"],

"outDir": "./build",

"strict": true,

"noImplicitAny": true,

},

"exclude": ["node\_modules", "tests"]

}

* + 1. You will see many more options available than what is above. Your application may require additional settings to be configured, but these are typically the main settings to start with. This file is where you can tell TypeScript how strict it should be while checking your code and what to ignore.

1. Importing modules
   1. *// Rename the module*

**import** 'name' **from** 'module';

*// Use destructuring to pull in specific functions when they are exported individually*

**import** {**function**, **function**} **from** 'module';

1. Exporting modules
   1. *// Export an individual function or other type of object in code*

**export** **const** **myFunction** = () => {};

*// Export a single item at the end*

**export** **default** object;

*// Export a list of objects*

**export** **default** {object1, object2};

1. Implicit Typing
   1. TypeScript will automatically assume types of objects if the object is not typed. It is best practice to allow TypeScript to type immutable variables and simple functions implicitly.
      1. **const** myNum = 3;

*// TypeScript implicitly types myNum as a number based on the variable*

* 1. Implicit Typing is a best practice when the app is self-contained (meaning that it does not depend on other applications or APIs) or variables are immutable.

1. Explicit Typing
   1. The developer does explicit typing when he applies a type to the object.
      1. **let** myVar: number = 3;
2. Basic Types
   1. string
   2. number
   3. boolean
   4. unknown
   5. undefined - used when a variable has yet to be defined

**const** **myFunc** = (student: string | **undefined**) => {

**if** ( student === **undefined** ){

*// do something*

}

};

* 1. null
  2. Union Types
     1. Used when more than one type can be used

**let** studentPhone: (number | string);

studentPhone = '(555) 555 - 5555';

studentPhone = 5555555555;

* 1. Return types
     1. void
     2. never - when the function will never return anything, such as with

functions that throw errors or infinite loops.

*(These effectively never reach the return statement)*

**const** myFunc = (student: any): any => {

*// do something*

};

* + 1. any - used when the type can be anything

*(Kind of defeats the purpose of typescript, it should be avoided)*

**const** myFunc = (student: any): any => {

*// do something*

};

* 1. Type Assertions
     1. Type Assertions are used to tell TypeScript that even though TypeScript thinks it should be one type, it is actually a different type. Common to see when a type is unknown

**const** myFunc = (student: unknown): string => {

newStudent = student **as** string;

**return** newStudent;

}

* 1. Object-Like Types
     1. Arrays
        1. Arrays can either accept a single type or multiple types.

*// only accepts strings*

**let** arr: string[] = [‘a’, ’b’, ‘c’];

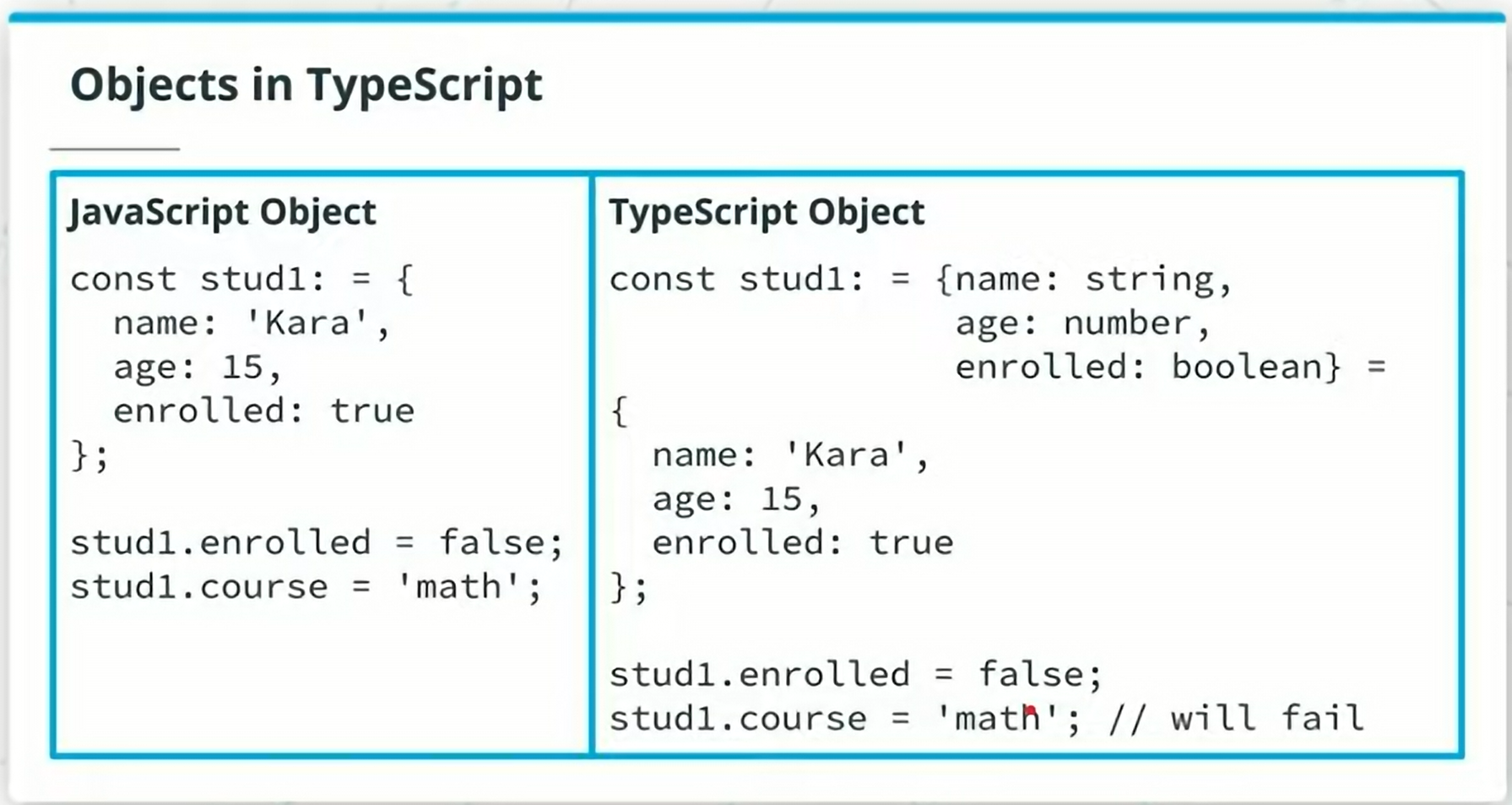
*// accepts strings or numbers*

**let** arr2: (string | number)[] = [1, ‘a’, ‘b’, 2];

* + 1. Tuples
       1. These are specific to TypeScript. JavaScript does not support them.
       2. Use tuples when you know exactly what data will be in the array, and you will not be adding to the array or modifying the type for any value.
          1. **let** arr: [string, number, string];

// ['cat', 7, 'dog']

* 1. Working with objects in TypeScript



* + 1. Translating JavaScript objects into TypeScript
       1. Difficult to read
       2. Can not add properties and Not extendable
       3. Better to use Interfaces or Type Aliases
    2. Interfaces
       1. With TypeScript, interfaces are simple used as the blueprint for the shape of something. Interfaces can be used to create functions but are most commonly seen to create objects.
       2. If you have an interface, you can declare that interface a second time and add additional properties to it. *(Is this interface overloading?)*
       3. Use PascalCase for naming interfaces
          1. Pascal case

ThisIsPascalCase

* + - * 1. Camel case

thisIsCamelCase

**interface** **Student** {

name: string,

age: number,

enrolled: boolean

};

**let** newStudent:Student = {name: 'Maria', age: 10, enrolled: true};

* 1. Duck Typing
     1. TypeScript uses duck typing for interfaces, meaning that even though you may say a function takes in an argument of interface A, if interface B has the same properties of A, the function will also accept B. Interface A is the duck, and Interface B walks and quacks like a duck, so we'll accept it as a duck too.
  2. Optional and ReadOnly properties
     1. TypeScript gives the ability to create both optional and read-only properties when working with object-like data.
     2. Optional
        1. Use when an object may or may not have a specific property by adding a ‘?’ at the end of the property name.

**interface** **Student** {

name: string,

age: number,

enrolled: boolean,

phone?: number *// phone becomes optional*

};

* + 1. ReadOnly
       1. use when a property should not be able to be modified after the object has been created. Keep in mind that this will only produce TypeScript errors and that the actual properties can still technically be changed as read-only does not exist in JavaScript. The closest thing in JavaScript is `Object.freeze` which will make all properties of the object unable to be modified.

**interface** **Student** {

name: string,

age: number,

enrolled: boolean,

readonly id: number *// id is readonly*

};

* 1. Type Aliases, TypeScript Classes and Factory Functions
     1. Type Aliases
        1. Type aliases do not create a new type; they rename a type. Therefore, you can use it to type an object and give it a descriptive name. But like the object type, once a type alias is created, it can not be added to; it can only be extended. Meaning, if you wanted to create an object from a type alias and then a second with additional properties, you would need to extend the type alias and make your second object with the extended alias. This makes interfaces the preferred method for creating objects.

type Student = {

name: string;

age: number;

enrolled: boolean;

};

**let** newStudent:Student = {name: 'Maria', age: 10, enrolled: true};

* + 1. Classes
       1. TypeScript classes are very similar to classes introduced in JavaScript ES6, except they also have types.

