MODULE

Week 1: Unit Administration and JavaScript Recap

Nawfal Ali
Updated 29 July 2021

Teaching Team

- Nawfal Ali (Clayton)
    - Nawfal.ali@monash.edu

Note: Begin email header with "FIT2095."

Changing Tutorial and Lab Allocations

- Academics have no privileges in the Allocate+ system – It's purely an admin issue
- There is a link in Allocate+ that allows you to submit a request for a change in allocation

Assessment – Non-Exam

- Contribution
    - Exam (60%)
    - Pre-reading Quizzes (10%): Max 10 Moodle MCQ's in each quiz
    - Workshop Quizzes (10%): Max 5 multi-part Moodle short answer questions in each quiz
    - Labs (20%)

Assessments

- **Exam**
    - Closed Book, 2 hours
    - Detailed Questions on non-trivial presented code fragments (50%)
        - You will not be required to code from scratch, BUT you will be asked to analyse non-trivial fragments of presented code including correcting

and interpreting the code
        - The presented code will not be unseen; it will come from labs, slide-sets, quizzes or tasks you have been asked to complete – Other questions (50%)
        - You will also be asked other questions that test your knowledge and understanding of the unit's content
        - These may include code-related issues
    - Sample questions will be posted towards the end of the semester

- **Pre-reading Quizzes**
    - From Week 2, There will be an online Moodle MCQ quiz each week
    - Quiz questions will be on:
        - The week's required reading (slide set + …)
            - Which you should have read before attempting the quiz
            - Two attempts allowed, correct answers only revealed after the quiz closes
        - They check you have done the pre-reading
    - Your answers to the quiz questions:
        - Are answered on Moodle
    - Correct answers and your marks are available as soon as the quiz closes
        - The mark will be available on Moodle.

- **Workshop Quizzes**
    - From Week 2, there will be a workshop quiz
    - Quiz questions will be on:
        - The week's required reading (slide set + …)
            - Which you have already read
            - Which you have proved by completing and passing a Moodle MCQ quiz
        - They check for a deep understanding and synthesis of the upcoming week's material
    - Your answers will be marked ASAP
        - The mark will be available on Moodle.

- **Labs**
    - We will build/modify Web applications including:
        - Server-side applications using Node.js, Express.js, and MongoDB

        - A rich client-side, “Web” application that demonstrates important new HTML5/CSS3 features and elementary and advanced JavaScript features and design patterns using Angular
    - You will be interviewed during your lab time to make sure you understand the code you are presenting for marking. Less than a confident understanding = 0 marks !!!
    - You may be asked to develop a little task that is very related to your lab to measure your understanding and the genuineness of your work
        - PRE-LAB PREPARATION IS ESSENTIAL
            - Labs are a place to tie up loose ends
                - e.g. get advice from your tutor on how to get around any roadblocks you have encountered
                  or ask questions to resolve difficulties you are having with the week’s material (you can do this in tutorials as well)

Tutorials

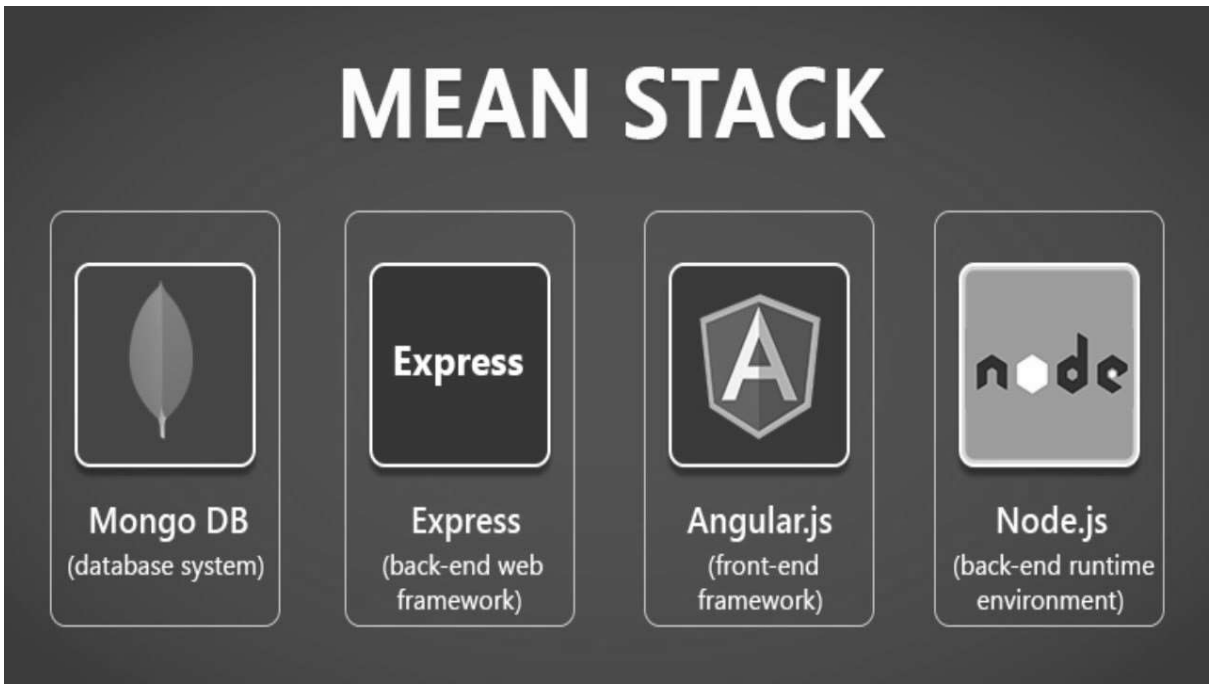
- The Point of the Tutorials
    - Preparation for the upcoming lab
        - Where appropriate this will involve tutors outlining possible solutions to the week’s lab work
        - Otherwise, it will be a Q and A session
    - Enforced consultation sessions focussing on the week’s material
    - To get around roadblocks
        - Tutor discussion and help
        - Peer discussion and help

Teaching approach

Lecture and/or tutorials or problem classes

This teaching and learning approach helps students to initially encounter information via prescribed pre-reading materials, discuss, explore and be quizzed on this information during a subsequent workshop (timetabled as a lecture), and put it into practice in a hands-on lab environment that is preceded by a lab preparation tutorial.

The MEAN Stack



MEAN is a free and open-source JavaScript software stack for building dynamic websites and web applications.

The MEAN stack is MongoDB, Express.js, Angular, and Node.js. Because all components of the MEAN stack support programs are written in JavaScript, MEAN applications can be written in **one language** for both server-side and client-side execution environments.

The components of the MEAN stack are as follows:

- MongoDB a NoSQL database
    - MongoDB is a NoSQL open-source, cross-platform document-oriented database program that JSON-like documents with schemas.
- Express.js a web application framework that runs on Node.js
    - Express is a light-weight web application framework to organise web application into an MVC architecture on the server-side.
- Angular, JavaScript MVC frameworks that run in-browser JavaScript engines,
    - Angular is a structural framework for dynamic web apps. It lets you use HTML as your template language and enables you to extend HTML's syntax to express your application's components clearly and succinctly.

- Node.js, an execution environment for event-driven server-side and networking applications

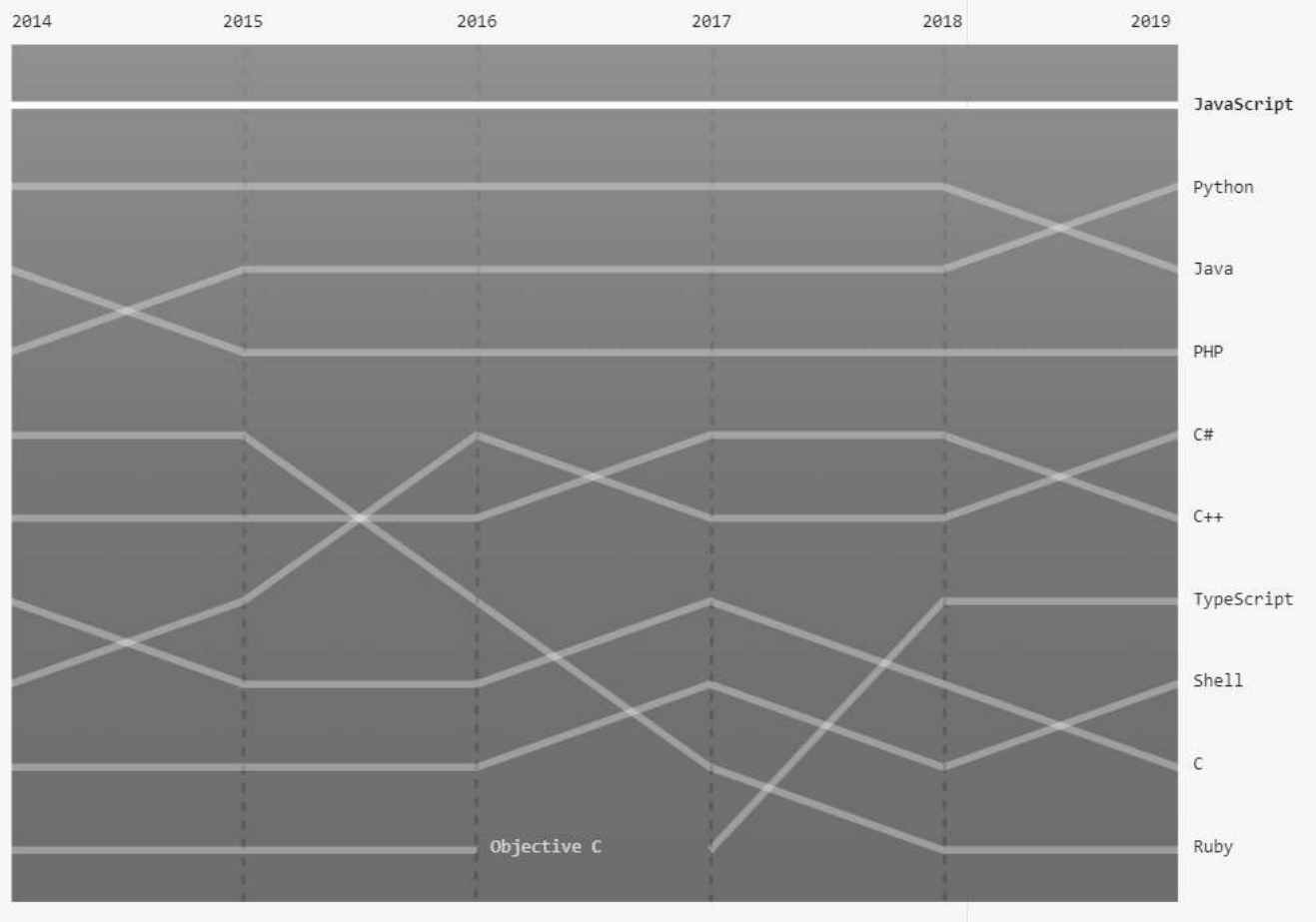
Several variations on the traditional MEAN stack are available by replacing one or more of the components with similar (typically Javascript-based) frameworks. For example, in a MEEN stack, the JavaScript MVC framework Ember.js is used instead of Angular.

Why JavaScript?

For many reasons:

- very easy to learn
- free and available everywhere
- huge community and support
- for frontend and backend development
- Node.js is very popular. It has more than 664726 free packages on NPM (Node Package Manager)

The following figure shows the number of pull requests for the fifteen most popular languages on GitHub.[source]



JavaScript: A Quick Guide

Question: is there a simple, light, online, and fast platform, where I can test and run my JavaScript code?

Answer: YES. Have a look at https://es6console.com/

Question: What if I need to run Javascript locally??

Answer: Save your code in a JS file (e.g. app.js) and run the node command (after installing node.JS):

```
$ node app.js
```

Question: Where can I find an online reference or a book for JavaScript??

Answer: For beginners, see MDN JavaScript tutorials: https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide. It has three levels, beginners, intermediate, and advanced.

For veteran programmers have you seen this https://eloquentjavascript.net/index.html

Binding (a variable name to value) in JavaScript: let, const, and var

- **let** allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used.
- This is unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

```
1. function varTest() {
2.   var x = 1;
3.   if (true) {
```

```
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}

varTest();
letTest();
```

- In the above code, variables declared by **let** have their scope in the block for which they are defined, as well as in any contained sub-blocks.
- The main difference is that the scope of a var variable is the entire enclosing function.

- **const**: Constants are block-scoped, much like variables defined using the let statement. The value of a constant cannot change through re-assignment, and it can't be redeclared.

```
const number = 42;

try {
  number = 99;
} catch(err) {
  console.log(err);
  // expected output: TypeError: invalid assignment to const `number'
  // Note - error messages will vary depending on browser
}

console.log(number);
```

```
// expected output: 42
```

More details:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
- https://eloquentjavascript.net/02_program_structure.html

Control flow

if...else statement

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false. An if statement looks as follows:

```
if (condition_1) {
  statement_1;
} else if (condition_2) {
  statement_2;
} else if (condition_n) {
  statement_n;
} else {
  statement_last;
}
//To execute multiple statements, group them within a block statement
if (condition) {
  statement_1_runs_if_condition_is_true;
  statement_2_runs_if_condition_is_true;
} else {
  statement_3_runs_if_condition_is_false;
  statement_4_runs_if_condition_is_false;
}
```

Example:

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}else{
  console.log('Your input is not a number');
}
```

Note: function ***prompt*** *is not defined in Node.js. It is for the frontend only.*

switch statement

```
switch (fruittype) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Apples':
    console.log('Apples are $0.32 a pound.');
    break;
  case 'Bananas':
    console.log('Bananas are $0.48 a pound.');
    break;
  default:
   console.log('Sorry, we are out of ' + fruittype + '.');
}
console.log("Is there anything else you'd like?");
```

Loops and iteration

for statement

Syntax:

```
for ([initialExpression]; [condition]; [incrementExpression])
  statement
```

Example:

```
let sum = 0;
for (let i = 1; i <= 50; i++) {
   sum = sum + i;
}
console.log("Sum = " + sum);     // => Sum = 1275
```

while loop

JavaScript loops are used to repeatedly run a block of code – until a certain condition is met. When developers talk about iteration or iterating over, say, an array, it is the same as looping. JavaScript offers several options to repeatedly run a block of code, including while, do-while, for and for-in.

```
let sum = 0;
let number = 1;
while (number <= 50) { // -- condition
  sum += number; // -- body
  number++; // -- updater
}
console.log("Sum = " + sum); // => Sum = 1275
```

Functions

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, { }.

For example, the following code defines a simple function named square:

```
function square(number) {
```

```
  return number * number;
}
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

let mycar = {make: 'Honda', model: 'Accord', year: 1998};
let x, y;

x = mycar.make; // x gets the value "Honda"

myFunc(mycar);
y = mycar.make; // y gets the value "Toyota"
                // (the make property was changed by the function)

console.log(x,y);
```

Function expressions

While the function declaration above is syntactically a statement, functions can also be created by a function expression. Such a function can be anonymous; it does not have to have a name. For example, the function square could have been defined as:

```
let square = function(number) { return number * number; };
let x = square(4); // x gets the value 16
```

or as a constant

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
```

Arrow functions

- Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character (not to be confused with the greater-than-or-equal operator, which is written >=). It is a syntactic shortcut.

Example:

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
console.log(power(2, 5));
```

and compare it to:

```
const power = function (base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
console.log(power(2, 5));
```

Optional Arguments

```
const power = function (base, exponent) {
  let result = 1;
  // lets assume if the default value of exponent is 0
  if(exponent===undefined){
    return 1;
  }
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
console.log(power(2, 0));
```

There is another way to provide a default value:

```
const power = function (base, exponent=2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};

console.log(power(2));
```

- In the above code, function power will use the default value of the parameter exponent.

Closures

Closures are an important and a fundamental JavaScript feature but a bit advanced right now. We will discuss it further when/if it becomes important to understand some code we are using.

JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the

variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function. Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the inner function execution, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

```
var createClient = function (name) {
    var age;
    var name;

    return {
        setName: function (newName) {
            name = newName;
        },

        getName: function () {
            return name;
        },

        getAge: function () {
            return age;
        },

        setAge: function (newAge) {
            if (newAge > 0 && newAge < 100) {
                age = newAge;
            }
        }
    }
}

var client = createClient('Tom');
console.log(client.getName());
client.setName('John');
client.setAge(50);
```

```
30. console.log(client.getAge());
31. console.log(client.getName());
```

Classes

What is a class in Object-Oriented Programming (OOP)?

A class is a blueprint of an object. It contains variables for storing data and functions for performing operations on these data.

Why do we need classes?

To have:

- Encapsulation: grouping data and methods that work on that data within one unit
- Inheritance: arrange data items and functions into a hierarchy
- Reusability
- Maintainability

JS has a very different form of objects and inheritance, and classes are a syntactic sugar overlay making JS work like more traditional OO languages (good/bad)

Classes are in fact “special functions”, and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

In the following piece of code, I will create a class called ‘User’ with a constructor and a method:

```
1.  class User {
2.
3.    constructor(name) {
4.      this.name = name;
5.    }
6.
7.    sayHi() {
8.      console.log(this.name);
9.    }
10.
11. }
12.
```

```
let user = new User("John");
user.sayHi();
```

The constructor method is a special method for creating and initialising an object created within a class. Here is another example:

```
class Polygon {
  constructor() {
    this.name = "Polygon";
  }
}

var poly1 = new Polygon();

console.log(poly1.name);
// expected output: "Polygon"
```

JavaScript class for Java and Python Developer: If you have done OOP before, the following examples show how to define and use a class in Java, Python, and JavaScript.

Java:

```
public class Rectangle {
    int width, height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;

    }
    int getArea() {
        return width * height;
    }
    public static void main(String[] args) {
        Rectangle rectangle=new Rectangle(2,3);
```

```
        System.out.println(rectangle.getArea());
    }
}
```

Python:

```
class Rectangle:
    def __init__(self, w=0, h=0):
        self.width = w
        self.height = h

    def get_area(self):
        area = self.width * self.height
        return area


rectangle = Rectangle(2, 3)

# Call getData() function
# Output: 2+3j
print(rectangle.get_area())
```

JavaScript:

```
class Rectangle{
    constructor(w,h){
        this.width=w;
        this.height=h;
    }
    getArea(){
        return this.width*this.height;
    }
}

```

```
let rectangle=new Rectangle(2,3);
console.log(rectangle.getArea())
```

Lab 1

The following source code represents the basic operations of the Queue data structure.

```
class Queue {
    constructor() {
        this.q = [];
    }
// get the current number of elements in the queue
//Getter function
    get length() {
        return this.q.length
    };
//Get all the elements
    get queue() {
        return this.q;
    }
// Boolean function: returns true if the queue is empty, false otherwise
    isEmpty() {
        return 0 == this.q.length;
    };
//adds new element to the end of the quue
    enqueue(newItem) {
        this.q.push(newItem)
    };
```

```
//Boolean function: returns true if an item is found (first occurnace); false otherwise
    inQueue(item) {
        let i = 0;
        let isFound = false;
        while (i < this.q.length && !isFound) {
            if (this.q[i] === item) {
                isFound = true;
            } else
                i++;
        }
        return (isFound);
    }
// pop an item from the queue
    dequeue() {
        if (0 != this.q.length) {
            let c = this.q[0];
            this.q.splice(0, 1);
            return c
        }
    };

};

let queue = new Queue();
queue.enqueue(10);
queue.enqueue(20);
console.log(queue.length);
console.log(queue.q);
queue.dequeue();
queue.enqueue(33);
console.log(queue.q);
console.log(queue.inQueue(33));
console.log(queue.inQueue(88));
```

Expected output

```
2
[ 10, 20 ]
[ 20, 33 ]
true
false
```

Tasks

- add a new function that removes all the elements in the queue
- add a new function that adds a set of items into the queue
    - E.g: queue.addAll([3,7,1,9])
- add a function that pops (dequeues) N elements from the queue. The function should reject the input if there is no enough element to be removed.
    - E.g: queue.dequeueN(2); // pop 2 elements
- add a new function that prints the content of the queue with their indexes. The output can be something like:
    - 1->34
    - 2->30
    - 3->11
    - 4->-3

Note: some text and codes have been taken from books and web pages.

References:

- https://en.wikipedia.org/wiki/MEAN_(software_bundle)
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/
- http://www.dofactory.com/tutorial/javascript-loops
- https://javascript.info/

MODULE

Week 2: Node.js: Fundamentals, Background and First Server App

Nawfal Ali
Updated 1 August 2021

MEAN is an abbreviation for MongoDB, Express, Angular, and Node.js. The concept behind it is to use only JavaScript-driven solutions to cover the different parts of your application. The advantages are significant and are as follows:

- A single language is used throughout the application
- All the parts of the application can support and often enforce the use of the MVC architecture
- Serialisation and deserialization of data structures are no longer needed because data marshalling is done using JSON objects

However, there are still a few essential questions that remain unanswered:

- How do you connect all the components?
- Node.js has a vast ecosystem of modules, so which modules should you use?
- JavaScript is paradigm agnostic, so how can you maintain the MVC application structure?
- JSON is a schema-less data structure, so how and when should you model your data?
- How do you handle user authentication?
- How should you use the Node.js non-blocking architecture to support real-time interactions?
- How can you test your MEAN application code base?
- What kind of JavaScript development tools can you use to expedite your MEAN application development process?

What software is required for full-stack MEAN web development?

- Node.js

- MongoDB
    - Linux
    - Windows
    - macOS

Install MongoDB on Mac systems

You can use the brew package manager to install MongoDB on Mac machines.

Step 1: to install brew, paste this command in the terminal

```
1. /usr/bin/ruby -e "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/master/insta
   ll)"
```

Step 2: install MongoDB, paste this command in the terminal

```
1. brew install mongodb
```

Step 3: start the MongoDB daemon:

```
1. ~/mongodb/bin/mongod
```

Introduction to Node.js

- At JSConf EU 2009, a developer named Ryan Dahl went onstage to present his project called Node.js.
- Starting in 2008, Dahl looked at the current web trends and discovered something odd in the way web applications worked.

- The problem was that web technologies didn't support two-way communication between the browser and the server.
- The test case he used was the Flickr upload file feature, where the browser was unable to know when to update the progress bar as the server could not inform it of how much of the file was uploaded.
- Dahl's idea was to build a web platform that would gracefully support the push of data from the server to the browser, but it wasn't that simple.
- When scaling to common web usage, the platform had to support hundreds (and sometimes thousands) of ongoing connections between the server and the browser.

JavaScript event-driven programming

Node.js uses the event-driven nature of JavaScript to support non-blocking operations in the platform, a feature that enables its excellent efficiency. JavaScript is an event-driven language, which means:

- that you register code to specific events, and
- that code will be executed once the event is emitted.

This concept allows you to seamlessly execute asynchronous code without blocking the rest of the program from running.

To understand this better, take a look at the following Java code example:

```
System.out.print("What is your name?");
String name = System.console().readLine();
System.out.print("Your name is: " + name);
```

In this example, the program executes the first and second lines, but any code after the second line will not be executed until the user inputs their name. This is synchronous programming, where I/O operations block the rest of the program from running. However, this is not how JavaScript works. Because it was initially written to support browser operations, JavaScript was designed around browser events. Even though it has vastly evolved since its early days, the idea was to allow the browser to take the HTML user events and delegate them to JavaScript code. Let's have a look at the following HTML example:

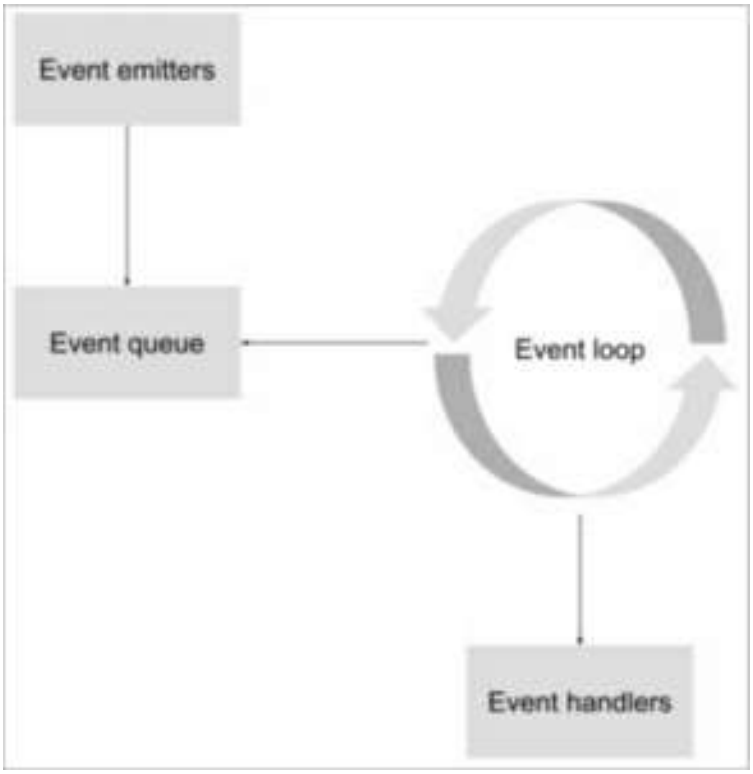
```
<span>What is your name?</span>
```

```
<input type="text" id="nameInput">
<input type="button" id="showNameButton" value="Show Name">
<script type="text/javascript">
let showNameButton = document.getElementById('showNameButton');

showNameButton.addEventListener('click', function() {
    alert(document.getElementById('nameInput').value);
});
// Rest of your code...
</script>
```

In the preceding example, we have a textbox and a button. When the button is pressed, it will alert the value inside the textbox. The primary function to watch here is the addEventListener() method.

As you can see it takes two arguments: the name of the event and an anonymous function that will run once the event is emitted. We usually refer to arguments of the latter kind as a callback function. Notice that any code after the addEventListener() method will execute accordingly regardless of what we write in the callback function.

The browser manages a single thread to run the entire JavaScript code using an inner loop, commonly referred to as the event loop. The event loop is a single-threaded loop that the browser runs infinitely. Every time an event is emitted, the browser adds it to an event queue. The loop will then grab the next event from the queue to execute the event handlers registered to that event. After all of the event handlers are executed, the loop grabs the next event, executes its handlers, grabs the next event, and so on. You can see a visual representation of this process in the following diagram:



Why Node.js?

Node.js uses asynchronous programming!

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting and continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

Node.js Basics and Syntax

Node.js was built on top of the Google Chrome V8 engine and its ECMAScript, which means most of the Node.js syntax is similar to front-end JavaScript (another implementation of ECMAScript), including objects, functions, and methods. In this section, we look at some of the most important aspects; let's call them Node.js/JavaScript fundamentals :

- Buffer—Node.js super data type
- Object literal notation
- Functions
- Arrays
- Prototypal nature
- Conventions

Data types

The latest ECMAScript standard defines seven data types:

- Six data types that are primitives:
  - Boolean
  - Null
  - Undefined
  - Number
  - String
  - Symbol (new in ECMAScript 2015)
- and Object

Node modules

JavaScript has turned out to be a powerful language with some unique features that enable efficient yet maintainable programming. Its closure pattern and event-driven behaviour have proven to be very helpful in real-life scenarios, but like all programming languages, it isn't perfect, and one of its primary design flaws is the sharing of a single global namespace.

In the Node.js module system, each file is treated as a separate module. For example, consider a file named foo.js:

```
1. const circle = require('./circle.js');
2. console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

On the first line, foo.js loads the module circle.js that is in the same directory as foo.js.

Here are the contents of circle.js:

```
const { PI } = Math;

exports.area = (r) => PI * r ** 2;

exports.circumference = (r) => 2 * PI * r;
```

The module circle.js has exported the functions area() and circumference(). Functions and objects are added to the root of a module by specifying additional properties on the particular exports object.

Variables local to the module will be private because the module is wrapped in a function by Node.js (see module wrapper). In this example, the variable PI is private to circle.js.

The module.exports property can be assigned a new value (such as a function or object).

Below, bar.js makes use of the square module, which exports a Square class:

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

The square module is defined in square.js:

```
// assigning to exports will not modify module, must use
module.exports
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
```

Built-in Modules

Node.js has a set of built-in modules which can be used without any further installation. Have a look: Built-in Modules Reference.

