Udacity- Full Stack JavaScript Nanodegree Notes

January 05, 2024 - N/A

Courtney Downs

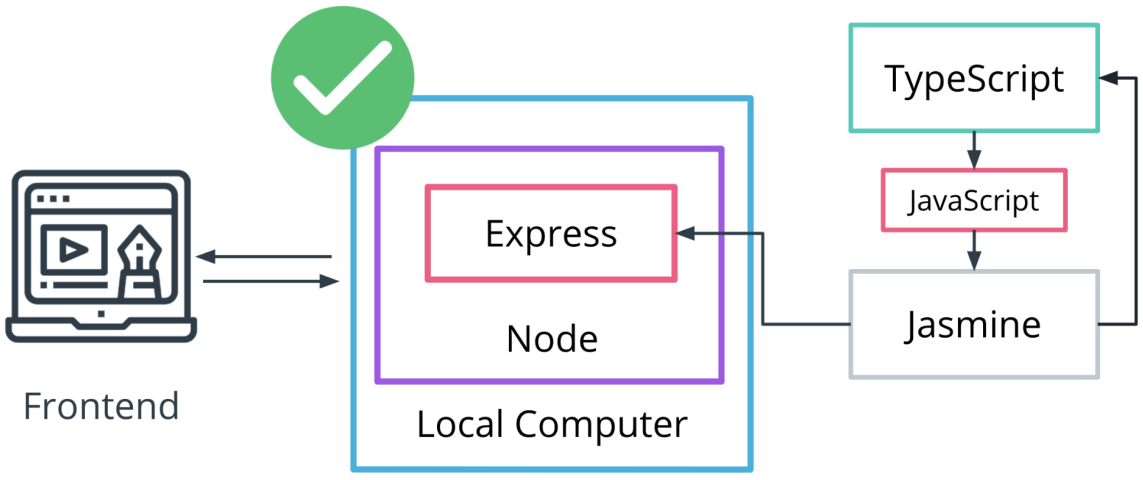
Foundations of Backend Development

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

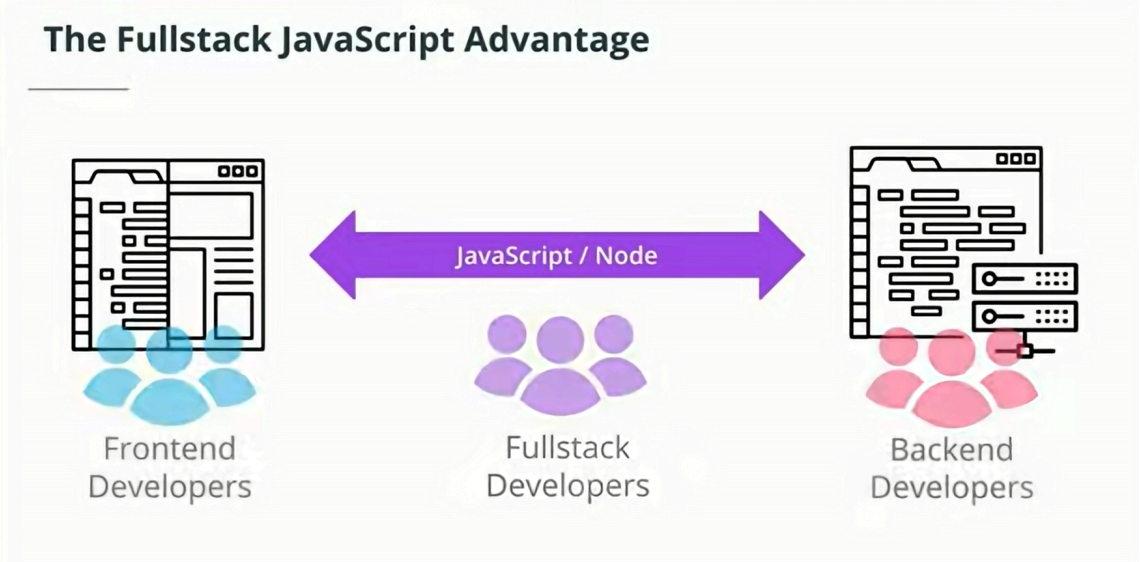
i. This arrangement approach means that each team member is an expert in a certain programming language, framework, or technology, and thus, fully responsible for their part of development. You can create teams with their own hierarchy and structure to complete one part of the project. It all depends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