**class** **Student** {

studentGrade: number;

studentId: number;

**constructor**(grade: number, id: number) {

**this**.studentGrade = grade;

**this**.studentId = id;

}

}

* + 1. Factory Functions
       1. If Factory Functions remain your preferred way of creating JavaScript objects, they are still usable within TypeScript. To create a factory function with explicit typing, create an interface with the object's properties and methods and use the interface as the return type for the function.

**interface** **Student** {

name: string;

age: number

**greet**(): **void**;

}

**const** studentFactory = (name: string, age: number): Student =>{

**const** greet = ():**void** => **console**.**log**('hello');

**return** { name, age, greet };

}

**const** myStudent = **studentFactory**('Hana', 16);

1. Generics
   1. Reusable components that can be used with different types
   2. Why not just use **any** instead of messing with generics?
      1. With generics, you can ensure that the types of your inputs match the types of your outputs of your functions.
   3. Uses angle brackets syntax <T> where T is the type parameter
      1. *Java Example: ArrayList<String>, ArrayList<Integer>, etc..*

**const getItem = <T>(arr: T[]): T**{

return arr[1];

}

**getItem([**‘cat’,’dog’**]); // ‘dog’**

**getItem([**5,6**]); // 6**

* 1. If you want to ensure that the function returns a number
     1. getItem**<number>**([5,6]); // Returns 6

**Code Example:**

**/ProjectsUdacityFullstackJS/AsyncAndPromises**

1. Asynchronous TypeScript
   1. async/await always returns a promise
   2. Use Promise<type> to set the type returned
      1. **const** myFunc = **async** ():**Promise**<**void**> => {//do stuff};

1. Promises
   1. A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value.
2. TypeScript with Third-Party Modules
   1. It’s common to want to use third-party modules in your project. Those modules typically contain functions that we want to execute in our code. Since we are importing modules in our own code to use, those modules need to be typed, since we are using typescript.
   2. In general, most packages found on <https://www.npmjs.com/> have type definitions that can be installed in addition to the package. Typically searching the name of the package followed by types will locate the type definitions, or you can simply try running:
      1. npm i --save-dev @types/packageName
      2. npm i --save-dev @typescript/packageName

**NOTE:** Type definitions are increasingly created by the creators of the package, but it's also common for them to be created by third parties and maintained by the open source community. So it's possible that type definitions could be outdated, missing specific functions, or were never created in the first place.

**Code Example:**

**/ProjectsUdacityFullstackJS/Lodash**

* 1. Lodash Module
     1. Very popular library for working with functional javascript. It provides great utilities for working with arrays, objects, strings, and other types of structures.
     2. <https://www.npmjs.com/package/lodash>
     3. Installation
        1. npm i lodash
        2. npm i –save-dev @types/lodash
  2. There are occasions when you are using a third-party module and there are no type definitions available. When this happens, the code will compile with errors but it will still run as expected. However, it is best practice to add type definitions when they are missing to help reduce errors when using third-party libraries.
  3. How to create type definitions when one is missing:
     1. Create a folder called **‘types’** in your root directory with a subfolder

called **‘3rdparty’**

* + 1. Create a file in your 3rdparty folder called **index.d.ts** (it could be a more specific name than index).
       1. The .d is standard for type definition files
       2. You’ll find those in node\_modules > @types
    2. Within your definitions file, import the node module with the missing definition.
    3. Use the declare keyword to declare which module the definition will be for, followed by curly braces to contain the definition.
    4. Write the definition which will likely be a class or an interface for a function. It's common to see interfaces for function since the function is actually defined elsewhere.
    5. Open tsconfig.json and find //"typeRoots": ,. Uncomment the line and update it to include your new types directory. **"typeRoots": ["./types"]**,
    6. The function should then be usable in your code.
       1. *For the example code in the project /ProjectsUdacityFullStackJS/Lodash if you wanted to make a type definition for the Lodash module’s function .multiply() then your index.d.js file above would look like the following:*

**import** \_ **from** 'lodash';

declare module 'lodash' {

**interface** **LoDashStatic** {

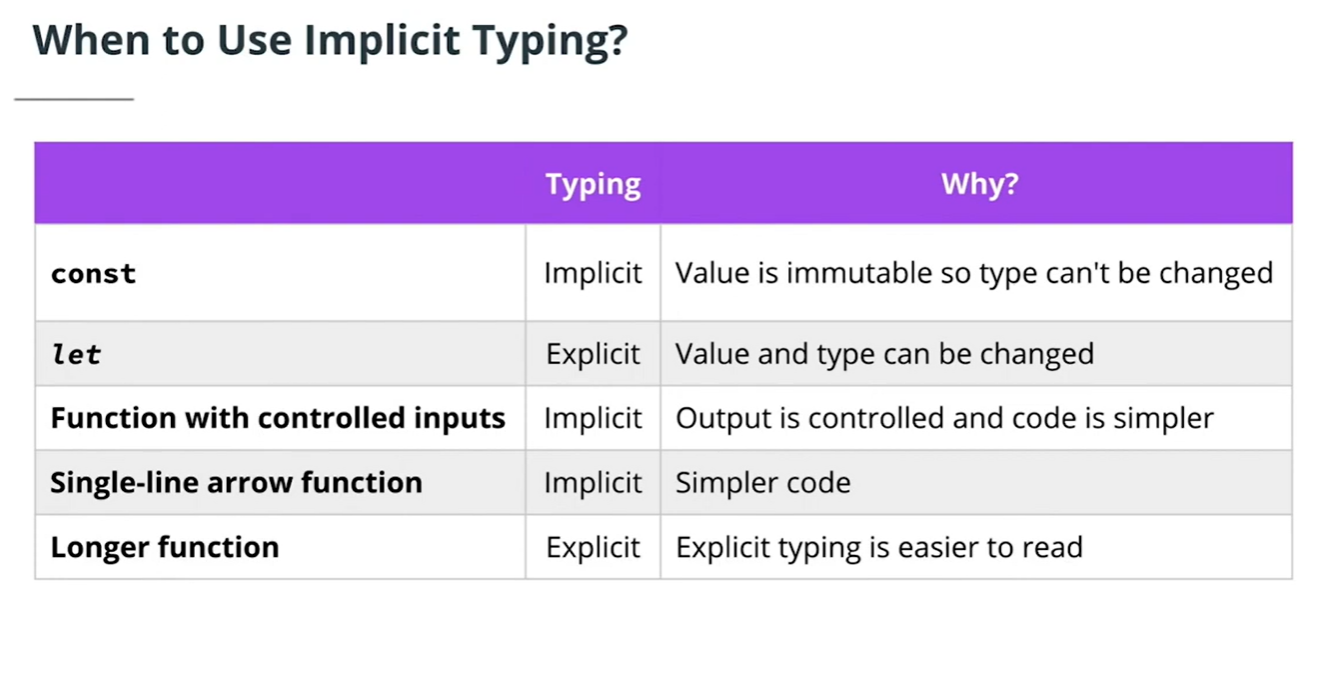
**multiply**(multiplier: number, multiplicand: number):

number;

}

}

1. TypeScript Best Practices
   1. Use noImplicitAny in **tsconfig.json** to prevent errors created by Typescript assuming Any type.
   2. Turn on all strict checking by setting strict to true in your tsconfig.json settings.



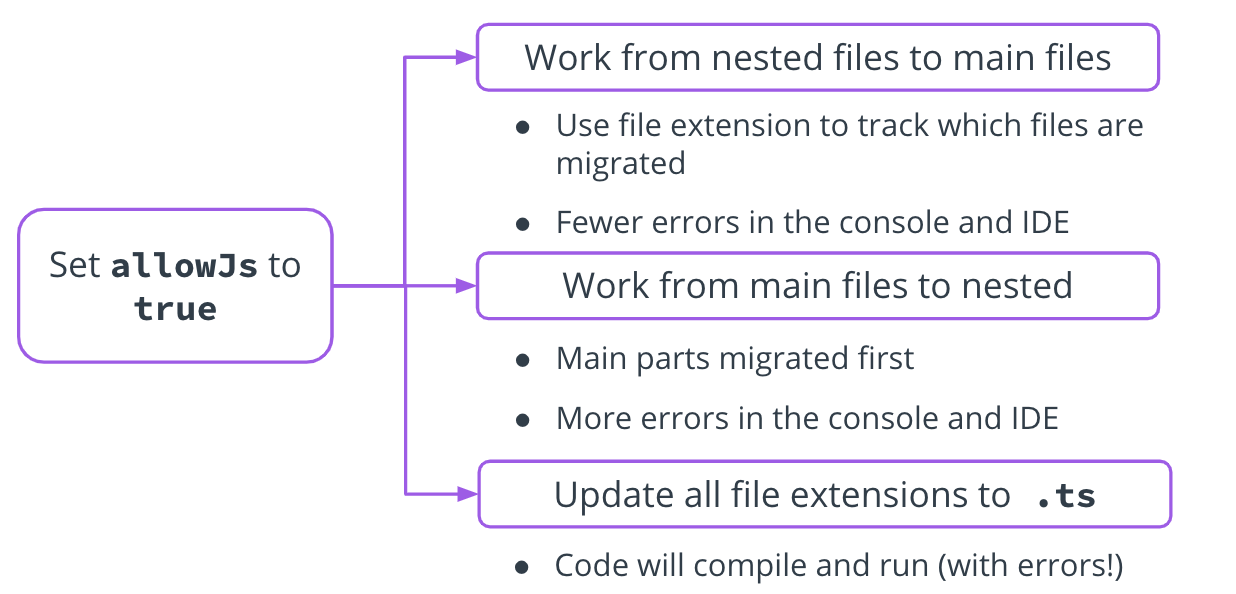
* 1. Take advantage of the latest EcmaScript features
     1. async/await
     2. ES6 modules
     3. Nullish coalescing
     4. Destructuring
     5. Spread operator