Example:

```
const http = require('http');
const fs=require('fs');
```

Modules, classes, and Inheritance

In the following example, we will demonstrate the concepts of having an application with three files (one main file and two modules).

person.js

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
module.exports = Person;
```

teacher.js

```
const Person = require('./person');
class Teacher extends Person {
    constructor(name, age, salary) {
        super(name, age);
        this.salary = salary;
    }
}
module.exports = Teacher;
```

student.js

```
const Person = require('./person');
class Student extends Person {
    constructor(name, age, unit) {
        super(name, age);
        this.unit = unit;
    }
}
module.exports = Student;
```

mainApp.js

```
const Student = require('./modules/student');

const Teacher = require('./modules/teacher');

const teacher = new Teacher('ABC', 29, 1500);
const student = new Student('XYZ', 19, 'FIT2095');

console.log(teacher.salary);
console.log(student.unit);
```

Now observe the following:

- Each class has its constructor.
- Classes Teacher and Student extend class Person. Wow!!! we have inheritance
- Classes Teacher and Student inherited the properties name and age from the parent class Person
- Classes Teacher and Student use the **super** method to invoke the parent's constructor
- To import a package, you have to use the **require** keyword
- Each package has to export its classes or functions using the **module.exports**= **className** or **functionName** expression
- The three classes, Person, Teacher, and Student, are located in a folder named "modules".

Initialising a Node.js Application

Every Node.js application or module will contain a **package.json** file to define the properties of that particular project. This file lives at the root level of your project. Typically, this file is where you'll specify the version of your current release, the name of your application, and the main application file. This file is essential for npm to save any packages to the node community online.

To create a new package.json file, run this command:

```
npm init
```

The output of the above command is a file named package.json contains:

```
{
  "name": "nodeapp",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

Now, let's install a package from the NPM repository.

```
npm install express
```

The above command will do two things:

- Download and install Express package into **node_modules** folder inside your project
- add express (latest version) to the list of dependencies in **pakcage.json file**

```
{
  "name": "nodeapp",
  "version": "1.0.0",
```

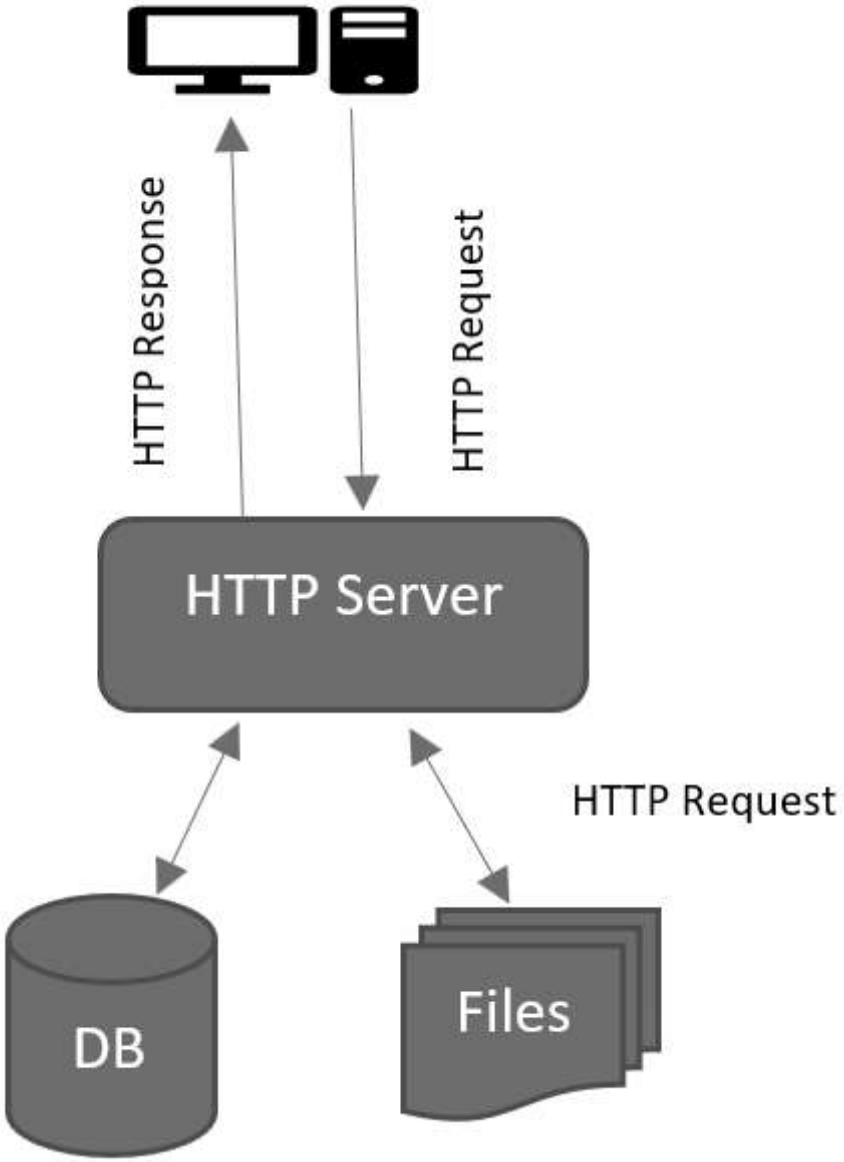
```
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

Basic HTTP Server in Node.js

In this section, we will build a simple HTTP server that accepts connections from clients and responds with a string or a page.

But, what is HTTP server?

- HTTP is the abbreviation of “HyperText Transfer Protocol”.
- The HTTP server is the program that serves data to clients using the HTTP protocol.



In the above figure,

- a client from a web browser sends a request to a web server (e.g. http://websitename.com)
- the server accepts the request and generates a response using its local database or files
- the server sends back its response to the client
- the client's browser displays/renders the response

Now, let's code our first server!!

Save the following code in a JS file server.js and run it using the node command

```
let http = require('http');

http.createServer(function (request, response) {
    console.log('request ', request.url);
```

```
    let d = new Date();
    let currentTime = d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds() + ":" + d.getMilliseconds();
    response.writeHead(200);
    response.write('Hello from FIT2095!! the time is : ' + currentTime);
    response.end();
}).listen(8080);
console.log('Server running at http://127.0.0.1:8080/');
```

to run:

```
$ node server.js
```

From the above code, observe the following:

- the program imports a built-in package called 'http'.
- The http.createServer() method turns your computer into an HTTP server.
- The http.createServer() method creates an HTTP Server object.
- The HTTP Server object can listen to ports on your computer and execute a function, a requestListener, each time a request is made.
- the **createServer** method requires a callback method (i.e. requestListener) as an input parameter
- with each new request, the callback method will be invoked and provided with two parameters: request and response
- using the request object, you can get details about the coming request such as the method, URL, headers, and body of the request
- the response object contains many useful methods for sending data back to the client
- The argument of the res.writeHead() method is the status code, 200 means that all is OK, 404 is page not found
- The response of the server is just a string with the current time

To test your server, navigate to http://localhost:8888

Now, let's send some files back to the client:

Here is the server.js

```
let http = require("http");
let fs = require("fs");
http.createServer(function (request, response) {
    console.log("request ", request.url);
    let filePath = "." + request.url;
    if (filePath === "./") {
      filePath = "./index.html";
    }
    fs.readFile(filePath, function (error, content) {
      if (error) {
        fs.readFile("./404.html", function (error, content) {
          response.writeHead(404, { // file not found
            "Content-Type": "text/html",
          });
          response.end(content, "utf-8");
        });
      } else {
        response.writeHead(200, { // OK
          "Content-Type": "text/html",
        });
        response.end(content, "utf-8");
      }
    });
  })
  .listen(8080);
console.log("Server running at http://127.0.0.1:8080/");
```

- **fs**: the Node.js file system module allows you to work with the file system on your computer. To include the File System module, use the **require()** method: **let fs = require('fs')**;
- **fs.readFile** asynchronously reads the entire contents of a file and invokes the callback upon completion.
- The callback is passed two arguments (err, data), where data is the contents of the file.
- in case of an error in reading or accessing the file, the err parameter will get a value that contains the error code and description.

index.html

```
<html>

<body>
    <h1>
        Hello from HTML file
    </h1>
</body>

</html>
```

login.html

```
<html>

<body>
    <label>Username</label>
    <input type="text" /> </br>
    <label>password</label>
    <input type="password" /> </br>

</body>

</html>
```

404.html

```
<html>

<body>
    <h1>File Not Found :(</h1>
</body>

```

```
7. </html>
```

now let's start the server:

```
1. $ node server.js
```

test your server by visiting pages like:

- http://localhost:8080
- http://localhost:8080/index.html
- http://localhost:8080/login
- http://localhost:8080/login.html
- http://localhost:8080/logout.html

Split the Query String

What is Query String? Is the part of a URL containing data that does not fit conveniently into a hierarchical path structure.

For example:

```
1. http://example.com/over/there?name=Tim&age=20
```

Note:

- starts with a question mark character (?)
- Parameters are key-value pairs that can appear inside URL path
- Query string appears after the path (if any)
- multiple query parameters are separated by the ampersand, "&"

How to split the query string into readable parts in Node.js?

Answer: using a function called 'parse', which is part of the 'url' module

```
let http = require("http");
let url = require("url");
http.createServer(function (req, res) {
    console.log("URL=" + req.url);
    res.writeHead(200, {
      "Content-Type": "text/html",
    });
    var baseURL = "http://" + req.headers.host + "/";
    var url = new URL(req.url, baseURL);
    let params = url.searchParams;
    console.log(params);
    let msg = params.get("year") + " " + params.get("month");
    res.end(msg);
  }).listen(8080);
```

Now:

- to import an external package, we have to use the '**require**' command
- searchParams returns an object representing the query parameters of the URL.
    - use the 'get' to extract the value of a given key

If you run the above code and navigate to this address:

```
http://localhost:8080/getDate?year=2021&month=7
```

where:

- localhost is the reserved address for the local computer, and it is equal to the IP address 127.0.0.1
- 8080 is the port number the server is listening on
- /getDate is the pathname
- year=2021&month=7 is the query string.
    - it consists of two pairs

The expected output in the server's terminal:

```
URL=/getDate?year=2021&month=7
URLSearchParams { 'year' => '2021', 'month' => '7' }
```

To sum up in a single diagram:

URL Routing

Now let's add some routing to our application. Let's assume; we have four pages: index.html, about.html, contanct.html and error.html and the proposed routing is:

- If the client navigates to http://localhost:8080/ (homepage), the server has to send back index.html
- If the client navigates to http://localhost:8080/about, the server has to send back about.html
- If the client navigates to http://localhost:8080/contact, the server has to send back contact.html
- If the client navigates to anywhere else, the server has to send back error.html

the server

```
1. let http = require('http');
2. let fs = require('fs');
3. let fileName = 'index.html';
4. http.createServer(function (request, response) {
5.     console.log('request ', request.url);
6.     let url = request.url;
7.     console.log('request ', url);
8.
9.     switch (url) {
```

```
        case '/':
            fileName = 'index.html';
            break;
        case '/about':
            fileName = 'about.html';
            break;
        case '/contact':
            fileName = 'contact.html';
            break;
        default:
            fileName = 'error.html';
            break;
    }
    fs.readFile(fileName, function (error, content) {
        response.writeHead(200, {
            'Content-Type': 'text/html'
        });
        response.end(content, 'utf-8');
    });
}).listen(8080);
console.log('Server running at http://127.0.0.1:8080/');
```

index.html

```
<html>

<body>

    <p>About:
        <a href="http://localhost:8080/about">About</a>
    </p>
    <p>Contact us:
        <a href="http://localhost:8080/contact">Contact US</a>
    </p>

```

```
</body>

</html>
```

The code above has a problem, which is all the URLs reference a specific server (localhost:8080). What will happen if the app is installed on a different server or served from a different port? All pages will be reached.

Therefore, the best practice is to make all the URLs relative to the server that serves the app by:

```
<html>

<body>

    <p>About:
        <a href="/about">About</a>
    </p>
    <p>Contact us:
        <a href="/contact">Contact US</a>
    </p>

</body>

</html>
```

about.html

```
<html>

<body>
    <p>
        <a href="/">Home</a>
    </p>
    About Page
</body>
```

```
9.
10. </html>
```

contact.html

```
1.  <html>
2.
3.  <body>
4.      <p>
5.          <a href="/">Home</a>
6.      </p>
7.      Contact Page
8.  </body>
9.
10. </html>
```

error.html

```
1.  <html>
2.
3.  <body>
4.      <h1>404 Page not found!!</h1>
5.      <p>
6.          <a href="/">Home</a>
7.      </p>
8.  </body>
9.
10. </html>
```

Now, run the server by executing this command:

```
1. node server.js
```

test your server by visiting pages like:

- http://localhost:8080
- http://localhost:8080/about
- http://localhost:8080/contact
- http://localhost:8080/login
- http://localhost:8080/About

Lab Week 2

You are required to build a web application for our unit FIT2095. It must include the following (but not limited to):

1. An HTTP server that serves clients requests on port 8080. **(4m)**
2. index.html that contains two hyperlinks to two different pages: **(2m)**
    a. Assessments.html: display the four assessments we have in the unit.
        i. example: http://localhost:8080/assessments
    b. Topics.html: display the 12 weeks topics
        i. example: http://localhost:8080/topics
3. 404.html: display an error message for unknown requests **(1m)**
    i. example: http://localhost:8080/testerror
4. a URL endpoint that accepts three parameters (day, month, year) and responds with: **(3m)**
    a. The current week since the first day of semester 2 which is the 26th of July 2021.
    b. An error message if the input date is before the first day of the semester.
    c. An error message if the input date is after week 14.
    d. example: http://localhost:8080/whichweek/?d=5&m=8&y=2020

All four pages must have a link to the home page (index.html)

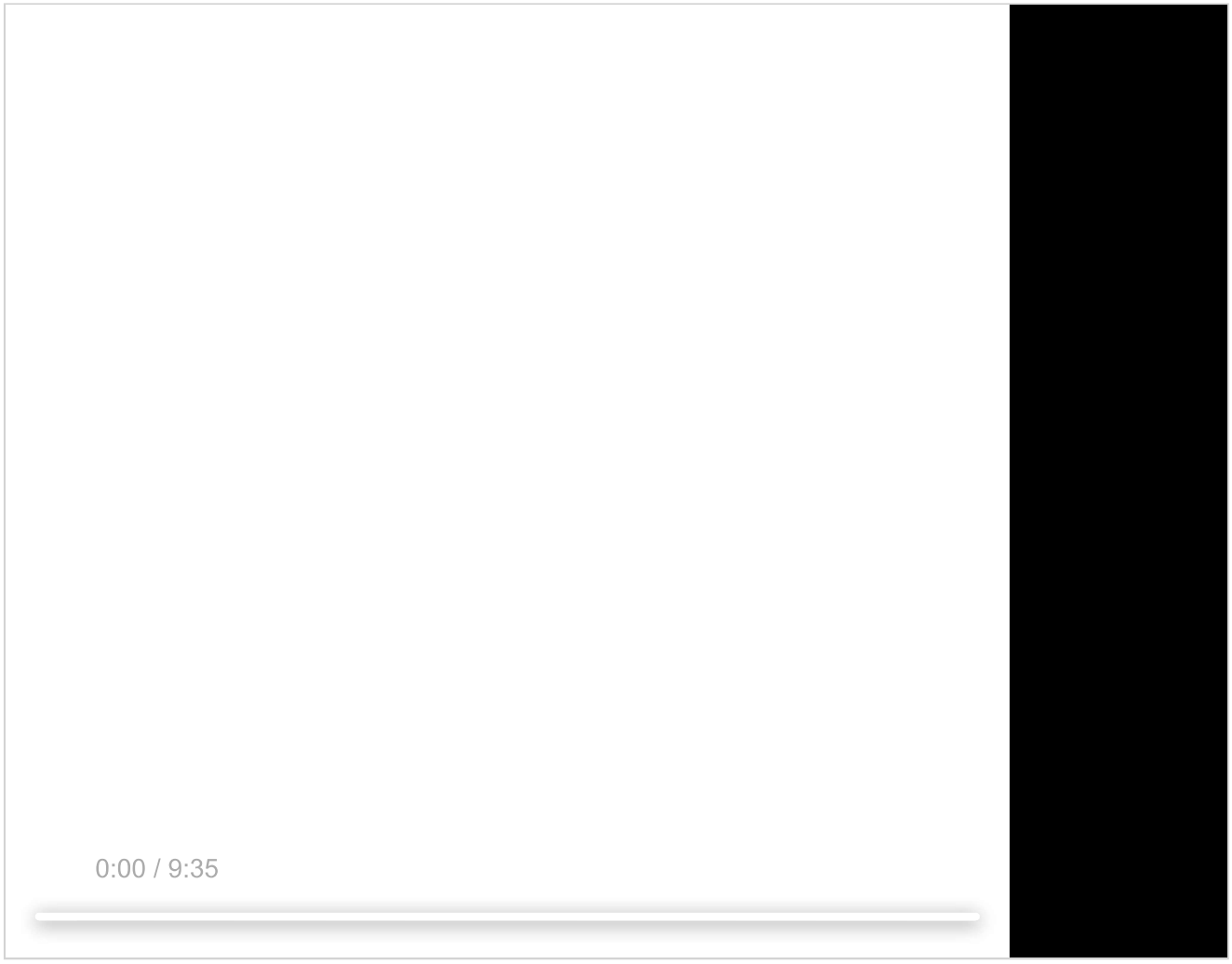
Note: You should not use Express.js

You can use the following function for task 4.

```
/**
 *
 * @param {day} d
 * @param {month} m
```

```
 * @param {year} y
 * @returns week number since August 3,2020; returns -1 if the input is before 3rd of August 2020
 */


function getDaysDiff(d, m, y) {
    let returnValue = -1;
    let currentDay = new Date();
    currentDay.setDate(parseInt(d));
    currentDay.setMonth(parseInt(m) - 1); // months start from 0
    currentDay.setYear(parseInt(y));

    let firstDay = new Date("8/3/2020"); // first day in semester 2

    if (currentDay >= firstDay) {
        var diffDays = parseInt((currentDay - firstDay) / (1000 * 60 * 60 * 24)); //gives day difference
        returnValue = (Math.floor(diffDays / 7) + 1);
    }
    return (returnValue);
}
```

Lab Requirements and Expected output



Support:

- FIT2095 unit preview
- HTML Lists (Unordered and Ordered)
- HTML Hyperlinks
- HTML Emoji

References:

- https://javascript.info
- Haviv, Amos Q.. MEAN Web Development (Kindle Location 905). Packt Publishing. Kindle Edition.
- https://www.w3schools.com/nodejs/nodejs_intro.asp
- https://nodejs.org/api/modules.html

MODULE

Week 3: Developing a Basic Site with Node.js and Express

Nawfal Ali
Updated 9 August 2021

Question: I am building a web server using plain node.js but I have to repeat the same code over and over again. Is there a framework that saves our time and reduces the number of lines of code?

Answer: YES!! Express.js

What is Express?

- Express (http://expressjs.com/) is a web application framework for Node.js.
- It provides all of the tools and basic building blocks you need to get a web server up and running by writing very little code.
- It puts the power to focus on writing your app and not worry about the nuts and bolts that go into making the basic stuff work in your hands.
- Express is like a toolbox. It has a lot of tools to do basic operations, allowing us to focus on the application's logic and content.

In general, most projects in Node.js perform two functions: run a server that listens on a specific port, and process incoming requests. Express is a wrapper for these two functionalities. The following is a basic code that runs the server:

```
1. let http = require('http');
2. http.createServer(function (req, res) {
3.   res.writeHead(200);
4.   res.end('Hello World\n');
```

```
}).listen(8080);
```

This is an example extracted from the official documentation of Node.js. As shown, we use the native module **http** and run a server on the port **8080**. There is also a request handler function, which simply sends the **Hello world** string to the browser. Now, let's implement the same thing but with the Express framework, using the following code:

```
let express = require('express');
let app = express();
app.get("/", function(req, res, next) {
  res.send("Hello world");
}).listen(8080);
```

They are almost the same, but, we don't need to:

- specify the response headers or
- add a new line at the end of the string because the framework does it for us.

> Develop a server that responds with the current time and date to requests with pathname '/time' and '/date' respectively.

```
let express = require('express');
let app = express();
app.get('/', function (req, res) {
    console.log('Hello World');
    res.send('Hello World');
});
app.get('/time', function (req, res) {
    let d = new Date();
    let msg = d.getHours() + ':' + d.getMinutes() + ':' +
d.getSeconds();
```

```
    res.send(msg);
});
app.get('/date', function (req, res) {
    let d = new Date();
    let msg = d.getDate() + '/' + (d.getMonth()+1) + '/' +
d.getFullYear();
    res.send(msg);
});

app.listen(8080);
```

To run this app:

- open the folder that contains the JS file
- open the integrated terminal
- initialize the **npm** package by running this command in the terminal:

```
npm init
```

- install the **express** package by:

```
npm install express
```

package.json

```
{
  "name": "expressapp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
```

```
    }
}
```

now run your application:

```
node server.js
```

To test your app, navigate to:

- http://localhost:8080
- http://localhost:8080/time
- http://localhost:8080/date

> Develop an application that:
>
> - has an array that works like a DB, and
> - accepts requests represent three DB operations
>   - '/list': the client needs the list of users in the database
>   - '/newuser': the client sends details of a new user through the URL to be inserted into the DB
>   - '/deleteitem': the client sends through the URL the id (index) of a user to be deleted
>   - '/': this is just the home page

Let's import the required packages:

```
let express = require('express');
let url = require('url');
```

create an instance of Express app:

```
let app = express();
```

Define the database:

```
// Database is an array of records
let db = [];
```

Insert an initial record:

```
//First record is an object contains the three attributes
let rec = {
    name: 'Tim',
    age: 23,
    address: 'Mel'
};
db.push(rec);
```

listen to port 8080:

```
app.listen(8080);
```

Build a function that generates a string contains the list of users:

```
function generateList() {
    let st = 'Name  Age   Address  </br>';
    for (let i = 0; i < db.length; i++) {
        st += db[i].name + ' | ' + db[i].age + ' | ' + db[i].address + '</br>';
    }
    return st;
}
```

listen to homepage requests (i.e. http://localhost:8080)

```
app.get('/', function (req, res) {
    res.send('Hello from FIT2095');
});
```

listen to '/list' path that responds with a string has all the users in the DB

```
app.get('/list', function (req, res) {

    res.send(generateList());
});
```

if the client wants to insert a new user to the DB, then he has to send a link like http://localhost:8080/newuser?name=Max&age=22&address=ACT, where the details of the new user can be found in the query string:

```
app.get("/newuser", function (req, res) {
    let baseURL = "http://" + req.headers.host + "/";
    let url = new URL(req.url, baseURL);
    let params = url.searchParams;
    console.log(params);

    let newRec = {
        name: params.get("name"),
        age: params.get("age"),
        address: params.get("address"),
    };
    db.push(newRec);
    res.send(generateList());
});
```

Note: the callback responds with the updated list of users

to delete a user from the DB, the client has to send an id (index) through the URL: http://localhost:8080/delete?id=2

```
app.get("/delete", function (req, res) {
    let baseURL = "http://" + req.headers.host + "/";
    let url = new URL(req.url, baseURL);
    let params = url.searchParams;
```

```
    console.log(params);
    deleteUser(params.get("id"));
    res.send(generateList());
});
```

Note: the callback responds with the updated list of users

Now let's put all the above fragments into one piece:

```
let express = require("express");
let app = express();
let url = require("url");
// Database is an array of records
let db = [];
//First record is an object contains the three attributes
let rec = {
    name: "Tim",
    age: 23,
    address: "Mel",
};
//Insert the first record to the db
db.push(rec);
app.get("/", function (req, res) {
    res.send("Hello from FIT2095");
});
app.get("/list", function (req, res) {
    res.send(generateList());
});
app.get("/newuser", function (req, res) {
    let baseURL = "http://" + req.headers.host + "/";
    let url = new URL(req.url, baseURL);
    let params = url.searchParams;
    console.log(params);

```

```
    let newRec = {
        name: params.get("name"),
        age: params.get("age"),
        address: params.get("address"),
    };
    db.push(newRec);
    res.send(generateList());
});
app.get("/delete", function (req, res) {
    let baseURL = "http://" + req.headers.host + "/";
    let url = new URL(req.url, baseURL);
    let params = url.searchParams;
    console.log(params);
    deleteUser(params.get("id"));
    res.send(generateList());
});
app.listen(8080);

function deleteUser(id) {
    db.splice(id, 1);
}

function generateList() {
    let st = "Name  Age   Address  </br>";
    for (let i = 0; i < db.length; i++) {
        st += db[i].name + " | " + db[i].age + " | " + db[i].address +
"</br>";
    }
    return st;
}
```

To run, the above code:

- install express using ‘npm install express –save’ command

- run the app using ‘node server.js’

Routing in Express.js

- Route paths, in combination with a request method, define the endpoints at which requests can be made
- Route paths can be strings, string patterns, or regular expressions
- The characters ?, +, *, and () are subsets of their regular expression counterparts
- The hyphen (-) and the dot (.) are interpreted literally by string-based paths

Examples:

- This route path will match requests to the root route, /.

  URL: http://localhost:8080

```
app.get('/', function (req, res) {
  res.send('root')
})
```

- This route path will match requests to /about.

  URL:http://localhost:8080/about

```
app.get('/about', function (req, res) {
  res.send('about')
})
```

- This route path will match requests to /random.text.

  URL: http://localhost:8080/random.text

```
app.get('/random.text', function (req, res) {
  res.send('random.text')
})
```

- Here are some examples of route paths based on string patterns. This route path will match acd and abcd.

```
app.get('/ab?cd', function (req, res) {
```

```
  res.send('ab?cd')
})
```

- This route path will match abcd, abbcd, abbbcd, and so on

```
app.get('/ab+cd', function (req, res) {
  res.send('ab+cd')
})
```

- This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

```
app.get('/ab*cd', function (req, res) {
res.send('ab*cd')
})
```

- This route path will match /abe and /abcde.

```
app.get('/ab(cd)?e', function (req, res) {
res.send('ab(cd)?e')
})
```

Examples of route paths based on regular expressions:

- This route path will match anything with an “a” in it.

```
app.get(/a/, function (req, res) {
res.send('/a/')
})
```

- This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.

```
app.get(/.*fly$/, function (req, res) {
res.send('/.*fly$/')
})
```

Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL.

The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

Now, let's reimplement the delete operation using the Express Route parameters.

```
app.get('/delete/:userId2Delete', function (req, res) {
console.log(req.params);
deleteUser(req.params.userId2Delete);
res.send(generateList());
})
```

Query String in Express

To get the list of query string parameters in a route, req.query can be used.

For example, if you run this server:

```
let express = require('express');
let app = express();

app.listen(8080);
app.get('/', function (req, res) {
```

```
    console.log(req.query.name);
    console.log(req.query.age);
    console.log(req.query.address);

});
```

and you send this request:

```
http://localhost:8080/?name=Tim&age=23&address=Mel
```

you will get this output:

```
Tim
23
Mel
```

Now, lets update '/newuser' and '/delete' to use req.query

```
app.get("/newuser", function (req, res) {
    // let baseURL = "http://" + req.headers.host + "/";
    // let url = new URL(req.url, baseURL);
    // let params = url.searchParams;
    // console.log(params);

    let newRec = {
        name: req.query.name,
        age: req.query.age,
        address: req.query.address,
    };
    db.push(newRec);
    res.send(generateList());
});

app.get("/delete", function (req, res) {
    //   let baseURL = "http://" + req.headers.host + "/";
```

```
    //   let url = new URL(req.url, baseURL);
    //   let params = url.searchParams;
    //   console.log(params);
    deleteUser(req.query.id);
    res.send(generateList());
});
```

Router-level middleware

For the sake of separation of concerns (SoC) and maintainability, it is possible to separate the routes from the main js file using Express.Router.

To demonstrate this, we need two JS files, router.js and server.js:

```
let express = require('express');
let router = express.Router();

router.get('/', function(req, res){
   res.send('Welcome to FIT2095 Home Page');
});
router.get('/about', function(req, res){
   res.send('This page is about FIT2095');
});

//export this router
module.exports = router;
```

Now the server.js

```
let express = require('Express');
let app = express();

let router = require('./router.js');

//both router.js and server.js should be in same directory
```

```
app.use('/', router);

app.listen(8080);
```

Lab Week 3

You are required to build a web application that represents a bookstore management system.

Specifications are:

1. The database is implemented as an array of items (books)
2. It uses Express.js as a middleware
3. Each item (book) has five attributes: id, title, author, topic,cost
4. the id for the new items is auto-generated.
    - HINT: use Math.random() method.
    - Example: let newId= Math.round(Math.random()*1000)
5. it offers the following operations:
    a. add a new book to the bookstore through the URL.
        - E.g: http://localhost:8080/addbook?title=Harry Potter&author= J. K. Rowling&topic=fiction&cost=15
        - Object {id:789, author:'J. K. Rowling', topic:'fiction',cost:15} should be saved in the database

    b. List all books
        - The output should have five columns: id, title, author, topic, cost
            - HINT: use function **generateList** as a reference
            - URL: http://localhost:8080/getallbooks

    c. delete a book by id
        - URL: http://localhost:8080/deleteid/938, where 938 is the id of the book that should be deleted

    d. get bookstore total value:
        - URL: http://localhost:8080/getbookstorevalue
        - where the bookstore value is equal to $\sum_0^n book.cost$ for all n items in the array DB

MODULE

Week 4: Advanced Express.js

Nawfal Ali
Updated 15 August 2021

Express.js Settings

In order to apply configurations on an Express.js app, we have to use **app.set** to define a value and use **app.get** to retrieve the value based on the key/name of the setting.

Port Number

To set the port number in an Express app:

```
1. app.set('port',8080);
```

where ‘port’ is the key and 8080 is the value (port number)

Now, to retrieve:

```
1. app.listen(app.get('port'));
```

Application-wide Variable

We can also use app.set() to expose variables to templates application-wide.

```
1. app.set('appName', 'FIT2095 App');
```

Working with Middleware

Third-party middleware

Third-party middleware functions can be used to add functionality to Express apps. Think about middleware as a function that has access to Request and Response objects and gets invoked by

Express to do some tasks. For a partial list of third-party middleware functions that are commonly used with Express, see https://expressjs.com/en/resources/middleware.html

In this section, we will cover 'morgan' middleware functions.

Morgan

Morgan is an HTTP request logger middleware for node.js. It generates logs automatically to any request being made. It has a set of predefined formats such as tiny, short, common, etc.

To access the full list of the predefined formats, click here: https://github.com/expressjs/morgan#predefined-formats

Try this example:

server.js

```
let express = require('express');
let morgan = require('morgan');

let app = express();

app.use(morgan('tiny'));

app.get('/', function (req, res) {
    res.send('Hello from FIT2095 server!')
});

app.listen(8080);
```

to run:

```
npm init
npm install express
npm install morgan
node server.js
```

the expected output for the ‘tiny’ format:

```
GET / 304 - - 2.783 ms
GET / 304 - - 0.553 ms
GET /login 404 144 - 2.348 ms
GET /logout 404 145 - 0.812 ms
```

output for the ‘short’ format:

```
::1 - GET / HTTP/1.1 304 - - 2.801 ms::1 - GET /login HTTP/1.1 404 144 - 3.681 ms
::1 - GET /logout HTTP/1.1 404 145 - 0.620 ms
```

output for the ‘common’ format

```
::1 - - [10/Aug/2019:07:45:53 +0000] "GET / HTTP/1.1" 304 -
::1 - - [10/Aug/2019:07:45:58 +0000] "GET /login HTTP/1.1" 404 144
::1 - - [10/Aug/2019:07:46:01 +0000] "GET /logout HTTP/1.1" 404 145
```

Application-level middleware

It is possible to intercept all incoming requests and apply some pre-processing by using middleware functions which can be added through app.use().

In the following example, the express app intercepts all the incoming requests and add the current timestamp.

```
let express = require("express");
let app = express();


app.use(function (req, res, next) {
  req.timestamp = new Date().toISOString();
```

```
    next();
});
```

The next() function represents the next middleware function to be executed. If you choose not to call function next(), the request will be left hanging.

POST Requests

Parse incoming request bodies in a middleware before your handlers, available under the req.body property.

> Develop a web app that its homepage sends data through a POST request to a server that uses Express.js as a framework. The server has to print out the received data and reply with 'Thank you'

index.html

```
<html>

<body>
    <form action="/data" method="POST">
        User Name:
        <input type="text" name="username" /> </br>
        User Age:
        <input type="number" name="userage" /> </br>
        <button>Submit</button>
    </form>
</body>

</html>
```

server.js

```
let express = require('express');
```

```
2.
3.  let app = express();
4.
5.  // parse application/x-www-form-urlencoded
6.  app.use(express.urlencoded({extended: true}));
7.  // parse application/json
8.  app.use(express.json())
9.
10. app.get('/', function (req, res) {
11.     res.sendFile(__dirname + '/index.html');
12. });
13. app.post('/data', function (req, res) {
14.     console.log(req.body.username);
15.     console.log(req.body.userage);
16.     res.send('Thank You')
17.
18. })
19. app.listen(8080);
```

to run:

```
1.  npm init
2.  npm install express
3.  node server.js
```

> Again:
>
> - GET request is used to get data from the server such as html files. Express.js handles GET requests by using app.get (lines 14-16).
> - POST request is used to send data from the client to the server such as username and passwoord fields. Express.js handles POST requests by using app.post (lines 17-22)

> NOTE: The server listens to two requests:
>
> - GET request that comes with pathname='/'
> - POST request that comes with pathname='/data
>
> The body parser middleware parses the data and makes it available under req.body property.

Webpage Rendering

A rendering engine enables you to use static files in your application. At runtime, the engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design a dynamic HTML page.

The most popular view engine is called 'EJS' (Embedded JavaScript).

> EJS is a simple templating language that lets you generate HTML markup with plain JavaScript. No religiousness about how to organize things. No reinvention of iteration and control-flow. It's just plain JavaScript.

To add the EJS engine to your app, you need to npm it first:

```
npm install ejs
```

Secondly, you have to configure the Express app to handle the engine:

```
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');
```

Now, each time you need to send an HTML file to a client, use res.render() method. It renders a view and sends the rendered HTML string to the client. It has two optional parameters:

- locals, an object whose properties define local variables for the view.

- callback, a callback function. If provided, the method returns both the possible error and rendered string but does not perform an automated response. When an error occurs, the method invokes next(err) internally.

On the HTML side (template), we have to use <%= %> tags to print out the value of the attribute.

Let's take an example:

> Develop a Node.js server that passes to the homepage a string and a random number that represent a name and an ID respectively.

server.js

```
let express = require('express');
let app = express();
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

app.get('/', function (req, res) {
    let randomId = Math.round(Math.random() * 100);
    res.render('index.html', { username: "admin",id: randomId});
});
app.listen(8080);
```

index.html

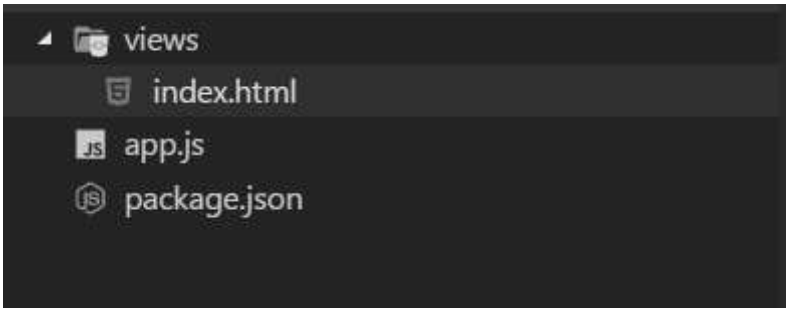
```
<html>
<body>
    <h1> Welcome <%= username%> </h1>
    </br>
    <h3>You ID is<%= id%> </h3>
</body>
```

```
7. </html>
```

to run:

```
1. npm init
2. npm install express
3. npm install ejs
4. node server.js
```

IMPORTANT: by default, res.render needs the HTML files to be in a directory called 'views'. Therefore, the project structure should be:



> Notes:
>
> - Line 8 invokes res.render with two parameters. The first one is the HTML file name. The second parameter is an object that contains two properties username and id. The view engine (EJS) substitutes <%= username%> and <%= id%> with "admin" and the random number respectively. Then, res.render sends the rendered HTML to the client.

Let's have another example:

> Develop a web app that its server sends an array of objects to the client's HTML page. The page has to display all the objects in an unordered list. Each Object contains car ID, car maker, car model, and car year.

server.js

```
let express = require('express');
let app = express();
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

let db = [];
db.push({
    carId: 0,
    carMake: 'BMW',
    carModel: '735',
    carYear: 2014
});
db.push({
    carId: 1,
    carMake: 'Mercedes',
    carModel: 'C250',
    carYear: 2017
});
db.push({
    carId: 3,
    carMake: 'Audi',
    carModel: 'A6',
    carYear: 2019
});

app.get('/', function (req, res) {
    res.render('index.html', {username: "Guest", carDb: db});
});
app.listen(8080);
```

index.html

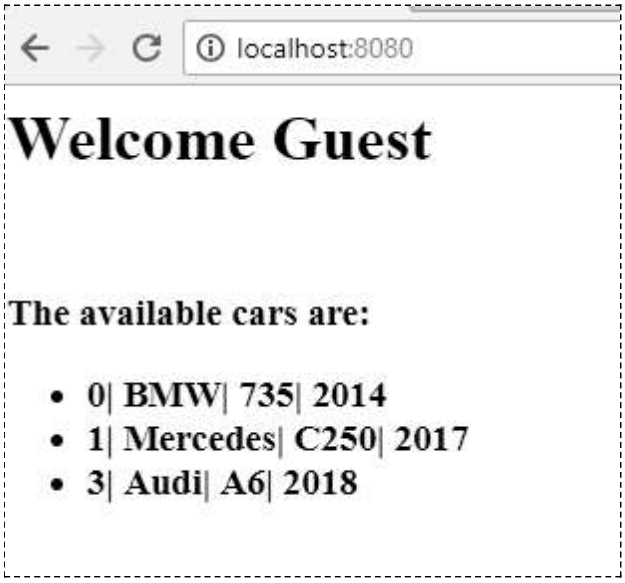
```
<html>

```

