Udacity - Full Stack JavaScript Nanodegree Notes

January 05, 2024 - N/A

Courtney Downs

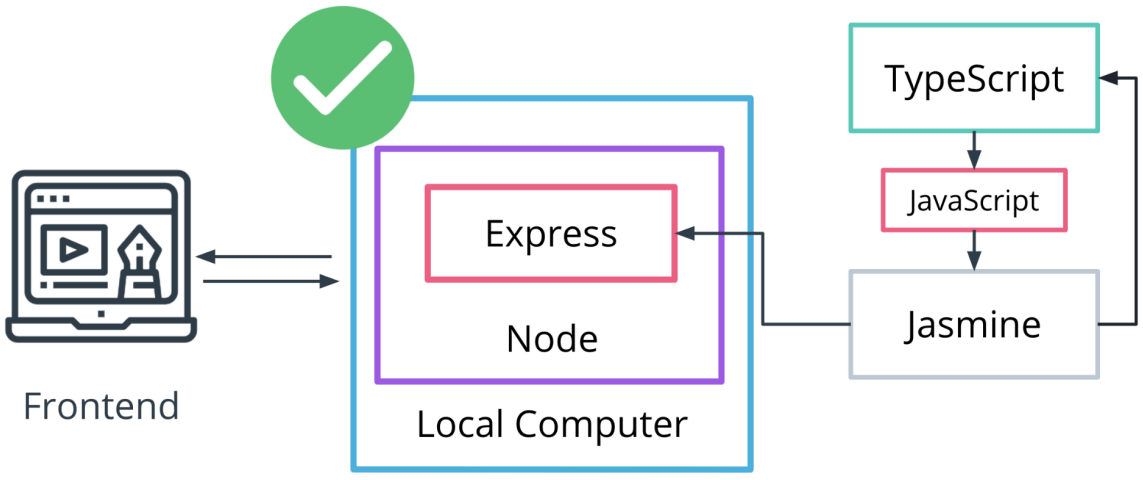
Foundations of Backend Development

1. Introduction to Backend Development with Node.js

    a. The backend has three parts
        i. **The server**: the computing resource that listens to requests from the frontend.
        ii. **The application**: code that runs on the server to process requests and return responses
        iii. **The database**: the part of the backend that is responsible for storing and organizing the data.

    b. 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.

    c. 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.



2. Jasmine
    a. 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.
        i. https://jasmine.github.io/

3. Containers
    a. 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.
    b. Docker containers
        i. https://www.docker.com/resources/what-container/

4. How to Structure a Development Team
    a. 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.
        i. https://stormotion.io/blog/6-tips-on-how-to-structure-a-development-team/

    b. Generalize Structure
        i. 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.
        ii. 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.
        iii.
    c. Specialist Structure

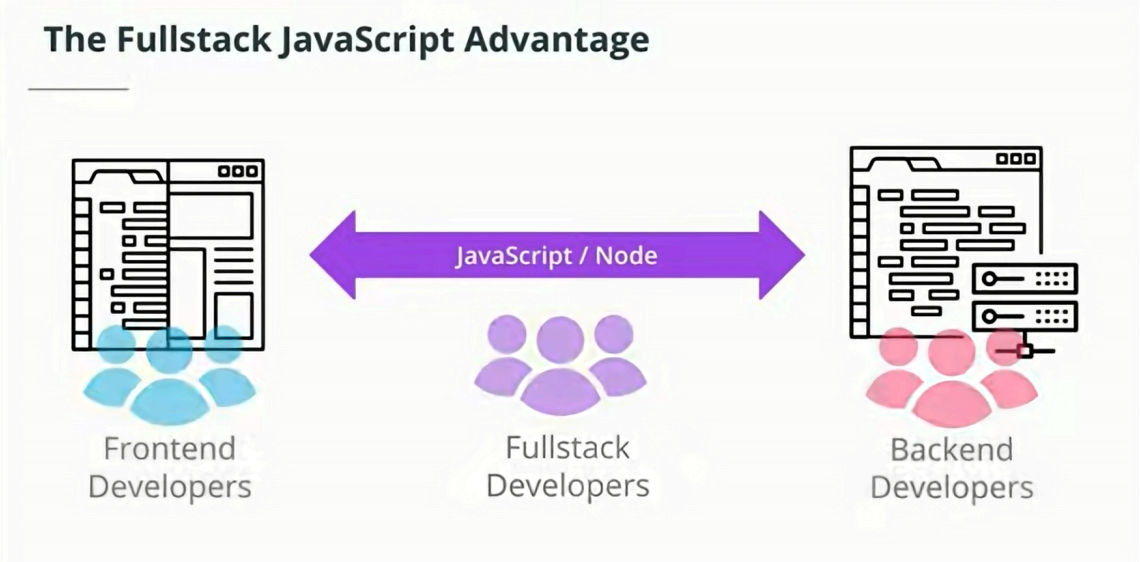
i. This arrangement approach means that each team member is an expert in a certain programming language, framework, or technology, and thus, fully responsible for their part of development. You can create teams with their own hierarchy and structure to complete one part of the project. It all depends on the scope of work.

d. Hybrid
   i. 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

e. Node.js advantages
   i. Node.js allows for JavaScript to be used on the frontend and backend.
   ii. Node.js allows for easy application scaling and maintenance.
   iii. Node.js is easy to learn.

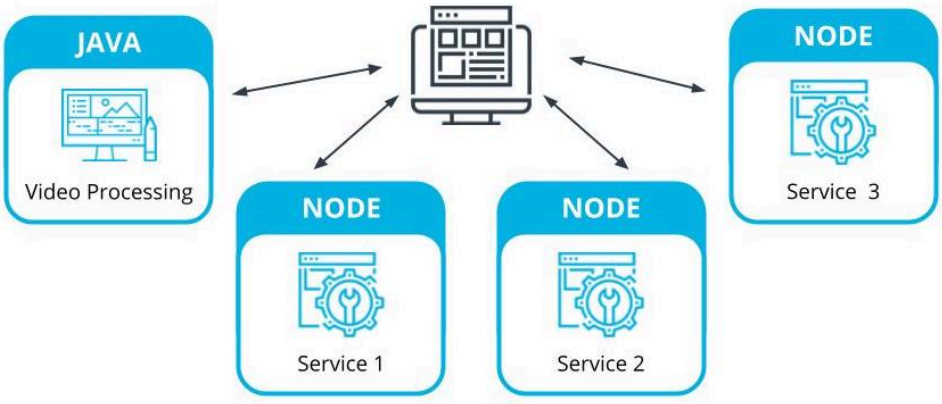
f. Documentation
   i. https://nodejs.org/api/documentation.html



g. Node.js disadvantages
   i. 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:
    a. 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.