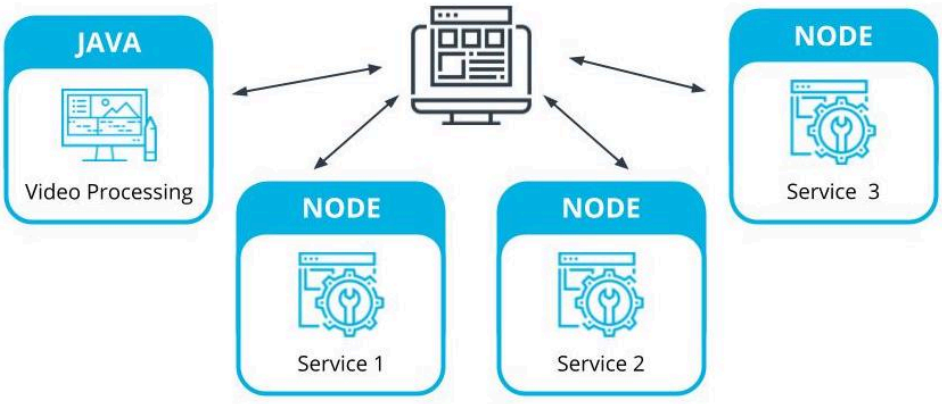
f. Documentation
   i. https://nodejs.org/api/documentation.html



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

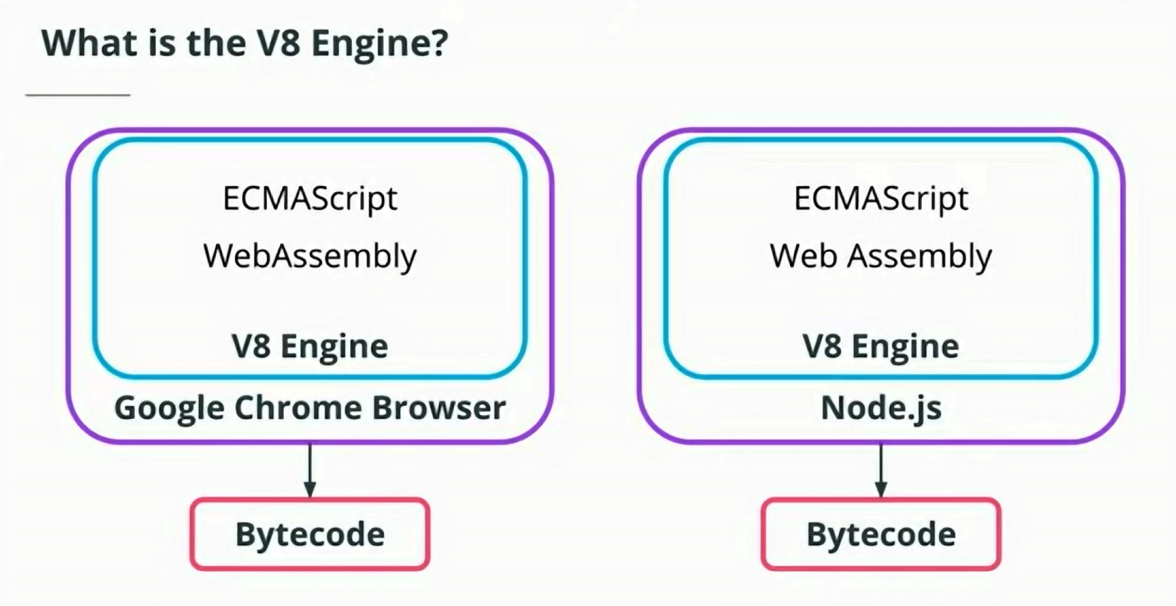
ii. **Microservices** are a piece of a larger application. In Microservices applications, the app is broken down into encapsulated parts that can be maintained individually.

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

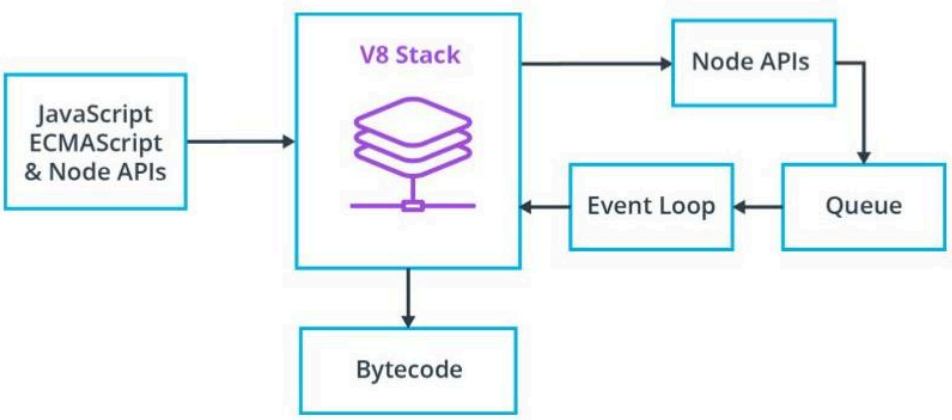
6. Node.js globals
    a. Node.js and the V8 Engine
        i. The google chrome browser is built on top of an open source V8 engine. The V8 engine processes the following:

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.