```
<body>
    <h1> Welcome  <%= username%> </h1>
    </br>
    <h3>The available cars are:
        <ul>
            <% for ( let i =0; i< carDb.length;i++){ %>
                <li>
                    <%= carDb[i].carId %>|<%= carDb[i].carMake %>|<%= carDb[i].carModel %>|<%= carDb[i].carYear %>
                </li>
                <% }%>
        </ul>
    </h3>
</body>

</html>
```

to run:

```
npm init
npm install express
npm install ejs
node server.js
```

The output:



> Notes:
>
> - db is an array with three objects (server.js lines 6-24)
> - each object has four properties
> - the server listens to GET requests only (server.js lines 26-28)
> - the server sends the index.html and an object has two properties username and carDb (server.js line 28)
> - the view engine substitutes <%= username%> (index.html line 4) with 'Guest'
> - at line 7, tag '<ul>' creates a new unordered list
> - at line 8, the for loop will iterate through all the object in carDb. In each iteration, a new item <il> is created.
> - line 10 provides the content of each item list
> - line 12 closes the begin of the for loop

> Now, let's improve the UI and generate a table rather than a list.

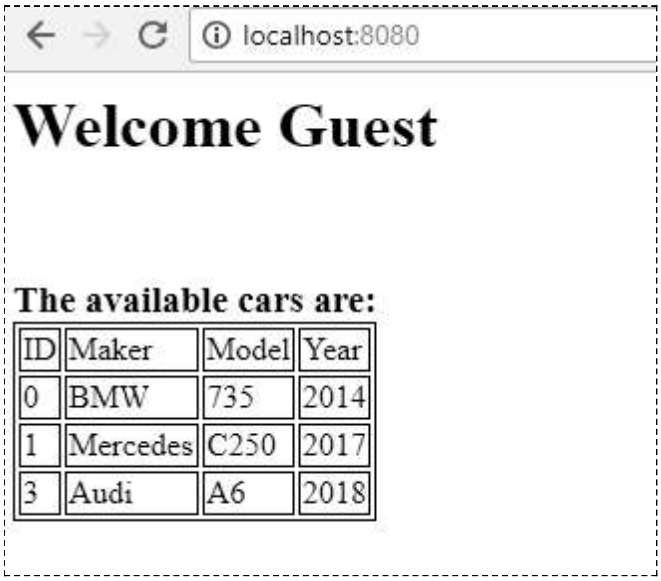
index.html

```
1. <html>
2.
3. <head>
```

```
    <style>
        table,th,td {
            border: 1px solid black;
        }
    </style>
</head>

<body>
    <h1> Welcome
        <%= username%>
    </h1>
    </br>
    <h3>The available cars are:
        <table>
            <tr>
                <th>ID</th>
                <th>Maker</th>
                <th>Model</th>
                <th>Year</th>
            </tr>
            <% for ( let i =0; i< carDb.length;i++){ %>
                <tr>
                    <td>
                        <%= carDb[i].carId %>
                    </td>
                    <td>
                        <%= carDb[i].carMake %>
                    </td>
                    <td>
                        <%= carDb[i].carModel %>
                    </td>
                    <td>
                        <%= carDb[i].carYear %>
                    </td>
                </tr>
                <% }%>
```

```
40.         </table>
41.       </h3>
42.   </body>
43.
44.   </html>
```

The output:



> Notes:
>
> - Line 17: tag <table> creates a new table
> - lines 18-23: tag <tr> creates a new table row that has four data items <td> which represent the header of the table
> - Line 24: is a for loop that iterates through all the object in the array carDb
> - In each iteration, a new table row will be created (lines 25-38)
> - lines 4-8 add some styles to the table

Express and the static assets

To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.

For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

```
app.use(express.static('public'))
```

Now, you can load the files that are in the public directory:

```
http://localhost:8080/images/logo.jpg
https://www.alexandriarepository.org/wp-content/uploads/20160610173720/style.css
http://localhost:8080/js/app.js
https://www.alexandriarepository.org/wp-content/uploads/20190507224749/fight-method-header.png
http://localhost:8080/form.html
```

To use multiple static assets directories, call the express.static middleware function multiple times:

```
app.use(express.static('public'))
app.use(express.static('files'))
```

> Develop a web app that uses Express.js to serve static assets.

server.js

```
let express = require('express');
let app = express();
//Setup the view Engine
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

//Setup the static assets directories
app.use(express.static('images'));
app.use(express.static('css'));


app.get('/', function (req, res) {
```

```
    let randomId = Math.round(Math.random() * 100);
    res.render('index.html', {
        username: "admin",
        id: randomId
    });
});
app.listen(8080);
```

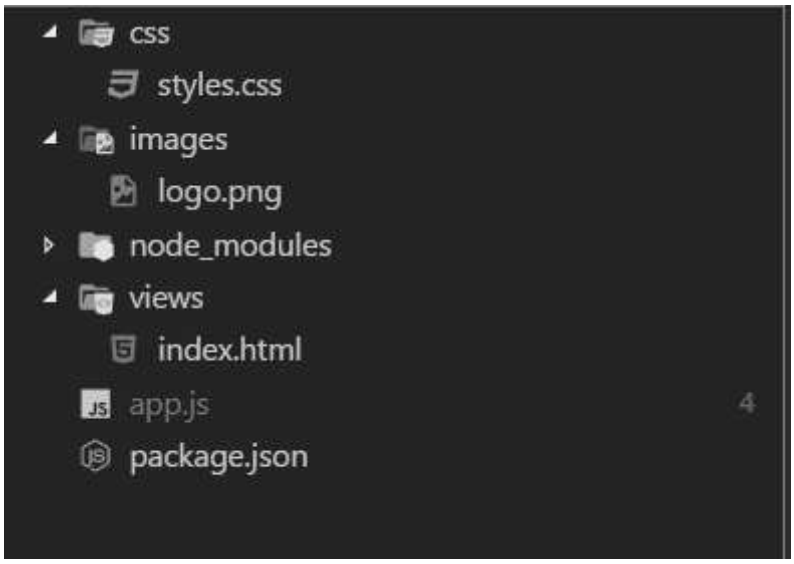
index.html

```
<html>

<head>
    <link rel="stylesheet" href="styles.css">
</head>

<body>
    <img src="logo.png">
    <h1> Welcome
        <%= username%>
    </h1>
    </br>
    <h3>You ID is
        <%= id%>
    </h3>
</body>

</html>
```

styles.css

```
body {
    background-color: bisque;
}
```

where the project structure should be:



and the output:

> Notes:
>
> - server.js, lines 8 and 9 tell express that all the static assets can be found in directories images and css
> - index.html, line 4 requests the file **styles.css** from the server. The css changes the background color of the page into bisque
> - index.html. line 8 adds an image to the page. The tag <img> requests the image 'logo.png' from the server.

MODULE

Express.js and POST Requests

Nawfal Ali
Updated 16 August 2021

This application represents a simple Customers Management System.

Using Node.js and Express, it implements the following concepts:

- GET and POST requests
- Response.sendFile and relative pathnames
- Response.render and the EJS engine
- Static assets

Click HERE to download the App in a ZIP format.

app.js

```
let express = require("express");
let app = express();

//List of customers
let db = [];

// viewPath is required for the response.sendFile function
//__dirname is the  directory name of the current module (i.e file/project).
let viewsPath = __dirname + "/views/";

/* Customer details
    firstName,
    lastName,
    email,
    phoneNumber
```

```
 */

//allow Express to understand the urlencoded format
app.use(express.urlencoded({ extended: true }));

// Express should be able to render ejs templates
app.engine("html", require("ejs").renderFile);
app.set("view engine", "html");

// we have some static assets such as images in this project
app.use(express.static("public/img"));
app.use(express.static("public/css"));

/*
          GET Requests
  */
//if a request to the home page (i.e. '/') arrives
app.get("/", function (req, res) {
  console.log("Homepage request");
  // generate the relative path
  let fileName = viewsPath + "index.html";
  // send index.html back to the client
  res.sendFile(fileName);
});

// a request to add a new customers
app.get("/getaddcustomer", function (req, res) {
  console.log("Add New Customer request");
  //Generate the relative path
  let fileName = viewsPath + "addcustomer.html";
  //send addcusotmer.html page back to the client
  res.sendFile(fileName);
});

//a request to get all customers
app.get("/getallcustomers", function (req, res) {
```

```
  console.log("Homepage request");
  // the content of the page should be generated dynamically.
  // a copy of the array (db) will be send to the rendering engine.
  res.render("allcustomers", {
    customers: db,
  });
});

// POST Requests

// when the user clicks on the submit button
app.post("/postnewcustomer", function (req, res) {
  console.log(req.body);
  db.push(req.body);

  // after pushing the new customer to the database, redirect the client to allcustomer.html
  res.render("allcustomers", {
    customers: db,
  });
});

app.listen(8080);
```

index.html

```
<html>
<body>
    <img src="homepagelogo.jpg" width="400" height="100" />
    <h1>Welcome to Customer Service App</h1>
    <a href="/getaddcustomer">Add New Customer</a> <br>
    <a href="/getallcustomers">List Customers</a>
</body>
</html>
```

addcustomer.html

```
<html>
  <body>
    <img src="homepagelogo.jpg" width="400" height="100" />
    <h1>Enter new Customer:</h1>
    <form action="postnewcustomer" method="POST">
      First Name <input type="text" name="firstName" /> <br />
      Last Name <input type="text" name="lastName" /> <br />
      Email <input type="text" name="email" /><br />
      Phone Number <input type="text" name="phoneNumber" /><br />
      <button>Submit</button>
    </form>
    Click <a href="/"> HERE</a> to return to home page!!
  </body>
</html>
```

allcustomers.html

```
<html>
<head>
    <link rel="stylesheet" href="customers.css">
</head>

<body>
    <table id="t01">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email</th>
            <th>Phone Number</th>
```

```
        </tr>
        <% for(let i=0;i<customers.length;i++){ %>
        <tr>
            <td><%= customers[i].firstName %></td>
            <td><%= customers[i].lastName %></td>
            <td><%= customers[i].email %></td>
            <td><%= customers[i].phoneNumber %></td>
        </tr>
        <%} %>
    </table>
</body>


Click <a href="/"> HERE</a> to return to home page!!

</html>
```

customers.css

```
table {width: 100%;}
table,th,td {
    border: 1px solid black;
    border-collapse: collapse;
}
th,td {
    padding: 15px;
    text-align: left;
}
table#t01 tr:nth-child(even) {
    background-color: #eee;
}
table#t01 tr:nth-child(odd) {
    background-color: #fff;
}
```

```
table#t01 th {
    background-color: black;
    color: white;
}
```

package.json

```
{
  "name": "week4_tute_app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "ejs": "^3.1.6",
    "express": "^4.17.1"
  }
}
```

MODULE

Week 5: MongoDB

Nawfal Ali
Updated 23 August 2021

What is MongoDB?

MongoDB is a free document database management system that offers high scalability, availability, and flexibility. It stores data in JSON-like documents, which means fields can vary from document to document and data structure can be changed over time. [source]

MongoDB Key Features [1,2]

Document DBs

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents. The following example shows a document (i.e. object) with five fields.

```
{
    name:'Tim',
    age:26,
    unit:'FIT2095',
    grade:'HD',
    mark:92.5,
    prerequisites:['FIT1051','FIT2081']
}
```

High Performance

High Availability

MongoDB's replication facility, called a replica set, provides:

– automatic failover and

– data redundancy.

Horizontal Scalability

MongoDB provides horizontal scalability as part of its core functionality:

– Sharding distributes data across a cluster of machines.

– Supports creating zones of data based on the shard key.

Indexing

You can index any field in a document.

Supports Ad-Hoc Queries

Rich Query Language

Terminology and Concepts

Many concepts in relational database systems have close analogs in MongoDB. The table below outlines the common concepts:

SQL	MongoDB
Database	Database
Table	Collection
Index	Index
Row	Document
Column	Field

> If you are a veteran SQL developer, this link compares between SQL and MongoDB in more details: https://docs.mongodb.com/manual/reference/sql-comparison/

Documents and Collections in MongoDB [2]

- MongoDB is a document database, which means that the equivalent of a record is a document or an object.
- A document is a data structure composed of field and value pairs.
- The values of fields may include other documents, arrays, and arrays of documents.
- MongoDB documents are similar to JSON objects.
- Compared to a JSON object, a MongoDB document has support not only for the primitive data types boolean, numbers, and strings, but also other common data types such as dates, timestamps, regular expressions, and binary data.
- A collection is like a table in a relational database. It is a set of documents, and you access each document via the collection.
- In MongoDB, you can have a primary key and indexes on the collection.
- A primary key is mandated in MongoDB, and it has the reserved field name _id.
- MongoDB creates the _id and auto-generates a unique key for every document if it is not supplied.
- MongoDB does not require to define a schema for a collection.
- The only requirement is that all documents in a collection must have a unique _id, but the actual documents may have completely different fields.

> **Question**: Is there any software to visualize, edit, and engineer MongoDB data?
>
> **Answer**: YES. Compass. The GUI for MongoDB

Compass Key Features [3]:

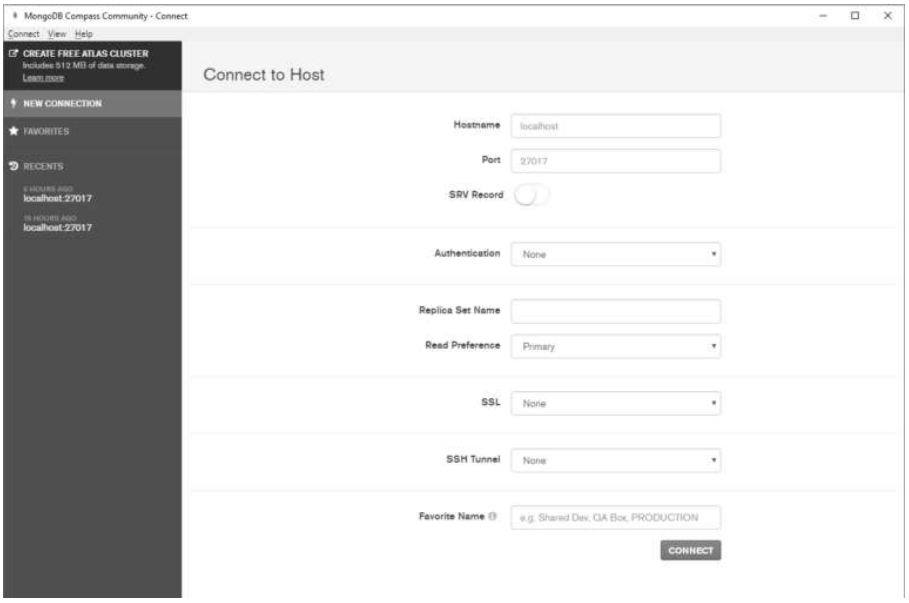
- Visually explore your data.
- Run ad hoc queries in seconds.
- Interact with your data with full CRUD functionality.
- View and optimize your query performance.
- Available on Linux, Mac, or Windows.
- Compass empowers you to make smarter decisions about indexing, document validation, and more.

How to get it?

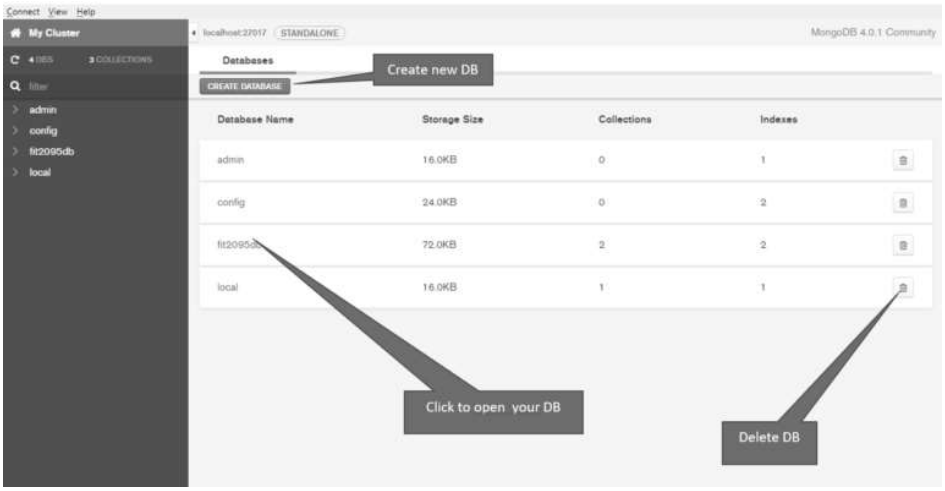
You can download it from https://www.mongodb.com/download-center/compass

How to use it?

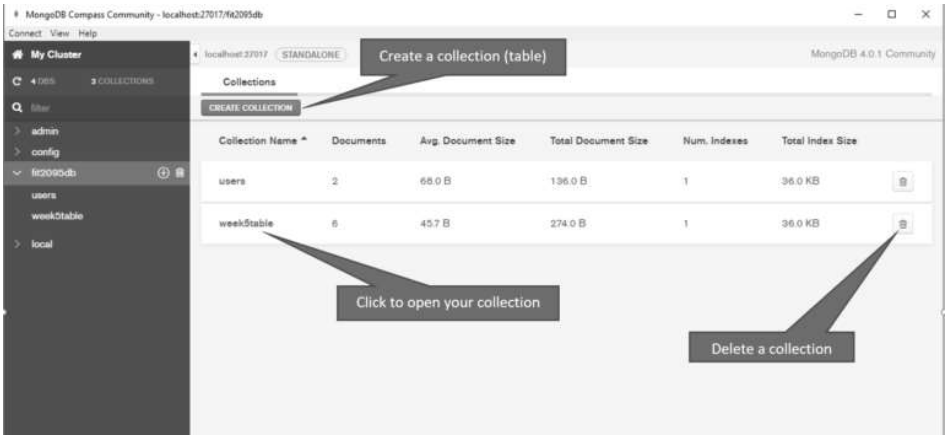
- After you run MongoDB Compass Community App, the connect to host window appears:



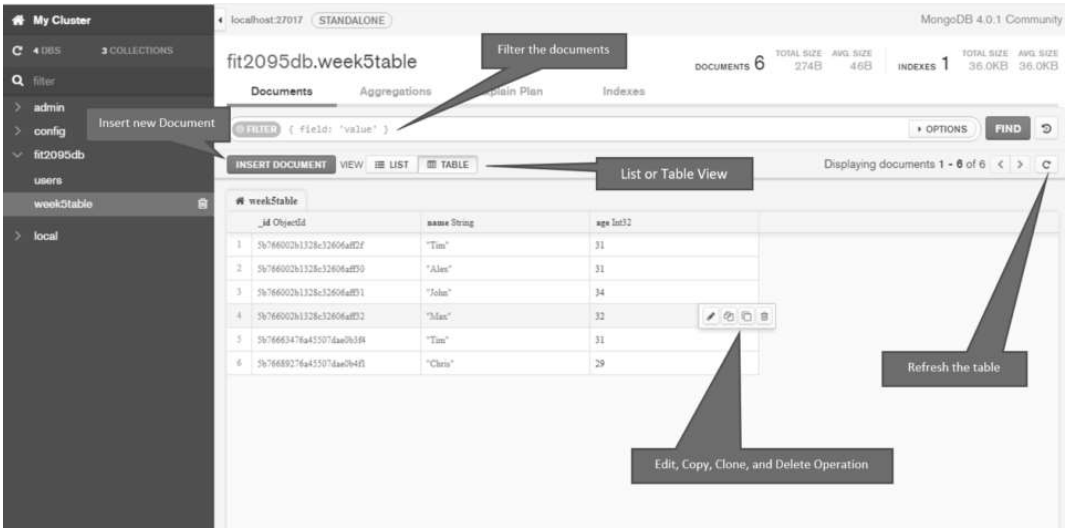
- keep the default values for all fields and click on CONNECT. The main dashboard window appears.



- Click on your database to get the list of collections window.



- Click on the collection name to get its documents



> Notes:
>
> - Collections and documents will be added in the next sections via Node.js
> - You might need to manually refresh Compass after each operation in order to see the effects

---

MongoDB CRUD operations

In this section, we will demonstrate how to integrate MongoDB into a Node.js app and how to perform the four CURD operations: Create, Update, Retrieve, and Delete.

Install MongoDB package (driver):

```
npm install mongodb --save
```

get an instance of MongoDB:

```
const mongodb = require('mongodb');
```

reference MongoDB client:

```
const MongoClient = mongodb.MongoClient;
```

the URL to the MongoDB server:

```
const url = 'mongodb://localhost:27017/';
```

Now let's connect our app to MongoDB server:

```
MongoClient.connect(url, {useNewUrlParser: true}, function (err, client) {
    if (err) {
        console.log('Err  ', err);
    } else {
        console.log("Connected successfully to server");
        db = client.db('fi2095table');
```

```
    }
});
```

- The first parameter is the server's URL, which is a local server listening at port 27017 (MongoDB default port)
- The second parameter is an object that is required for the latest version of MongoDB (version >4/0)
- The third parameter is a callback function that will get executed after the connect operation finishes. It has two parameters: **err** object that gets value if an error occurs and **client** which is used later to access the database as shown in line 6 that connects to a database named 'fit2095db'.

Insert Documents

To insert a new document into the collection, we need a reference to the collection:

```
db.collection('week5table')
```

The above statement references a collection named 'week5table'. The statement creates the collection if it does not exist.

Now, let's insert a document that has one property {name:'Tim'}

```
db.collection('week5table').insertOne({name: 'Tim'});
```

Let's check our document in Compass:

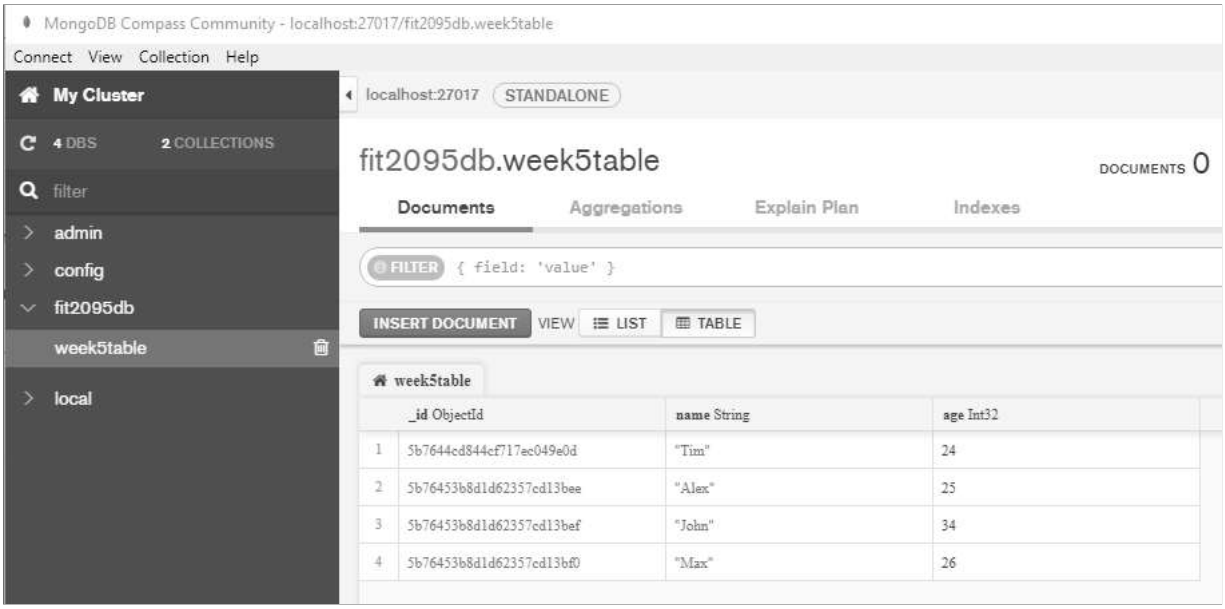
> NOTE: the field **_**id is generated automatically by MongoDB!!

You can also insert several documents at once using **inserMany** function

```
db.collection('week5table').insertMany([
    { name: 'Alex', age: 25 },
    { name: 'John', age: 34 },
    { name: 'Max', age: 26 }
]);
```

where the input to the insertMany function is an array of objects.

Now, our collection:



Retrieve Documents

findOne

The function findOne() returns one document that satisfies the specified query criteria on the collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.

```
db.collection("week5table").findOne({ name: "Tim" }, function (err, result) {
  console.log(result);
});
```

The above statement returns the first document it finds that has name='Tim'.

find()

The **find()** method returns all occurrences in the selection.

```
db.collection("week5table").find({}).toArray(function (err, result) {
  console.log(result);
});
```

The output should be:

```
[ { _id: 5b7644cd844cf717ec049e0d, name: 'Tim', age: 24 },
  { _id: 5b76453b8d1d62357cd13bee, name: 'Alex', age: 25 },
  { _id: 5b76453b8d1d62357cd13bef, name: 'John', age: 34 },
  { _id: 5b76453b8d1d62357cd13bf0, name: 'Max', age: 26 } ]
```

Query the database

In order to filter the result of find() method, a query (i.e. criteria)object should be used.

```
let query = { name: 'Alex' };
db.collection("week5table").find(query).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

We can also use Regular Expressions in the query object:

For example, get all the documents where the name starts with 'T':

```
let query = { name: /^T/ };
db.collection("week5table").find(query).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

and the output:

```
[ { _id: 5b7644cd844cf717ec049e0d, name: 'Tim', age: 24 } ]
```

Finds all the documents where the name ends with 'x':

```
let query = { name: /x$/ };
db.collection("week5table").find(query).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

and the output:

```
[ { _id: 5b76453b8d1d62357cd13bee, name: 'Alex', age: 25 },
  { _id: 5b76453b8d1d62357cd13bf0, name: 'Max', age: 26 } ]
```

Comparison Expression Operators

The comparison expressions take two argument expressions and compare both value and type.

> **Syntax**
>
> { <field>: { Operator: <value> } }

For example, to return all the documents that have an **age** greater than or equal to 25:

```
let query = { age: { $gte: 25 } };
db.collection("week5table").find(query).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

The list of Operators

Operators	Description
$CMP	RETURNS 0 IF THE TWO VALUES ARE EQUIVALENT, 1 IF THE FIRST VALUE IS GREATER THAN THE SECOND, AND -1 IF TH
$eq	Returns true if the values are equivalent.
$gt	Returns true if the first value is greater than the second.
$gte	Returns true if the first value is greater than or equal to the second.
$lt	Returns true if the first value is less than the second.

$lte	Returns true if the first value is less than or equal to the second.
$ne	Returns true if the values are not equivalent.

Sorting the result

The sort() method can be used to sort the result in ascending or descending order.

The sort parameter contains field and value pairs, in the following form:

> **Syntax:**
>
> { field: value }

Specify in the sort parameter the field or fields to sort by and a value of 1 or -1 to specify an ascending or descending sort respectively.

For example: return all the objects that have an age greater than 25 in descending order by 'name'.

```
let query = { age: { $gte: 25 } };
let sortBy={name:-1}
db.collection("week5table").find(query).sort(sortBy).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

and the output:

```
[ { _id: 5b76453b8d1d62357cd13bf0, name: 'Max', age: 26 },
  { _id: 5b76453b8d1d62357cd13bef, name: 'John', age: 34 },
  { _id: 5b76453b8d1d62357cd13bee, name: 'Alex', age: 25 } ]
```

Another example: sort all the documents in a descending by age and ascending by name

```
let query = { age: { $gte: 25 } };
let sortBy={age:-1,name:1}
db.collection("week5table").find(query).sort(sortBy).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

Limit the Result

To limit the result in MongoDB, we use the **limit**() method.

Example: return the top 5 documents that have age>=25.

```
let query = { age: { $gte: 25 } };
let sortBy = { age: -1 }
db.collection("week5table").find(query).sort(sortBy).limit(5).toArray(function (err, result) {
  if (err) throw err;
  console.log(result);
});
```

Delete Documents

collection.deleteOne() removes a single document from the collection based on the filter (i.e. criteria) .

Example: To delete a document that has name='Tim'

```
db.collection("week5table").deleteOne({ name: 'Tim' }, function (err, obj) {
  console.log(obj.result);
});
```

The return object **obj** contains the number of deleted documents.

```
{ n: 1, ok: 1 }
```

To delete more than one: **collection.deleteMany()** removes multiple documents from the collection based on the filter.

Example: To delete all the documents that have age >= 25

```
db.collection("week5table").deleteMany({age: { $gte: 25 }}, function (err, obj) {
  console.log(obj.result);
});
```

the output:

```
{ n: 3, ok: 1 }
```

Update Documents

collection.updateOne() updates a single document based on the filter.

> **Syntax**
>
> db.collection.updateOne(<filter>, <update>, <option>)

Where:

filter: the selection criteria for the update.

Example: {name:'Tim'}

update: the modifications to apply.

we have to use a set of operators to apply the modifications such as **$set** to set a value and **$inc** to increment the value of the field by the specified amount.

Example: {$set:{age:31}}

The full list of operators:

Name	Description
$currentDate	Sets the value of a field to current date, either as a Date or a Timestamp.
$inc	Increments the value of the field by the specified amount.
$min	Only updates the field if the specified value is less than the existing field value.

$max	Only updates the field if the specified value is greater than the existing field value.
$mul	Multiplies the value of the field by the specified amount.
$rename	Renames a field.
$set	Sets the value of a field in a document.
$setOnInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update oper
$unset	Removes the specified field from a document.

option: a set of options to the update method such as upset that creates a new document if no documents match the filter.

Example: update the age of a document with **name**='Tim' to be 31.

```
1. db.collection("week5table").updateOne({ name: 'Tim' }, { $set: {age: 31 } }, { upsert: true }, function (err,
   result) {
2. });
```

Example: update the age of a document with **name**='Chris' to be 29. Insert if 'Chris' does not exist.