Use Node.js in the Appropriate Modules

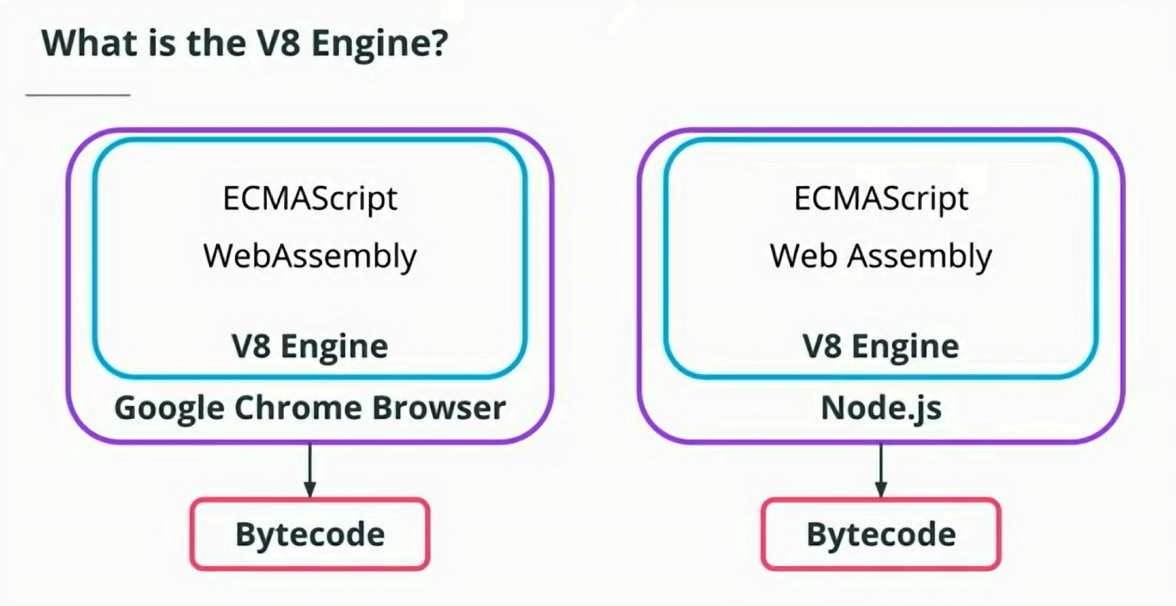
ii. **Microservices** are a piece of a larger application. In Microservices applications, the app is broken down into encapsulated parts that can be maintained individually.

5. How to run JavaScript code
    a. Run JS code in REPL(Read, evaluate, print, loop) terminal
        i. Open powershell
        ii. Type node
        iii. Ctrl+d (to exit)
    b. Run JS code file
        i. node /path/to/file.js

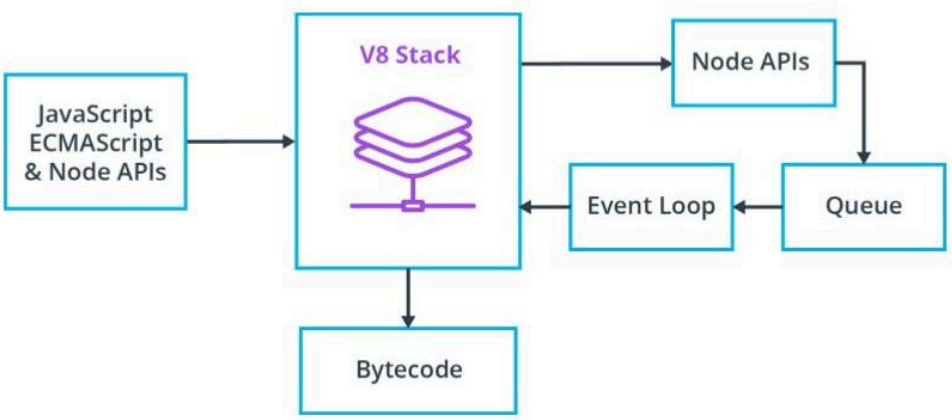
6. Node.js globals
    a. Node.js and the V8 Engine
        i. The google chrome browser is built on top of an open source V8 engine. The V8 engine processes the following:

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)

ii. The V8 engine translates the web assembly into byte code.



b. 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.



How JavaScript is Processed

c. 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.

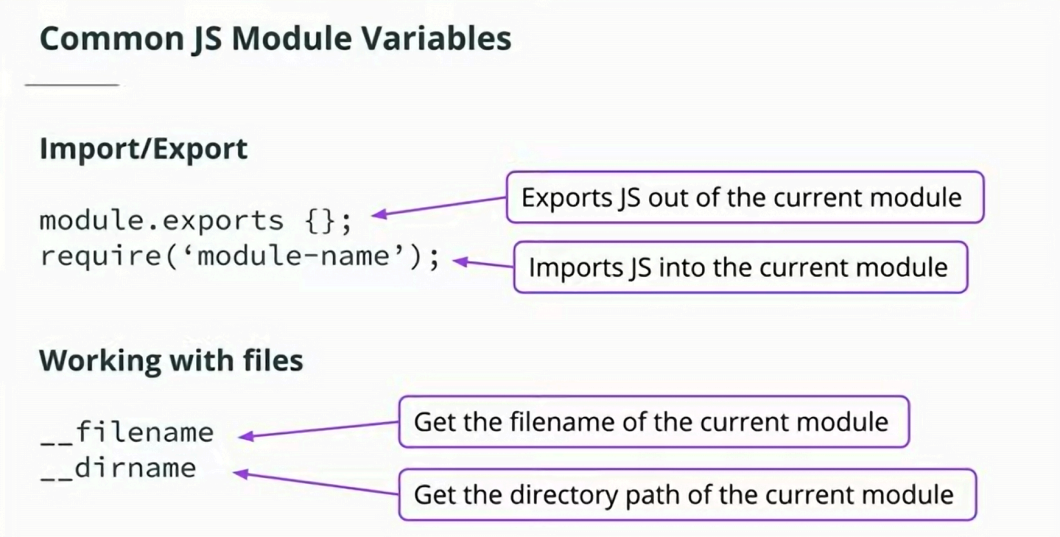
7. Importing and Exporting Modules
    a. Modules are how JavaScript allows you to break your code into focused functional chunks. You can export chunks as modules and import and use them as part of a larger program. Some modules are in node.js by not the browser apparently.
        i. Browser APIs
            1. Timers
            2. Console
            3. Fetch
            4. DOM
            5. Canvas
            6. Etc…