1. Migrating to TypeScript
   1. TypeScript makes migrating from JavaScript easy. Even for monolithic legacy JS projects.
      1. Different project structures will have to be migrated differently.
         1. Microservices vs. Monolithic
      2. Decide whether to migrate all at once or file-by-file.
         1. It’s common to migrate slowly, file-by-file and then conducting migration testing to ensure that nothing breaks along the way.
      3. Add Typescript to *each* service if the project uses microservice architecture.
      4. For monolithic architecture, move to a src/dist to keep working files separate from compiled Javascript.
         1. Check if this affects any of the other paths within the project, as they might not be automatically updated (although most IDEs do).
         2. If it doesn't automatically update, you can use a path module.
      5. To exclude folders you don't want to be migrated, utilize the configuration file.
      6. Make sure to add all type definitions for Third-Party Modules

### 

Third-Party Module Type Definitions

* To find the definitions, search through dependencies and dev-dependencies going through each dependency and adding definitions for each. If a dependency doesn't have definitions, you can create your own.



3 Typescript Migration Strategies

By setting allowJS to true in the config file, you. can follow the following approaches:

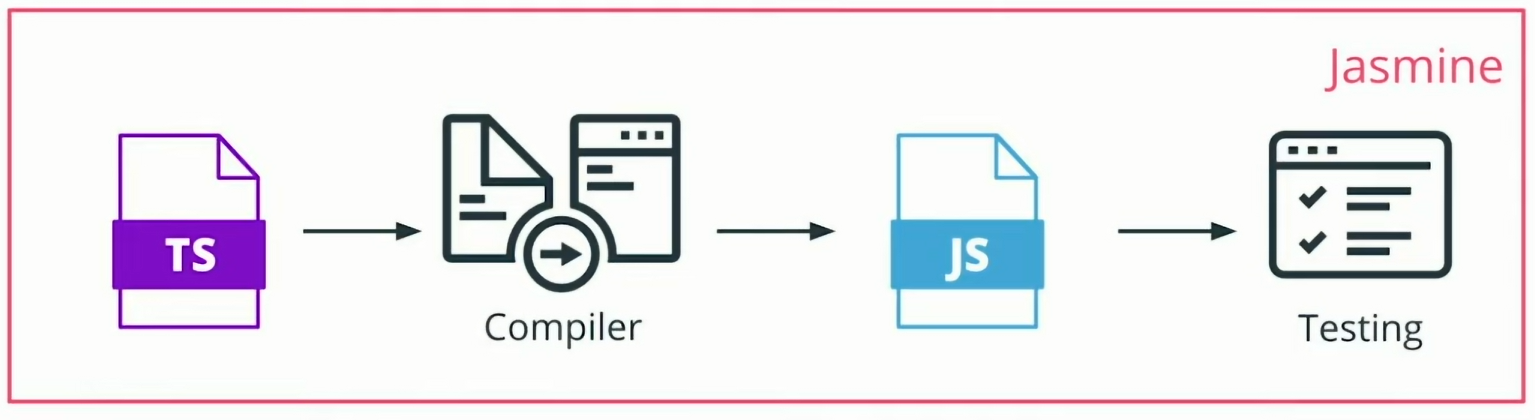
* Work for nested files to main files
  + Use file extension to track which files are migrated
  + Fewer errors in the console and IDE
* Work from main files to nested:
  + Main parts migrated first
  + More errors in console and IDE
* Update all files to .ts
  + Code will compile, but run with errors.

# Unit Testing with Jasmine

1. What we’ve done so far to reduce the chance of errors:
   1. *Added Prettier library for standard code formatting*
   2. *Added ESLint library to improve style and structure*
   3. *Using TypeScript to reduce type errors with JS.*
2. Other JavaScript testing frameworks
   1. Jest
   2. Mocha
   3. Puppeteer
      1. What are the differences between them and why choose Jasmine ?
         1. <https://raygun.com/blog/js-toolbox-part-3/#what-to-choose>
3. Jasmine allows us to integrate automated unit testing to ensure the following:
   1. Tests are run when code is integrated
   2. Alerts us when refactored code affects other code
   3. Find issues early, when they are easier to fix
   4. Encourages simple, modular, easy-to-maintain code
4. Testing Approaches *(Jasmine supports both)*
   1. Behavior Driven Development (**BDD**)
      1. You write tests for your application based on user behavior.
   2. Test Driven Development (**TDD**)
      1. You write tests that look for specific results when the code is run.
         1. A feature request comes in
         2. Before writing code for the feature, tests are written for the most simple functionality of the feature that includes edge cases and failure expectations
         3. Tests fail due to lack of code for the feature
         4. Code is written to make tests pass
         5. Code is refactored to be most concise and easy to read
      2. Cycle continues until the feature is complete.
   3. The tests remain in the codebase and as the feature is built upon or other features are added, the tests will ensure the feature continues to work as expected.
5. How to come up with tests before writing the code?
   1. <https://www.browserstack.com/guide/what-is-test-driven-development>
   2. <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>
6. Unit Testing Best Practices
   1. Pseudo-code

* Test file structure and file names should match the app.
* Describe and name the tests to be easy to read and maintain.
* Design app features with pseudo code to inform tests.
  + Pseudo code provides an overview of the application complexity and finds the easiest pieces of the test to write, build, refactor, and reiterate.
  1. DRY (Don't Repeat Yourself)
* Write short tests that allow you to pinpoint why the test is failing.
  + Try writing short, uncomplicated tests by first starting with an object with data that should pass and test each value in the object.
  + Try this again with an object with data that should fail unless the appropriate error is passed to ensure error handling is a standard, not an afterthought.
  1. Tests should be reliable
* Tests should only fail when there are bugs in the tested code.
* Avoid conflicts with other tests.
* Call the correct objects for each test. The wrong objects may have the wrong input and create an error.
* Import the correct file for the test to avoid errors.

1. Configuring Jasmine
   1. When working with Jasmine, server-side, Jasmine tests the compiled JavaScript.



* 1. **Install** 
     1. npm i jasmine
     2. npm i jasmine-spec-reporter
     3. npm i --save-dev @types/jasmine
  2. **Add testing script** 
     1. Inside of package.json, add this to the scripts option.

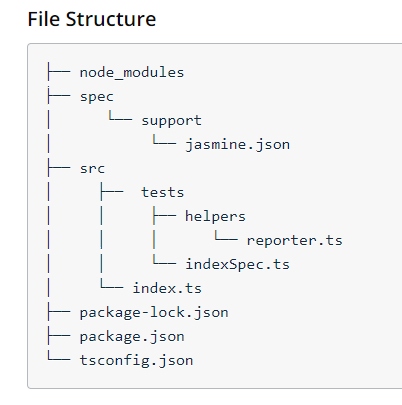
“scripts”:{

…

“jasmine”: “jasmine”

}

* 1. **Set up file structure** 
     1. At the root level create a new folder called ***spec***
     2. Inside of spec create another folder called ***support***
     3. Inside of support create a file called ***jasmine.json***
     4. Inside of src create a new folder called ***tests***
     5. Inside of tests folder, create a file called ***indexSpec.ts***
        1. *Has the tests related to index.ts*
     6. Inside of tests folder, create another folder called ***helpers***
     7. Inside of helpers, create a file called ***reporter.ts***



* 1. Best Practices For File Naming
     1. When creating files for tests, a best practice is to name the **.ts** file the same as the **.js** file to be tested with **Spec** appended to the end. The more tests needed to be run, the more test files will need to be created. Be sure to follow this best practice to keep track of the **test** file that contains the tests for each **.js** file.
  2. To run Jasmine tests you need to first run the build script and then the test script.
     1. **npm** run build
     2. **npm** run jasmine
        1. To avoid having to always run these commands one after the other, you can update the package.json file to have another script:
           1. “test”: ”npm run build && npm run jasmine”
           2. Now you can just use “**npm run test**”
     3. **IMPORTANT**: The TemplateProject in the ProjectsUdacityFullStackND repo already has all of the setup listed above that is needed to run jasmine unit tests.
     4. Additional resources for Jasmine
        1. <https://www.npmjs.com/package/jasmine-spec-reporter/v/6.0.0>
        2. <https://github.com/bcaudan/jasmine-spec-reporter/tree/master/examples/typescript>