```
1. db.collection("week5table").updateOne({ name: 'Chirs' }, { $set: { name: 'Chris', age: 29 } }, { upsert: true },
   function (err, result) {
2. });
```

To update multiple documents, db.collection.updateMany() can be used.

Example: increate the field **age** by 2 to all documents with **name** field ends by 'x'

```
1. db.collection("week5table").updateMany({ name: /x$/ }, { $inc: { age: 2 } }, { upsert: true }, function (err,
   result) {
2. });
```

Now, let's tie all the pieces together.

> Develop an application that represents a simple human resources management system. It should have the following functions:
>
> - Add a new user, where each user has a name, age, and address
> - Delete a user by the name
> - Update the user by the name
> - List all users
>
> Notes:
>
> - Data should be saved persistently
> - redirect the client to 'list users' after the insert, delete, and update operations.

server.js

```
1. //Import packages
2. const express = require("express");
3. const mongodb = require("mongodb");
```

```
const morgan = require("morgan");
const ejs = require("ejs");
//Configure Express
const app = express();
app.engine("html", ejs.renderFile);
app.set("view engine", "html");
app.use(express.static("public"));
app.use(express.urlencoded({ extended: false }));
app.use(morgan("common"));
app.listen(8080);
//Configure MongoDB
const MongoClient = mongodb.MongoClient;
// Connection URL
const url = "mongodb://localhost:27017/";
//reference to the database (i.e. collection)
let db;
//Connect to mongoDB server
MongoClient.connect(url, { useNewUrlParser: true }, function (err, client) {
  if (err) {
    console.log("Err  ", err);
  } else {
    console.log("Connected successfully to server");
    db = client.db("fit2095db");
  }
});
//Routes Handlers
//Insert new User
//GET request: send the page to the client
app.get("/", function (req, res) {
  res.sendFile(__dirname + "/views/index.html");
});
//POST request: receive the details from the client and insert new document (i.e. object) to the collection (i.e.
table)
app.post("/addnewuser", function (req, res) {
  let userDetails = req.body;
  db.collection("users").insertOne({
    name: userDetails.uname,
    age: userDetails.uage,
    address: userDetails.uaddress,
  });
  res.redirect("/getusers"); // redirect the client to list users page
});
//List all users
//GET request: send the page to the client. Get the list of documents form the collections and send it to the
rendering engine
app.get("/getusers", function (req, res) {
  db.collection("users")
    .find({})
    .toArray(function (err, data) {
      res.render("listusers", { usersDb: data });
    });
});
//Update user:
//GET request: send the page to the client
```

```
app.get("/updateuser", function (req, res) {
  res.sendFile(__dirname + "/views/updateuser.html");
});
//POST request: receive the details from the client and do the update
app.post("/updateuserdata", function (req, res) {
  let userDetails = req.body;
  let filter = { name: userDetails.unameold };
  let theUpdate = {
    $set: {
      name: userDetails.unamenew,
      age: userDetails.uagenew,
      address: userDetails.uaddressnew,
    },
  };
  db.collection("users").updateOne(filter, theUpdate);
  res.redirect("/getusers"); // redirect the client to list users page
});
//Update User:
//GET request: send the page to the client to enter the user's name
app.get("/deleteuser", function (req, res) {
  res.sendFile(__dirname + "/views/deleteuser.html");
});
//POST request: receive the user's name and do the delete operation
app.post("/deleteuserdata", function (req, res) {
  let userDetails = req.body;
  let filter = { name: userDetails.uname };
  db.collection("users").deleteOne(filter);
  res.redirect("/getusers"); // redirect the client to list users page
});
```

views/index.html

```
<html>
<head>
        <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1> Insert User</h1> </br>
    <h1>
        <a href="/">Insert Users</a>
        <a href="/getusers">List Users</a>
        <a href="/updateuser">Update User</a>
        <a href="/deleteuser">Delete User</a>
    </h1>
    <form action="/addnewuser" method="POST">
        User Name <input type="text"  name="uname" /> </br>
        User Age <input type="number"  name="uage" /> </br>
        User Address <input type="text"  name="uaddress" /> </br>
        <input type="submit" value="Submit" /> </br>
    </form>
```

```
</body>

</html>
```

views/updateuser.html

```
<html>
        <head>
                <link rel="stylesheet" href="styles.css">
        </head>
<body>
    <h1>Update User</h1></br>
    <h1>
        <a href="/">Insert Users</a>
        <a href="/getusers">List Users</a>
        <a href="/updateuser">Update User</a>
        <a href="/deleteuser">Delete User</a>
    </h1>

    <form action="/updateuserdata" method="POST">

        User Old Name <input type="text"  name="unameold" /> </br>
        User New Name <input type="text"  name="unamenew" /> </br>
        User New Age <input type="number"  name="uagenew" /> </br>
        User New Address <input type="text" name="uaddressnew" /> </br>
        <input type="submit" value="Submit" /> </br>
    </form>
</body>

</html>
```

views/deleteuser.html

```
<html>

<head>
    <link rel="stylesheet" href="styles.css">
</head>

<body>
    <h1>Delete User</h1></br>
    <h1>
        <a href="/">Insert Users</a>
        <a href="/getusers">List Users</a>
        <a href="/updateuser">Update User</a>
        <a href="/deleteuser">Delete User</a>
    </h1>

    <form action="/deleteuserdata" method="POST">

        User name <input type="text"  name="uname" /> </br>
        <input type="submit" value="Submit" /> </br>
```

```
    </form>
</body>

</html>
```

view/listusers.html

```
<html>

<head>
    <head>
        <link rel="stylesheet" href="styles.css">
    </head>
</head>

<body>
    <h1>List Users</h1> </br>
    <h1> <a href="/">Insert Users</a>
        <a href="/getusers">List Users</a>
        <a href="/updateuser">Update User</a>
        <a href="/deleteuser">Delete User</a>
    </h1>
    <table>
        <tr>
            <td>Name</td>
            <td>Age</td>
            <td>Address</td>
        </tr>
        <% for (let i=0;i<usersDb.length;i++){ %>
            <tr>
                <td><%= usersDb[i].name %> </td>
                <td> <%= usersDb[i].age %> </td>
                <td><%= usersDb[i].address %></td>
            </tr>
            <%}%>
    </table>
</body>

</html>
```

to run this app:

```
npm init --y
npm install express ejs mongodb morgan
node server.js
```

Expected Output:

Homepage or insert a new user

List Users

Delete User

Update User

Refereces:

1. https://docs.mongodb.com/manual/introduction/
2. Pro MERN Stack,Vasan Subramanian [book]
3. https://www.mongodb.com/products/compass

MODULE

Week 6: Advanced MongoDB

Nawfal Ali
Updated 25 August 2021

Question: MongoDB is very flexible. It allows me to insert documents in collections with different shapes. But, can I have strongly-typed documents? and is there any way to validate the input data?

Answer: Using MongooseJS, you can have strongly-typed schemas, data validation, data pre-processing, and many other features.

Question: I am developing a web app that stores books and each book has one Author. How can I create a one to one relationship?

Approach 1: the first way to do that is to store the author's object as a property of the book

For example:

```
let Author = {
    _id: 1234,
    firstName: 'Tim',
    lastName: 'John',
    age: 35
};

let book1 = {
    _id: 789,
    title: 'FIT2095 Book',
    author: {
        id: 1234,
        firstName: 'Tim',
```

```
        lastName: 'John',
        age: 35
    },
    isbn: 9876
}
```

But this approach has several problems. I have to iterate through all books documents each time I need to update an author.

Approach 2: Let's save the author's ID as a property of his book.

```
let Author = {
    _id: 1234,
    firstName: 'Tim',
    lastName: 'John',
    age: 35
};

let book1 = {
    _id: 789,
    title: 'FIT2095 Book',
    author_id: 1234,
    isbn: 9876
}
```

> **Question**: Is there any tool or package that helps to create such a relation?
>
> **Answer**: Yes, Mongoose.

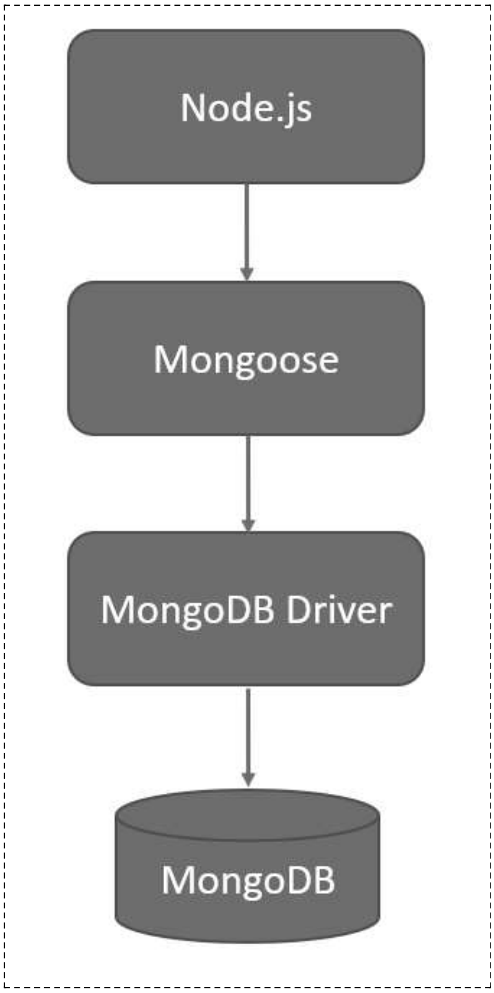
What is Mongoose?

Sources:[2,3,4]

Mongoose is an object data modelling (ODM) library that provides a modeling environment for your collections. It enforces structure as needed while still keeping the flexibility and scalability of MongoDB.

Mongoose is a JavaScript framework that is commonly used in a Node.js application with a MongoDB database. Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in typecasting, validation, query building, business logic hooks and more, out of the box.

It uses MongoDB driver to interact with MongoDB storage.



Mongoose provides an incredible amount of functionality around creating and working with schemas. Mongoose currently contains eight SchemaTypes that a property is saved as when it is persisted to MongoDB. They are:

1. String
2. Number
3. Date
4. Buffer
5. Boolean
6. Mixed
7. ObjectId
8. Array
9. Decimal128
10. Map

SchemaTypes handle definition of path

1. defaults
2. validation
3. getters
4. setters
5. field selection defaults for queries

Further to these common options, certain data types allow you to further customize how the data is stored and retrieved from the database. For example, a String data type also allows you to specify the following additional options:

- convert it to lowercase
- convert it to uppercase
- trim data prior to saving
- a regular expression that can limit data allowed to be saved during the validation process
- an enum that can define a list of strings that are valid

We will discuss them in details soon.

> **In a nutshell, we will use Mongoose functions and options and perform all the CRUD (Create, Retrieve, Update, Delete) operations on MongoDB database.**

Schema in Mongoose

Let's start by creating a database that consists of two collections (Authors and Books). We will implement a simple relationship between them which is One-To-One. In other words, each book has an author.

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

```
var mongoose = require('mongoose');
```

Initially, get a reference from Mongoose package

Create your schema:

```
var authorSchema = mongoose.Schema({});
```

Now let's add our fields:

```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
             firstName:String,
             lastName:String,
            },
    age: Number,
    created: Date
});
```

Using the above schema, each document will get four items, which are: _id, name, age, and created.

Now, I need to add some roles to my schema:

- Field name is mandatory (i.e. required)

```
name: {
        firstName: {
            type: String,
```

```
            required: true
        },
        lastName: String
    }
```

From the above, to make a field required, set the field an object with two properties: the **type** and **required** which needs a boolean value.

- Field age should be a number between 10 and 110

```
age: {
        type: Number,
        validate: {
            validator: function (ageValue) {
                return ageValue >= 10 && ageValue <= 110;
            },
            message: 'Age should be a number between 10 and 110'
        }
    }
```

Similar to name, age becomes an object with two properties: the type and validate which has a boolean function that takes an input represents the field's (age) value and returns true if the value is valid (based on our logic) and false otherwise. The **message** property is a string represents the output that will be printed out when an invalid value is detected.

There is another way to validate the minimum and maximum values of numeric fields by using the **min** and **max** properties:

```
age: { type: Number, min: 5, max: 20 }
```

- Field created has a default value

```
created: {
    type: Date,
    default: Date.now
}
```

> The field created is an object with two properties: the type and the default value of the field if no value is provided.

So, the schema in one piece:

```
const mongoose = require('mongoose');

let authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
        lastName: String
    },
    age: {
        type: Number,
        validate: {
            validator: function (ageValue) {
                return ageValue >= 10 && ageValue <= 110;
            },
            message: 'Age should be a number between 10 and 110'
        }
    },
    created: {
        type: Date,
        default: Date.now
    }
});
```

Now its time to create the Book schema, which has to have the following fields: _id (ObjectId), title (String and required), isdn (String), author (ObjectId, reference wit Author Schema), and created(Date with the default value to the current date/time).

- field **_id**:

```
_id: mongoose.Schema.Types.ObjectId,
```

- field **title:**

```
title: {
        type: String,
        required: true
    },
```

- field **isbn**:

```
isbn: String,
```

- field **author**:

```
author: {
         type: mongoose.Schema.Types.ObjectId,
         ref: 'Author'
     },
```

> Field **author** is an object with two properties: **type** and **ref**. The type is ObjectId because it will only have the Id of the author's document. the **ref** indicates the name of the schema, which is, in this case, the **Author**.

- field **created**:

```
created: {
         type: Date,
         default: Date.now
     }
```

So, the Book schema in one piece:

```
const mongoose = require('mongoose');

let bookSchema = mongoose.Schema({
     _id: mongoose.Schema.Types.ObjectId,
```

```
    title: {
        type: String,
        required: true
    },
    isbn: String,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
    created: {
        type: Date,
        default: Date.now
    }
});
```

Mongoose Models

'Models' are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database. A Mongoose model is a wrapper on the Mongoose schema. A Mongoose schema defines the structure of the document, default values, validators, etc., whereas a Mongoose model provides an interface to the database for creating, querying, updating, deleting records, etc. [4]

So, in order to export a model, we need to invoke the **model** constructor and pass it a string represents **the name of the collection** and a reference to the schema.

For example, the **Author** schema:

```
const mongoose = require('mongoose');

let authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
```

```
        lastName: String
    },
    age: {
        type: Number,
        validate: {
            validator: function (ageValue) {
                return ageValue >= 10 && ageValue <= 110;
            },
            message: 'Age should be a number between 10 and 110'
        }
    },
 //   age     : { type: Number, min: 5, max: 20 },
    created: {
        type: Date,
        default: Date.now
    }
});

module.exports = mongoose.model('Author', authorSchema);
```

and the **Book** model:

```
const mongoose = require('mongoose');

let bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: {
        type: String,
        required: true
    },
    isbn: String,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
```

```
    created: {
        type: Date,
        default: Date.now
    }
});

module.exports = mongoose.model('Book', bookSchema);
```

Now, our database has two collections ‘Author’ and ‘Book’.

Inserting Documents

Now, its time to develop the main app js file and insert new documents into the DB.

Let’s start by referencing Mongoose package

```
const mongoose = require('mongoose');
```

referencing our schemas:

```
const Author = require('./models/author');
const Book = require('./models/book');
```

Create a Mongoose URL string which has syntax: mongodb**://ServerAddress: Port//DbName**

```
let url='mongodb://localhost:27017/libDB';
```

and then connect

```
mongoose.connect(url, function (err) {});
```

The **err** parameter will get a value if an error occurs.

Let's create a new author:

```
let author1 = new Author({
    _id: new mongoose.Types.ObjectId(),
    name: {
        firstName: 'Tim',
        lastName: 'John'
    },
    age: 80
});
```

Now, let's save it:

```
author1.save(function (err) {});
```

the save function has a callback that will get executed after the save operation is completed.

To test the data type validation, try to insert a document where the age is a non-numeric value:

```
let author1 = new Author({
    _id: new mongoose.Types.ObjectId(),
    name: {
        firstName: 'Tim',
        lastName: 'John'
    },
    age: '8a'
});
```

Next, I need to add two books for this author (author1 or 'Tim John'):

```
var book1 = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'FIT2095 Book ',
            author: author1._id,
            isbn: '123456',
```

```
6.          });
```

Line 4, field **author**, which is of type ObjectId and referencing Author Schema (look at line 11 in the Book Schema) is set to the ID of the author we have just created (author1).

Similar to the author, we have to save the new document by calling the function **save**.

```
1.  book1.save(function (err) {
2.      if (err) throw err;
3.      console.log('Book1 successfully Added to DB');
4.  });
```

Let's add another book:

```
1.  var book2 = new Book({
2.      _id: new mongoose.Types.ObjectId(),
3.      title: 'MEAN Stack with FIT2095',
4.      author: author1._id
5.  });
```

and let's save it

```
1.  book2.save(function (err) {
2.      if (err) throw err;
3.
4.      console.log('Book2 successfully add to DB');
5.  });
```

Now, the app.js in one piece:

```
1.  const mongoose = require('mongoose');
2.
3.  const Author = require('./models/author');
4.  const Book = require('./models/book');
5.
6.  mongoose.connect('mongodb://localhost:27017/libDB', function (err) {
7.      if (err) {
```

```
        console.log('Error in Mongoose connection');
        throw err;
    }

    console.log('Successfully connected');

    let author1 = new Author({
        _id: new mongoose.Types.ObjectId(),
        name: {
            firstName: 'Tim',
            lastName: 'John'
        },
        age: 80
    });

    author1.save(function (err) {
        if (err) throw err;

        console.log('Author successfully Added to DB');

        var book1 = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'FIT2095 Book ',
            author: author1._id,
            isbn: '123456',
        });

        book1.save(function (err) {
            if (err) throw err;
            console.log('Book1 successfully Added to DB');
        });

        var book2 = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'MEAN Stack with FIT2095',
            author: author1._id
```

```
44.         });
45.
46.         book2.save(function (err) {
47.             if (err) throw err;
48.
49.             console.log('Book2 successfully add to DB');
50.         });
51.     });
52. });
```

> Note: the creating and saving of **book1** and **book2** have been implemented insode the callback of **author1.save** function. Why??
>
> Node.js is asynchrounus and both **book1** and **book2** required the author's ID (lines 31 and 43). Therefore, we have to create them after the save operation of the author is done.

.insertMany() function

This function takes as input an array of documents (objects) and inserts them if they are valid.

Querying Documents with Mongoose

Mongoose models have several static functions that can be used for CRUD operations.

- Model.deleteMany()
- Model.deleteOne()
- Model.find()
- Model.findById()
- Model.findByIdAndDelete()
- Model.findByIdAndRemove()
- Model.findByIdAndUpdate()
- Model.findOne()
- Model.findOneAndDelete()
- Model.findOneAndRemove()
- Model.findOneAndUpdate()
- Model.replaceOne()

- Model.updateMany()
- Model.updateOne()

> For more details about all the above functions, please navigate to: https://mongoosejs.com/docs/queries.html

Finding Documents

Simple Query

```
Book.find({ 'name.firstName': 'Tim' }, function (err, docs) {
  // docs is an array
});
```

Retrieving only certain fields

Example: get the age of all documents with first name = 'Tim'

```
Author.find({ 'name.firstName': 'Tim' }, 'age', function (err, docs) {
    //docs is an array
    console.log(docs);

});
```

Find only one

```
Model.findOne({'name.firstName':'Tim'}, 'age', function (err, doc) {
//doc is a document
});
```

Find by ID

```
Author.findById(author1._id, 'age', function (err, docs) {
```

```
    console.log(docs);
});
```

Using the ‘where’ clause

Using ‘where’, we can create complex expressions.

Example: Find all documents that have firstName starts with the letter ‘T’ and the age >= 25.

```
Author.where({ 'name.firstName': /^T/
}).where('age').gte(25).exec(function (err, docs) {
     console.log(docs);

 });
```

> Note **exec** indicates the end of the chain and invokes the callback function.

and age <=35

```
Author.where({ 'name.firstName': /^T/
}).where('age').gte(25).lte(35).exec(function (err, docs) {
    console.log(docs);

});
```

limit the result to 10 only:

```
Author.where({ 'name.firstName': /^T/
}).where('age').gte(25).lte(35).limit(10).exec(function (err, docs) {
    console.log(docs);

});
```

and sort the results in ascending order by the **age**

```
Author.where({ 'name.firstName': /^T/
}).where('age').gte(25).lte(35).limit(10).sort('age').exec(function
(err, docs) {
    console.log(docs);

});
```

Populate

Mongoose has a powerful aggregation operator called populate(), which allows you reference documents in other collections.

For exmaple, to get the books and their authors' details:

```
Book.find({}).populate('author').exec(function (err, data) {
        console.log(data);
    });
```

Update Documents

.updateOne() function

This document updates only the first document that matches criteria.

```
Author.updateOne({ 'name.firstName': 'Alex' }, { $set: {
'name.firstName': 'John' } }, function (err, doc) {
    console.log(doc);
});
```

.updateMany() function

Similar to updateOne() except MongoDB will update all the document that matche criteria.

Delete Documents

.deleteOne() function

```
Author.deleteOne({ 'name.firstName': 'Tim' }, function (err, doc) {
    console.log(doc);

});
```

deleteOne() function deletes a document based on condition (criteria).

.deleteMany()

This function deletes all the documents that match the criteria.

```
Author.deleteMany({ 'name.firstName': 'Tim' }, function (err, doc) {
    console.log(doc);

});
```

> The full list of Model's functions can be found here: https://mongoosejs.com/docs/api.html#Model

References:

1. https://mongoosejs.com/
2. Practical Node.js, Building Real-World Scalable Web Apps [Book]
3. https://mongoosejs.com/docs/guide.html
4. https://medium.freecodecamp.org/introduction-to-mongoose-for-mongodb-d2a7aa593c57

MODULE

Week 7: Create a RESTFul Application

Nawfal Ali
Updated 1 September 2021

What is RESTFul API?

RESTFul API is an application programming interface that uses HTTP requests to perform the four CRUD operations (Create, Retrieve, Update, Delete) on data. REST stands for Representational State Transfer. It is a stateless architecture (i.e. no information is retained by either sender or receiver) that uses the Web's existing protocols and technologies.

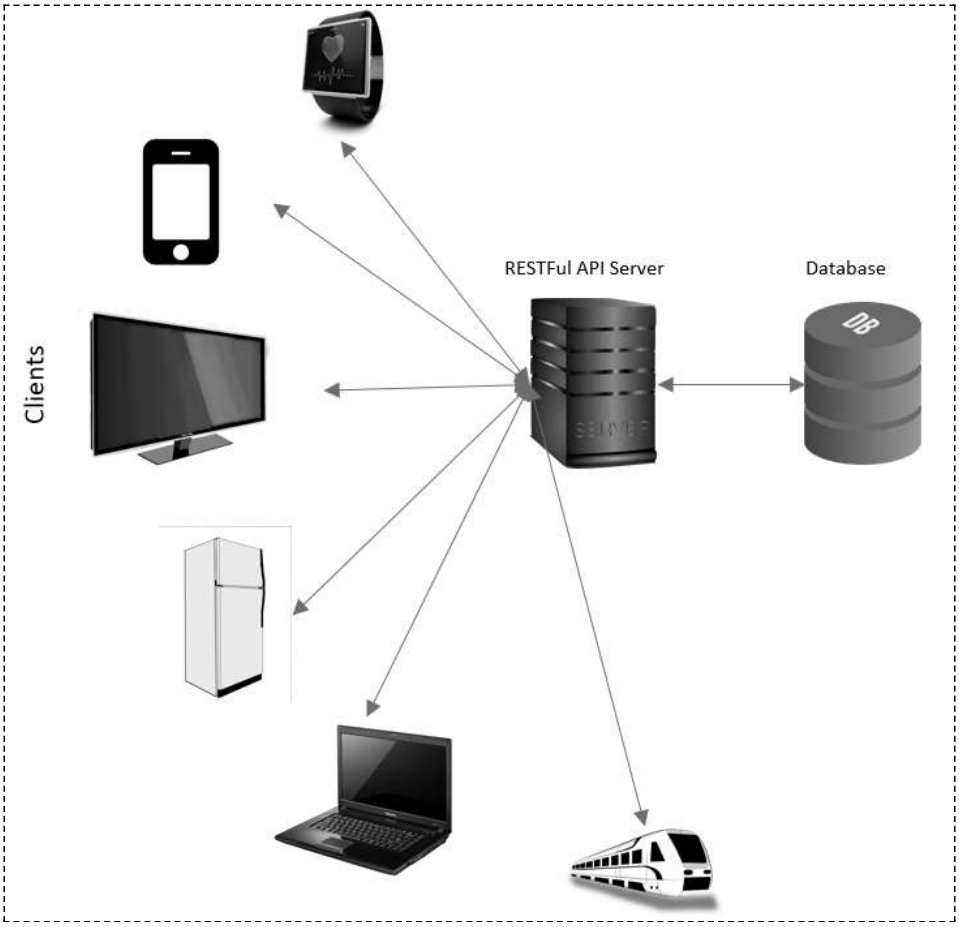
The REST architecture consists of:

- clients
- servers
- resources
- a vocabulary of HTTP operations known as request methods (such as GET, PUT, or DELETE).

Benefits of REST

- The separation between the client and the server
- The REST API is always independent of the type of platform or languages.
    - With REST, data that is produced and consumed is separated from the technologies that facilitate production and consumption. As a result, REST performs well, is highly scalable, simple, and easy to modify and extend
- Since REST is based on standard HTTP operations, it uses verbs with specific meanings such as "get" or "delete" which avoids ambiguity. Resources are assigned individual URIs, adding flexibility.

The following diagram depicts the RESTEFul architecture that consists of a set of different types of clients sending requests and receiving responses from a RESTFul API server which is in turn connected to a database (such as MongoDB).



HTTP methods

HTTP has defined a set of methods which indicates the type of action to be performed on the resources.

- **GET** method requests data from the resource and should not produce any side effect.
  E.g **GET /actors** returns the list of all actors.
- **POST** method requests the server to create a resource in the database, mostly when a web form is submitted.
  E.g **POST /movies** creates a new movie where the movies property could be in the body of the request.
  POST is non-idempotent which means multiple requests will have different effects.
- **PUT** method requests the server to update a resource or create the resource if it doesn't exist.
  E.g. PUT /movie/23 will request the server to update, or create if doesn't exist, the movie

with id =23.
PUT is idempotent which means multiple requests will have the same effects.

- **DELETE** method requests that the resources, or its instance, should be removed from the database.
  E.g DELETE /actors/103 will request the server to delete an actor with ID=103 from the actor's collection.

HTTP response status codes

When the client sends an HTTP request to the server, the client should get feedback, whether it passed, or failed.

HTTP status codes are a set of standardized codes which has various explanations in various scenarios. For example:

- Successful responses
  - 200 OK
  - 201 Created
  - 202 Accepted
- Client error responses
  - 400 Bad Request
  - 404 Not Found
  - 405 Method Not Allowed
- Server error responses
  - 500 Internal Server Error
  - 501 Not Implemented
  - 503 Service Unavailable

The full list of status codes can be found HERE.

Develop an application that represents a library for Movies. You have to be able to add, update, retrieve, and delete data.

Each movie has:

- id
- title
- year
- list of actors

Each actor has:

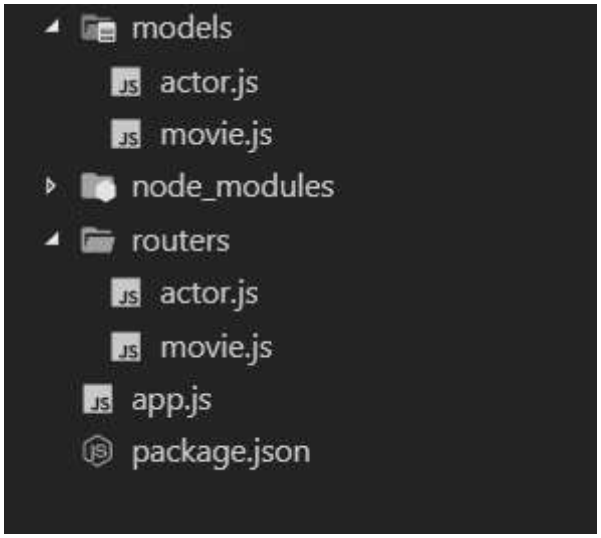
- id
- name
- birth year
- list of movies

Each movie may have one or more actors, and each actor may have participated in multiple movies. In this case, you have many movies related to many actors (many to many relationships).

The server has to respond with status codes:

- 400 if an error occurs
- 404 if no document can be found
- 200 (which is the default) if the request is processed successfully

The structure of the app will be:



The Actor schema

The schema needs access to mongoose library:

```
const mongoose = require('mongoose');
```

Create a new schema

```
const actorSchema = new mongoose.Schema({});
```

Declare the ID of each actor (_id)

```
_id: mongoose.Schema.Types.ObjectId,
```

the name is a mandatory field of type string

```
name: {
    type: String,
    required: true
},
```

the birth year is a mandatory field of type Integer

```
bYear: {
        validate: {
            validator: function (newYear) {
                if (Number.isInteger(newYear))
                    return true;
                else return false
            },
            message: 'Birth year should be integer'
        },
        type: Number,
```

```
        required: true

    },
```

The validator is a boolean function that returns true if the birth year is integer and false otherwise.

In a simpler way, we can code it:

```
bYear: {
    validate: {
        validator: Number.isInteger,
        message: 'Birth year should be integer'
    },
    type: Number,
    required: true

}
```

Now, let's define the list of moves, which is an array of references (i.e. ids) to 'Movie' collection

```
movies: [{
    type: mongoose.Schema.ObjectId,
    ref: 'Movie'
}]
```

The last step is to export the model that uses the above schema.

```
module.exports = mongoose.model('Actor', actorSchema);
```

The Actor Schema ('/models/actor.js) in one piece:

```
const mongoose = require('mongoose');

const actorSchema = new mongoose.Schema({
```

```
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        type: String,
        required: true
    },
    bYear: {
        validate: {
            validator: function (newAge) {
                if (Number.isInteger(newAge))
                    return true;
                else return false
            },
            message: 'Birth year should be integer'
        },
        type: Number,
        required: true

    },
    movies: [{
        type: mongoose.Schema.ObjectId,
        ref: 'Movie'
    }]
});
module.exports = mongoose.model('Actor', actorSchema);
```

The name of the model is ‘Actor’, therefore, mongoose will create a collection named ‘actors’.

The Movie Schema

The movie Schema (‘/models/movie.js’) is pretty similar to Actor.

```
const mongoose = require('mongoose');

const movieSchema = new mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: {
        type: String,
```

```
7.          required: true
8.      },
9.      year: {
10.         type: Number,
11.         required: true
12.     },
13.     actors: [{
14.         type: mongoose.Schema.ObjectId,
15.         ref: 'Actor'
16.     }]
17. });
18.
19. module.exports = mongoose.model('Movie', movieSchema);
```

The name of the model is ‘Movies’, therefore, mongoose will create a collection named ‘movies’.

Lines 13-17 define actors, which is an array of references (i.e. ids) to Actor collection. By doing so, each movie document can reference a set of Actors.

To have better management and to apply the Separation of Concerns principle, two routers for actors and movies will be implemented.

The Actor Router

In this file, all the operation of the ‘Actor’ collection will be implemented.

Firstly, the router needs access to both models (Actor and Movie) and to the Mongoose library.

```
1. const mongoose = require('mongoose');
2.
3. const Actor = require('../models/actor');
4. const Movie = require('../models/movie');
```

Next, due to having more than one function to be exported, the router will export an object where each function is a member (property) of that object.

```
module.exports = {
// Implement your functions
}
```

getAll: is a function that retrieves all the documents from the Actor collection and sends them back as a response.

```
getAll: function (req, res) {
    Actor.find(function (err, actors) {
        if (err) {
            return res.json(err);
        } else {
            res.json(actors);
        }
    });
},
```

> res.json() function sends the response in a JSON format. More details: https://expressjs.com/en/api.html#res.json

createOne: is a function that creates a new document based on the parsed data in 'req.body' and saves it in 'Actor' collection.

```
createOne: function (req, res) {
    let newActorDetails = req.body;
    newActorDetails._id = new mongoose.Types.ObjectId();

    let actor = new Actor(newActorDetails);
    actor.save(function (err) {
        console.log('Done');
        res.json(actor);
    });
},
```

There is another way to insert a new document into a collection by using 'Model.create' function:

```
createOne: function (req, res) {
    let newActorDetails = req.body;
    newActorDetails._id = new mongoose.Types.ObjectId();

    Actor.create(newActorDetails, function (err, actor) {
        if (err)
            return res.json(err);
        res.json(actor);
    });
},
```

getOne: this function finds one document by an ID

```
getOne: function (req, res) {
    Actor.findOne({ _id: req.params.id })
        .populate('movies')
        .exec(function (err, actor) {
            if (err) return res.json(err);
            if (!actor) return res.json();
            res.json(actor);
        });
},
```

> .populate replaces each ID in the array 'movies' with its document.
>
> For example, if there is an actor with two movies, then the output of the above function will be:

```
{
    "movies": [
        {
            "actors": [],
            "_id": "5b89971ce7ef9220bcada5c2",
```

```
            "title": "FIT2095",
            "year": 2015,
            "__v": 0
        },
        {
            "actors": [],
            "_id": "5b8997c4e7ef9220bcada5c3",
            "title": "FIT1051",
            "year": 2020,
            "__v": 0
        }
    ],
    "_id": "5b89692872b3510c78aa14f2",
    "name": "Tim",
    "bYear": 2013,
    "__v": 2
}
```

But if we omit function populate('movies'), the output will be:

```
{
    "movies": [
        "5b89971ce7ef9220bcada5c2",
        "5b8997c4e7ef9220bcada5c3"
    ],
    "_id": "5b89692872b3510c78aa14f2",
    "name": "Tim",
    "bYear": 2013,
    "__v": 2
}
```

More details, please navigate to: https://mongoosejs.com/docs/populate.html

UpdateOne: this function finds a document by its ID and sets new content that is retrieved from 'req.body'

```
updateOne: function (req, res) {
```

```
    Actor.findOneAndUpdate({ _id: req.params.id }, req.body, function (err, actor) {
        if (err) return res.status(400).json(err);
        if (!actor) return res.status(404).json();

        res.json(actor);
    });
},
```

deleteOne: this function deletes the document that matches the criteria.

```
deleteOne: function (req, res) {
    Actor.findOneAndRemove({ _id: req.params.id }, function (err) {
        if (err) return res.status(400).json(err);

        res.json();
    });
},
```

AddMovie: this function adds a movie ID to the list of movies in an actor's document.