        ii. Node APIs
            1. Timers
            2. Console
            3. Fetch - 3rd party
            4. Process
            5. File System

    b. Timers module

i. Browser APIs
    1. setTimeout()
    2. setInterval()

ii. Node.js APIs
    1. setImmediate()
        a. Allows you to run Asynchronous code within the input/output blocks, without circling back to the start of the event loop.



c. Export module

i.
```
// working file = util/logger.js

// exports as object
module.exports = {
    myFirstFunction: myFirstFunction,
    mySecondFunction: mySecondFunction
}

// using ES6 shorthand property names
module.exports = {
    myFirstFunction,
    mySecondFunction
}
```

d. Require module

i.
```
// 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');
```

e. Path module

i. **Path.join**: concatenates strings to create a path that works across operating systems.

```
1. console.log(path.join('/app', 'src', 'util',
   '..', '/index.js'));
```

ii. **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
```

iii. **Path.normalize**: normalizes a path by removing dots and double slashes

```
1. console.log(path.normalize('./app//src//util/'));
2. // prints app/src/util
```

8. The Event Loop

a. Code Example:

i. /ProjectsUdacityFullstackJS/EventLoop

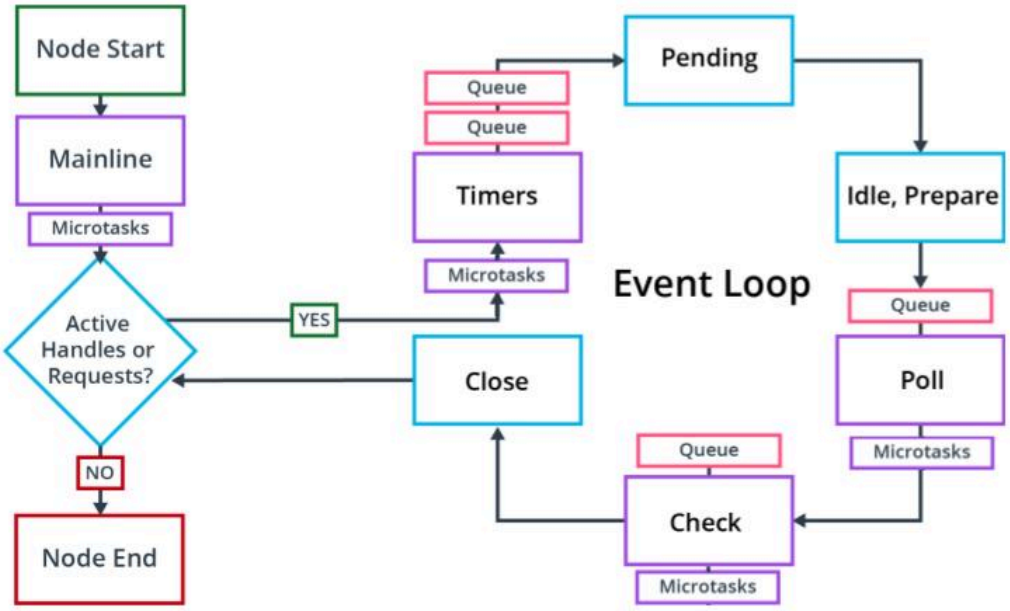
b. Nearly every Node.js feature is considered to be asynchronous (non-blocking). This means that we can request an API using **promises** and have our application continue running while that request is being waited for. But how does Node.js process that asynchronous request? Both the Browser and Node.js take advantage of something called the Event Loop.

c. 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.

    i. 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

d. Six phases of the event loop
    i. **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.
    ii. **Pending** - *internal phase*
    iii. **Idle/Prepare** - *internal phase*
    iv. **Poll** - process I/O callbacks
    v. **Check** - execute any setImmediate() timers added in the Poll phase
    vi. **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.



9. Best Practices for Server Side Development
    a. 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.

        i. 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.
        ii. 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.
        iii. 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** (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** and how it can improve an enterprise project.
    a. https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/

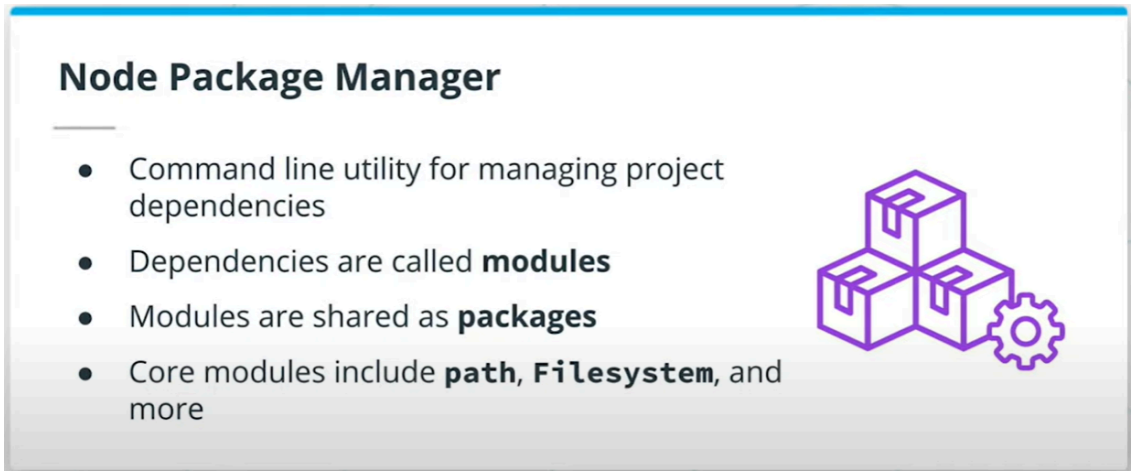
iv. Handle Errors

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.

v. Node.js best practices

1. https://github.com/goldbergyoni/nodebestpractices?tab=readme-ov-file

10. What is NPM ?
    a. **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.



    b. Initializing npm and creating a package.json file
        i. Initializing npm will create a package.json within the root of your application folder containing general information about the project.

ii. To initialize npm and go through all of the settings use:

```
1. npm init
```

iii. To automatically select all default use:

```
1. npm init -y
```

c. 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.

d. 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.

i.
```
npm i module-name
// install module to dependencies
```

ii.
```
npm i --save-dev module-name
// install to dev dependencies
```

iii.
```
npm i --save-dev module-name@1.19
// install a specific version (1.19 here) of module
```

iv. Installing dependencies adds the dependency to your package.json file in the format:

```
1.  "devDependencies": {
        "prettier": "^2.2.1"
    }
```

Pay special attention to the version listed. The format is as follows.

- First number = major version
- Second number = minor release
- Third number = patch

The version states what was installed, but it also clarifies how it can be updated should you remove the node_modules and package-lock.json files and reinstall all dependencies with `$ npm install`.

The additional included characters (or lack thereof) tell npm how to maintain your dependencies.

- `*` means that you'll accept all updates
- `^` means that you'll only accept minor releases
- `~` means that you'll only accept patch releases
- `>`, `>=`, `<=`, `<` are also valid for saying you'll accept versions greater/less/equal to the listed version
- `||` allows you to combine instructions `"prettier": "2.2.1 || >2.2.1 < 3.0.0"` which says use prettier greater than 2.2.1 and less than version 3.0.0
- You can also leave off a prefix and only accept the listed version

e. 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** depending on the project.

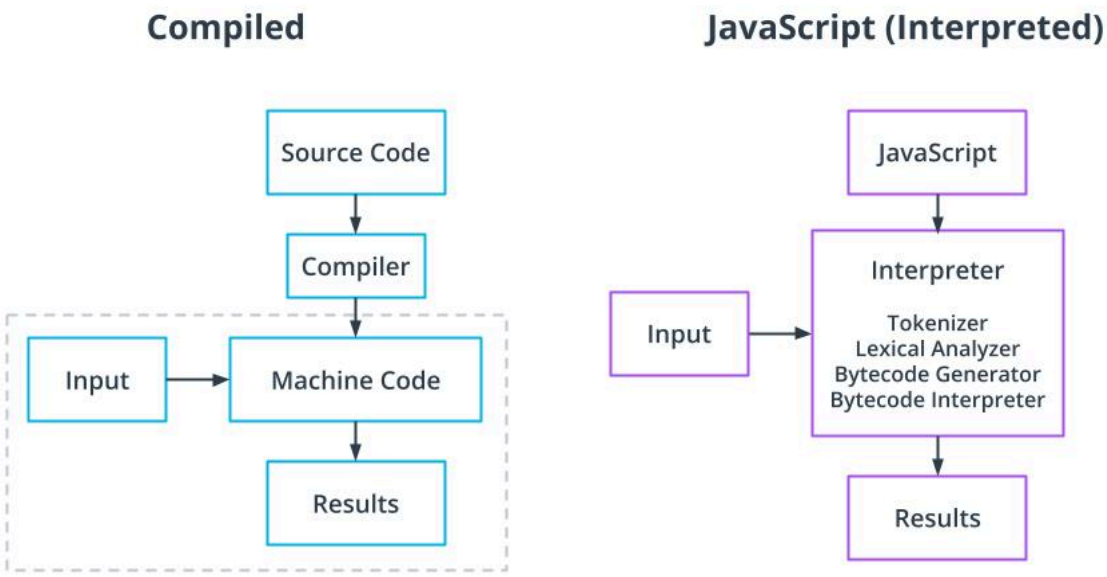
i.
```
// example config file, path structure to check, and write fixes
    "prettier": "prettier --config .prettierrc 'src/**/*.js' --write"
                        // or
    "prettier": "prettier --config .prettierrc src/**/*.js --write"
```

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

11. Is Node.js single threaded
    a. JavaScript is single threaded, is Node.js?
        i. Compiled vs. Interpreted language

1. Compiled language
    a. Language is written and compiled to machine code inside of an application
    b. Errors are detected during compiling
    c. The code won't compile until it's error-free
    d. Examples: C, Java, C++, Erlang, Go

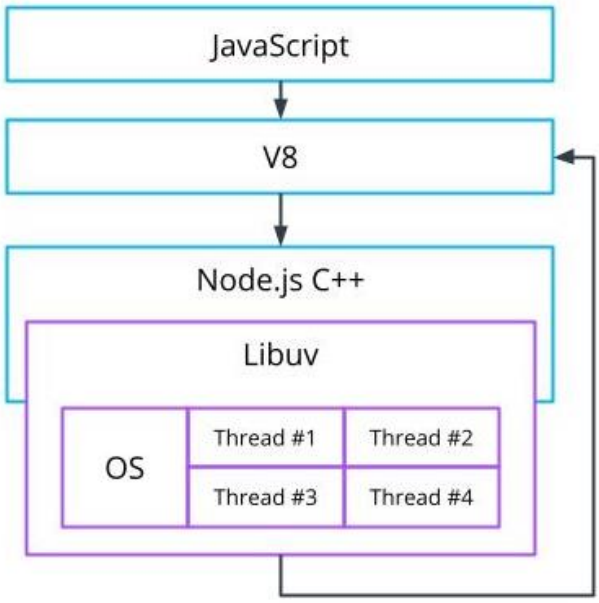
2. JavaScript is an interpreted language
    a. Errors are found when the code is run
    b. The interpreter translates and runs code one statement at a time
    c. Interpreted code runs more slowly
    d. JavaScript runs the same on different systems because it is run in the browser not the CPU.



Compiled vs, Interpreted Languages

b. Node.js is an interpreter
   i. Node.js is mostly written in C++.
   ii. Many node modules also include some C++ code
   iii. Other modules include python or C

c. C Libuv library gives Node.js access to a thread pool
   i. Main thread runs async JavaScript
   ii. Libuv takes advantage of the operating system's asynchronous interfaces before engaging the thread pool
   iii. The thread pool is engaged for events that require more processing power including compression and encryption tasks
   iv. The default thread pool includes four threads



Libuv and Node.js

Developing with TypeScript

12. Code Example:

- **a. /ProjectsUdacityFullstackJS/TypeScriptInit**
- **b. /ProjectsUdacityFullstackJS/TemplateProject** ***← for creating new .ts projects***

13. Why TypeScript is Important
    a. 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.
       i. `sum(2,'2') => '22'`

    b. 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)*

    c. How to run .ts?
       i. **Option 1:**

1. npm install -g typescript
2. tsc .\<filename>.ts
3. node .\<filename>.js

ii. **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"
}
```