1. Writing Unit Tests
   1. Jasmine uses Suites and Specs
      1. **Spec**: an individual test
      2. **Suite**: a collection of similar tests related to one function
      3. Tests should cover all intended behaviors
      4. Error handling should also be tested
      5. <https://jasmine.github.io/api/3.6/matchers.html>
2. Jasmine Syntax

**Code Example:**

**/ProjectsUdacityFullstackJS/Jasmine**

(run test with command: “*npm run test*” )

* 1. Use the **describe** keyword followed by a short description of what the suite is testing and one or more specs.
  2. A best practice is to start a sentence with “it” and then complete the sentence with the description of what the suite is testing.
     1. **describe**(“suite description”, () => {

**it**(“describes the spec”, () => {

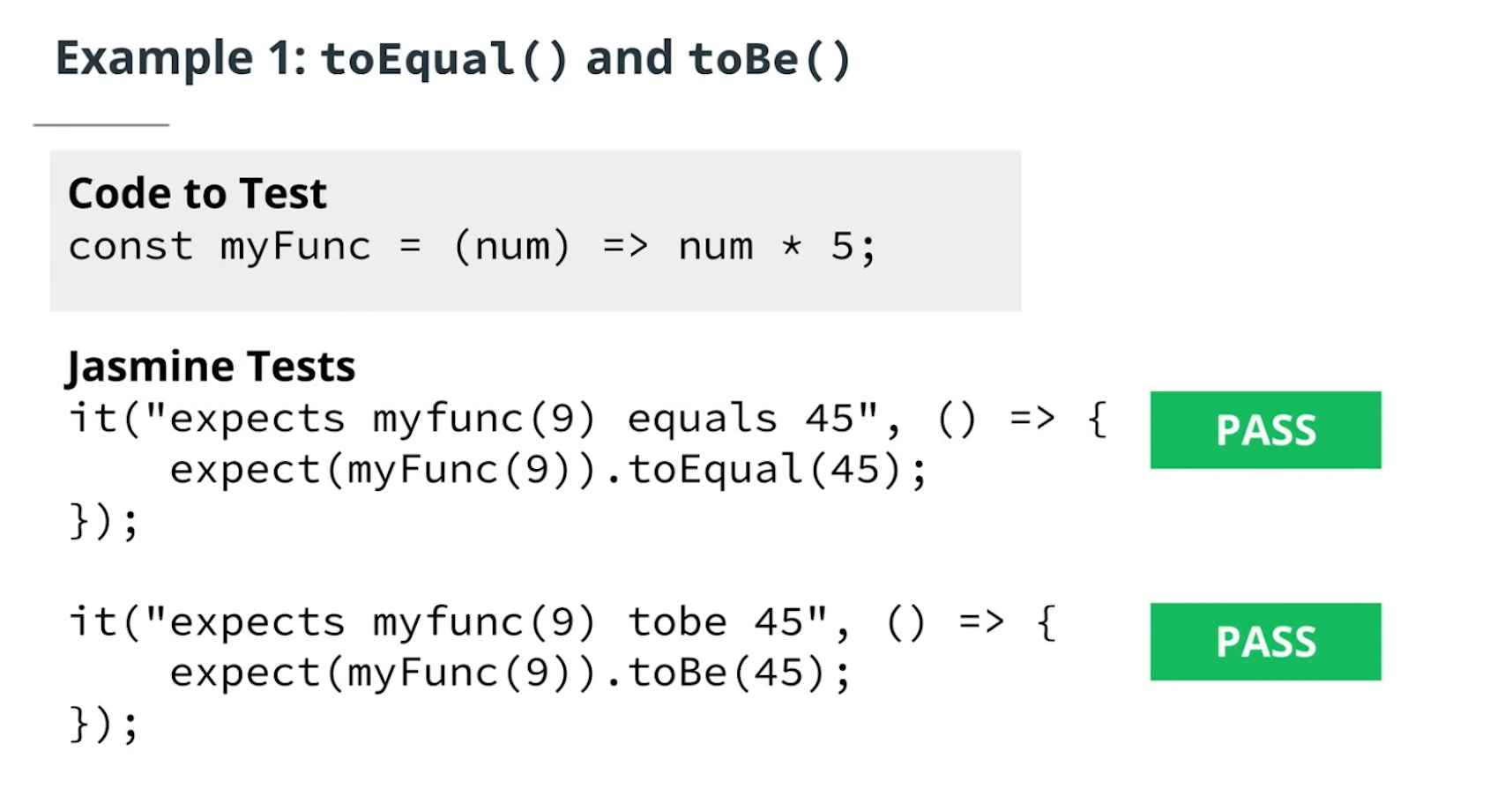
**const** myVar = true;

**expect**(myVar).**toBe**(true);

});

});

1. Comparisons
   1. Can compare strings, numbers, objects, or arrays
   2. .**toEqual**(expected value)
      1. deep checks if the tested value is ***equal to the expected value***
   3. .**toBe**(expected reference)
      1. checks if tested object is ***the same object***