```
addMovie: function (req, res) {
    Actor.findOne({ _id: req.params.id }, function (err, actor) {
        if (err) return res.status(400).json(err);
        if (!actor) return res.status(404).json();

        Movie.findOne({ _id: req.body.id }, function (err, movie) {
            if (err) return res.status(400).json(err);
            if (!movie) return res.status(404).json();

            actor.movies.push(movie._id);
            actor.save(function (err) {
                if (err) return res.status(500).json(err);

```

```
                res.json(actor);
            });
        })
    });
},
```

The first step is to retrieve the actor's document (line 2) where the id can be found in the URL's params. The next step is to retrieve the movie (line 6) where its ID is saved in the 'req.body'. After pushing the movie into the actor's document, we save (line 11) it back to the database.

The actor router './routers/actor.js' in one piece:

```
const mongoose = require('mongoose');

const Actor = require('../models/actor');
const Movie = require('../models/movie');

module.exports = {

    getAll: function (req, res) {
        Actor.find(function (err, actors) {
            if (err) {
                return res.status(404).json(err);
            } else {
                res.json(actors);
            }
        });
    },

    createOne: function (req, res) {
        let newActorDetails = req.body;
        newActorDetails._id = new mongoose.Types.ObjectId();

        let actor = new Actor(newActorDetails);
        actor.save(function (err) {
            res.json(actor);
```

```
25.         });
26.     },
27.
28.     getOne: function (req, res) {
29.         Actor.findOne({ _id: req.params.id })
30.             .populate('movies')
31.             .exec(function (err, actor) {
32.                 if (err) return res.status(400).json(err);
33.                 if (!actor) return res.status(404).json();
34.                 res.json(actor);
35.             });
36.     },
37.
38.
39.     updateOne: function (req, res) {
40.         Actor.findOneAndUpdate({ _id: req.params.id }, req.body,
    function (err, actor) {
41.             if (err) return res.status(400).json(err);
42.             if (!actor) return res.status(404).json();
43.
44.             res.json(actor);
45.         });
46.     },
47.
48.
49.     deleteOne: function (req, res) {
50.         Actor.findOneAndRemove({ _id: req.params.id }, function (err)
    {
51.             if (err) return res.status(400).json(err);
52.
53.             res.json();
54.         });
55.     },
56.
57.
58.     addMovie: function (req, res) {
```

```
        Actor.findOne({ _id: req.params.id }, function (err, actor) {
            if (err) return res.status(400).json(err);
            if (!actor) return res.status(404).json();

            Movie.findOne({ _id: req.body.id }, function (err, movie)
{
                if (err) return res.status(400).json(err);
                if (!movie) return res.status(404).json();

                actor.movies.push(movie._id);
                actor.save(function (err) {
                    if (err) return res.status(500).json(err);

                    res.json(actor);
                });
            })
        });
    }
};
```

Movie Router

This router is pretty similar to the actor's router.

getAll: this function uses the Movie model to retrieve all the documents

```
getAll: function (req, res) {
    Movie.find(function (err, movies) {
        if (err) return res.status(400).json(err);

        res.json(movies);
    });
},
```

createOne: this function creates a new movie document

```
createOne: function (req, res) {
    let newMovieDetails = req.body;
    newMovieDetails._id = new mongoose.Types.ObjectId();
    Movie.create(newMovieDetails, function (err, movie) {
        if (err) return res.status(400).json(err);

        res.json(movie);
    });
},
```

getOne: this function uses Movie model to retrieve a document (a movie) using its _id

```
getOne: function (req, res) {
    Movie.findOne({ _id: req.params.id })
        .populate('actors')
        .exec(function (err, movie) {
            if (err) return res.status(400).json(err);
            if (!movie) return res.status(404).json();

            res.json(movie);
        });
},
```

updateOne and **deleteOne** functions have the same concepts as their counterparts in '/router/actor.js'

The movie router './routers/movie.js' in one piece

```
var Actor = require('../models/actor');
var Movie = require('../models/movie');
const mongoose = require('mongoose');

module.exports = {
```

```
6.
7.      getAll: function (req, res) {
8.          Movie.find(function (err, movies) {
9.              if (err) return res.status(400).json(err);
10.
11.             res.json(movies);
12.         });
13.     },
14.
15.
16.     createOne: function (req, res) {
17.         let newMovieDetails = req.body;
18.         newMovieDetails._id = new mongoose.Types.ObjectId();
19.         Movie.create(newMovieDetails, function (err, movie) {
20.             if (err) return res.status(400).json(err);
21.
22.             res.json(movie);
23.         });
24.     },
25.
26.
27.     getOne: function (req, res) {
28.         Movie.findOne({ _id: req.params.id })
29.             .populate('actors')
30.             .exec(function (err, movie) {
31.                 if (err) return res.status(400).json(err);
32.                 if (!movie) return res.status(404).json();
33.
34.                 res.json(movie);
35.             });
36.     },
37.
38.
39.     updateOne: function (req, res) {
40.         Movie.findOneAndUpdate({ _id: req.params.id }, req.body,
    function (err, movie) {
```

```
            if (err) return res.status(400).json(err);
            if (!movie) return res.status(404).json();

            res.json(movie);
        });
    }
};
```

Now, it's time to implement app.js, which is the server-side.

References to the required packages:

```
const express = require('express');
const mongoose = require('mongoose');
```

References to the routers:

```
const actors = require('./routers/actor');
const movies = require('./routers/movie');
```

Create an app from Expressjs and configure it:

```
const app = express();

app.listen(8080);

app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

Ask mongoose to connect to a database named 'movies' :

```
mongoose.connect('mongodb://localhost:27017/movies', function (err) {
    if (err) {
```

```
        return console.log('Mongoose - connection error:', err);
    }
    console.log('Connect Successfully');

});
```

Now, its time to create the RESTFul endpoints.

- If a GET request arrives with pathname ='/actors', execute actor.getAll

```
app.get('/actors', actors.getAll);
```

- If a POST request arrives with pathname ='/actors', execute actor.createOne

```
app.post('/actors', actors.createOne);
```

- If a GET request arrives with pathname ='/actors/:id', where **id** is the ID of the actor, execute actor.getOne

```
app.get('/actors/:id', actors.getOne);
```

- If a PUT request arrives with pathname ='/actors/:id', where **id** is the ID of the actor, execute actor.updateOne

```
app.put('/actors/:id', actors.updateOne);
```

- if a DELETE request arrives with pathname ='/actors/:id', where **id** is the ID of the actor, execute actor.deleteOne

```
app.delete('/actors/:id', actors.deleteOne);
```

the RESTFul endpoints for the Movie Schema have the same concepts as their counterparts in Actor

So, the app.js in one piece:

```
//https://hub.packtpub.com/building-movie-api-express/
const express = require('express');
```

```
const mongoose = require('mongoose');

const actors = require('./routers/actor');
const movies = require('./routers/movie');

const app = express();

app.listen(8080);


app.use(express.json());
app.use(express.urlencoded({ extended: false }));

mongoose.connect('mongodb://localhost:27017/movies', function (err) {
    if (err) {
        return console.log('Mongoose - connection error:', err);
    }
    console.log('Connect Successfully');

});

//Configuring Endpoints
//Actor RESTFul endpoionts
app.get('/actors', actors.getAll);
app.post('/actors', actors.createOne);
app.get('/actors/:id', actors.getOne);
app.put('/actors/:id', actors.updateOne);
app.post('/actors/:id/movies', actors.addMovie);
app.delete('/actors/:id', actors.deleteOne);


//Movie RESTFul  endpoints
app.get('/movies', movies.getAll);
app.post('/movies', movies.createOne);
app.get('/movies/:id', movies.getOne);
app.put('/movies/:id', movies.updateOne);
```

Testing RESTFul Applications

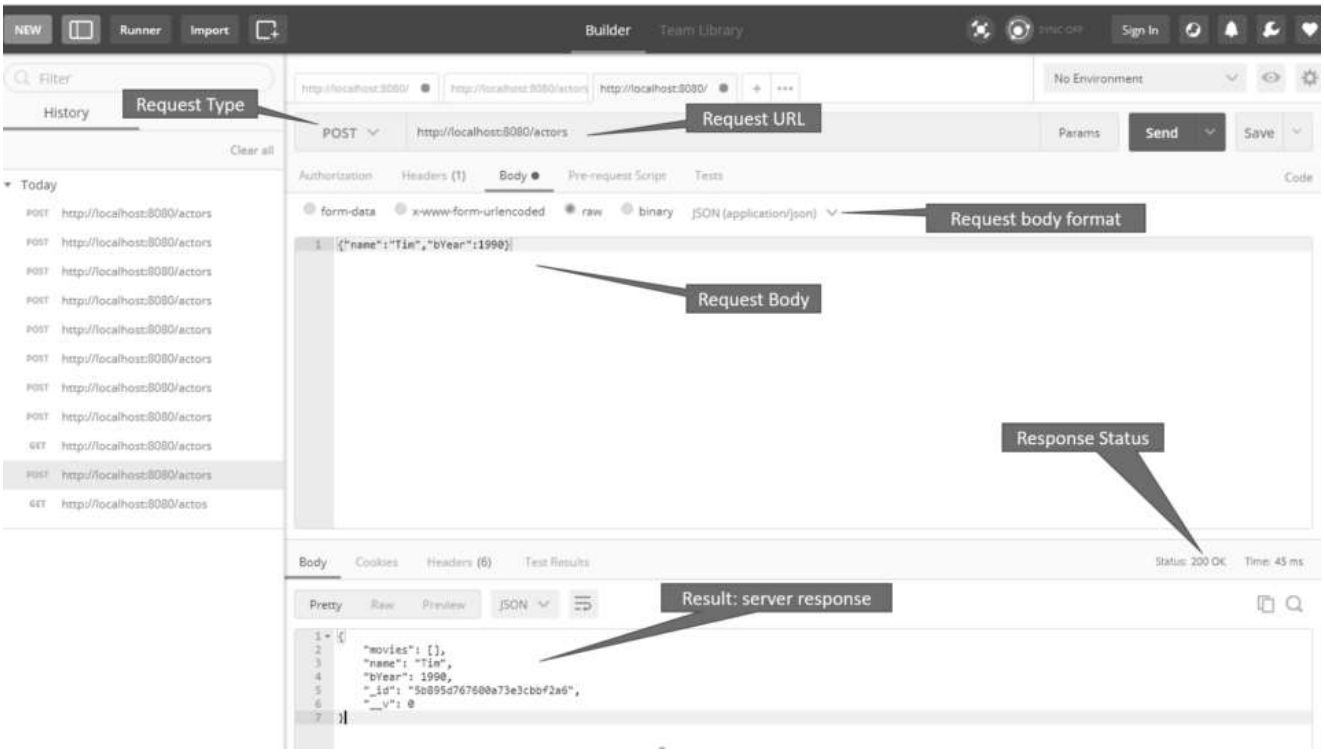
There are several ways to test the RESTFul applications. The first approach is to develop frontend pages that generate HTTP requests (what we have done in the previous weeks). This approach is not generic and flexible enough to test many cases and it needs a lot of time and resources. The second approach is using RESTFul clients that simulate the client-side and generate requests based on our design. This section demonstrates how to install and use Postman RESTFul client plugin.

Postman is an application that enables developers to send HTTP requests to servers.

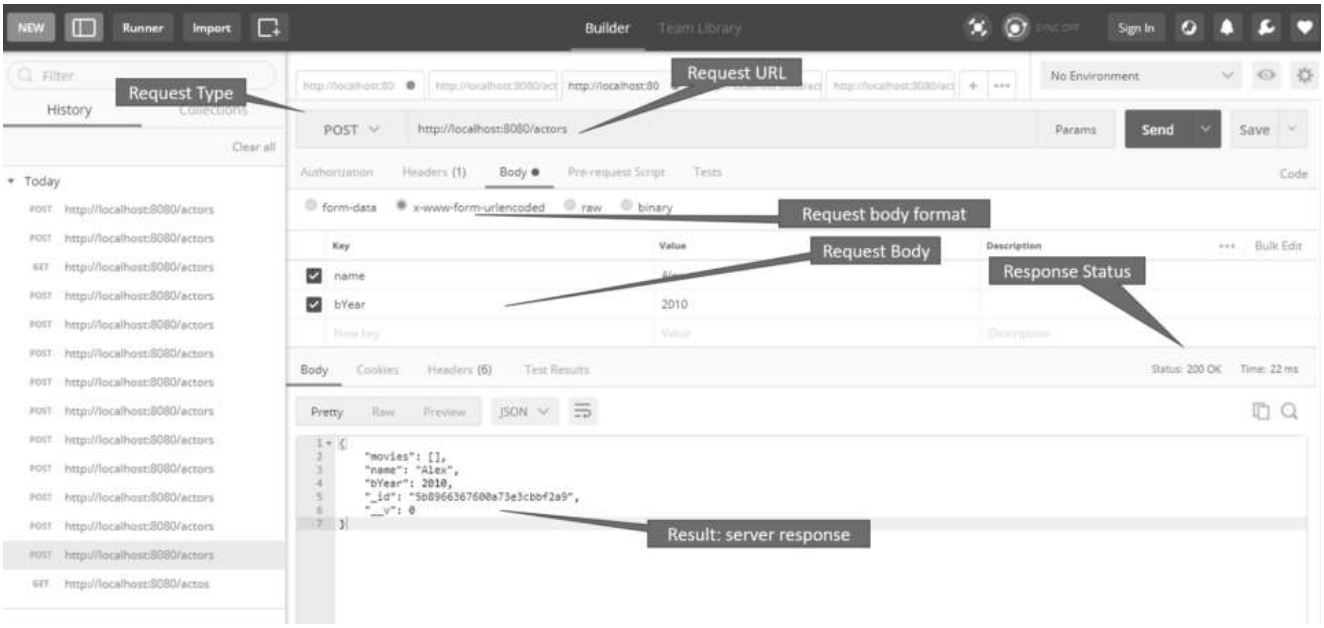
To install Postman as a native app on your machine, navigate to https://www.getpostman.com/downloads/

Now, open Postman and Select Request.

The below picture depicts the required settings to send a POST request with URL **http://localhost:8080/actors** and a body in JSON format:



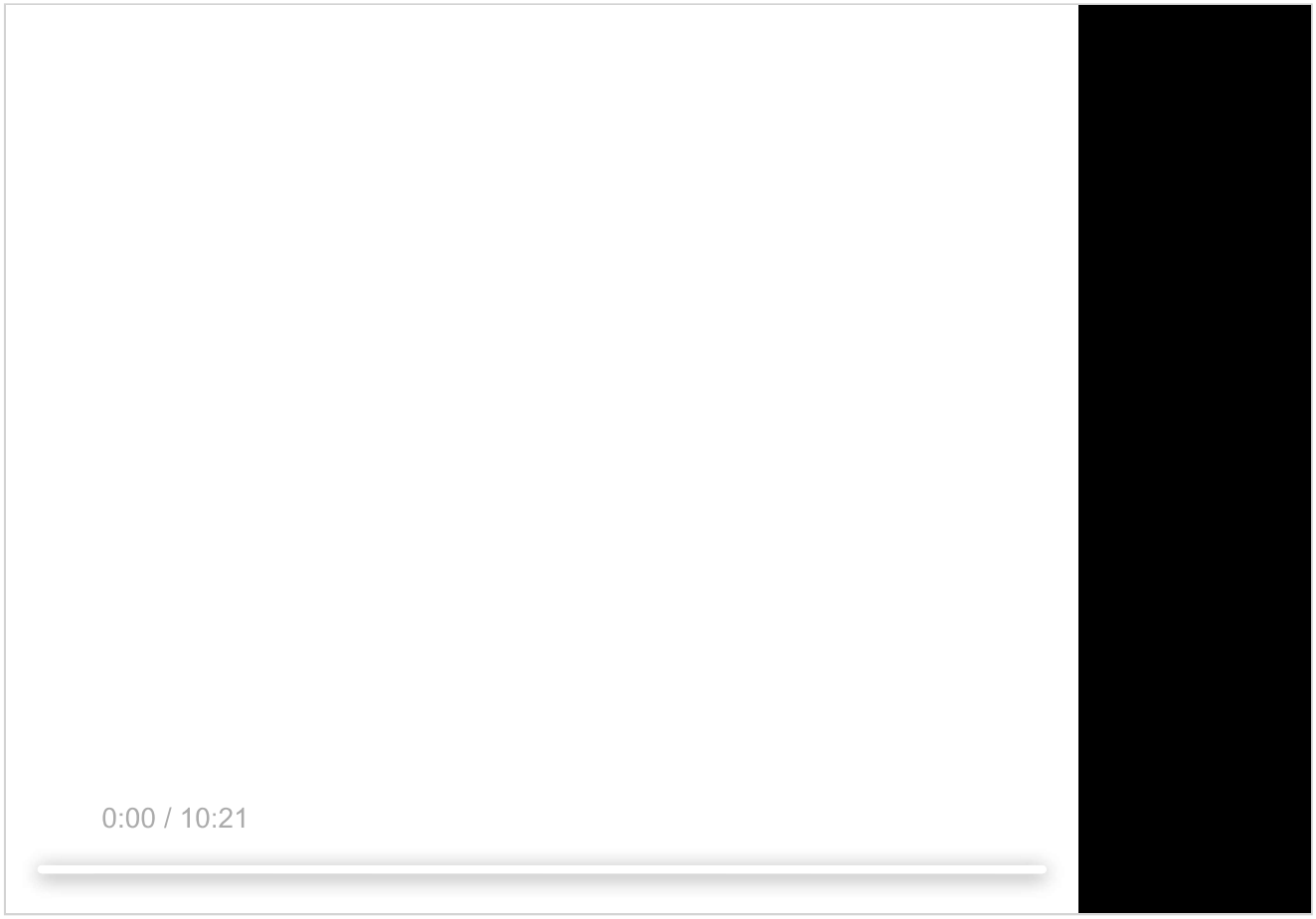
The same post request but the body in urlencoded format:



> Remember, our express app is configured to understand both formats:JSON and urlencoded
>
> ```
> 1. app.use(express.json());
> 2. app.use(express.urlencoded({ extended: false }));
> ```

to send a delete request:

Postman: How to use?



MONASH University | Alexandria BETA

MODULE

Week 8: Angular

Nawfal Ali
Updated 14 September 2021

What is Angular?

Angular is a platform and framework for building client applications in HTML and TypeScript (TypeScript is a strict superset of JavaScript). Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps. Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or desktop.

What is TypeScript?

Typescript is a typed superset of JavaScript that compiles to plain JavaScript.

TypeScript is a strongly typed, object-oriented, compiled language. In other words, TypeScript is JavaScript plus some additional features. TypeScript code is typically transpiled to standards-based JavaScript that all browsers can understand and run.

Main Features of TypeScript

- TS supports strong static typing

```
let flag: boolean = false;
let age: number = 6;
let firstName: string = "bob";
let listA: number[] = [1, 2, 3];
let anotherList: Array<number> = [5, 6, 7];
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

```
let anyValue: any = 4;
anyValue= "maybe a string instead";
anyValue= false; // okay, definitely a boolean

function showMessage(data: string): void {
  alert(data);
}
showMessage('hello');
```

- TS supports Object-Oriented Programming
- TS tools save your developers time

More Details:

- https://msdn.microsoft.com/en-us/magazine/dn890374.aspx
- https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

Angular Main Blocks

The main building blocks of Angular are:

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

Modules

In Angular, a module is a mechanism to group components, directives, pipes and services that are related, in such a way that can be combined with other modules to create an application. An Angular application can be thought of as a puzzle where each piece (or each module) is needed to be able to see the full picture.

Another analogy to understand Angular modules is classes. In a class, we can define public or private methods. The public methods are the API that other parts of our code can use to interact with it while the private methods are implementation details that are hidden. In the same way, a

module can export or hide components, directives, pipes and services. The exported elements are meant to be used by other modules, while the ones that are not exported (hidden) are just used inside the module itself and cannot be directly accessed by other modules of our application.

Modules consist of one or more components. They do not control any HTML. Your modules declare which components can be used by components belonging to other modules, which classes will be injected by the dependency injector and which component gets bootstrapped. Modules allow you to manage your components to bring modularity to your app.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule.

Component

Components are basically classes that interact with the .html file of the component, which gets displayed on the browser.

The component is the template(view) + a class (Typescript code) containing some logic for the view + metadata(to tell angular about from where to get data it needs to display the template).

Components control views (HTML). They also communicate with other components and services to bring functionality to your app.

Templates

The view of the component is defined through templates. Templates are basically the HTML we use to show on our page.

Services

Angular services are singleton objects which get instantiated only once during the lifetime of an application. They contain methods that maintain data throughout the life of an application, i.e. data does not get refreshed and is available all the time. The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application.

Dependency Injection (DI)

Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

In Angular, the DI framework provides declared dependencies to a class when that class is instantiated.

Prerequisites

Angular comes with a Command Line Interface (CLI). The **Angular CLI** makes it easy to create an application that already works, right out of the box. It follows TS programming best practices.

To install:

```
sudo npm install -g @angular/cli
```

> No need for sudo on windows machines

Angular CLI can be used to:

- ng new
  The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows our best practices!
- ng generate
  Generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these.
- ng serve
  Easily test your app locally while developing. It is a simple HTTP server that serves your application on address: http://localhost:4200, where 4200 is the default port number for 'ng serve'.

ng new

To create a new Angular application, run:

```
ng new w8ang
```

a new folder named ‘w8ang’ will be created with several folders and files. The ‘ng new’ command will also install all the required packages into ‘node_modules’ subfolder.

Now, get into the folder

```
cd w8ang
```

and open it using your IDE.

Project structure and explanation of some components can be found below:

Inside ‘/src/app’ folder, you will find five files:

- app.module.ts or the root module. It is to start up your application and set the links to your other modules.
- app.component.ts: This is the file that contains the logic of the root component.
- app.component.html: The view of the component
- app.component.css: The style sheet of the component
- app.component.spec.ts: Unit testing for the component's logic. We will ignore it.

app.component.ts

This file declares the root component of our application. By default, it contains:

```
1.  import { Component } from '@angular/core';
2.
3.  @Component({
4.    selector: 'app-root',
5.    templateUrl: './app.component.html',
6.    styleUrls: ['./app.component.css']
7.  })
8.  export class AppComponent {
9.    title = 'w8ang';
10. }
```

> - The **@Component** decorator identifies the class immediately below it as a component class, and specifies its metadata.
> - **selector**: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-root></app-root>, then Angular inserts an instance of the AppComponent view between those tags. You can find the '<app-root></app-root>' in **index.html** file.
> - **templateUrl**: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
> - **providers**: An array of providers for services that the component requires. We will cover it in the comming weeks.

app.component.html

Open this file and replace its code with:

```
<p>
  Welcome to {{title}}!!
</p>>
```

> The {{title}} interpolation (see the data binding section for more details) displays the component's **title** property value within the <p> element.

To run the application, run this command:

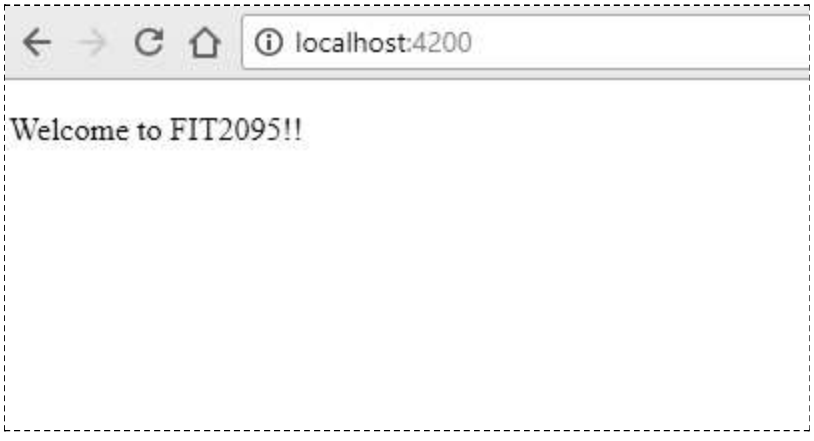
```
ng serve
```

> Keep the terminal session alive as 'ng serve' will monitor your changes and recompile then reload your borower automatically.

Now, open 'app.component.ts' and change the value of the title to 'FIT2095'

```
title = 'FIT2095';
```

Your browser should display:



ng generate

Generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these.

To add a new component to the project, run this command in your terminal

```
ng generate component StudentLogin
```

or, to add a new service:

```
ng generate service RetriveDataServer
```

> More details: https://github.com/angular/angular-cli/wiki

ng serve

It builds the application and starts a web server.

```
ng serve
```

Angular CLI comes with a simple HTTP server that can be used for testing your application during the development phase.

Data Binding

Data binding is the automatic synchronization of data between the model and view components. It allows you to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data.

There are three types of data binding:

- Property Binding
- Event Binding
- Two Way Binding

Property Binding (Component to HTML)

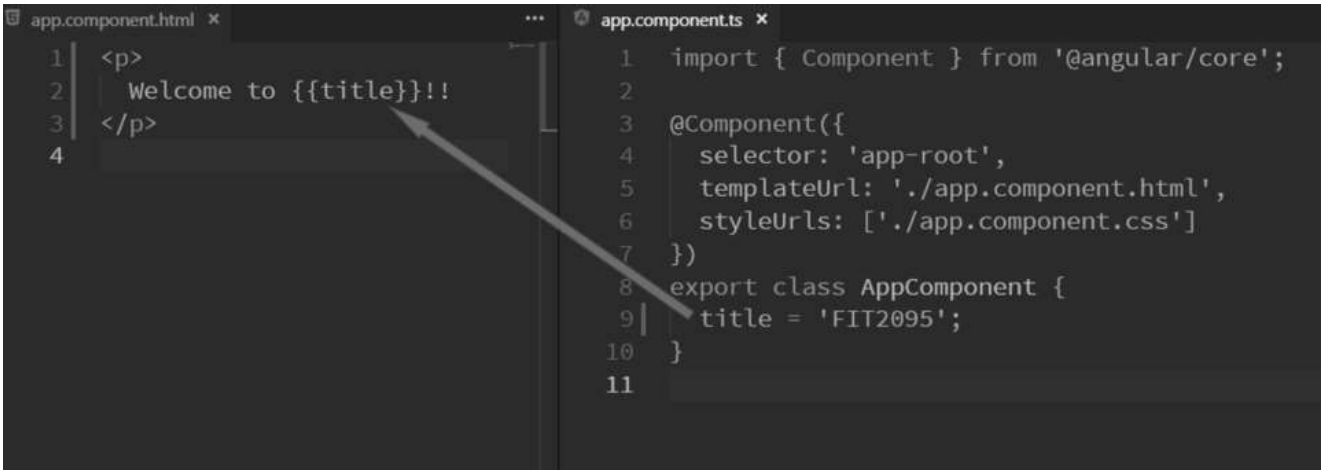
Property binding simply means passing data from the component class (e.g. app.component.ts) and setting the value of a given element in the view (e.g. app.component.html). Property binding is one way, in that the data is transferred from the component to the class. The benefit of

property binding is that it allows you to control element property values from the component and change them whenever needed.

For example:

```
<p>
  Welcome to {{title}}!!
</p>
```

The text between the braces (title) is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. In the example above, Angular evaluates the title the application title on the HTML.



Event binding (HTML to Component)

The property binding flows data in one direction: from a component to an element.

Users usually enter text into input boxes, pick items from lists and/or click buttons. Such actions may result in a flow of data in the opposite direction: from an element to a component.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches. You declare your interest in user actions through Angular event binding.

Example: Build a click event counter using Angular

open app.component.ts and paste this code:

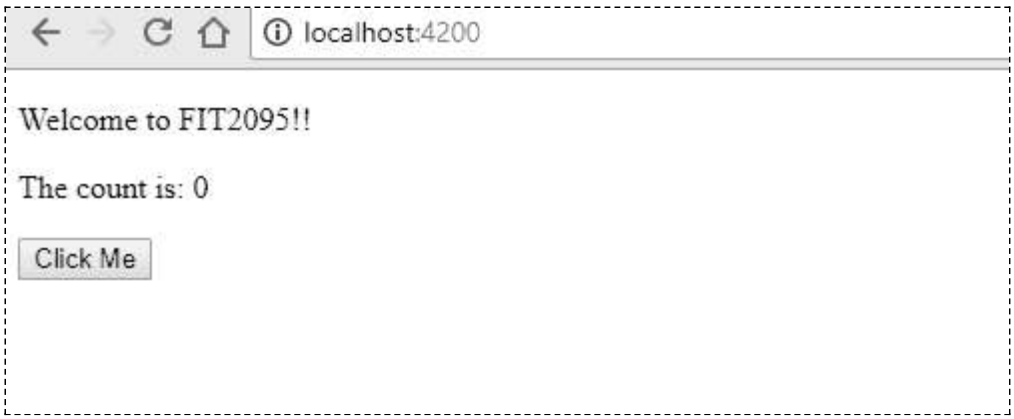
```
1.  import { Component } from '@angular/core';
2.
3.  @Component({
4.    selector: 'app-root',
5.    templateUrl: './app.component.html',
6.    styleUrls: ['./app.component.css']
7.  })
8.  export class AppComponent {
9.    title = 'FIT2095';
10.   counter: number = 0;
11.
12.   incCounter() {
13.     this.counter++;
14.   }
15. }
```

Line 10 declares the counter and sets its initial value to zero. Lines 12-14 declares a function '**incCounter**' that increments the value of '**counter**' by one.

open app.component.html and paste this code:

```
1.  <p>
2.    Welcome to {{title}}!!
3.  </p>
4.  <p>
5.    The count is: {{counter}}
6.  </p>
7.  <button (click)="incCounter()" >Click Me</button>
```

Line 5 has property binding as it displays the value of '**counter**'. Line 7 adds a button to the page and binds the 'click' event to '**incCounter**' function, which is declared in **app.component.ts**.



Let's add a reset button.

Change app.component.ts to be:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'FIT2095';
  counter: number = 0;

  incCounter() {
    this.counter++;
  }

  resetCounter(){
    this.counter=0;
  }
}
```

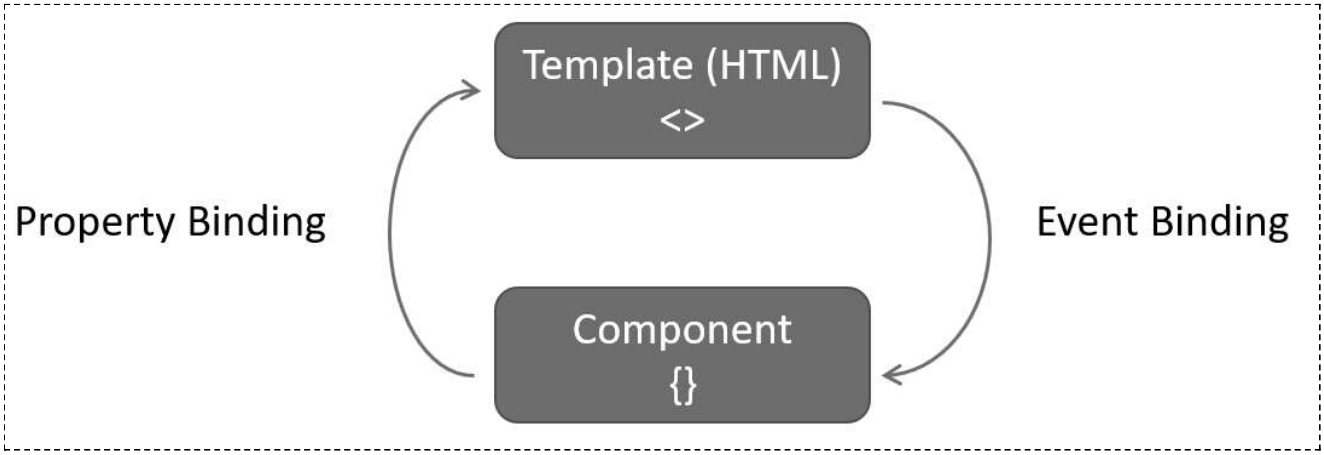
and app.component.html

```
<p>
  Welcome to {{title}}!!
</p>
```

```
4.  <p>
5.    The count is: {{counter}}
6.  </p>
7.
8.
9.  <button (click)="incCounter()">Click Me</button>
10.
11.
12. <button (click)="resetCounter()">Reset</button>
```

Each time the user hits the 'Reset' button, function 'resetCounter()' gets invoked and sets the counter back to zero.

the below picture visualizes the relationship between the template (HTML) and the component from data binding point of view.



Let's have another example. Add two buttons to a page to increase and decrease the font size of a '**div**' element.