5. npm run build
6. npx tsc - -init *(installs the tsconfig.json file)*

d. Update the default Typescript configuration file **tsconfig.json**

i. 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"]
}
```

ii. 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.

14. Importing modules
    a.
```
// Rename the module
import 'name' from 'module';

// Use destructuring to pull in specific functions when they are exported individually
import {function, function} from 'module';
```
15. Exporting modules
    a.
```
// 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};
```

16. Implicit Typing
    a. 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.

        i.
```
const myNum = 3;
// TypeScript implicitly types myNum as a number based on the variable
```

    b. 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.

17. Explicit Typing
    a. The developer does explicit typing when he applies a type to the object.
        i.
```
let myVar: number = 3;
```

18. Basic Types

    a. string
    b. number
    c. boolean
    d. unknown
    e. undefined - used when a variable has yet to be defined

```
const myFunc = (student: string | undefined) => {
  if ( student === undefined ){
    // do something
  }
};
```

f. null

g. Union Types
   i. Used when more than one type can be used

```
let studentPhone: (number | string);
studentPhone = '(555) 555 - 5555';
studentPhone = 5555555555;
```

h. Return types
   i. void
   ii. 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
};
```

   iii. 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
};
```

i. Type Assertions
   i. 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;
```

```
}
```

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

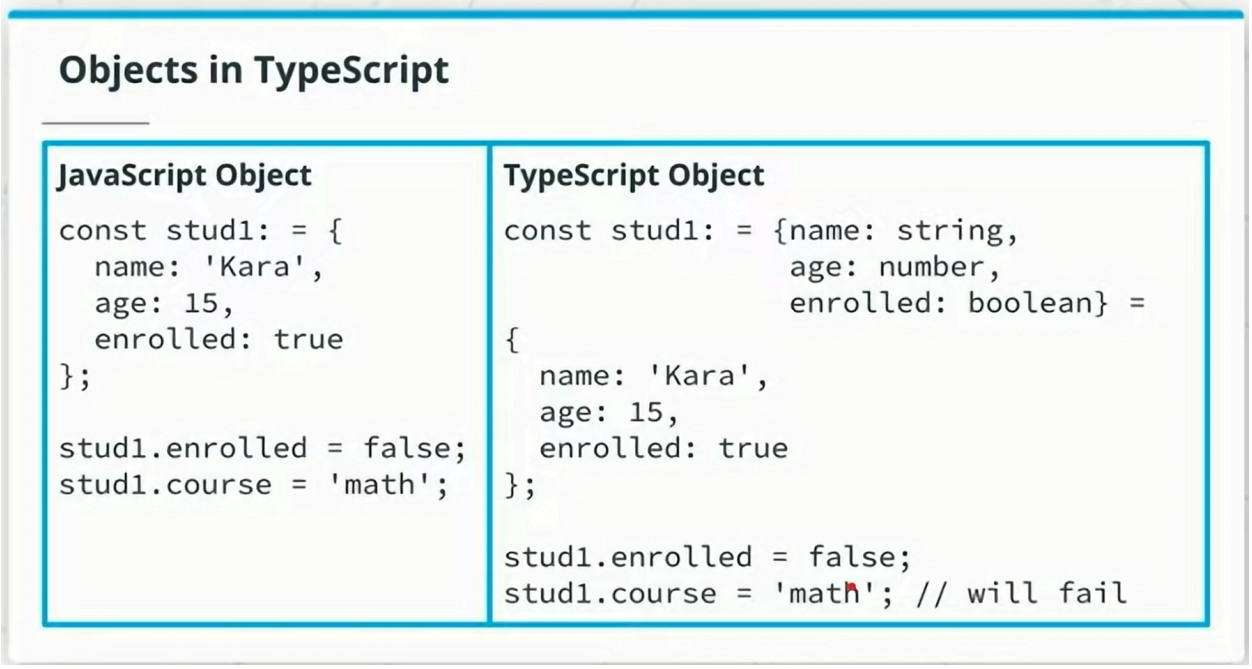
```
// only accepts strings
let arr: string[] = [‘a’, ’b’, ‘c’];

// accepts strings or numbers
let arr2: (string | number)[] = [1, ‘a’, ‘b’, 2];
```

   ii. 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.

```
a. let arr: [string, number, string];
   // ['cat', 7, 'dog']
```

k. Working with objects in TypeScript



   i. 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

ii. 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
    a. Pascal case
        i. ThisIsPascalCase
    b. Camel case
        i. thisIsCamelCase

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean
};

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

l. Duck Typing
i. 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.

m. Optional and ReadOnly properties
i. TypeScript gives the ability to create both optional and read-only properties when working with object-like data.

ii. 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
};
```

iii. 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
};
```

n. Type Aliases, TypeScript Classes and Factory Functions

i. 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};
```

ii. 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;
  }
}
```

iii. 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);
```

19. Generics
    a. Reusable components that can be used with different types
    b. Why not just use **any** instead of messing with generics?
        i. With generics, you can ensure that the types of your inputs match the types of your outputs of your functions.

    c. Uses angle brackets syntax <T> where T is the type parameter
        i. Java Example: ArrayList<String>, ArrayList<Integer>, etc..