******

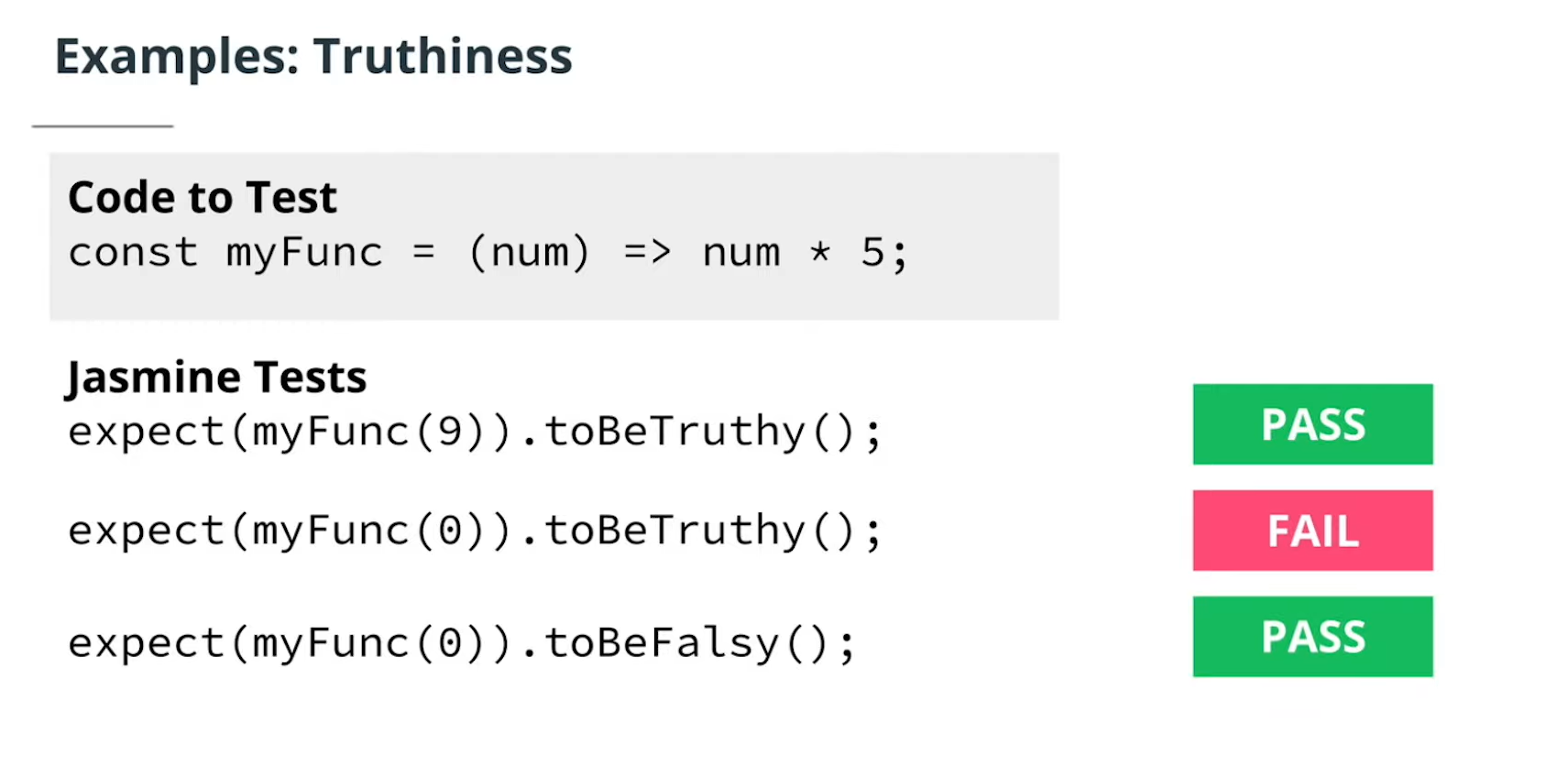
1. Truthiness

**Code Example:**

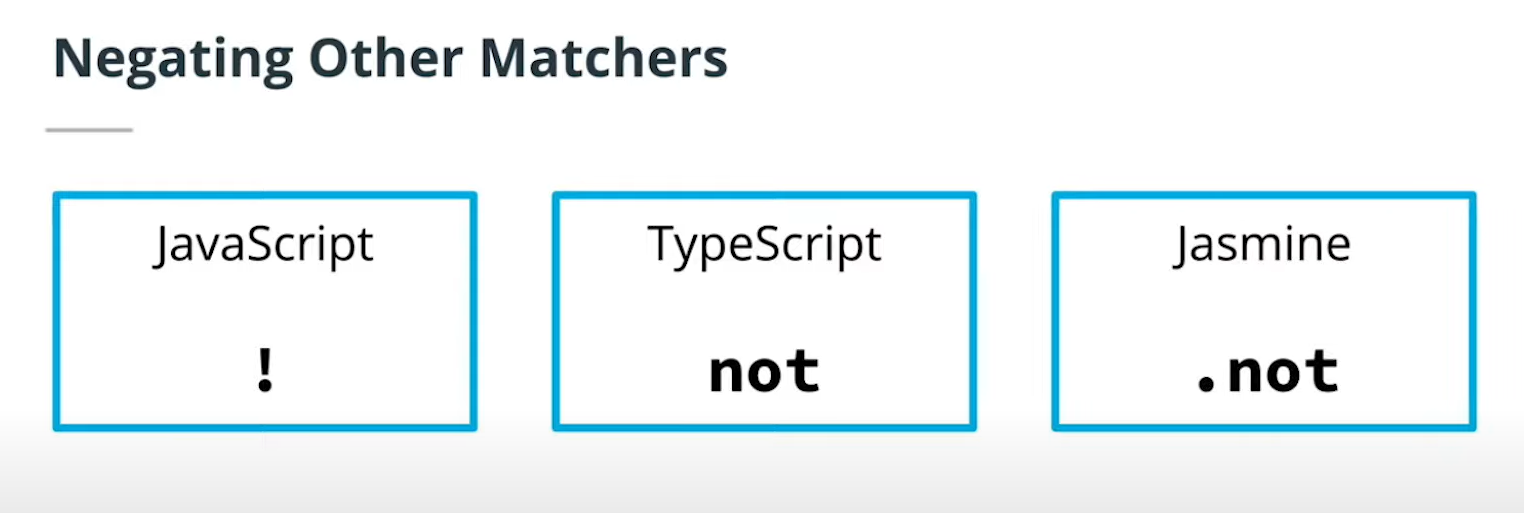
**/ProjectsUdacityFullstackJS/JasmineSetupTeardown**

(run test with command: “*npm run test*” )

* 1. These are tests that are useful with boolean types.
  2. .**toBeTruthy()** passes when
     1. The expectation has any non-zero value
     2. The expectation evaluates to true
  3. .**toBeFalsy()** passes when the value is:
     1. 0
     2. '' (an empty string)
     3. undefined
     4. null
     5. NaN
  4. **.toEqual()**
     1. Use this if you only need a Boolean true or false
     2. When comparing two objects, .toEqual() checks if they have the same keys (properties) and values recursively.
     3. When comparing two arrays, .toEqual() checks if they have the same elements in the same order. It also performs a deep comparison for nested arrays or objects within the arrays.
     4. For other types like strings, numbers, booleans, or null/undefined, .toEqual() behaves similarly to the strict equality operator (===).



1. Numerical Matchers
   1. .**toBeCloseTo**(expected value, precision )
      1. Passes if the value is within a specified precision of the expected value
      2. Precision is optional and represents the number of decimal points to check *(defaults to 2)*
   2. **.toBeGreaterThan**(expected value)
   3. **.toBeLessThan**(expected value)
   4. **.toBeGreaterThanOrEqual**(expected value)
   5. **.toBeLessThanOrEqual**(expected value)



1. Other Matchers
   1. .**toContain**(expected value)
   2. .**toMatch**(expected value)
   3. .**toBeDefined**()
   4. .**toBeUndefined**()
   5. .**toBeNull**()
   6. .**toBeNaN**()
   7. Custom mathers that you create.
      1. <https://jasmine.github.io/tutorials/custom_matcher>
2. Exceptions in Jasmine
   1. .**toThrow**(expected value)
   2. .**toThrowError**(expected value, expected message)
   3. Expected value and expected message parameters are optional

1. Testing Asynchronous Code

**Code Example:**

**/ProjectsUdacityFullstackJS/JasmineAsync**

(run test with command: “*npm run test*” )

* 1. The key to testing async code is letting Jasmine know when it’s ready to be tested.
     1. Using **async/await** syntax makes testing easier
        1. Jasmine syntax is very similar to JavaScript syntax.
        2. Add async before the asynchronous function call
        3. Add await before the return
        4. Testing occurs after the return

**it**(‘expects asyncFun() result to equal value’,

async () => {

**const** result = await asyncFun();

**expect**(result ).**toEqual**(value);

});

* + 1. Using **promise** syntax with Jasmine
       1. Promise values are included in the return statement
       2. Test is run in the **.then()** statement that is chained to the return value.

**it**(‘expects asyncFun() result to equal value’, () => {

**return** asyncFun().then( result => {

**expect**(result ).**toEqual**(value);

});

});

* 1. Testing promise resolution and rejection with **[ES6 Promise Matchers Library](https://www.npmjs.com/package/jasmine-es6-promise-matchers)**
     1. **.toBeResolved()** tests if a promise is resolved and will return true if the promise is resolved
     2. **.toBeRejected()** tests if a promise is rejected and will return true if the promise is rejected
     3. **.toBeRejectedWith(expected value)** tests if the expected error is returned
  2. With both async/await and promises, should the promise be rejected, or throw an error, the test will fail.

1. Endpoint Testing
   1. An endpoint is the URL of the REST API with the method that gets, adds to, or modifies the data of an API in some way.
   2. **Why does endpoint testing matter?**
      1. Confirms that the server is working
      2. Confirms that endpoints are configured properly
      3. More efficient than manual testing
   3. The ‘supertest’ framework
      1. Jasmine doesn’t support endpoint testing, so we need to use another framework called **Supertest** to test the status of responses from servers.
   4. Setting up Supertest
      1. Install the dependency

**npm** i supertest

* + 1. Add type definition to allow the code to compile without TypeScript errors.

**npm** i --save-dev @types/supertest

* + 1. Import supertest in the spec file

**FILE: /src/tests/indexSpec.ts**

**import** supertest **from** 'supertest';

**import** app **from** '../index';

**const** request = **supertest**(app);

**describe**('Test endpoint responses', () => {

**it**('gets the api endpoint', **async** (done) => {

**const** response = **await** request.**get**('/api');

**expect**(response.status).**toBe**(200);

**done**();

}

)});

**FILE: /src/index.ts**

**import** express **from** 'express';

**const** app = **express**();

**const** port = **5000**;

// set endpoint

**app.get**('/api', (req, res) => {

**res**.send('Hello, world.');

});

// check for port to avoid already in use error.

**app.**listen( port, () => console.log(

'Listening on port ${port} !’));

**export default** app;

* + 1. Create and run tests

**npm** run test

1. Performing Tasks Before and After Tests
   1. For example, there are times when running a test may require connecting to a database before you can run your test and disconnecting afterwards.
   2. Run only a specific test/suite
      1. Add **f** in front of **describe** or **it**
      2. Removes clutter in the terminal

**fdescribe**("A spec", **function**() {

**it**("is just a function, so it can contain any code",

()=> {

**expect**(foo).**toEqual**(1);

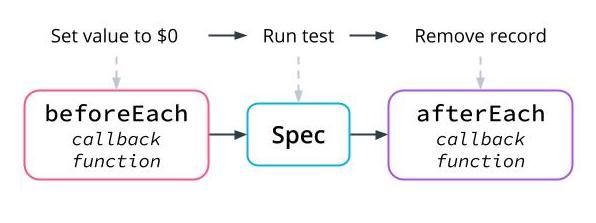
});

});

* 1. Skip a test or suite
     1. Add **x** in front of **describe** or **it**
     2. Helpful to avoid a time consuming test

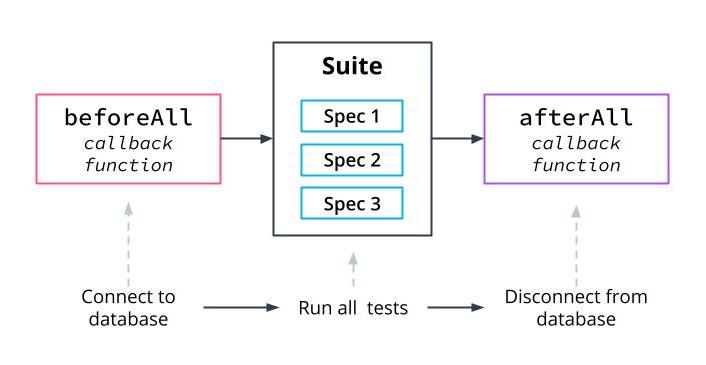
**beforeEach** and **afterEach**

* beforeEach takes a callback function where we can tell the test to perform a task ***before each test is run.***
* afterEach is used if there is a task to be run ***after each test is complete***.