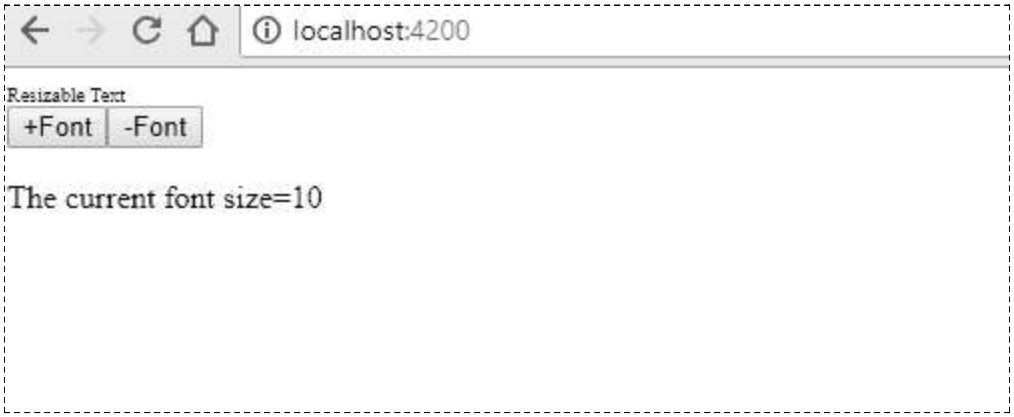
Here is app.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  size = 10;

  dec() {this.size--;}
  inc() {this.size++;}

}
```

The component has a variable called ‘size’ and two functions to increment and decrement its value.

app.component.html:

```
<div [style.font-size.px]="size">Resizable Text</div>
<button (click)="inc()">+Font</button>
<button (click)="dec()">-Font</button>
<p>
  The current font size={{size}}
</p>
```



Question: Can we listen to double-click events?

Answer: just replace (click) with (dblclick)

Question: How about Mouse Over events?

Answer: replace (click) with (mouseover)

Question: Where can I find the list of HTML DOM Events?

Answer: Here: https://www.w3schools.com/jsref/dom_obj_event.asp

Two-way Binding

In order too both display a data property and update that property when the user makes changes, two-way binding should be used.

On the element side that takes a combination of setting a specific element property and listening for an element change event.

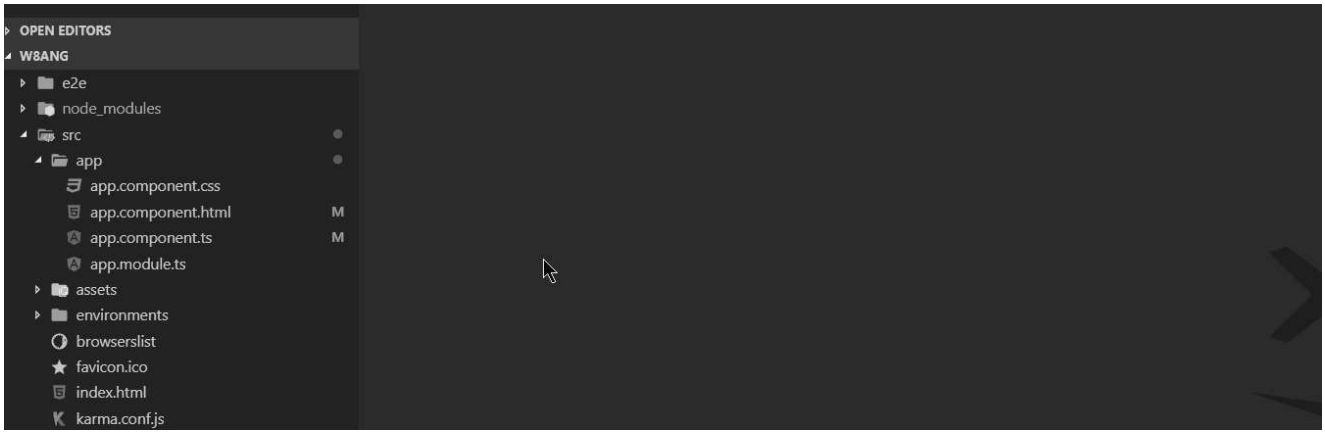
Angular offers a special two-way data binding syntax for this purpose, [(x)]. The [(x)] syntax combines the brackets of property binding, [x], with the parentheses of event binding, (x).

> [( )] = BANANA IN A BOX

When developing data entry forms, you often both display a data property and update that property when the user makes changes. Two-way data binding with the NgModel directive makes that easy. Here's an example:

```
<input [(ngModel)]="studentName">
```

Before using the ngModel directive in a two-way data binding, you must import the FormsModule and add it to the NgModule's imports list.



Now, let's have this example that implements two-way binding using the 'ngModel' directive

app.component.ts:

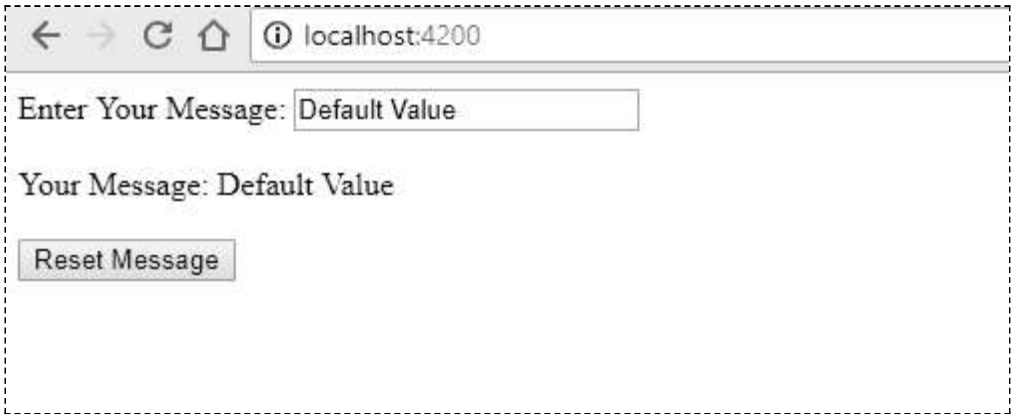
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  message = "Default Value";

  resetMsg() {
    this.message = "";
  }
}
```

app.component.html

```
Enter Your Message:
```

```
<input [(ngModel)]="message" />
<br>
<br>
<div>Your Message: {{message}}</div>
<br>
<button (click)="resetMsg()" >Reset Message </button>
```

The output:



How does it work?

Each time the user updates the input element (HTML line) -which is bound to 'message'- Angular sets the component variable 'message' with the new value and re-evaluates all the expressions that use 'message'.

Directive

Directive helps us to add behavior to the DOM elements.

Built-in structural directives

*ngIf

You can add or remove an element from the DOM by applying an NgIf directive to that element (called the host element). Bind the directive to a condition expression like isActive in this example.

```
<div *ngIf="isActive"></div>
```

The following example adds elements into an array, shows the number of elements if the array has at least one, and shows another message when the length reaches 5.

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  data:any = [];
  item = "";
  newItem() {
    this.data.push(this.item);
  }
  clearItems() {
    this.data = [];
  }
}
```

app.component.html

```
Enter Your Items:
<input [(ngModel)]="item" />
<br>

<button (click)="newItem()">Add Item</button>
<button (click)="clearItems()">Clear Items</button>
<br>
<div *ngIf="data.length>0">You have {{data.length}} items</div>
<div *ngIf="data.length==5">Caution:  5 items in your array</div>
<br>
```

Here is the output:

*ngFor

NgForOf is a repeater directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed. You tell Angular to use that block as a template for rendering each item in the list.

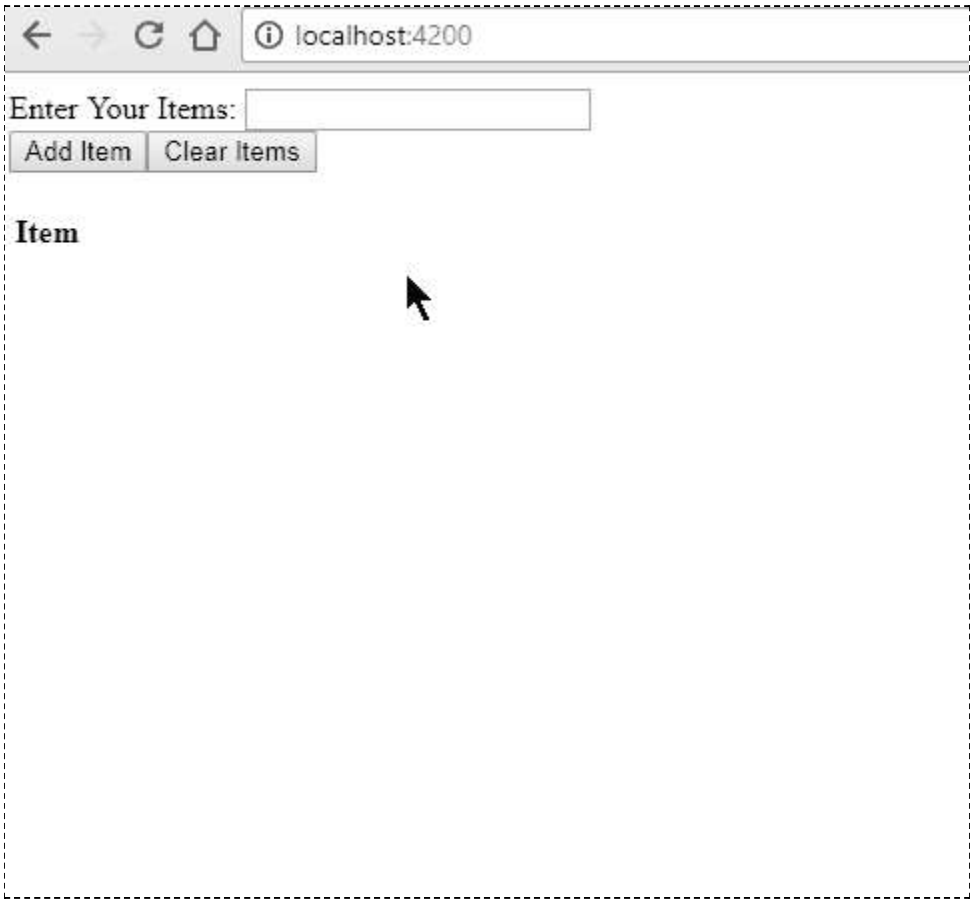
Example: Improve the above application to display the content of the array in a table.

Update app.component.html to be:

```
Enter Your Items:
<input [(ngModel)]="item" />
<br>

<button (click)="newItem()">Add Item</button>
<button (click)="clearItems()">Clear Items</button>
<br>
<div *ngIf="data.length>0">You have {{data.length}} items</div>
<div *ngIf="data.length==5">Caution: 5 items in your array</div>
<br>

<table>
  <tr>
    <th> Item </th>
```

```
    </tr>
    <tr *ngFor="let item of data">
      <td>{{item}}</td>
    </tr>
  </table>
```

Here is the output:

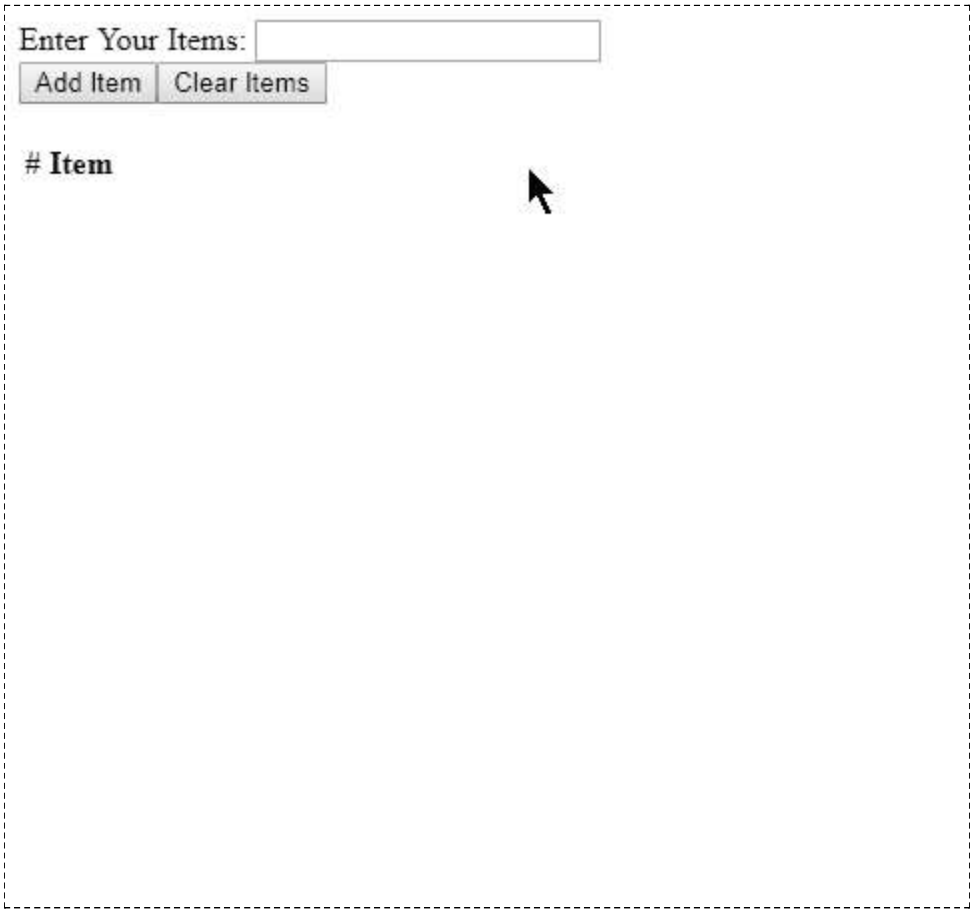


The following update adds the index of items as a column to the table:

```
Enter Your Items:
<input [(ngModel)]="item" />
<br>

<button (click)="newItem()">Add Item</button>
<button (click)="clearItems()">Clear Items</button>
<br>
<div *ngIf="data.length>0">You have {{data.length}} items</div>
<div *ngIf="data.length==5">Caution: 5 items in your array</div>
```

```
<br>

<table>
  <tr>
    <th> # </th>
    <th> Item </th>
  </tr>
  <tr *ngFor="let item of data; let i = index">
    <td>{{i+1}}</td>
    <td>{{item}}</td>
  </tr>
</table>
```

The output:



NgSwitch

NgSwitch is like the JavaScript switch statement. It can display one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

NgSwitch is actually a set of three, cooperating directives: NgSwitch, NgSwitchCase, and NgSwitchDefault as seen in this example.

Example:

Implement the following JavaScript Switch-case statement using Angular NgSwitch directive:

```
username="Tim"
switch(username) {
  case 'Tim':
      //set the button class to class="btn btn-primary"
      break;
  case 'Alex':
      //set the button class to class="btn btn-success"
      break;
  case 'John':
      //set the button class to class="btn btn-info"

  default:
  //set the button class to class="btn btn-danger"
}
```

app.component.html:

```
<div class="form-group">
  <label for="item">Enter Your Name:</label>
  <input class="form-control" [(ngModel)]="username" id="item" />
</div>
<br>

<div [ngSwitch]="username">
  <button *ngSwitchCase="'Tim'" class="btn btn-primary">{{username}}
</button>
```

```
  <button *ngSwitchCase="'Alex'" class="btn btn-success">{{username}}
</button>
  <button *ngSwitchCase="'John'" class="btn btn-info">{{username}}
</button>
  <button *ngSwitchDefault class="btn btn-danger">N/A</button>

</div>
```

and here is the output:

Adding Bootstrap to an Angular CLI project

Step1: Install bootsrap from NPM

```
npm install bootstrap
```

Step 2: Importing Bootstrap CSS

- open angular.json file and add the bootstrap.min.css reference to the list of styles as shown below:

```
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Step 3: Add bootstrap classes to your DOM elements. For example, to get a blue button, add “btn btn-primary” classes to a button’s styles.

```
<button class="btn btn-primary" (click)="newItem()">Add Item</button>
```

Update your app.component.html to be:

```
<div class="form-group">
  <label for="item">Enter Your Items:</label>
  <input class="form-control" [(ngModel)]="item" id="item" />
</div>
<br>

<button class="btn btn-primary" (click)="newItem()">Add Item</button>
<button class="btn btn-danger" (click)="clearItems()">Clear Items</button>
```

```
<br>
<div *ngIf="data.length>0">You have {{data.length}} items</div>
<div *ngIf="data.length==5">Caution: 5 items in your array</div>
<br>

<table class="table table-hover">
  <tr>
    <th> # </th>
    <th> Item </th>
  </tr>
  <tr *ngFor="let item of data; let i = index">
    <td>{{i+1}}</td>
    <td>{{item}}</td>
  </tr>
</table>
```

Note: a Non-trivial amount of text has been taken from Angular documentation and other resources.

References

1. [Book] Angular 5: From Theory To Practice: Build the web applications of tomorrow using the new Angular web framework from Google.
2. [Book] MEAN Web Development.
3. https://angular.io/docs
4. https://angular.io/guide/architecture
5. https://www.w3schools.com/angular/
6. https://dzone.com/articles/components-of-angular2-architecture
7. https://www.typescriptlang.org/docs/home.html
8. https://www.tutorialspoint.com/typescript/typescript_overview.htm

---

MODULE

Week 9: Angular II

Nawfal Ali
Updated 16 September 2021

Angular II

Topics of this week:

- Angular Services
- Dependency Injection
- Create a component
- HTTP CRUD operation through Angular
- Serving Angular UI using Express

Angular Services

> A service is typically a class with a narrow, well-defined purpose. A Service is a broad category encompassing any value, function, or feature that an app needs.
>
> Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.
>
> A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console.

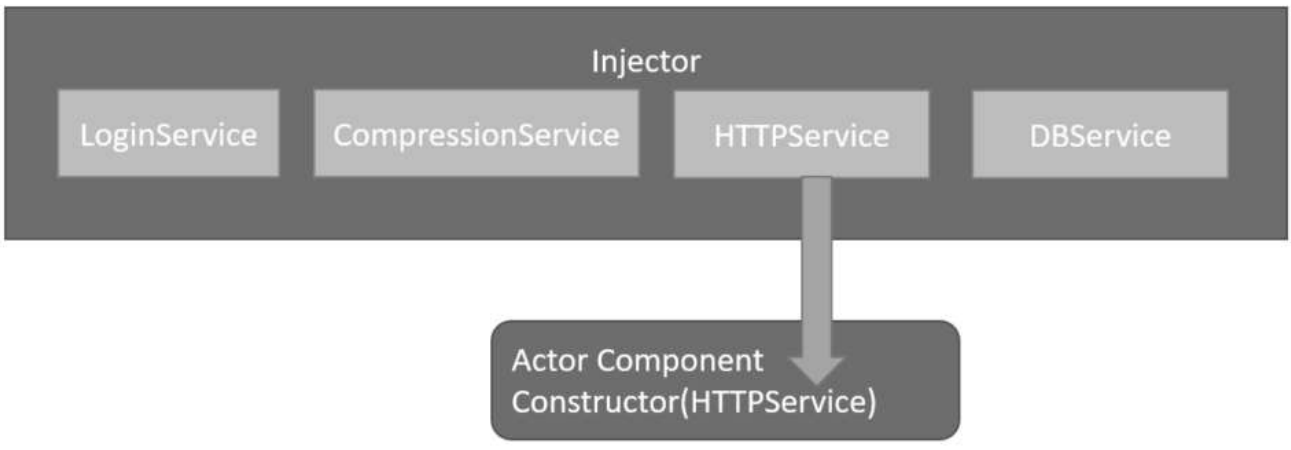
Srouce: https://angular.io/guide/architecture-services

Dependency injection (DI)

> Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested. DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.
>
> When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of **ActorComponent** needs **HTTPService**.
>
> When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider, and adds it to the injector before returning the service to Angular.

sources:

- https://angular.io/guide/architecture-services
- https://docs.angularjs.org/guide/di



Application

Using Angular, develop a frontend for the RESTFul application that we developed in week 7 (https://www.alexandriarepository.org/module/week-7-create-a-restful-application/)

The front end has to provide the following CRUD operations:

- Add new Actor
- Delete actor
- Update actor
- Retrieve all actors

Create an Angular application using the ng CLI:

```
1. ng new movieAng
```

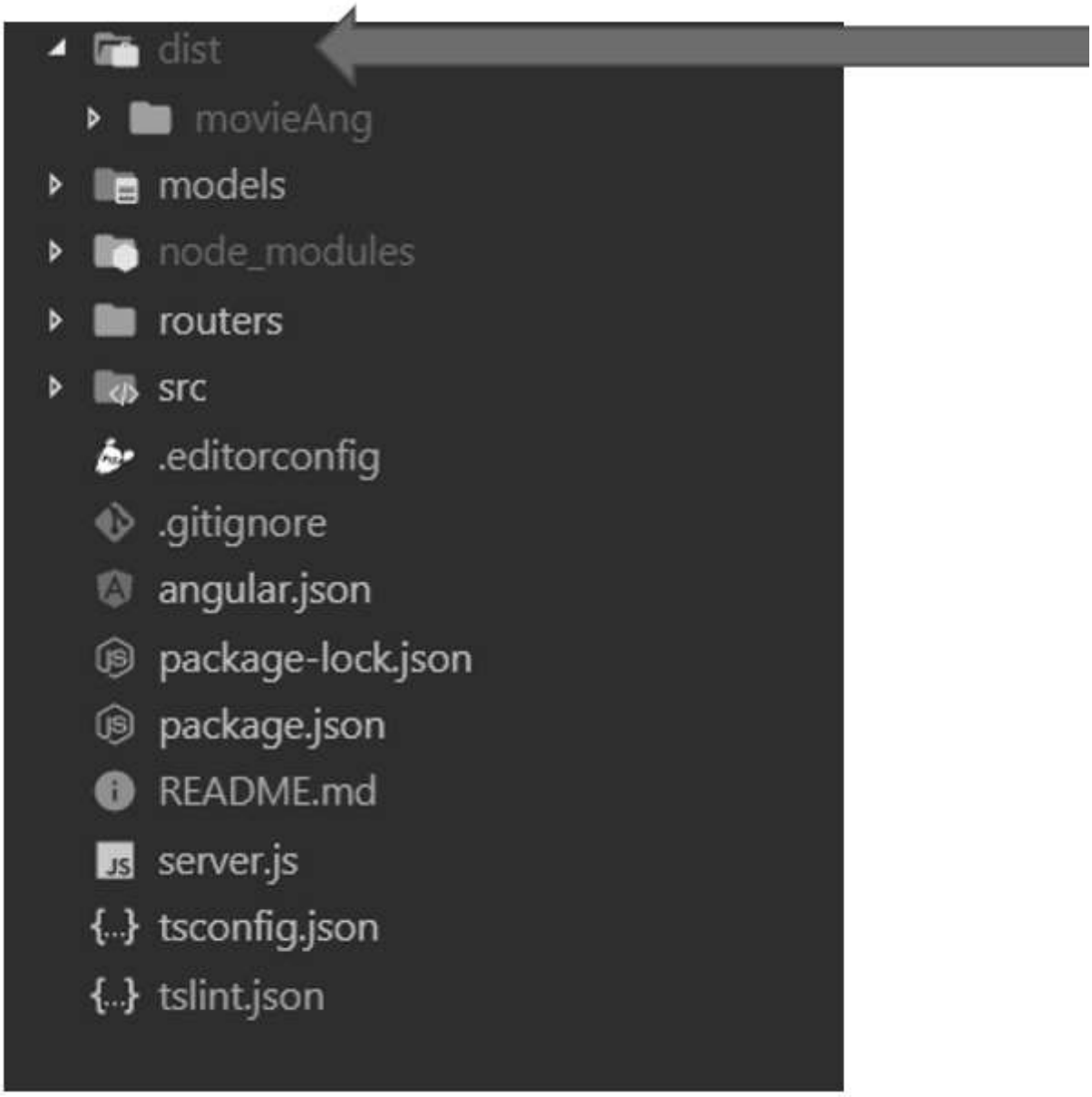
open the new project using VSCode.

from week 7, copy **models** and **routers** folders and **server.js** and paste them in the root of **movieAng** folder. The project structure should be:

The project will be served using node HTTP server, therefore, we need to build the Angular project each time a file at the client side gets changes. To build, open your terminal and run:

```
1. ng build
```

This command compiles your Angular project and generates new content (new HTML and JS files) in a folder named "dist" located at the root of your project.



Now, you have to use the 'dist' folder as a source for the client UI. In other words, any request other than the CRUD operation should be redirected to this folder. To do this, add this line to app.js

```
1. app.use("/", express.static(path.join(__dirname, "dist/movieAng")));
```

> The Path module provides a way of working with directories and file paths. To add it to your app:
>
> let path = require('path');

Now, start your server and navigate to **http://localhost:8080** to get the default Angular page.

Adding New Component

In order to add a new component, we will use the Angular CLI (ng) '**ng generate component [name]**' to generate all the default files, folders, and settings. In your terminal, run:

```
ng generate component Actor
```

or

```
ng g c Actor
```

where 'Actor' is the name of the new component. In the project structure, you will find a new folder named 'actor' contains the default files for the component:

- actor.component.html: the template in HTML format and its initial code will be:

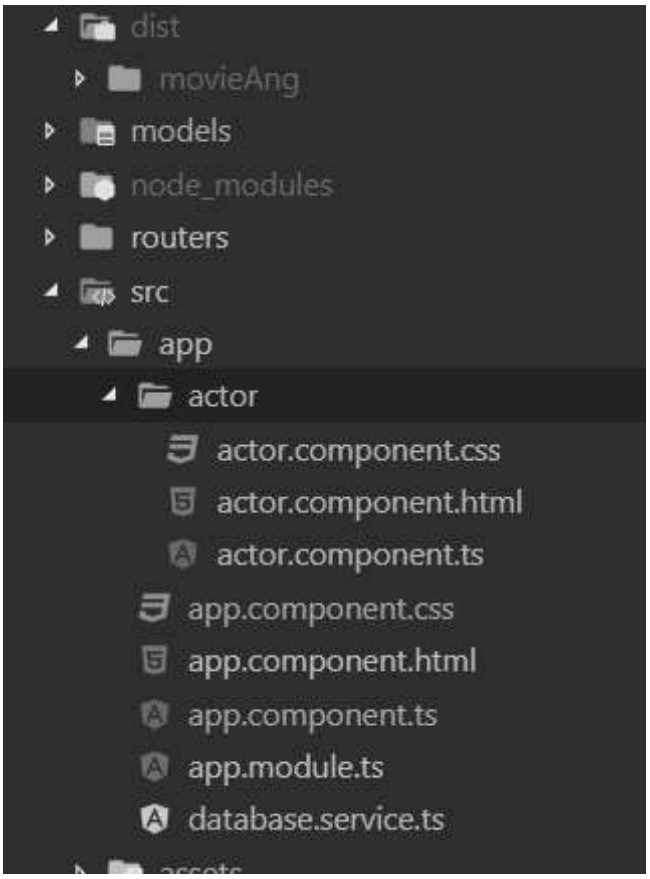
```
<p>
  actor works!
</p>
```

- actor.component.css: the style sheet for the template
- actor.component.ts: where the business logic is coded in a class named 'ActorComponent'

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

@Component({
  selector: 'app-actor',
  templateUrl: './actor.component.html',
  styleUrls: ['./actor.component.css']
})
export class ActorComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

> The new component comes with a selector (i.e. tag) '<app-actor></app-actor>' which will be substite by the template later.

- actor.componet.specs.ts: for unit testing (I have deleted this file)



You will also notice that the **ActorComponent** class has been imported and added to **app.module.ts** automatically by the ng command.

Now, replace the **app.component.html** code with the selector (i.e. tag) of the actor component.

```
1. <app-actor></app-actor>
```

> index.html and the UI seen is rendered from app.component.html.

By doing so, you are instantiating a new instance of Actor Component and placing its HTML (template) inside app.component.html.

To run the app, we have to build Angular's component first. In order to automate the process of building the Angular project with each change of its files, this command should be used in a new terminal:

```
ng build --watch
```

> It is possible to split the VSCode terminal into two sessions by using: Ctrl+'\'

In the first terminal run the server by:

```
node server.js
```

Navigate to http://localhost:8080 to get 'actor works! '

Adding a new Service

An angular service is an object that gets instantiated only once during the lifetime of an application. It contains methods that maintain data throughout the life of an application. The main benefit of having a service is to organize and share methods, models, or data with different components of an Angular application.

We will add a service that is responsible for the communication with the RESTFul server.

Again, the CLI command will be used to generate a new service:

```
ng generate service database
```

or

```
ng g s database
```

A new file 'database.service.ts' has been added to the project and it contains:

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

@Injectable({
  providedIn: 'root'
})
export class DatabaseService {

```

```
8.    constructor() { }
9.  }
```

'@Injectable' is a marker metadata that marks a class as available to Injector for creation. In order to make this service available to the whole application, we will keep the default value of the property 'proividedIn' as root. But, if we need to be allocated to a certain module, then we need to update it. All the functionalities that will be provided by this service will be coded inside the class **DatabaseService**.

The constructor is used to provide references to dependencies and can also be used to initialize the process of the service.

The last step is to tell the app module that this new service should be available application-wide. To do so, add '**DatabaseService**' to the list of providers in **app.module.ts**.

```
1.  providers: [DatabaseService],
```

Dependency Injection

The database service depends on the HTTP service in its communications with the RESTFul servers. To achieve that:

import the Angular HttpClientModule in the root AppModule app.module.ts.

```
1.  imports: [BrowserModule, FormsModule, HttpClientModule]
```

and add this line to list of imports at the list of imports

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

where **BrowserModule** exports required infrastructure for all Angular apps and **FormsModule e**xports the required providers and directives for template-driven forms (i.e. it is required by the two-way binding directive ngModel)

open database.service.ts and import both **HttpClient** and **HttpHeaders** from angular http package:

```
1.  import { HttpClient, HttpHeaders } from "@angular/common/http";
```

In the service's constructor, add a reference to the **httpClient**:

```
constructor(private http: HttpClient) {}
```

By doing so, Angular will inject the httpClient to our service and make it available with reference **http**. Now, any method inside the service will be able to make http requests by calling **this.http** (for instance **this.http.get** for a GET request).

GET Request

In order to fetch all the actors, we need to send a GET http request to '/actors'

> '/actors' is relative to the server's address that servers the client side

```
getActors() {
    return this.http.get("/actors");
 }
```

getActors is a function that sends a GET request and returns the array of actors.

to retrieve one actor by its ID, we have to send a GET request with URL '/actors/id' :

```
getActor(id: string) {
  let url = "/actors/" + id;
  return this.http.get(url);
}
```

POST Request

POST requests are required to create new documents in the database.

```
createActor(data) {
  return this.http.post("/actors", data, httpOptions);
}
```

where **data** is an object contains the actor's details that are collected from the UI elements and **httpOptions** is an object that specifies a set of options for the request such as the format of the body (JSON in our case).

```
const httpOptions = {
  headers: new HttpHeaders({ "Content-Type": "application/json" }),
};
```

PUT Request

PUT requests are used to update a document already exist in the database. The following function accepts as input two parameters, the id of the document and the update.

```
updateActor(id, data) {
  let url = "/actors/" + id;
  return this.http.put(url, data, httpOptions);
}
```

DELETE Request

To send a delete request, an id is required:

```
deleteActor(id) {
  let url = "/actors/" + id;
  return this.http.delete(url, httpOptions);
}
```

So, database.service.ts in one piece:

```
import { Injectable } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
const httpOptions = {
  headers: new HttpHeaders({ "Content-Type": "application/json" }),
```

```
};

@Injectable({
  providedIn: "root",
})
export class DatabaseService {
  constructor(private http: HttpClient) {}
  result: any;

  getActors() {
    return this.http.get("/actors");
  }
  getActor(id: string) {
    let url = "/actors/" + id;
    return this.http.get(url);
  }
  createActor(data) {
    return this.http.post("/actors", data, httpOptions);
  }
  updateActor(id, data) {
    let url = "/actors/" + id;
    return this.http.put(url, data, httpOptions);
  }
  deleteActor(id) {
    let url = "/actors/" + id;
    return this.http.delete(url, httpOptions);
  }
}
```

Observables and Subscribe

In order to handle asynchronous operation, Angular uses observables.

So, what are observables?

Observables are declarative—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

To execute the observable and begin receiving notifications, you call its subscribe() method, passing an observer (i.e. callback function). This is a JavaScript object that defines the handlers for the notifications you receive.

Therefore, all the http functions (**get(),put(),post(), and delete()**) return an observable output and our component has to call their **subsucribe()** methods to get them executed.

Source and More Details: https://angular.io/guide/observables

Inject the database service to the Actor Component

As the actor component needs to perform the CRUD operations with the RESTFul server, we have to provide this service to it.

First, import the service:

```
1. import { DatabaseService } from "../database.service";
```

Second, inject. Similar to injecting the HTTP service to the database service, we will use the constructor to provide a reference to the database service :

```
1. constructor(private dbService: DatabaseService) {}
```

Angular will create a new instance of DatabaseService (if not available) and inject it under the name ‘dbService’.

Managing Different UI Sections

Similar to the previous week, we will create a set of div sections that are controlled by ‘*ngIf’ directive to only show one section at a time but using an integer variable instead of boolean.

The integer variable is called ‘section’ and it is declared in the actor.component.ts:

```
1. section = 1;
```

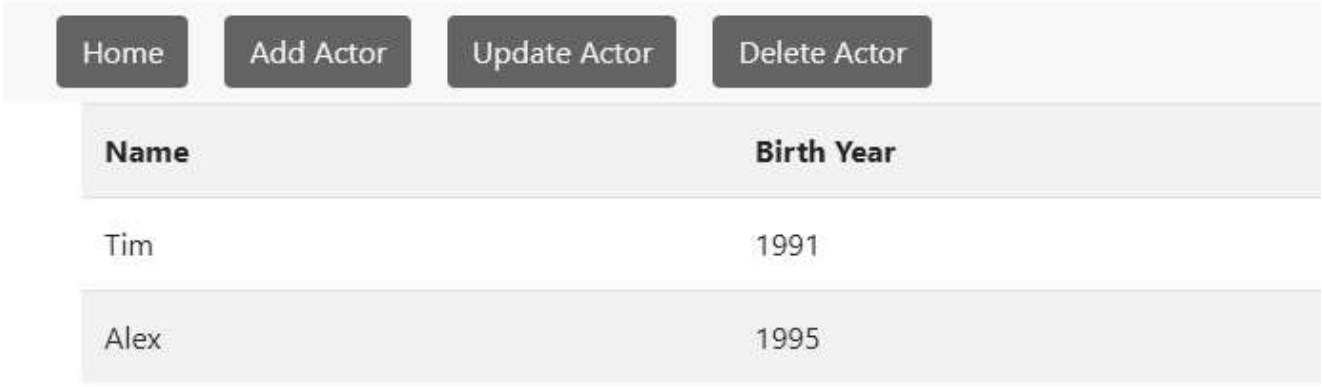
To change its value, a function named '**changeSection(id)**' is developed. This function is invoked by four buttons represent the navigation bar of the application. Each button will send to the '**changeSection()**' function a unique number between 1..4 represents the section we intend to make visible.