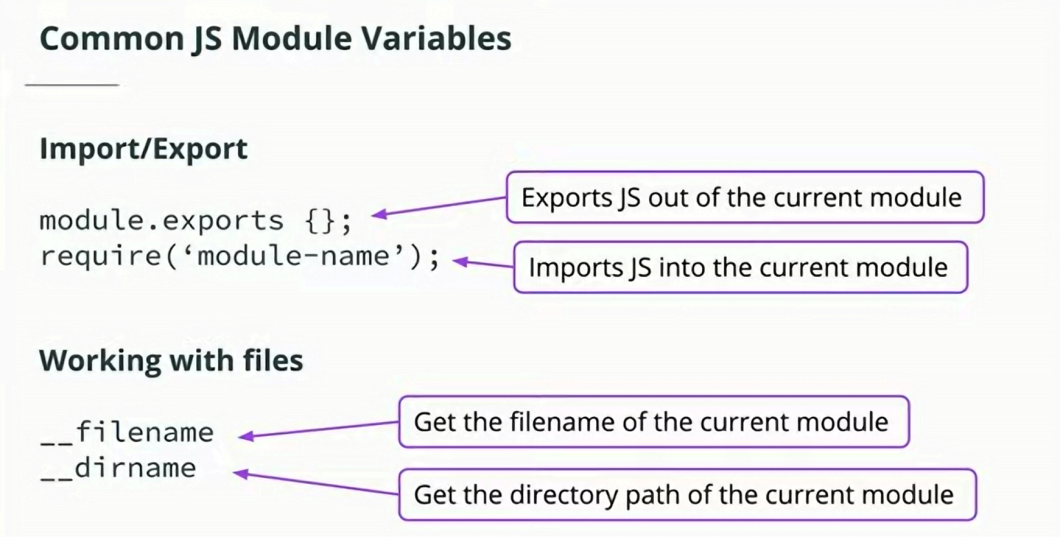
7. Importing and Exporting Modules
    a. Modules are how JavaScript allows you to break your code into focused functional chunks. You can export chunks as modules and import and use them as part of a larger program. Some modules are in node.js by not the browser apparently.
        i. Browser APIs
            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

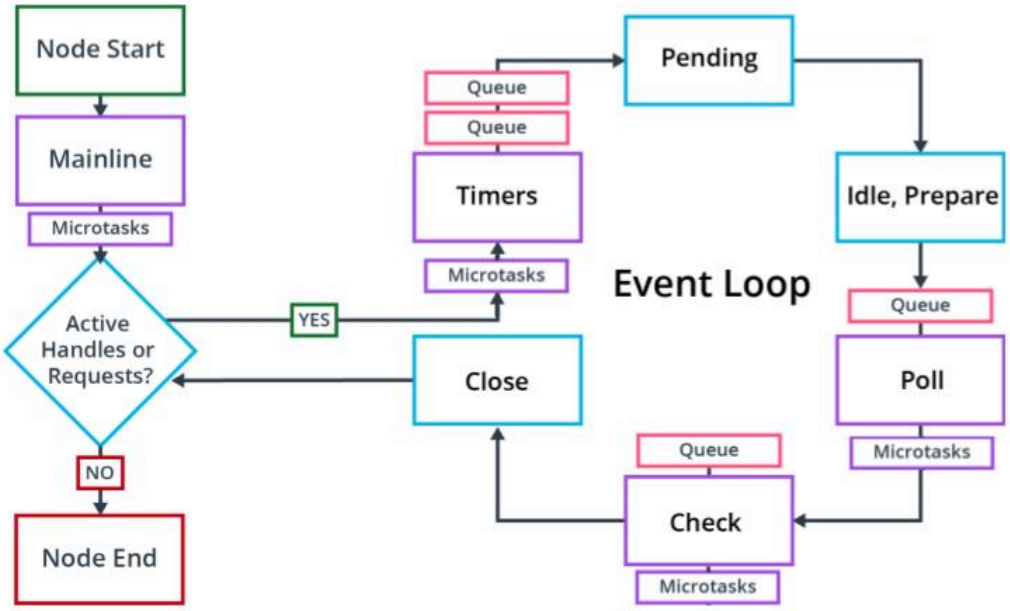
b. Nearly every Node.js feature is considered to be asynchronous (non-blocking). This means that we can request an API using **promises** and have our application continue running while that request is being waited for. But how does Node.js process that asynchronous request? Both the Browser and Node.js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