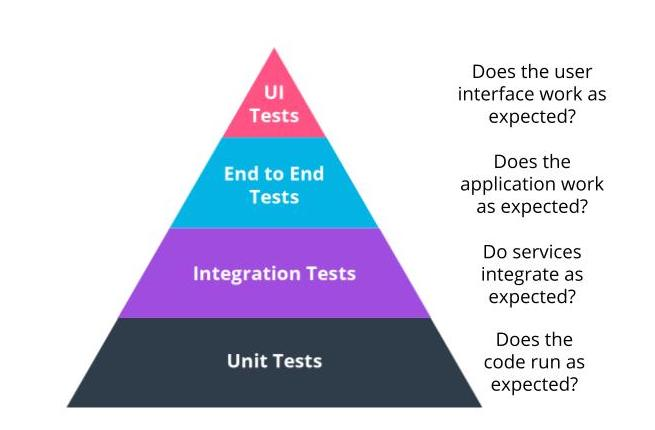


**beforeAll** and **afterAll**

* To perform an operation ***once before*** all the specs in a suite, use beforeAll
* To perform an operation ***once after*** all the specs in a suite, use afterAll.



1. Additional points on Unit Testing
   1. Microservice Testing: Unit Tests
      1. <https://medium.com/@nathankpeck/microservice-testing-unit-tests-d795194fe14e>
   2. The Testing Pyramid



* + 1. **UI Tests:** 
       1. Automated UI test
          1. Scraping the website to ensure that there are no broken links.
       2. Manual UI test
          1. Resizing the screen to test that all images are responsive
    2. **End to End Tests:**
       1. Focused on how all of the services of the application work together by going through the expected process of the user from start to finish.
       2. For an e-commerce solution, this may be using a chat service that has you fill out a form before entering the chat and then ensuring the service is able to pull your order successfully and give it to the customer service representative, who is then able to respond.

That's just one path the user could take on the application. All possible interactions across all of your services should be tested.

Use Jasmine with **Selenium** to emulate user interactions

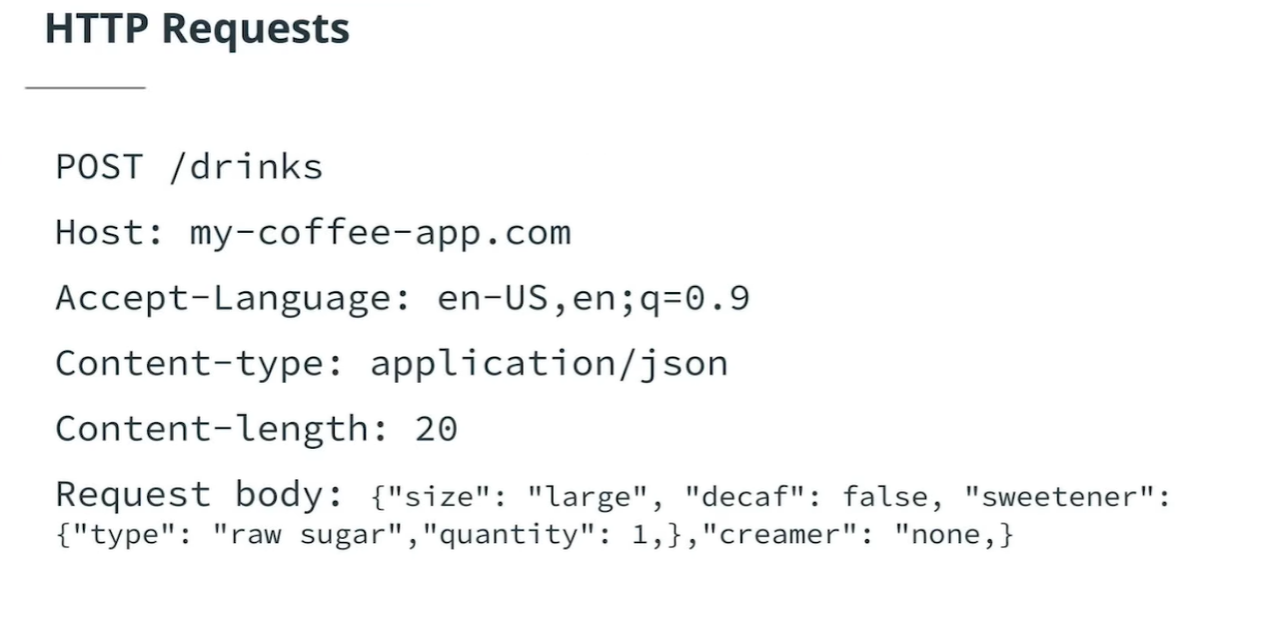
* + 1. **Integration Test:**
       1. It is broadly defined as testing combined modules or components to make sure they integrate as expected.
       2. An example would be an e-commerce app with a shopping cart and purchase page, maybe two different components. But the purchase page needs to be able to pull from the cart in order to work.
    2. **Unit Test:** 
       1. Focuses on whether an individual piece of code can be run as expected.

Unit Test Additional Help:

* What makes a good unit test design ?
* How to know that you are covering all of the possible unit tests that your app needs??

# Building a Server

1. How does a server work?
   1. When the user opens a URL in the browser, the browser sends a request to the server. The server processes the request, and the response is sent back to the browser. The requests that the browser sends are HTTP requests.
   2. HTTP request includes the following:
      1. The request will include a method and where to access that request.
         1. **Example below**: POST is the method, /drinks is where to access the request.
      2. Headers, which include information about where to find the resource, information about the request
      3. Request Body



1. Common HTTP Requests
   1. **GET** - retrieves data from the server
   2. **POST** - sends data to the server
   3. **DELETE** - removes data from the server
   4. **PUT** - replaces data on the server
   5. **PATCH** - updates data on the server
2. Query Parameters
   1. Query strings are parameters in the URL, identified by a ‘?’
   2. To chain multiple parameters together in a query string, use ‘&”

https://coffee.com/search**?decaf=true**

Query String

https://coffee.com/search**?decaf=true&size=large&creamer=soy**

Multiple parameters separated by ampersands

1. HTTP Response Status Codes
   1. After a request is made, the server sends a response, including the status code. The status code is a 3 digit number about how the server handled the request.

**Status Code Range Example Code**

* + 1. **100-199**: Information 100 Response not ready yet.
    2. **200-299**: request was successful 200 Request was successful
    3. **300-399**: request was redirected 301 Resource moved Permanently

307 Temporary Redirect

* + 1. **400-499**: client side error 400 Bad Request from poor syntax

401 Incorrect permission for request

405 Method Not Allowed

* + 1. **500-599**: server side error 500 Internal Server Error

1. Idempotency

*“A request is said to be idempotent when making multiple requests to the API that are identical produce the same result”*

* 1. Idempotency and API Methods:
     1. The only method not considered idempotent is POST.
     2. POST adds a new resource each time; on the other hand, GET, DELETE, PATCH, and PUT act on the same resource each time with the same result.
  2. Idempotency and Security
     1. Get:
        1. Safe because the database doesn’t change
        2. Endpoint is stored in session history
        3. Can be cached
        4. Often logged
     2. Post:
        1. Endpoint **not** stored in session history
        2. Protects user data from being inadvertently exposed

1. Why Use Express
   1. Node.js has the ability to create a server with the HTTP module. So, you might be asking yourself why use Express in the first place? Express is a framework that builds on top of the HTTP module and makes it easy to use less code to do the following:
      1. Set up the server
      2. Work with routes
      3. Apply middleware
         1. <https://expressjs.com/en/guide/writing-middleware.html>
         2. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction#introducing_express>
   2. Express Entry Point
      1. Convention is to create the server entry point as a file called **index.js**, **server.js**, or **app.js**. This file creates the server, and tells you where to go to access your routes.
      2. Package.json
         1. Displays the author and copyright information of the project. It also lists all of the pieces that made the project happen.
         2. Shows a way of starting the server and working with other needed scripts, for testing and compiling.
      3. Routes
         1. Routes are stored in a routes folder. There is 1 file per route. That file contains all of the endpoints for available actions.
            1. Example Route: /books
            2. Example Endpoints:

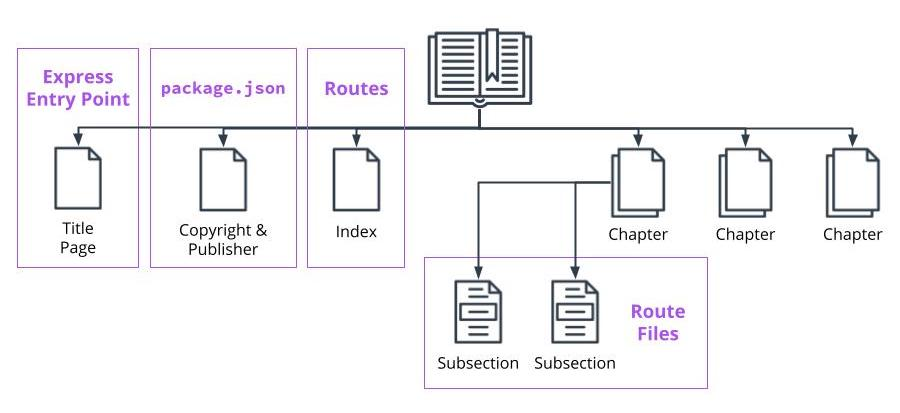
**get** books, **post** books, **put** books, **delete** books, etc.