```
const getItem = <T>(arr: T[]): T{
  return arr[1];
}

getItem(['cat','dog']); // 'dog'
getItem([5,6]); // 6
```

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

Code Example:
/ProjectsUdacityFullstackJS/AsyncAndPromises

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

21. Promises
    a. A promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value.

22. TypeScript with Third-Party Modules
    a. 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.

    b. 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:

        i. `npm i --save-dev @types/packageName`
        ii. `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

    c. Lodash Module
        i. Very popular library for working with functional javascript. It provides great utilities for working with arrays, objects, strings, and other types of structures.
        ii. https://www.npmjs.com/package/lodash
        iii. Installation

```
1. npm i lodash
2. npm i –save-dev @types/lodash
```

d. 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.

e. How to create type definitions when one is missing:
   i. Create a folder called **'types'** in your root directory with a subfolder called **'3rdparty'**
   ii. 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
   iii. Within your definitions file, import the node module with the missing definition.
   iv. Use the declare keyword to declare which module the definition will be for, followed by curly braces to contain the definition.
   v. 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.
   vi. Open tsconfig.json and find //"typeRoots": ,. Uncomment the line and update it to include your new types directory. **"typeRoots": ["./types"]**,

   vii. 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;
  }
}
```

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

When to Use Implicit Typing?

	Typing	Why?
`const`	Implicit	Value is immutable so type can't be changed
`let`	Explicit	Value and type can be changed
Function with controlled inputs	Implicit	Output is controlled and code is simpler
Single-line arrow function	Implicit	Simpler code
Longer function	Explicit	Explicit typing is easier to read

    c. Take advantage of the latest EcmaScript features
        i. async/await
        ii. ES6 modules
        iii. Nullish coalescing
        iv. Destructuring
        v. Spread operator

24. Migrating to TypeScript
    a. TypeScript makes migrating from JavaScript easy. Even for monolithic legacy JS projects.
        i. Different project structures will have to be migrated differently.
            1. Microservices vs. Monolithic

        ii. 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.

iii. Add Typescript to *each* service if the project uses microservice architecture.

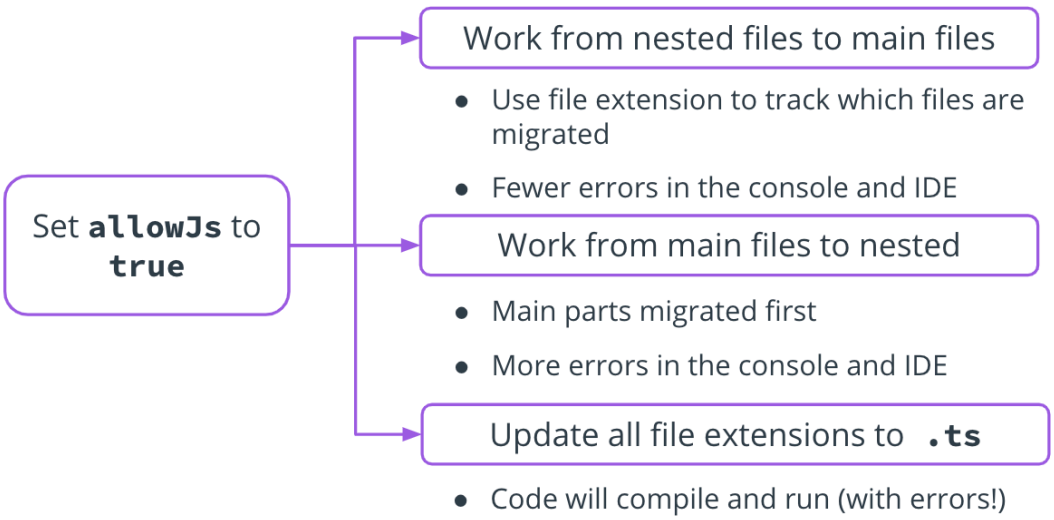
iv. For monolithic architecture, move to a src/dist to keep working files separate from compiled Javascript.
    1. Check if this affects any of the other paths within the project, as they might not be automatically updated (although most IDEs do).
    2. If it doesn't automatically update, you can use a path module.

v. To exclude folders you don't want to be migrated, utilize the configuration file.

vi. 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

25. What we've done so far to reduce the chance of errors:
    a. *Added Prettier library for standard code formatting*
    b. *Added ESLint library to improve style and structure*
    c. *Using TypeScript to reduce type errors with JS.*

26. Other JavaScript testing frameworks
    a. Jest
    b. Mocha
    c. Puppeteer
        i. What are the differences between them and why choose Jasmine ?
            1. https://raygun.com/blog/js-toolbox-part-3/#what-to-choose

27. Jasmine allows us to integrate automated unit testing to ensure the following:
    a. Tests are run when code is integrated
    b. Alerts us when refactored code affects other code
    c. Find issues early, when they are easier to fix
    d. Encourages simple, modular, easy-to-maintain code

28. Testing Approaches *(Jasmine supports both)*

    a. Behavior Driven Development (**BDD**)
        i. You write tests for your application based on user behavior.

    b. Test Driven Development (**TDD**)
        i. 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

ii. Cycle continues until the feature is complete.

c. 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.

29. How to come up with tests before writing the code?
    a. https://www.browserstack.com/guide/what-is-test-driven-development
    b. https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters

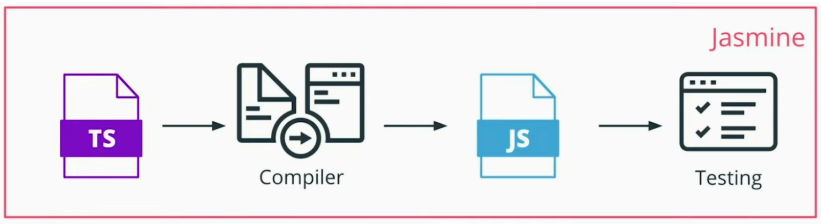
30. Unit Testing Best Practices

a. 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.

b. 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.

c. 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.