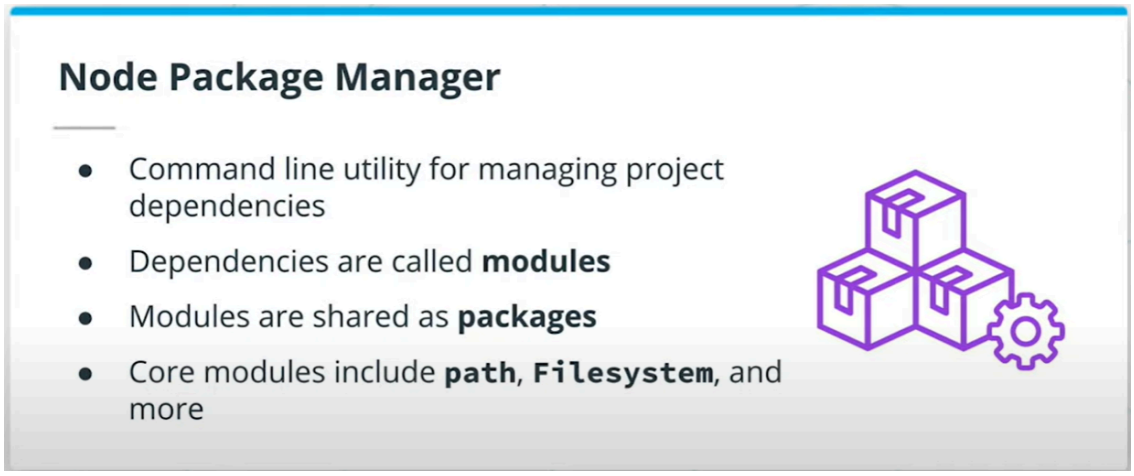
iv. Handle Errors

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.

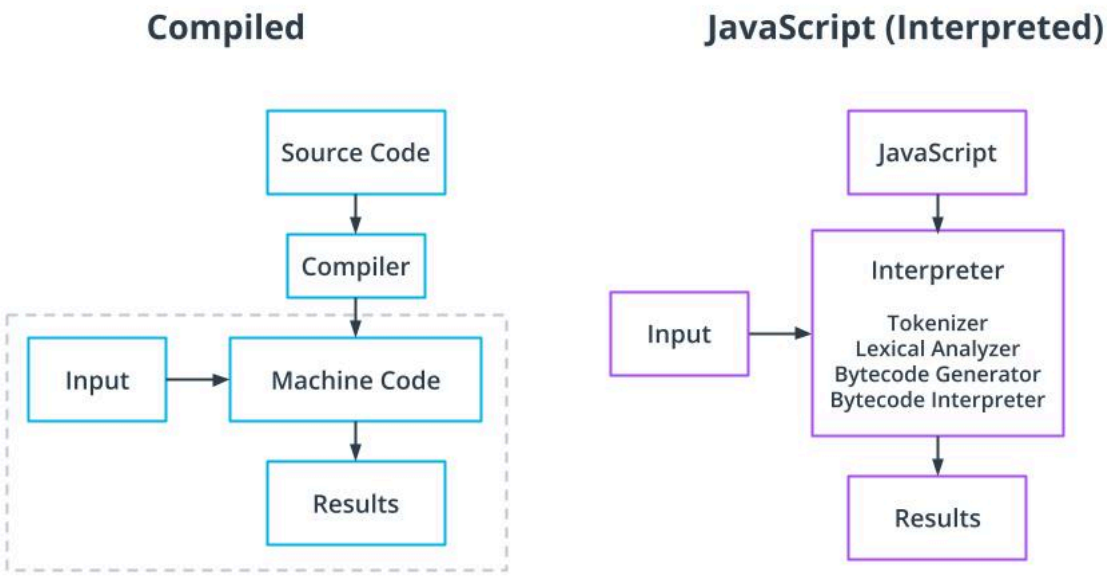
i.
```
// example config file, path structure to check, and write fixes
    "prettier": "prettier --config .prettierrc 'src/**/*.js' --write"
                    // or
    "prettier": "prettier --config .prettierrc src/**/*.js --write"
```