* + - 1. The URLs that point to a specific location in the server.

Ex: drinks-api/coffee/34

* + 1. Endpoints
       1. Use routes to perform an action on the data.

Ex: GET drinks-api/coffee/34



1. Creating a server

const app = express();

* 1. The application method that creates the webserver at the specified host and port is called **.listen()** 
     + 1. app.listen(port, host, callback)
     1. Leaving out port and host arguments will force .listen() to use default values for those. ( localhost:3000 )
  2. **.get()** and **.post()**
     1. get - retrieves data from a route
     2. post - sends data to a route
        1. app.get(path, callback)
        2. app.post(path, callback)
        3. Callback function includes **req** and **res** parameters.
           1. req - represents the HTTP request object
           2. res - represents the HTTP response object
  3. The **Request** Object
     1. Based on Node.js HTTP request object.
     2. Has many properties which allow you to get things like:
        1. Cookies
        2. IP of the request
        3. Full URL or parts of it
        4. Query parameters from the request



* 1. The **Response** Object
     1. It is returned to the server after the request has been received.
     2. Available methods
        1. res.cookie() - sets cookies
        2. res.clearCookies() - clear a cookie
        3. res.redirect() - redirects to different URL
        4. res.sendStatus(code) - sets and send the response status
        5. res.sendFile(path) - sends a file to the browser
        6. res.send(body) - sends a response *(which could be an object, an array or a string of HTML)* from the server to the browser

**Code Example:**

**/ProjectsUdacityFullstackJS/ExpressExample**

(run with command: “*npm run start*” )

1. Setup express
   1. Install Node modules. ***Note:*** if you complete the project in the provided workspace, you can skip this step because it has already been completed
      1. **npm** install
   2. Install express
      1. **npm** i express
   3. Install type definitions for express
      1. **npm** i --save-dev @types/express
   4. Install nodemon
      1. **npm** i --save-dev nodemon
   5. Add a start script for **nodemon** in the **package.json** file
      1. “start” : ”nodemon src/index.ts”
   6. Import express into index.ts
      1. **import** express **from** 'express';
   7. Create your application object with express()
      1. **const** app = **express**();
   8. Set a port
      1. **const** port = 3000;
   9. Add an API endpoint to get a route, then send a response to the browser
      1. app.**get**('/api’, (req, res)=>{

res.**send**('Hello, world!’);

});

* 1. Set your application to listen on your port and output a message to the console with app.listen
     1. app.**listen**(port, ()=>{

**console**.**log**('server started at localhost:${port}’

);

});

* 1. Run **npm run start** to see the application start in development mode.
  2. Run **npm run build** and node **build/.** to build and run your file project.

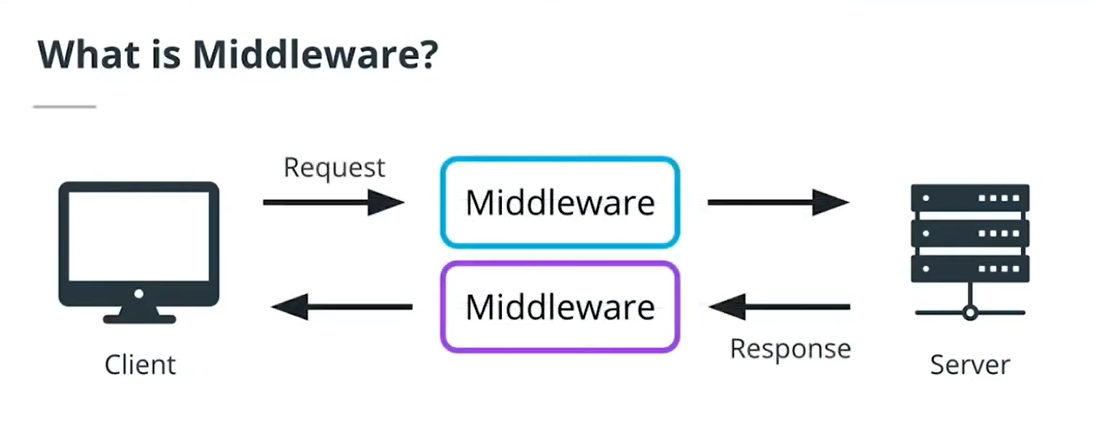
1. Nodemon
   1. Allows the server to restart anytime a file is saved.

**Code Example:**

**/ProjectsUdacityFullstackJS/ExpressMiddleware**

(run with command: “*npm run start*” )

1. Middleware
   1. Middleware is a function that is applied between the request and response.
      1. <https://blog.logrocket.com/express-middleware-a-complete-guide/>
      2. <https://expressjs.com/en/guide/using-middleware.html>
   2. Meaning you get the request, do something with it, and then send the response. Common uses of middleware include:
      1. Checking the authentication status of a user before sending a response
      2. Logging the request before sending the response.
         1. See the **/ExpressMiddleware** code example.
      3. Compressing server responses to improve performance
   3. Types of middleware
      1. Express’s built-in middleware
         1. express.static - for serving static files
         2. .json - for parsing incoming JSON
         3. .urlencoded - for parsing incoming urlencoded data
      2. Third party middleware
         1. Installed through NPM
         2. morgan - popular HTTP request logger
      3. Custom middleware
         1. Middleware that you create specifically for your own project.



* 1. How to add middleware to your application ?
     1. Application/Route Level
        1. Apply to entire application using app.use();
           1. Applied between every request and response
           2. Used for logging, compression, and CORS middleware

//can take in a route, and middleware as arguments.

app.**use**(middleware);

* + 1. Endpoint Level
       1. Applies middleware to a specific endpoint
       2. Often used for authentication. For example, allowing only authorized users onto the site. After the request, the middleware checks the authorization of the user.

students.**get**(‘/’,middleware, (req,res) =>

{

//do stuff

});

* 1. Applying multiple middleware
     1. It’s possible to send the response of one middleware to another middleware and then another one before eventually sending the request to the server.
     2. Using an Array

**const** middleware = [cors, logger];

app.**use**(middleware);

students.**get**(‘/’,middleware, (req,res) =>

{

//do stuff

});

* + 1. Listing Middleware

app.**use**(**cors**(), logger);//app level

students.**get**(‘/’,**cors**(), logger, (req, res)=>

{

//do stuff

});//endpoint level

* + 1. Writing Middleware
       1. Middleware is really just a function that is applied between the request and response. As such, if writing middleware, you create a function.
       2. A middleware function takes at least 3 arguments (req, res, next); a 4th is also available of err (err, req, res, next) for use in writing error-handling middleware. Then you write the necessary code to complete your functionality followed by calling the next(); method.

#### **next();**

* + - 1. The next method is a method from the express router. next() calls the next middleware in a chain of middlewares. Without adding next to your middleware function, your application will get stuck on the middleware.

**const middleware** = (req, res, next) => {

//do stuff

**next**();

};

1. Working with Routes

**Code Example:**

**/ProjectsUdacityFullstackJS/ExpressRoutes**

(run with command: “*npm run start*” )

* 1. Router Object
     1. When building an express application, it’s best practice to keep the server and application endpoints and functionality separate. With the router object, you’re able to create a directory of routes and separate the functionality of each route onto its own file.
        1. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#routes_primer>
        2. <https://expressjs.com/en/guide/routing.html>
  2. Router();
     1. The router method is applied to the top-level express object. With this method, you are able to create a route object that you can apply your endpoints to rather than the application object.