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



b. Install

```
 i.   npm i jasmine
ii.   npm i jasmine-spec-reporter
iii.  npm i --save-dev @types/jasmine
```

c. Add testing script

i. Inside of package.json, add this to the scripts option.

```
"scripts":{
      ...
      "jasmine": "jasmine"
}
```

d. Set up file structure

i. At the root level create a new folder called ***spec***

ii. Inside of spec create another folder called ***support***

iii. Inside of support create a file called ***jasmine.json***

iv. Inside of src create a new folder called ***tests***

v. Inside of tests folder, create a file called ***indexSpec.ts***
   1. *Has the tests related to index.ts*

vi. Inside of tests folder, create another folder called ***helpers***

vii. Inside of helpers, create a file called ***reporter.ts***

File Structure

```
├── node_modules
├── spec
│       └── support
│           └── jasmine.json
├── src
│      ├──  tests
│      │     ├── helpers
│      │     │      └── reporter.ts
│      │     └── indexSpec.ts
│      └── index.ts
├── package-lock.json
├── package.json
└── tsconfig.json
```

e. Best Practices For File Naming

i. 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.

f. To run Jasmine tests you need to first run the build script and then the test script.

i. `npm run build`

ii. `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:

   a. “test”: ”npm run build && npm run jasmine”

   b. Now you can just use **“npm run test”**

iii. **IMPORTANT**: The TemplateProject in the ProjectsUdacityFullStackND repo already has all of the setup listed above that is needed to run jasmine unit tests.

iv. 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

32. Writing Unit Tests
    a. Jasmine uses Suites and Specs
        i. **Spec**: an individual test
        ii. **Suite**: a collection of similar tests related to one function
        iii. Tests should cover all intended behaviors
        iv. Error handling should also be tested
        v. https://jasmine.github.io/api/3.6/matchers.html

33. Jasmine Syntax

Code Example:
/ProjectsUdacityFullstackJS/Jasmine
(run test with command: "*npm run test*")

a. Use the **describe** keyword followed by a short description of what the suite is testing and one or more specs.
b. A best practice is to start a sentence with "it" and then complete the sentence with the description of what the suite is testing.

i.
```
describe("suite description", () => {
    it("describes the spec", () => {
        const myVar = true;
        expect(myVar).toBe(true);
    });
});
```

34. Comparisons
    a. Can compare strings, numbers, objects, or arrays
    b. .**toEqual**(expected value)
        i. deep checks if the tested value is ***equal to the expected value***
    c. .**toBe**(expected reference)
        i. checks if tested object is ***the same object***

Example 1: `toEqual()` and `toBe()`

Code to Test

```
const myFunc = (num) => num * 5;
```

Jasmine Tests

```
it("expects myfunc(9) equals 45", () => {
    expect(myFunc(9)).toEqual(45);
});
```

PASS

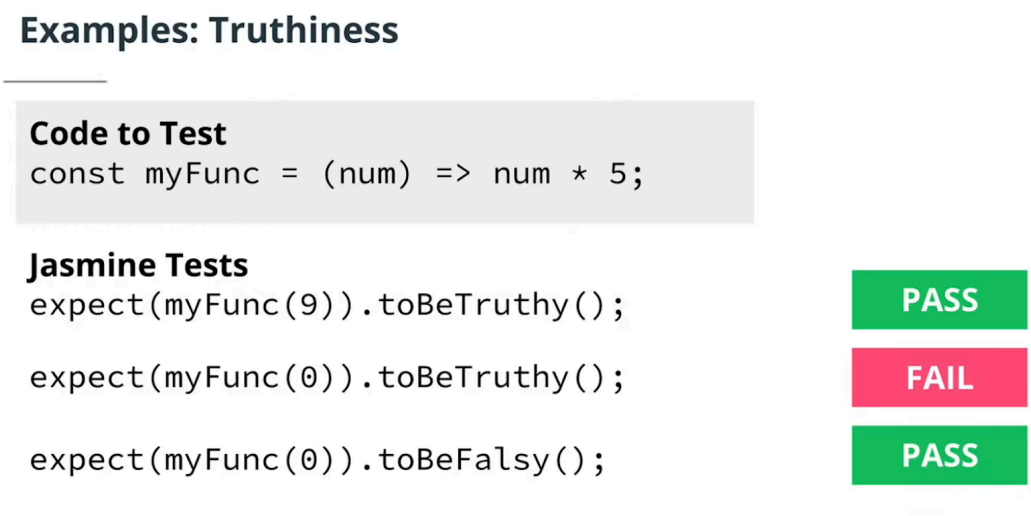
```
it("expects myfunc(9) tobe 45", () => {
    expect(myFunc(9)).toBe(45);
});
```

PASS

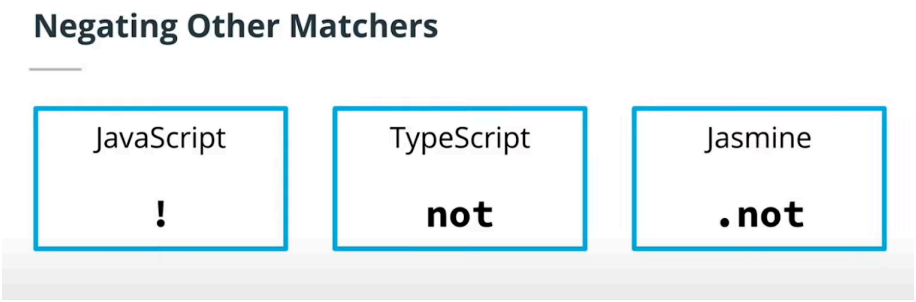
35. Truthiness

Code Example:
/ProjectsUdacityFullstackJS/JasmineSetupTeardown
(run test with command: "*npm run test*")

a. These are tests that are useful with boolean types.
b. .**toBeTruthy()** passes when
    i. The expectation has any non-zero value
    ii. The expectation evaluates to true
c. .**toBeFalsy()** passes when the value is:
    i. 0
    ii. " (an empty string)
    iii. undefined
    iv. null
    v. NaN
d. **.toEqual()**
    i. Use this if you only need a Boolean true or false
    ii. When comparing two objects, .toEqual() checks if they have the same keys (properties) and values recursively.
    iii. 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.
    iv. For other types like strings, numbers, booleans, or null/undefined, .toEqual() behaves similarly to the strict equality operator (===).