```
changeSection(sectionId) {
  this.section = sectionId;
}
```

and the navigation bar is:

```
<nav class="navbar navbar-expand-sm bg-light">
  <ul class="navbar-nav">
    <li class="nav-item" style="padding-left:0.2in">
      <a class="btn btn-primary" (click)="changeSection(1)">Home</a>
    </li>
    <li class="nav-item" style="padding-left:0.2in">
      <a class="btn btn-primary" (click)="changeSection(2)">Add Actor</a>
    </li>
    <li class="nav-item" style="padding-left:0.2in">
      <a class="btn btn-primary" (click)="changeSection(3)">Update Actor</a>
    </li>
    <li class="nav-item" style="padding-left:0.2in">
      <a class="btn btn-primary" (click)="changeSection(4)">Delete Actor</a>
    </li>
  </ul>
</nav>
```

Source: https://www.w3schools.com/bootstrap4/bootstrap_navbar.asp



The first section is to display all the actors in a table format. Open actor.component.html and paste this code:

```
<div class="section" *ngIf="section==1">
  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
    </tr>
  </table>
</div>
```

This section will be visible if the value of the variable '**section**' is equal to 1. '**actorsDB'** is an array that is declared in the **actor.component.ts** and contains all the actors retrieved by the GET request.

in the actor.component.ts:

```
actorsDB: any[] = [];
```

to get all the actors and save them into **actorsDB**, this function will invoke **getActors** function which is developed earlier in the database service:

```
onGetActors() {
```

```
    this.dbService.getActors().subscribe((data: any[]) => {
      this.actorsDB = data;
    });
  }
```

> the output of getActors is observable and we have to subscribe to get the method executed. We have to provide a function that represents the handler of the output of method getActors()

The second section will be responsible for creating a new actor and its section value is equal to 2 (visible if section=2).

```
<div class="section" *ngIf="section==2">
  <div class="form-group">
    <label for="actorName">Full Name</label>
    <input type="text" class="form-control" id="actorName"
[(ngModel)]="fullName">
  </div>
  <div class="form-group">
    <label for="actorName">Birth Year</label>
    <input type="number" class="form-control" id="actorName"
[(ngModel)]="bYear">
  </div>
  <button type="submit" class="btn btn-primary"
(click)="onSaveActor()">Save Actor</button>
</div>
```

This section contains two two-way binding variables: **fullName** and **bYear**, which are declared in the **actor.component.ts** as string and number respectively.

```
fullName: string = "";
bYear: number = 0;
```

It also has a button that invokes a method named ‘onSaveActor()’, which will be responsible for calling the database service function ‘createActor()’

```
onSaveActor() {
    let obj = { name: this.fullName, bYear: this.bYear };
```

```
    this.dbService.createActor(obj).subscribe(result => {
      this.onGetActors();
    });
  }
```

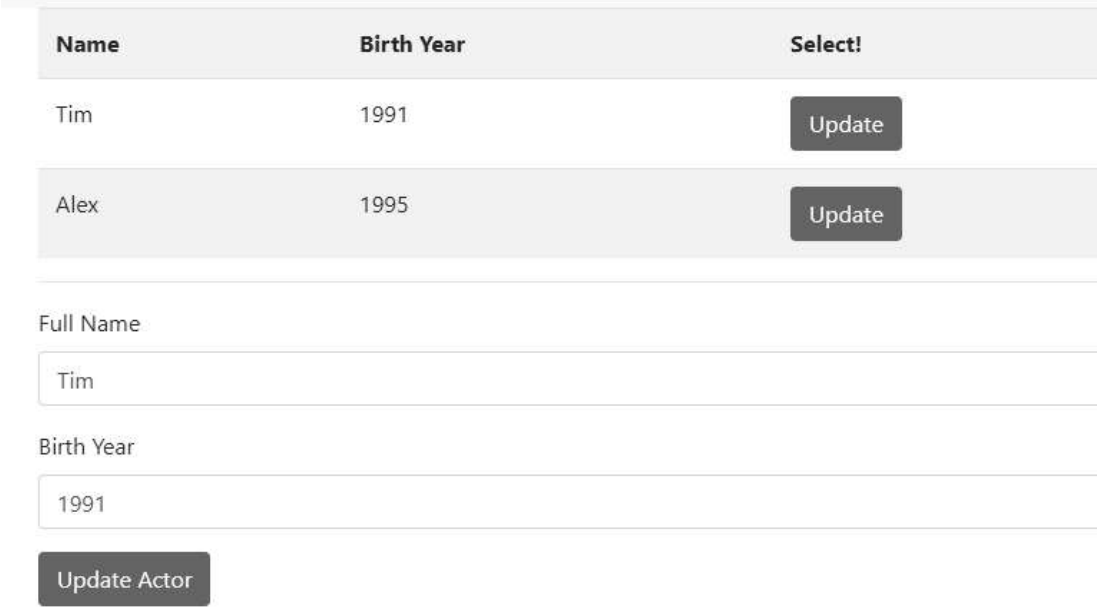
After adding the new actor, the function calls '**onGetActors()**' to update the list of actors in the array **actorsDB.**

The third section, which will be visible of the value of section is equal to 3, is the update section.

```
<div class="section" *ngIf="section==3">
  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
      <th>Select!</th>
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
      <td><button type="submit" class="btn btn-primary"
(click)="onSelectUpdate(item)">Update</button></td>
    </tr>
  </table>
  <hr>
  <div *ngIf="actorsDB.length>0">
    <div class="form-group">
      <label for="actorName">Full Name</label>
      <input type="text" class="form-control" id="actorName"
[(ngModel)]="fullName">
    </div>
    <div class="form-group">
      <label for="actorName">Birth Year</label>
      <input type="number" class="form-control" id="actorName"
[(ngModel)]="bYear">
    </div>
```

```
    <button type="submit" class="btn btn-primary"
(click)="onUpdateActor()">Update Actor</button>
  </div>
</div>
```

This section shows the list of actors in a table with a button to select the actor for the update.



The '**update**' button invokes a method named '**onSelectUpdate**' that takes as input the selected actor object and extracts its details into global variables to be used later by the **onUpdateActor** method.

```
onSelectUpdate(item) {
  this.fullName = item.name;
  this.bYear = item.bYear;
  this.actorId = item._id;
}
onUpdateActor() {
  let obj = { name: this.fullName, bYear: this.bYear };
  this.dbService.updateActor(this.actorId, obj).subscribe(result => {
    this.onGetActors();
  });
}
```

The last section is the delete actor section which will be visible if the value of the variable **section** is equal to 4.

```
<div class="section" *ngIf="section==4">
  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
      <th>Delete?</th>
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
      <td><button type="submit" class="btn btn-primary"
(click)="onDeleteActor(item)">Delete</button></td>
    </tr>
  </table>
</div>
```

It shows the list of available actors in a table with a button for each one to invoke a method named '**onDeleteActor**()'.

```
onDeleteActor(item) {
  this.dbService.deleteActor(item._id).subscribe(result => {
    this.onGetActors();
  });
}
```

Name	Birth Year	Delete?
Tim	1991	Delete
Alex	1995	Delete

In order to reset the input fields after each operation, a ‘resetValue’ method is developed:

```
resetValues() {
  this.fullName = "";
  this.bYear = 0;
  this.actorId = "";
}
```

So, the actor.component.ts in one piece:

```
import { Component, OnInit } from "@angular/core";
import { DatabaseService } from "../database.service";

@Component({
  selector: "app-actor",
  templateUrl: "./actor.component.html",
  styleUrls: ["./actor.component.css"],
})
export class ActorComponent implements OnInit {
  actorsDB: any[] = [];

  section = 1;

  fullName: string = "";
  bYear: number = 0;
  actorId: string = "";

  constructor(private dbService: DatabaseService) {}

  //Get all Actors
  onGetActors() {
    this.dbService.getActors().subscribe((data: any[]) => {
      this.actorsDB = data;
    });
```

```
25.   }
26.   //Create a new Actor, POST request
27.   onSaveActor() {
28.     let obj = { name: this.fullName, bYear: this.bYear };
29.     this.dbService.createActor(obj).subscribe(result => {
30.       this.onGetActors();
31.     });
32.   }
33.   // Update an Actor
34.   onSelectUpdate(item) {
35.     this.fullName = item.name;
36.     this.bYear = item.bYear;
37.     this.actorId = item._id;
38.   }
39.   onUpdateActor() {
40.     let obj = { name: this.fullName, bYear: this.bYear };
41.     this.dbService.updateActor(this.actorId, obj).subscribe(result =>
    {
42.       this.onGetActors();
43.     });
44.   }
45.
46.   //Delete Actor
47.   onDeleteActor(item) {
48.     this.dbService.deleteActor(item._id).subscribe(result => {
49.       this.onGetActors();
50.     });
51.   }
52.   // This lifecycle callback function will be invoked with the
    component get initialized by Angular.
53.   ngOnInit() {
54.     this.onGetActors();
55.   }
56.
57.   changeSection(sectionId) {
58.     this.section = sectionId;
```

```
59.       this.resetValues();
60.     }
61.
62.     resetValues() {
63.       this.fullName = "";
64.       this.bYear = 0;
65.       this.actorId = "";
66.     }
67.   }
```

and actor.component.html in one piece:

```
1.  <div class="jumbotron text-center">
2.    <h1>Actors DB System</h1>
3.  </div>
4.
5.  <!-- The navigation bar -->
6.  <nav class="navbar navbar-expand-sm bg-light">
7.    <ul class="navbar-nav">
8.      <li class="nav-item" style="padding-left:0.2in">
9.        <a class="btn btn-primary" (click)="changeSection(1)">Home</a>
10.     </li>
11.     <li class="nav-item" style="padding-left:0.2in">
12.       <a class="btn btn-primary" (click)="changeSection(2)">Add
    Actor</a>
13.     </li>
14.     <li class="nav-item" style="padding-left:0.2in">
15.       <a class="btn btn-primary" (click)="changeSection(3)">Update
    Actor</a>
16.     </li>
17.     <li class="nav-item" style="padding-left:0.2in">
18.       <a class="btn btn-primary" (click)="changeSection(4)">Delete
    Actor</a>
19.     </li>
20.   </ul>
```

```
21. </nav>
22.
23. <!-- First section: Display all in a tble -->
24. <div class="section" *ngIf="section==1">
25.   <table class="table table-striped">
26.     <tr>
27.       <th>Name</th>
28.       <th>Birth Year</th>
29.     </tr>
30.     <tr *ngFor="let item of actorsDB">
31.       <td>{{item.name}}</td>
32.       <td>{{item.bYear}}</td>
33.     </tr>
34.   </table>
35. </div>
36.
37. <!-- Second Section: Add new actor -->
38. <div class="section" *ngIf="section==2">
39.   <div class="form-group">
40.     <label for="actorName">Full Name</label>
41.     <input type="text" class="form-control" id="actorName"
    [(ngModel)]="fullName">
42.   </div>
43.   <div class="form-group">
44.     <label for="actorName">Birth Year</label>
45.     <input type="number" class="form-control" id="actorName"
    [(ngModel)]="bYear">
46.   </div>
47.   <button type="submit" class="btn btn-primary"
    (click)="onSaveActor()">Save Actor</button>
48. </div>
49.
50. <!-- Section 3: Update actor -->
51.
52. <div class="section" *ngIf="section==3">
53.   <table class="table table-striped">
```

```
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
      <th>Select!</th>
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
      <td><button type="submit" class="btn btn-primary"
(click)="onSelectUpdate(item)">Update</button></td>
    </tr>
  </table>
  <hr>
  <div *ngIf="actorsDB.length>0">
    <div class="form-group">
      <label for="actorName">Full Name</label>
      <input type="text" class="form-control" id="actorName"
[(ngModel)]="fullName">
    </div>
    <div class="form-group">
      <label for="actorName">Birth Year</label>
      <input type="number" class="form-control" id="actorName"
[(ngModel)]="bYear">
    </div>
    <button type="submit" class="btn btn-primary"
(click)="onUpdateActor()">Update Actor</button>
  </div>
</div>

<!-- Section 4: Delete Actor -->
<div class="section" *ngIf="section==4">
  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
      <th>Delete?</th>
```

```
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
      <td><button type="submit" class="btn btn-primary"
(click)="onDeleteActor(item)">Delete</button></td>
    </tr>
  </table>
</div>
```

and app.module.ts

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { AppComponent } from "./app.component";
import { ActorComponent } from "./actor/actor.component";
import { DatabaseService } from "./database.service";
import { HttpClientModule } from "@angular/common/http";
import { FormsModule } from "@angular/forms";

@NgModule({
  declarations: [AppComponent, ActorComponent],
  imports: [BrowserModule, HttpClientModule, FormsModule],
  providers: [DatabaseService],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

The app:

MODULE

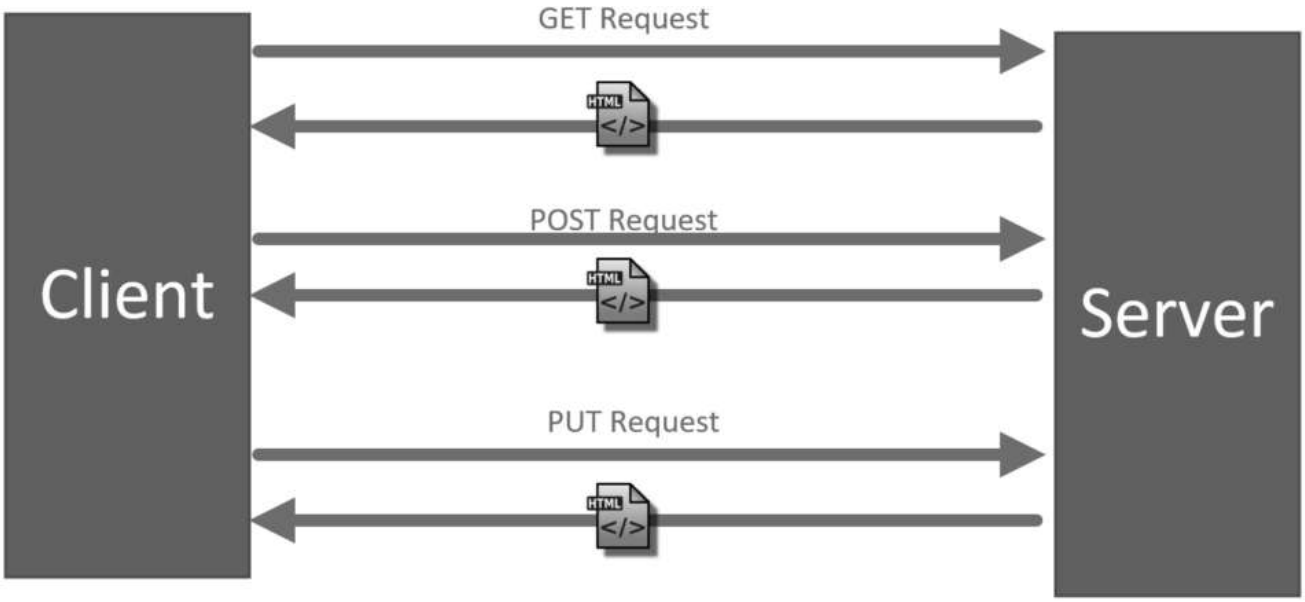
Week 10: Single Page Application Using MEAN Stack

Nawfal Ali
Updated 6 October 2021

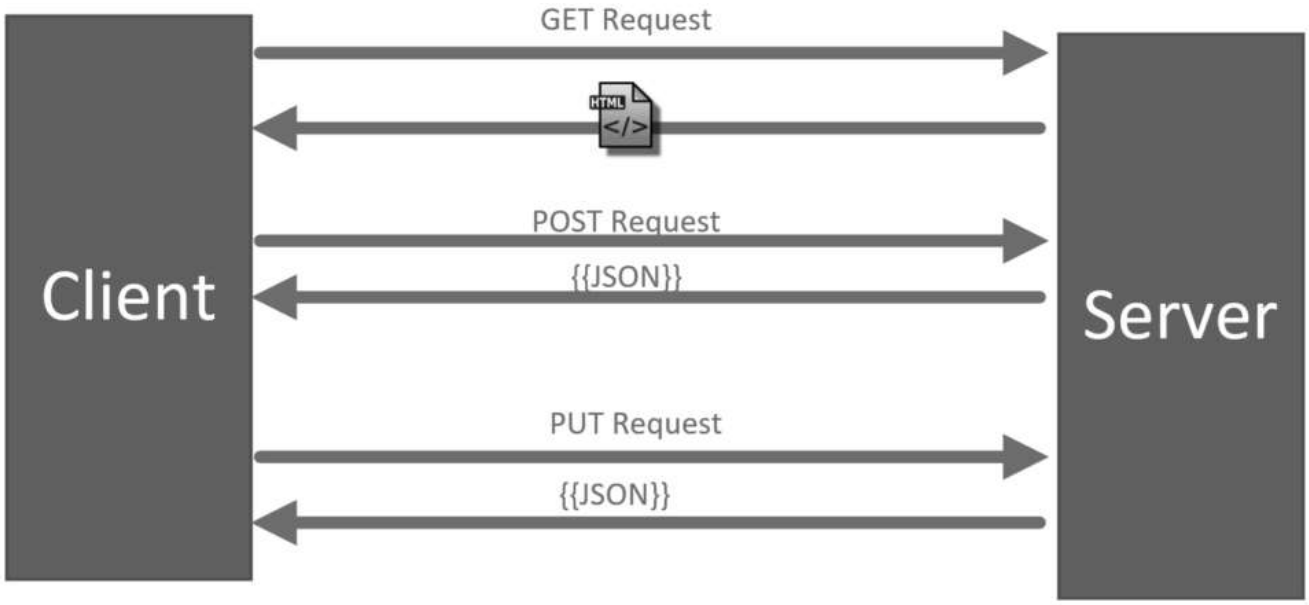
Single Page Application

Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

In other words, in Multi-Page Applications (MPAs), the server responds with HTML files with each request. The client (such as browsers) has to render the whole page again. Sending a new HTML file might also include its static assets such as CSS, JS, png files.



In the case of SPAs, clients load a single HTML that represents most of the web app and communicate with their servers via HTTP requests.



Disadvantages of Single Page Applications

- History: Browsers such as Chrome store history so that web pages load quickly when users hit the back button. The back button takes you the previous page, not the previous state (Component) of your app.
- Loading CSS and JS: SPAs might get higher booting time as they have to load most of the assets in advance.
- Analytics: Analytics tools track page views by default, not components or app states.
- Security: Part of the web app engine will be on the client side.

Routing & Navigation [1,2]

The Angular Router enables navigation from one view to the next as users perform application tasks.

In most web applications, users navigate from one page to the next as they perform application tasks. Users can navigate in these ways:

- Entering a URL in the address bar Following links,
- clicking buttons,
- and so on Going backward or forward in the browser history

In Angular applications, users can navigate in the same three ways but they're navigating through components (the building blocks of Angular apps). We can navigate because we have the Angular router. The router can interpret a browser URL as an instruction to navigate to a component and pass optional parameters (which contain information) to the component to give it contextual information and help it decide which specific content to present or what it needs to do. We can bind the router to links on a page, and it will navigate to the appropriate component when the user clicks a link. We can navigate imperatively when the user clicks a button, selects from a drop-down, or responds to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

Router imports

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, @angular/router. Import what you need from it as you would from any other Angular package.

```
1. import { RouterModule, Routes } from '@angular/router';
```

RouterConfiguration

A routed Angular application has one singleton instance of the Router service. When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.

A router has no routes until you configure it. The following example creates four route definitions, configures the router via the RouterModule.forRoot method, and adds the result to the AppModule's imports array.

```
1. const appRoutes: Routes = [
2.   { path: "listactors", component: ListactorsComponent },
3.   { path: "addactor", component: AddactorComponent },
4.   { path: "updateactor", component: UpdateactorComponent },
5.   { path: "deleteactor", component: DeleteactorComponent },
6.   { path: "", redirectTo: "/listactors", pathMatch: "full" },
7. ];
```

> The appRoutes array of routes describes how to navigate. Pass it to the **RouterModule.forRoot** method in the module imports to configure the router.
>
> Each Route maps a URL path to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, allowing you to use both relative and absolute paths when navigating between application views.
>
> The empty path in the fifth route represents the default path for the application, the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the /listactors URL and, therefore, will display the ListactorsComponent.
>
> It is possible to add ** path which is a wildcard. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration. This is useful for displaying a "404 – Not Found" page or redirecting to another route.
>
> The order of the routes in the configuration matters and this is by design. The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. In the configuration above, routes with a static path are listed first, followed by an empty path route, that matches the default route. The wildcard route comes last because it matches every URL and should be selected only if no other routes are matched first.

Router outlet

Given this configuration, when the browser URL for this application becomes /listactors, the router matches that URL to the route path /listactors and displays the ListactorsComponent after a RouterOutlet that you've placed in the host view's HTML.

```
<router-outlet></router-outlet>
<!-- Routed views go here -->
```

Router links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result

of some user action such as the click of an anchor tag.

Consider the following template:

```
<a class="nav-link" routerLink="/listactors"
routerLinkActive="active">List </a>
```

> The RouterLink directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the routerLink (a "one-time" binding).
>
> Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters (the link parameters array). The router resolves that array into a complete URL.
>
> The RouterLinkActive directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route. The router adds the active CSS class to the element when the associated RouterLink becomes active. You can add this directive to the anchor or to its parent element.

Base Ref

Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

As the app folder is the application root, set the href value exactly to '/'. This line should be added to /src/index.html

```
<base href="/">
```

More details on Routing and Navigation in Angular: https://angular.io/guide/router

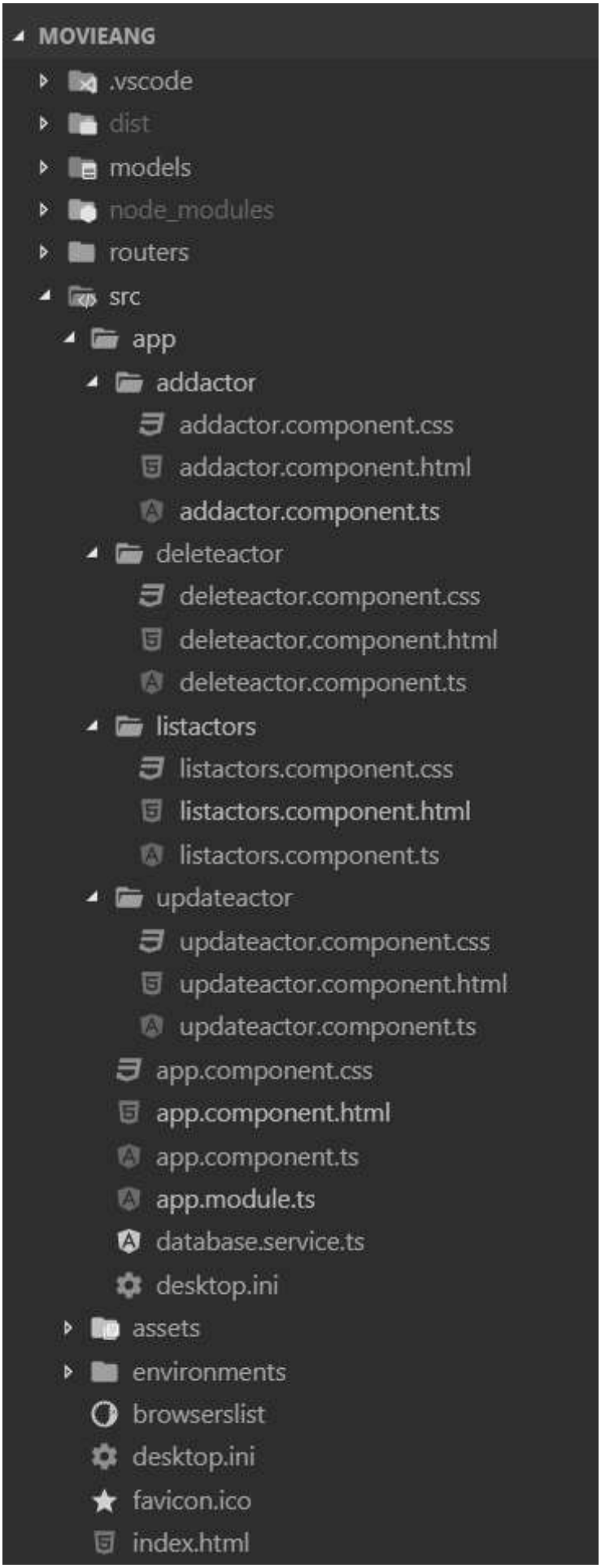
Now, let's reimplement the movie database application to be a multi-component app that is controlled by a router.

We will add four components that represent the four CRUD operations. These components work as child components to the App-root component as depicted below:

To add the four components using Angular CLI

```
1. ng generate component listactors
2. ng generate component addactor
3. ng generate component updateactor
4. ng generate component deleteactor
```

The project structure will be:



List Actors Component

This component will be responsible for fetching and displaying all the actors in the database.

listactors.component.html

```
1.  <div class="section">
2.    <table class="table table-striped">
3.      <tr>
4.        <th>Name</th>
5.        <th>Birth Year</th>
6.      </tr>
7.      <tr *ngFor="let item of actorsDB">
8.        <td>{{item.name}}</td>
9.        <td>{{item.bYear}}</td>
10.     </tr>
11.   </table>
12. </div>
```

> Line 7 iterates through all items in the array actorsDB. It generates a new <tr> for each item. Lines 8 and 9 use string interpolation {{}} to display the values of name and bYear respectively.

The backend logic of the component is coded in a file named: listactors.component.ts

```
1.  import { Component, OnInit } from "@angular/core";
2.  import { DatabaseService } from "../database.service";
3.
4.  @Component({
5.    selector: "app-listactors",
6.    templateUrl: "./listactors.component.html",
7.    styleUrls: ["./listactors.component.css"],
8.  })
9.  export class ListactorsComponent implements OnInit {
10.   actorsDB: any[] = [];
11.
12.   constructor(private dbService: DatabaseService) {}
13.
14.   ngOnInit() {
```

```
15.       console.log("Hi From ListActors ngIOnit");
16.
17.       this.dbService.getActors().subscribe((data: any[]) => {
18.         this.actorsDB = data;
19.       });
20.     }
21.   }
```

> Line 7, which is the constructor of the component, registers the need for a dependency which is the DatabaseService. The callback function ngOnInit which will be invoked by Angular as part of the component's lifecycle, uses the dbService to fetch all the actors from the database.

Here is output listactors component looks like:

Add Actor Component

Add actor component has two input fields and a button which is linked to a function named onSaveActor (addactor.component.ts@line17).

```
1.  <div class="section">
2.    <div class="form-group">
3.      <label for="actorName">Full Name</label>
4.      <input type="text" class="form-control" id="actorName"
    [(ngModel)]="fullName">
```

```
  </div>
  <div class="form-group">
    <label for="actorName">Birth Year</label>
    <input type="number" class="form-control" id="actorName"
[(ngModel)]="bYear">
  </div>
  <button type="submit" class="btn btn-primary"
(click)="onSaveActor()">Save Actor</button>
</div>
```

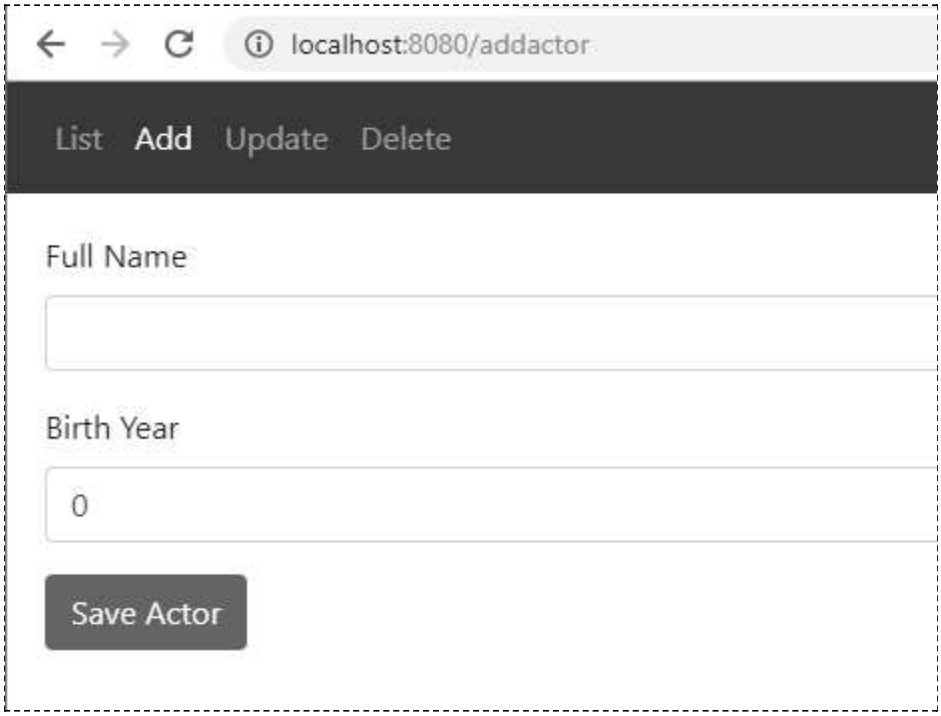
addactor.component.ts

```
import { Component, OnInit } from "@angular/core";
import { DatabaseService } from "../database.service";
import { Router } from "@angular/router";

@Component({
  selector: "app-addactor",
  templateUrl: "./addactor.component.html",
  styleUrls: ["./addactor.component.css"],
})
export class AddactorComponent {
  fullName: string = "";
  bYear: number = 0;
  actorId: string = "";

  constructor(private dbService: DatabaseService, private router:
Router) {}

  onSaveActor() {
    let obj = { name: this.fullName, bYear: this.bYear };
    this.dbService.createActor(obj).subscribe(result => {
      this.router.navigate(["/listactors"]);
    });
  }
```

```
23.
24.  }
```

> The constructor of this component has two dependencies: the DatabaseService and the Router as shown in line 15. The onSaveActor function generates the new actor objects and passes it to the service to be sent to the server via a post request. Line 20 uses the router service to redirect the client to another component. In other words, replace the current component with another one.

The addactor component:



Update Actor Component

To select the actor we want to update, this component lists all the available actors first in a table format.