**import express from 'express';**

**const routes = express.Router();**

**routes.get('/', (req, res) => {**

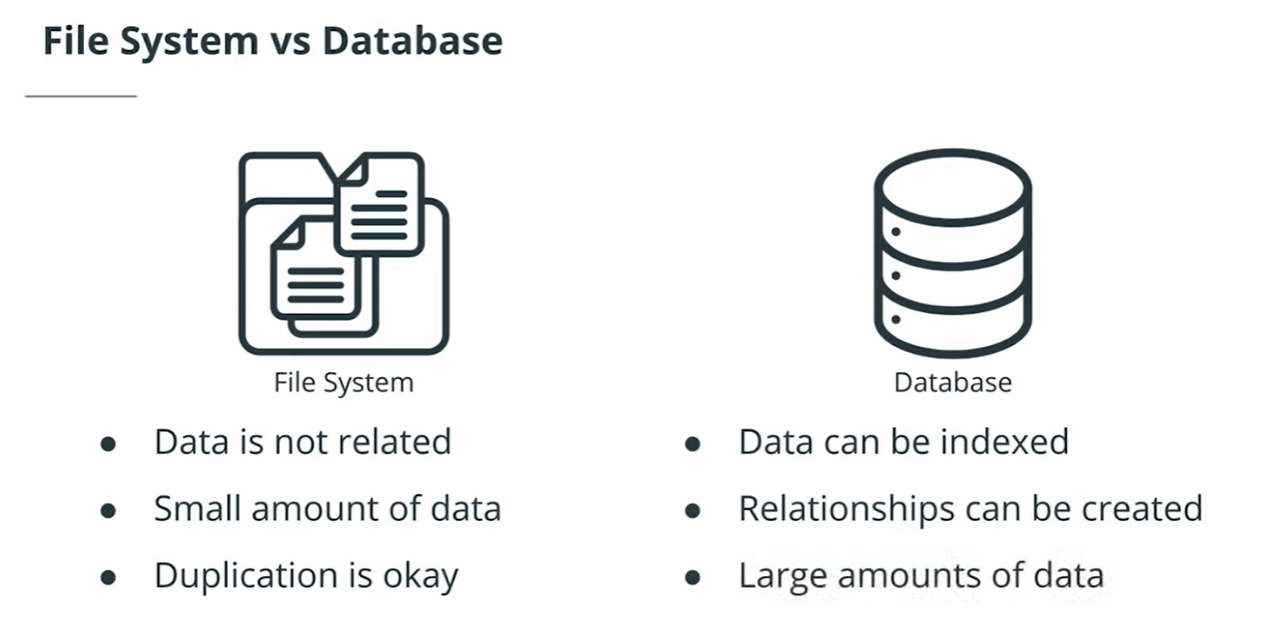
//do something

**});**

**export default routes;**

* 1. Using the Router
     1. To use the router you have created, you must first export the router. Then on your main application entry point, you can import your routes module. Then use **app.use();** to use your routes modules as middleware.

1. Introduction to Postman
   1. Helpful tool for testing endpoints!
   2. Alternatives
      1. <https://www.pcstacks.com/postman-alternatives/>
2. Reading and Writing with File System
   1. Technically, databases are just files. If you've ever tried to look at an SQL file, you may have seen this in action. They can be single files containing gigabytes of text data. So why use a database when it's ultimately a file as well?
   2. With Databases the content is structured, can be relational, and indexed. With File System, you can only control where you write to the file, and where you read from the file, so File System is only good for simple data storage.
      1. <https://www.digitalocean.com/community/tutorials/how-to-work-with-files-using-the-fs-module-in-node-js>



1. File System Flays

These are used for identifying read/write operations available when opening a file.

* **r** - open the file for the reading of a file
* **r+** - open for reading and writing of a file. Writing to file will overwrite

content at the start of the file but leave remaining content.

* **w+** - open for reading and writing of a file, will create a file if it does not

yet exist

* **a** - open for reading and writing of a file and will append new content to

the end of the file, not overwriting current content

* **a+** - open for reading and writing of a file, will create a file if it does not yet

exist, and will append new content to the end of the file, not overwriting

current content.

1. Writing to a File
   1. .open() - Used to open a file. Takes a filename and flag as arguments.
      1. Allows you to perform multiple functions without opening the file multiple times. (open -> add -> read -> write -> close)

**const** **writeData** = **async** () => {

**const** myFile = **await** fsPromises.**open**('myfile.txt', 'a+');

}

* 1. .write() - Used to write to a file that is already open. Use this when you want to

append data to an existing file.

**const** **writeData** = **async** () => {

**const** myFile = **await** fsPromises.**open**('myfile.txt', 'a+');

**await** myFile.**write**('add text');

}

* 1. .writeFile() - Used to write to a file, overwriting any content that may already exist in the file. Takes a filename, data, and options as arguments.

const writeData = async () => {

const myFile = await fsPromises.writeFile(

'myfile.txt', 'add text');

}

1. Reading, Moving, Renaming and Deleting Files
   1. .read() - Used to read a file. The file must be opened first. Allows for reading only a portion of a file, but requires the creation of a buffer to do so. Takes a buffer and options as arguments.

const readData = async () => {

const buff = new Buffer.alloc(26);

const myFile = await fsPromises.open('myfile.txt', a+);

await myFile.read(buff, 0, 26);

console.log(myFile);

}

* 1. .readFile() - Used to read the entire contents of a file. Takes a path and options as arguments. Is the preferred method for reading files when the entire content needs to be read.

const readData = async () => {

const myFile = await fsPromises.readFile(

'myfile.txt', 'utf-8');

console.log(myFile);

}

* 1. .rename() - Used to rename or move a file. Takes the old file path and new file path as arguments.

const moveData = async () => {

await fsPromises.rename('old-name.txt', 'new-name.txt');

}

* 1. .mkdir() - Used to make new directories. Takes a directory path as an argument.

const makeDir = async () => {

await fsPromises.mkdir('src');

}

* 1. .unlink() - Used to remove a file. Takes a file path as an argument.

const removeFile = async () => {

await fsPromises.unlink('myFile.txt');

}

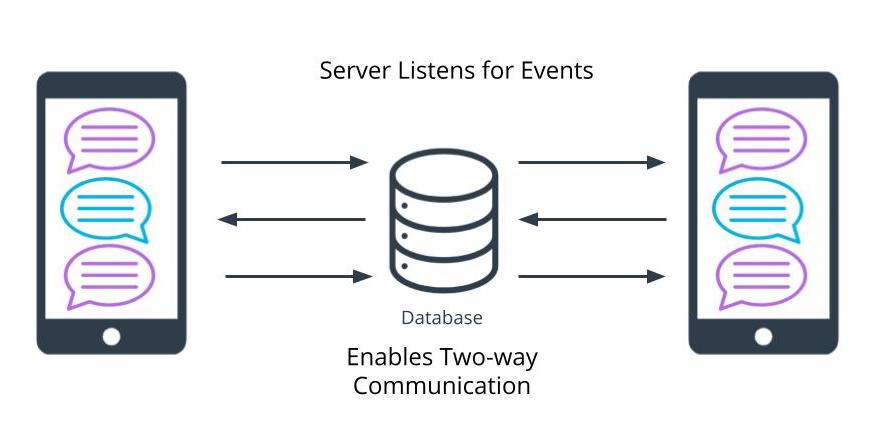
* 1. .rmdir() - Used to remove an empty directory. Takes a directory path as an argument.

const removeFile = async () => {

await fsPromises.rmdir('src');

}

1. WebSocket APIs
   1. Everything we've done in this lesson so far has been the foundation of what's known as a **REST API (Representational State Transfer)**. It's an architectural standard for APIs and currently the most popular type of API in use. RESTful APIs are considered to be stateless meaning the user/browser is independent of the server and they don't care what the other is doing.
   2. But what if you need the interactions between the server and the user to be stateful meaning the server is aware of what the user is doing. Think about when you are waiting for someone to send a text message and you see the dots pop up letting you know they are working on a response--that's stateful. So what do we do when we need a real-time application? Instead of a REST API, you create a **WebSocket API.**
   3. Websocket APIs are stateful and allow for real-time communication between the user and the server allowing for one user to know what another user is doing. Websockets do not use the HTTP protocol and have their own WebSocket protocol.
   4. The most popular library for working with WebSockets in Node.js is socket io
      1. <https://socket.io/>



1. GraphQL
   1. GraphQL isn't specifically a type of API, although it is generally referred to as one. GraphQL is a query language for working with APIs. It's becoming increasingly popular and does work well with Express. Once you're familiar with REST APIs it's very easy to learn GraphQL.
      1. <https://graphql.org/>

# Introduction to Building APIs with Postgres and Express

# Databases and SQL

# Create an API with a PostgreSQL connection

# Create an API with Express

# Authentication and Authorization in a Node API

# SQL for advanced API functionality

# Foundations of Angular

# Angular Overview

# Components

# Libraries and Services

# Data

# Foundations of Deployment Process

# Setting up a Production Environment

# Interact with Cloud Services via a CLI

# Write Scripts for Web Applications

# Configure and Document a Pipeline

# 

**Self-Hosting Questions:**

* Host your own server. The free approach. Reasons to do it, and reasons not to do it.
  + <https://www.youtube.com/watch?v=QdHvS0D1zAI>
* How do I host my own server after creating a local express.js application?

Answer:

1. **Choose a Hosting Provider**: Select a hosting provider that supports Node.js applications. Popular options include **Heroku, AWS, DigitalOcean,** and Google Cloud Platform.

2. **Set Up an Account**: Sign up for an account with your chosen hosting provider.

3. **Install the Necessary Tools**: Install the command-line interface (CLI) tools provided by your hosting provider. For example, if you're using Heroku, you'll need to install the Heroku CLI.

4. **Configure Your Hosting Provider**: Follow the instructions provided by your hosting provider to create a new application or server instance. This may involve configuring settings such as the Node.js version, environment variables, and any other necessary configurations.

5. **Deploy Your Application**: Use the CLI tools provided by your hosting provider to deploy your application. This typically involves pushing your code to a remote Git repository associated with your hosting account.

# 

Other Web Technology to be aware of:

* Tailwind
* Next.js
* Prisma