- 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

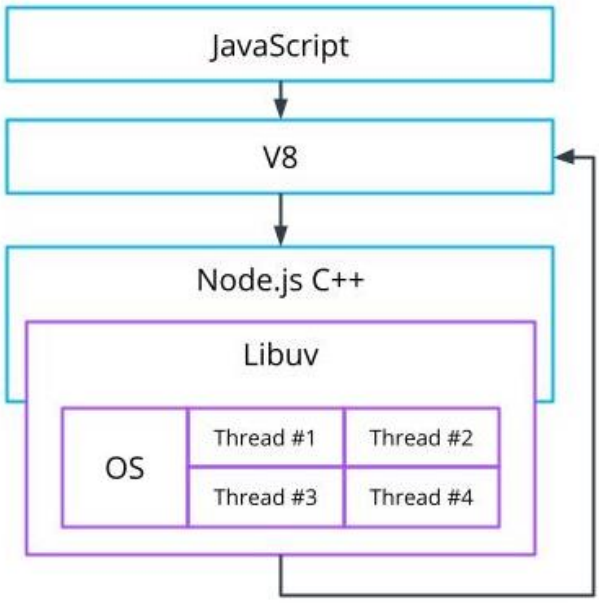
2. JavaScript is an interpreted language
    a. Errors are found when the code is run
    b. The interpreter translates and runs code one statement at a time
    c. Interpreted code runs more slowly
    d. JavaScript runs the same on different systems because it is run in the browser not the CPU.



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**

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.

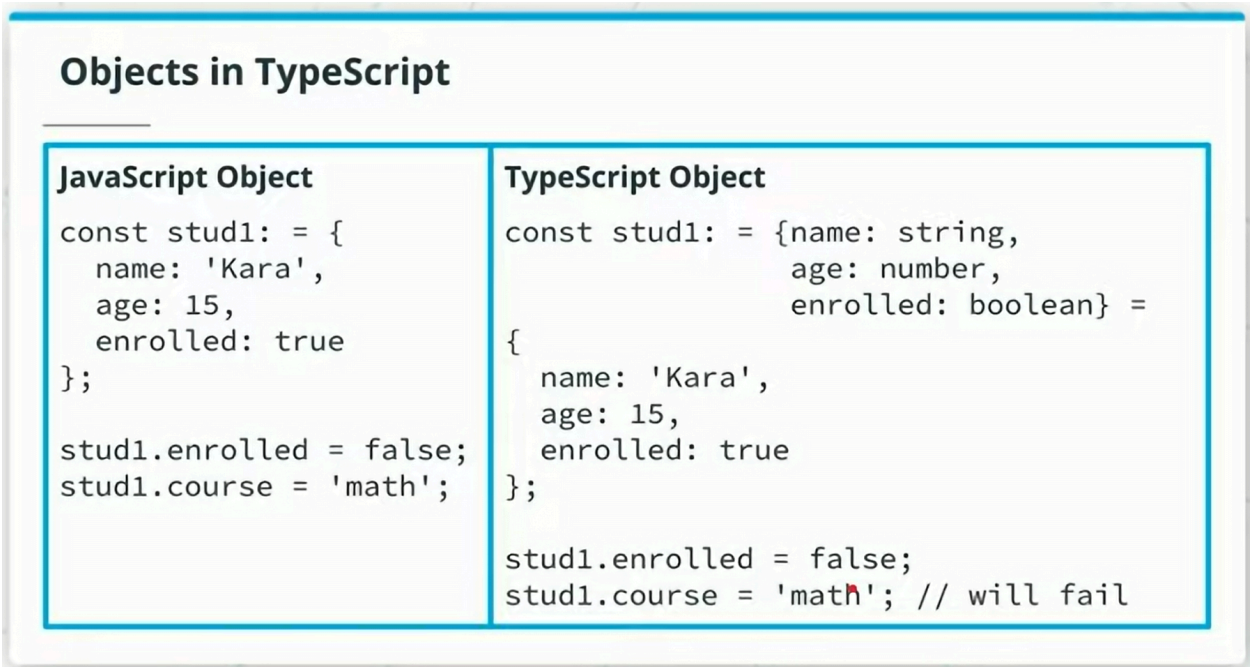
```
// only accepts strings
let arr: string[] = [‘a’, ’b’, ‘c’];

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

    c.

Unit Testing with Jasmine

Building a Server

Introduction to Building APIs with Postgres and Express

Databases and SQL

Create an API with a PostgresSQL 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