36. Numerical Matchers
    a. .**toBeCloseTo**(expected value, precision)
        i. Passes if the value is within a specified precision of the expected value
        ii. Precision is optional and represents the number of decimal points to check *(defaults to 2)*
    b. **.toBeGreaterThan**(expected value)
    c. **.toBeLessThan**(expected value)
    d. **.toBeGreaterThanOrEqual**(expected value)
    e. **.toBeLessThanOrEqual**(expected value)



37. Other Matchers
    a. .**toContain**(expected value)
    b. .**toMatch**(expected value)
    c. .**toBeDefined**()
    d. .**toBeUndefined**()
    e. .**toBeNull**()

f. .**toBeNaN**()
g. Custom mathers that you create.
    i. https://jasmine.github.io/tutorials/custom_matcher

38. Exceptions in Jasmine
    a. .**toThrow**(expected value)
    b. .**toThrowError**(expected value, expected message)
    c. Expected value and expected message parameters are optional

39. Testing Asynchronous Code

Code Example:
/ProjectsUdacityFullstackJS/JasmineAsync
(run test with command: "*npm run test*")

a. The key to testing async code is letting Jasmine know when it's ready to be tested.
    i. 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);
});
```

    ii. 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);
        });
    });
```

b. Testing promise resolution and rejection with **ES6 Promise Matchers Library**
   i. **.toBeResolved()** tests if a promise is resolved and will return true if the promise is resolved
   ii. **.toBeRejected()** tests if a promise is rejected and will return true if the promise is rejected
   iii. **.toBeRejectedWith(expected value)** tests if the expected error is returned

c. With both async/await and promises, should the promise be rejected, or throw an error, the test will fail.

40. Endpoint Testing

a. 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.

b. Why does endpoint testing matter?
   i. Confirms that the server is working
   ii. Confirms that endpoints are configured properly
   iii. More efficient than manual testing

c. The 'supertest' framework
   i. Jasmine doesn't support endpoint testing, so we need to use another framework called **Supertest** to test the status of responses from servers.

d. Setting up Supertest
   i. Install the dependency

```
npm i supertest
```

   ii. Add type definition to allow the code to compile without TypeScript errors.

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

iii. 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;
```

iv. Create and run tests

```
npm run test
```

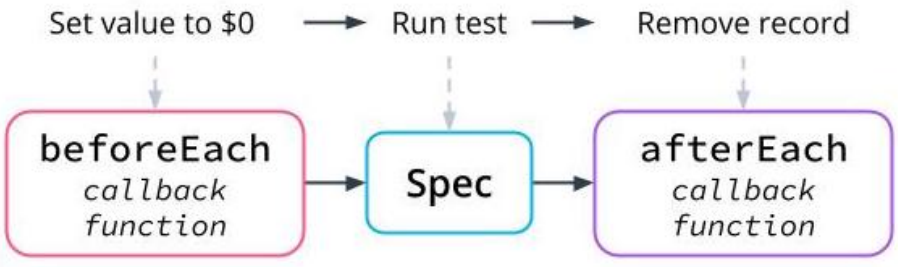
41. Performing Tasks Before and After Tests
    a. For example, there are times when running a test may require connecting to a database before you can run your test and disconnecting afterwards.
    b. Run only a specific test/suite
        i. Add **f** in front of **describe** or **it**
        ii.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

```
fdescribe("A spec", function() {
    it("is just a function, so it can contain any code",
      ()=> {
      expect(foo).toEqual(1);
    });
});
```

    c. Skip a test or suite
        i. Add **x** in front of **describe** or **it**
        ii. Helpful to avoid a time consuming test

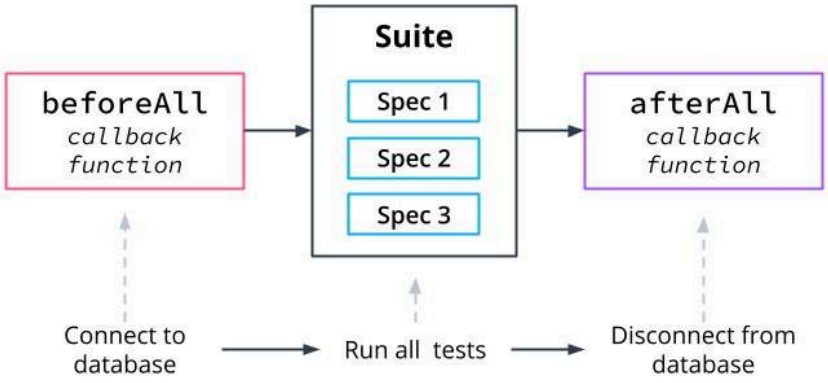
beforeEach and **afterEach**

- 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`**

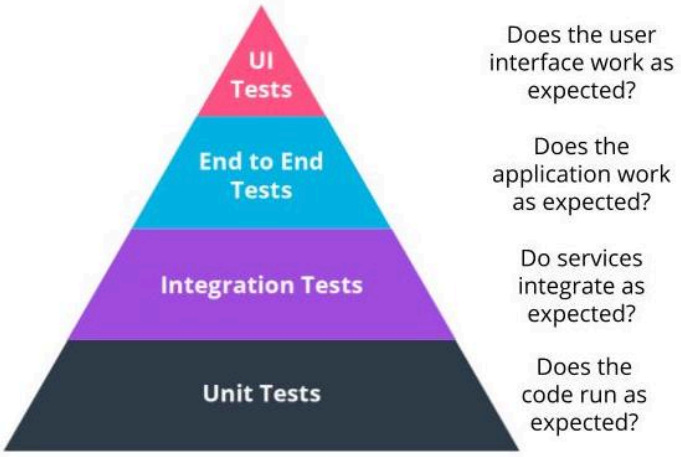
- To perform an operation ***once before*** all the specs in a suite, use `beforeAll`
- To perform an operation ***once after*** all the specs in a suite, use `afterAll`.



42. Additional points on Unit Testing

a. Microservice Testing: Unit Tests
   i. https://medium.com/@nathankpeck/microservice-testing-unit-tests-d795194fe14e

b. The Testing Pyramid



   i. **UI Tests:**

1. Automated UI test
    a. Scraping the website to ensure that there are no broken links.
2. Manual UI test
    a. Resizing the screen to test that all images are responsive

ii. 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

iii. 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.

iv. 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

43. How does a server work?
    a.

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

Other Web Technology to be aware of:

- Tailwind
- Next.js
- Prisma
-