```
1.  <div class="section">
2.    <table class="table table-striped">
3.      <tr>
4.        <th>Name</th>
5.        <th>Birth Year</th>
6.        <th>Select!</th>
7.      </tr>
8.      <tr *ngFor="let item of actorsDB">
```

```
9.        <td>{{item.name}}</td>
10.       <td>{{item.bYear}}</td>
11.       <td><button type="submit" class="btn btn-primary"
(click)="onSelectUpdate(item)">Update</button></td>
12.     </tr>
13.   </table>
14.   <hr>
15.   <div *ngIf="actorsDB.length>0">
16.     <div class="form-group">
17.       <label for="actorName">Full Name</label>
18.       <input type="text" class="form-control" id="actorName"
[(ngModel)]="fullName">
19.     </div>
20.     <div class="form-group">
21.       <label for="actorName">Birth Year</label>
22.       <input type="number" class="form-control" id="actorName"
[(ngModel)]="bYear">
23.     </div>
24.     <button type="submit" class="btn btn-primary"
(click)="onUpdateActor()">Update Actor</button>
25.   </div>
26. </div>
```

> Line 15 has the *ngIf directive that will show the update form only if there is at least one actor to update.

udpateactor.component.ts

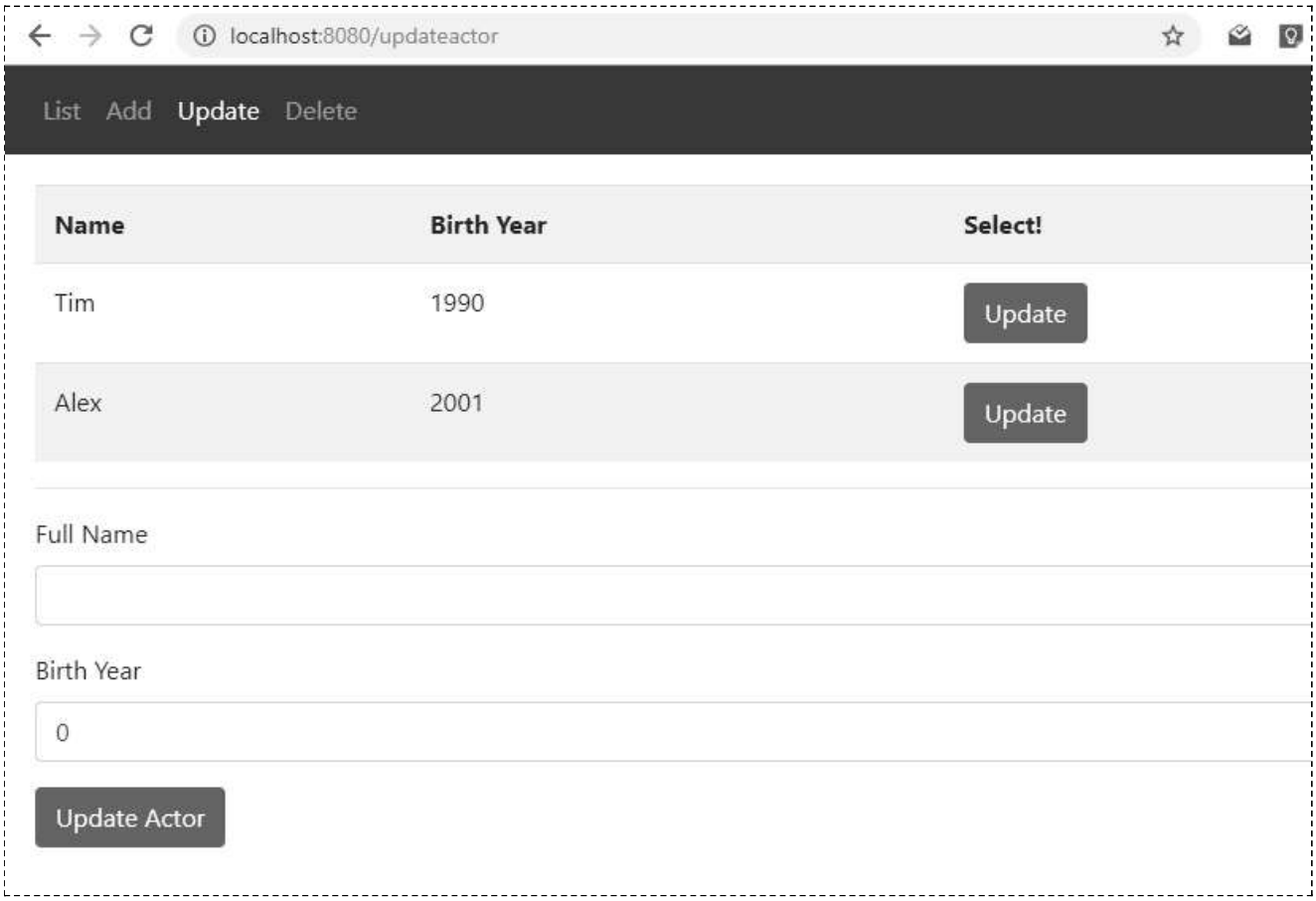
```
1.  import { Component, OnInit } from "@angular/core";
2.  import { DatabaseService } from "../database.service";
3.  import { Router } from "@angular/router";
4.
5.  @Component({
6.    selector: "app-updateactor",
7.    templateUrl: "./updateactor.component.html",
```

```
  styleUrls: ["./updateactor.component.css"],
})
export class UpdateactorComponent implements OnInit {
  fullName: string = "";
  bYear: number = 0;
  actorId: string = "";

  actorsDB: any[] = [];

  constructor(private dbService: DatabaseService, private router:
Router) {}

  //Get all Actors
  onGetActors() {
    console.log("From on GetActors");

    return this.dbService.getActors().subscribe((data: any[]) => {
      this.actorsDB = data;
    });
  }

  // Update an Actor
  onSelectUpdate(item) {
    this.fullName = item.name;
    this.bYear = item.bYear;
    this.actorId = item._id;
  }
  onUpdateActor() {
    let obj = { name: this.fullName, bYear: this.bYear };
    this.dbService.updateActor(this.actorId, obj).subscribe(result =>
{
      this.onGetActors();
      this.router.navigate(["/listactors"]);
    });
  }

```

```
42.    ngOnInit() {
43.      this.onGetActors();
44.    }
45.  }
```

> The backend logic of the update component requests two services to be injected (by the dependency injection) which are: DatabaseService and Router.
>
> DatabaseService will be used to fetch all the actors form the database while the Router service will be used to redirect the client side to another component at the end of the update operation.

The updateactor component:



Delete Actor Component

This component lists all the available actors and deletes the selected one.

deleteactor.component.html

```
1.  <div class="section">
```

```
  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Birth Year</th>
      <th>Delete?</th>
    </tr>
    <tr *ngFor="let item of actorsDB">
      <td>{{item.name}}</td>
      <td>{{item.bYear}}</td>
      <td><button type="submit" class="btn btn-primary"
(click)="onDeleteActor(item)">Delete</button></td>
    </tr>
  </table>
</div>
```

deleteactor.component.ts

```
import { Component, OnInit } from "@angular/core";
import { DatabaseService } from "../database.service";
import { Router } from "@angular/router";

@Component({
  selector: "app-deleteactor",
  templateUrl: "./deleteactor.component.html",
  styleUrls: ["./deleteactor.component.css"],
})
export class DeleteactorComponent implements OnInit {
  actorsDB: any[] = [];

  constructor(private dbService: DatabaseService, private router:
Router) {}
  //Get all Actors
  onGetActors() {
    console.log("From on GetActors");

```

```
18.       return this.dbService.getActors().subscribe((data: any[]) => {
19.         this.actorsDB = data;
20.       });
21.     }
22.
23.     //Delete Actor
24.     onDeleteActor(item) {
25.       this.dbService.deleteActor(item._id).subscribe(result => {
26.         this.onGetActors();
27.         this.router.navigate(["/listactors"]);
28.       });
29.     }
30.
31.     // This callback function will be invoked with the component get
    initialized by Angular.
32.     ngOnInit() {
33.       this.onGetActors();
34.     }
35.   }
```

This is how the deleteactor component looks like:

Routing

To configure our routing, add this array to app.module.ts just before the @NgModule decorator.

```
const appRoutes: Routes = [
  { path: "listactors", component: ListactorsComponent },
  { path: "addactor", component: AddactorComponent },
  { path: "updateactor", component: UpdateactorComponent },
  { path: "deleteactor", component: DeleteactorComponent },
  { path: "", redirectTo: "/listactors", pathMatch: "full" },
];
```

Import the RouterModule and provide the appRoutes array as input.

```
RouterModule.forRoot(appRoutes)
```

Don't forget to import the Routes and RouterModule from "@angular/router":

```
import { RouterModule, Routes } from "@angular/router";
```

So, the app.module.ts will be:

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { AppComponent } from "./app.component";
import { DatabaseService } from "./database.service";
import { HttpClientModule } from "@angular/common/http";
import { FormsModule } from "@angular/forms";
import { ListactorsComponent } from
"./listactors/listactors.component";
import { AddactorComponent } from "./addactor/addactor.component";
import { DeleteactorComponent } from
"./deleteactor/deleteactor.component";
import { UpdateactorComponent } from
"./updateactor/updateactor.component";
import { RouterModule, Routes } from "@angular/router";

```

```
14. const appRoutes: Routes = [
15.   { path: "listactors", component: ListactorsComponent },
16.   { path: "addactor", component: AddactorComponent },
17.   { path: "updateactor", component: UpdateactorComponent },
18.   { path: "deleteactor", component: DeleteactorComponent },
19.   { path: "", redirectTo: "/listactors", pathMatch: "full" },
20. ];
21.
22. @NgModule({
23.   declarations: [
24.     AppComponent,
25.     ListactorsComponent,
26.     AddactorComponent,
27.     UpdateactorComponent,
28.     DeleteactorComponent,
29.   ],
30.   imports: [
31.     RouterModule.forRoot(appRoutes),
32.
33.     BrowserModule,
34.     HttpClientModule,
35.     FormsModule,
36.   ],
37.   providers: [DatabaseService],
38.   bootstrap: [AppComponent],
39. })
40. export class AppModule {}
```

Remember:

1. import @line30: to import modules only and not the services or components.
2. declare @line23: to declare components only.
3. provide @line37: to list the required services only.

Now its time to build the navigation bar which will be located in the parent component ‘app-root’.

Open app.component.html and paste this code:

```
1.  <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
2.      <ul class="navbar-nav">
3.          <li class="nav-item">
4.              <a class="nav-link" routerLink="/listactors"
    routerLinkActive="active">List </a>
5.          </li>
6.          <li class="nav-item">
7.
8.              <a class="nav-link" routerLink="/addactor"
    routerLinkActive="active">Add </a>
9.          </li>
10.         <li class="nav-item">
11.
12.             <a class="nav-link" routerLink="/updateactor"
    routerLinkActive="active">Update</a>
13.         </li>
14.         <li class="nav-item">
15.
16.             <a class="nav-link" routerLink="/deleteactor"
    routerLinkActive="active">Delete</a>
17.         </li>
18.     </ul>
19. </nav>
20.
21. <router-outlet></router-outlet>
```

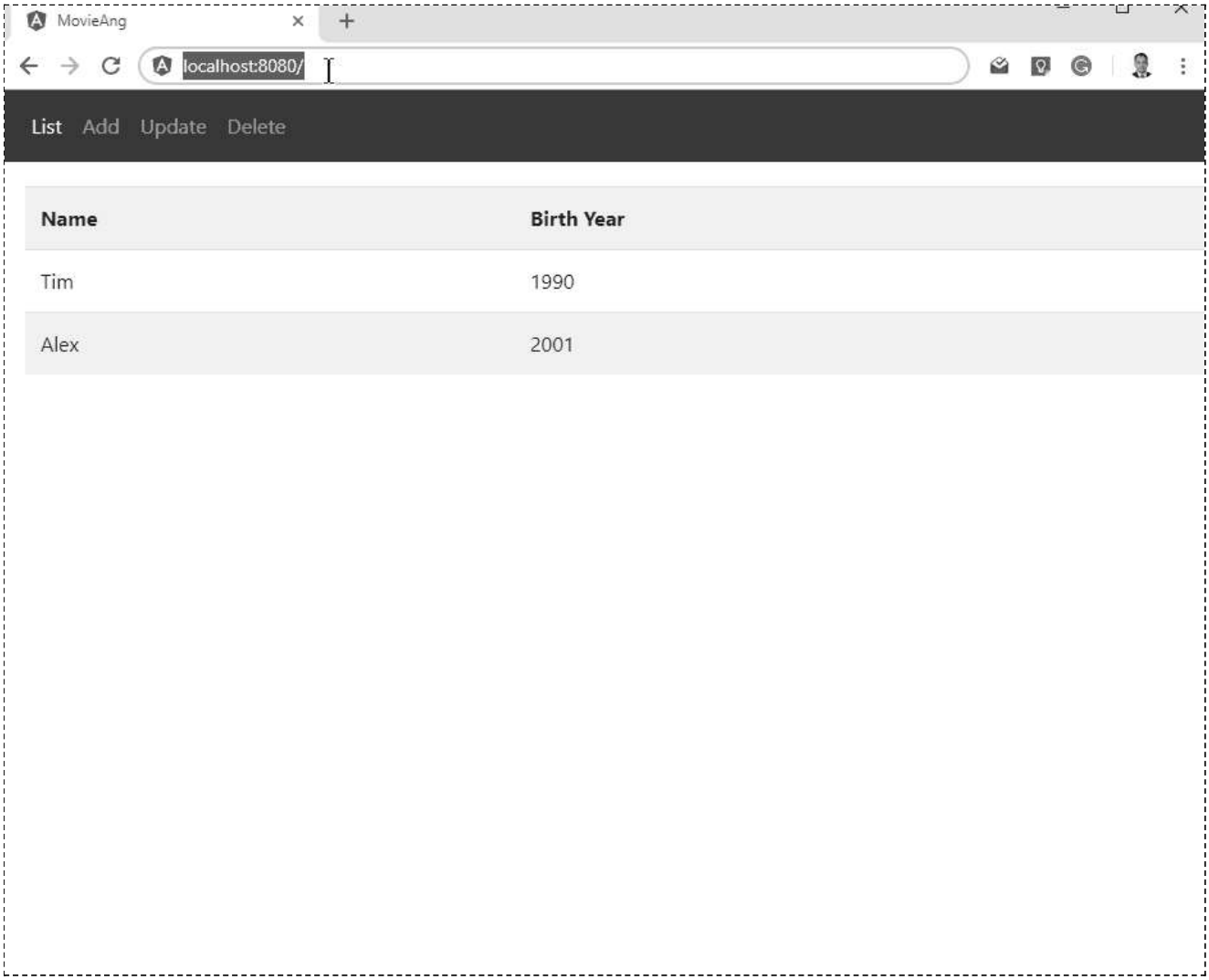
> source: https://www.w3schools.com/bootstrap4/bootstrap_navbar.asp

Line 21 represents the Router outlet which is where Angular has to place the selected component as discussed HERE.

To run your app:

```
ng build
node server.js
```

Here is how the app looks like:



Angular Pipes

Angular pipes are functions that are used to transform data in Angular templates. These functions take input data values, transform/process them and return a single output value.

Now, let's build a pipe that takes a string data value and two integer parameters. The pipe's function needs to return a substring using the first parameter as the starting index and the second parameter as the last index.

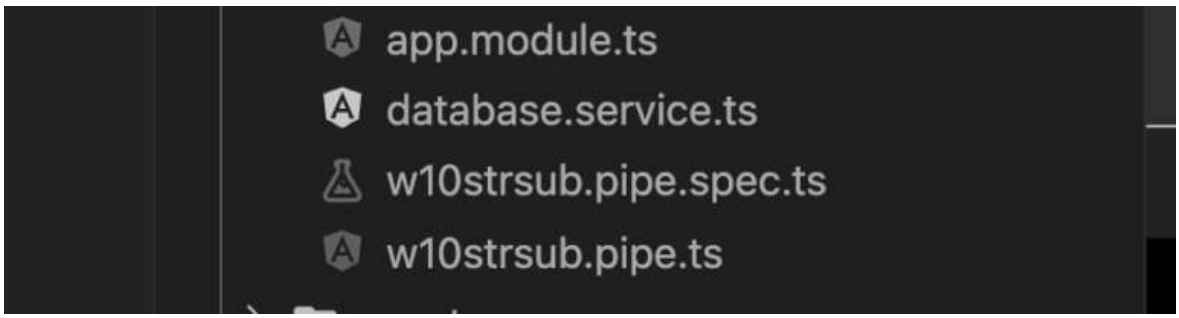
Step 1: add a pipe to your project

```
ng generate pipe w10strsub
```

or

```
ng g p w10strsub
```

Now, you got two new files in your project



Now open w10strsub.pipe.ts and update the code to be:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'w10strsub'
})
export class W10strsubPipe implements PipeTransform {

  transform(value: string, ...args: number[]): string {
    return null;
  }

}
```

where ‘value’ is the value the pipe function is invoked on (i.e. the value before the pipe character) and the parameters are the values that are placed after the pipe function.

Therefore, the syntax for calling the pipe is:

```
{{value | pipeFunction:param1:param2:param3}}
```

Now, let’s put the logic:

```
import {
  Pipe,
```

```
  PipeTransform
} from '@angular/core';

@Pipe({
  name: 'w10strsub'
})
export class W10strsubPipe implements PipeTransform {

  transform(value: string, ...args: number[]): string {

    let transformedStr = '';
    let startingIndex = args[0];
    let stopIndex = args[1];
    transformedStr = value.substring(startingIndex, stopIndex);
    return transformedStr;
  }

}
```

To test our new pipe:

Add the following statements to your templates

```
<td>{{item.name| w10strsub:1:2}}</td>
<td>{{"Week 10 Lab"| w10strsub:5:9 }}
```

Progressive Web Apps

Progressive web apps (PWA) are web applications that are capable to give users an experience similar to native apps. The main advantages of PWA apps are:

- installable: can be installed locally (works offline)
- progressively enhanced
- responsively designed: it works on multiple platforms such as desktop, mobile, and tables
- discoverable: can be discovered by the search engines
- network-independent: it works offline

- secure: much more secure than ordinary web applications

> *A progressive web application (PWA) is a type of application software delivered through the web, built using common web technologies including HTML, CSS and JavaScript. It is intended to work on any platform that uses a standards-compliant browser, including both desktop and mobile devices.*
>
> *Since a progressive web app is a type of webpage or website known as a web application, they do not require separate bundling or distribution. Developers can just publish the web application online, ensure that it meets baseline "installability requirements", and users will be able to add the application to their home screen. Publishing the app to digital distribution systems like Apple App Store or Google Play is optional.*

source: Wikipedia (https://en.wikipedia.org/wiki/Progressive_web_application)

What are the requirements to make your app installable?

In order to make your app comply with PWA standards you need the following:

1. web application (Angular application)
2. manifest file: a JSON file contains information about the app such as:
    1. name
    2. description
    3. language
    4. set of icons
    5. theme
    6. start_url
3. Service Worker: it's a JavaScript file that:
    1. runs in the background (not the main browser thread)
    2. intercepting all the network requests from/to your app
    3. saving (caching) or retrieving resources from the cache of your app
    4. sending push messages
4. icon for the app
5. secure connection (the web app is served from an HTTPS server). If you serve your PWA app from HTTP (unsecured), then your app will be installable on 'localhost' only.

How to make my Angular application a PWA app?

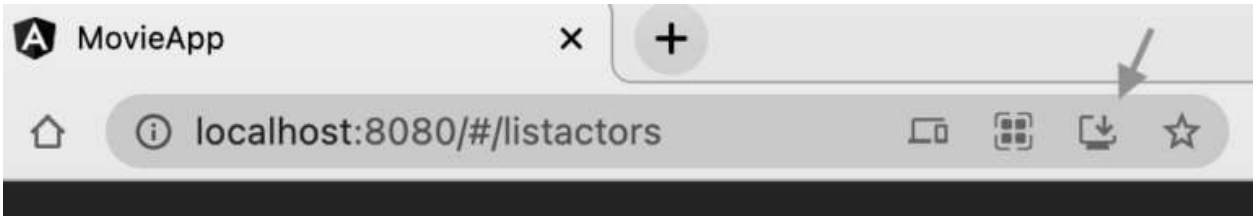
Good news!! Angular CLI can do it for you. All you need to do is to execute the following command on the root of you project:

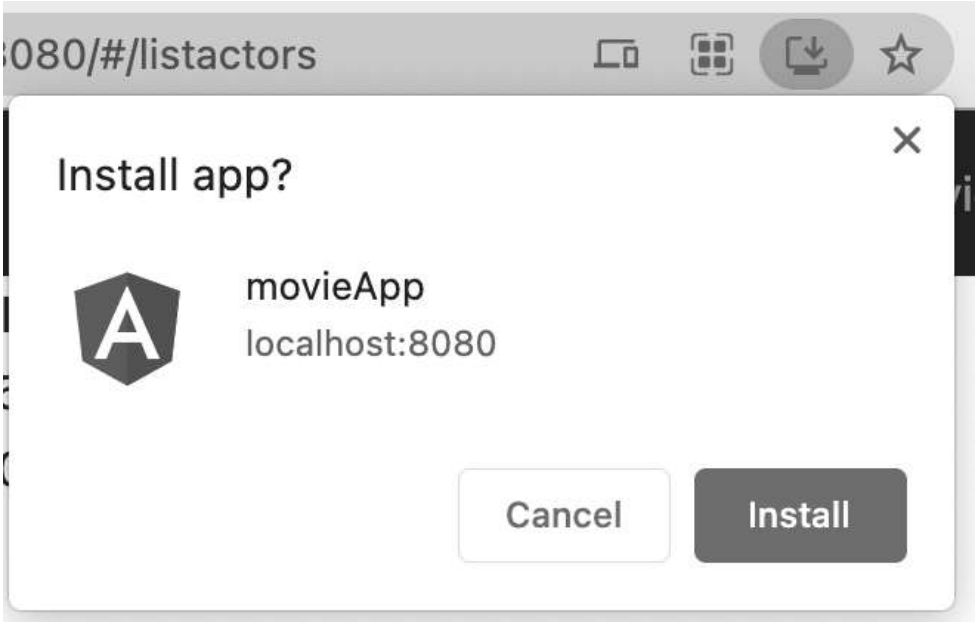
```
ng add @angular/pwa
```

Now build your app again:

```
ng build
```

To test, navigate to your app and you must see the install icon as depicted below:





MODULE

Week 11: Real-Time Bidirectional Event-Based Communication & Cloud Service

Nawfal Ali
Updated 11 October 2021

Question: If a server wants to push (send) data to a client, the client has to send an HTTP first; is there a way to send messages from the server to clients in real-time?

Answer: Yes. WebSocket protocol provides a full-duplex communication channel through a single TCP/IP connection.

Question: Is there a package that simplifies the usage of WebSockets?

Answer: Yes. Socket.io

What is Socket.io?

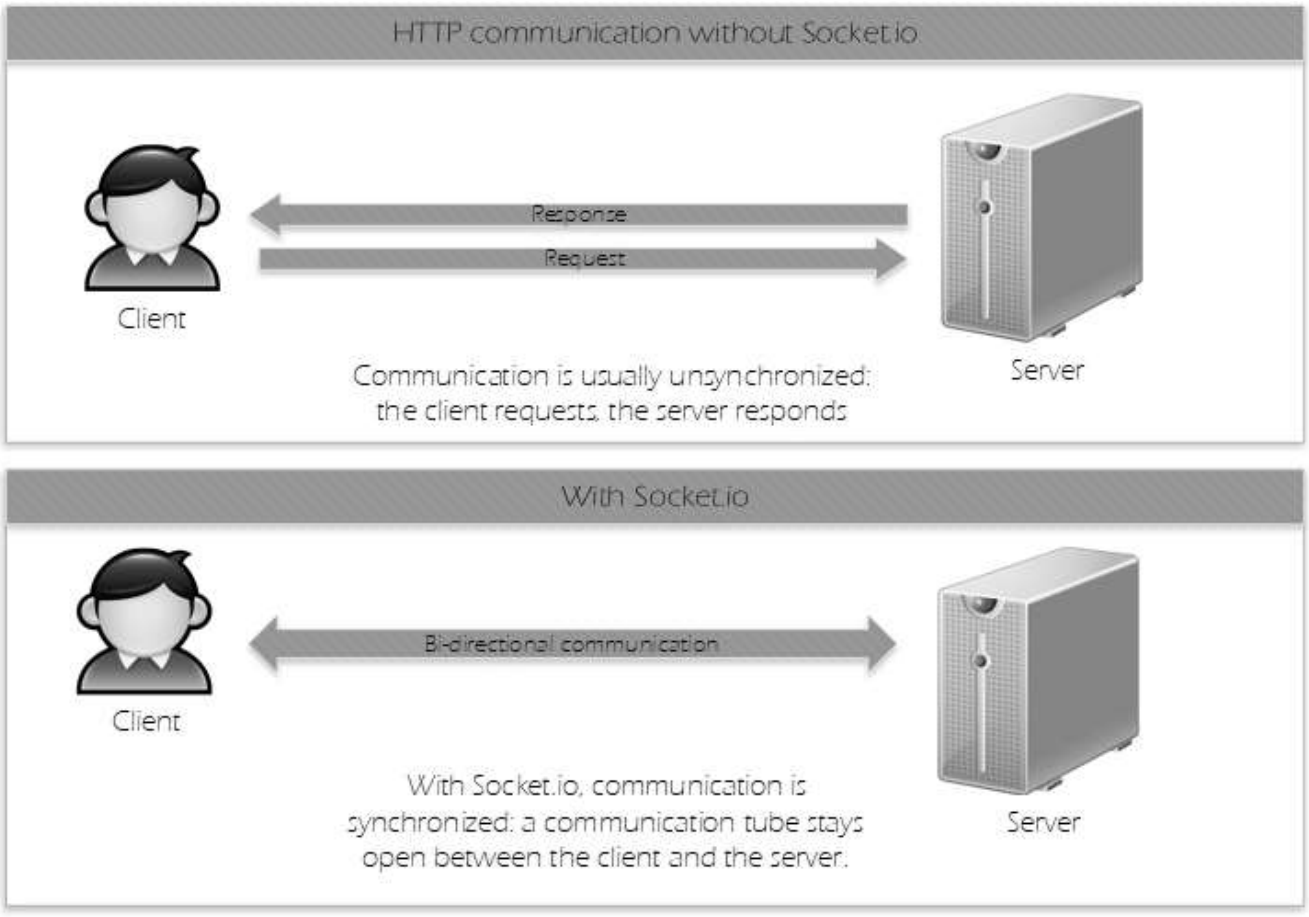
Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of:

- a Node.js server
- a Javascript client library for the browser

Socket.io main features [1]

- Reliability
- Auto-reconnection support:

    - Unless instructed otherwise a disconnected client will try to reconnect forever until the server is available again.
- Disconnection detection
    - A heartbeat mechanism is available, allowing both the server and the client to know when the other one is not responding anymore.
- Binary support:
    - Any serializable data structures can be emitted.
- Multiplexing support:
    - In order to create the separation of concerns within your application (for example per module, or based on permissions), Socket.IO allows you to create several Namespaces, which will act as separate communication channels but will share the same underlying connection.
- Room support:
    - Within each Namespace, you can define arbitrary channels, called Rooms, that sockets can join and leave. You can then broadcast to any given room, reaching every socket that has joined it. This is a useful feature to send notifications to a group of users, or to a given user connected on several devices for example.



A Chat Application using Node.js, Express, Angular, and Socket.io

Server Side (Node.js and Express)

- Create a file named server.js
- Install Socket.io server-side library

```
npm install  socket.io
```

- Install Express

```
npm install  express
```

- Create an instance of Express application

```
var app = require('express')();
var server = require('http').Server(app);
```

> Line 2: Socket.io is attached to an instance of http.Server and adds handlers to it.

- Create an instance of Socket.io server

```
let io = require("socket.io")(server);
```

- Wait for connections:

```
io.on("connection", function(socket) {
  console.log("new connection made");
}
```

- Now, wait for events (messages):

```
  socket.on("eventName", function(data){
    console.log("got data ");
}
```

- If you have more than one event:

```
  socket.on("event1", function(data){
    console.log("got data from event 1 ");
}
  socket.on("event2", function(data){
    console.log("got data from event 2 ");
}
```

- To send a message to all connected sockets (i.e. clients)

```
io.sockets.emit("anotherEventName", data);
```

- or to send to a single client:

```
socket.emit("justAnotherEvent", data);
```

So, the server side in one piece

```
let express = require("express");
let path = require("path");
let app = express();
let server = require("http").Server(app);

let io = require("socket.io")(server);

let port = 8080;

```

```
app.use("/", express.static(path.join(__dirname, "dist/chatApp")));

io.on("connection", socket => {
  console.log("new connection made from client with ID="+socket.id);

  socket.on("newMsg", data => {
    io.sockets.emit("msg", { msg: data, timeStamp: getCurrentDate()
});
  });
});

server.listen(port, () => {
  console.log("Listening on port " + port);
});

function getCurrentDate() {
  let d = new Date();
  return d.toLocaleString();
}
```

Client-Side

- Create a new Angular Application using:

```
ng new chatapp
```

- Add the server-side source code ‘server.js’ to the root folder of the Angular app
- Install socket.io client-side libraries

```
npm install socket.io-client
npm install @types/socket.io-client
```

- open app.component.ts and import socket.io-client

```
import { io } from 'socket.io-client';
```

- Create an instance of the socket.io client

```
let socket: any;
```

- Send a connection request to the server:

```
this.socket = io();
```

- The client has to listen to messages from its server:

```
this.socket.on("eventName", theMsg => {});
```

- If the client wants to send a message to the server:

```
this.socket.emit("justAnotherEventName", this.messageText);
```

- So, the client-side app.component.ts in one piece:

```
import { Component } from "@angular/core";
import { io } from 'socket.io-client';
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  messageText: string;
  messages: Array<any> = [];
  socket:any;
  constructor() {
    this.socket = io();
  }
```

```
  ngOnInit() {
    this.messages = new Array();
    this.listen2Events();
  }
  listen2Events() {
    this.socket.on("msg", data => {
      this.messages.push(data);
    });
  }

  sendMessage() {
    this.socket.emit("newMsg", this.messageText);
    this.messageText = "";
  }
}
```

- and app.component.html:

```
<div id="chatContainer">
  <div class="chatform">
    <div class="form-group">
      <label for="userName">Your Name</label>
      <input type="text" class="form-control" id="userName"
placeholder="Your name" [(ngModel)]="userName">
    </div>
    <div class="form-group">
      <label for="message">Message</label>
      <input type="text" class="form-control" id="message"
placeholder="Your message" [(ngModel)]="messageText">
    </div>

    <button (click)="sendMessage()" type="submit" class="btn btn-
primary">Send </button>
  </div>

```

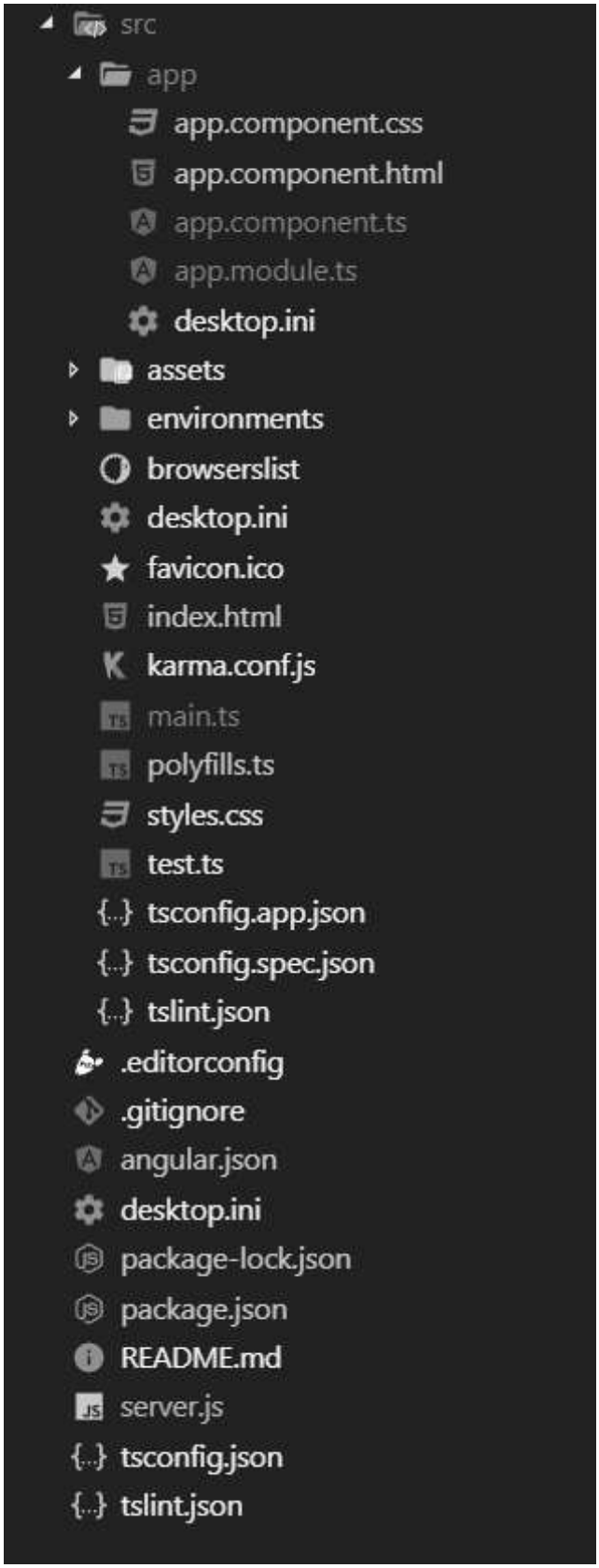
```
15.
16.   <div class="container" *ngFor="let item of
    messages.slice().reverse() ">
17.     <p>From: {{item.userName}}</p>
18.     <div>
19.       <h3>{{item.msg}}</h3>
20.     </div>
21.     <span class="time-right">{{item.timeStamp}}</span>
22.   </div>
23. </div>
```

- and app.component.css

```
1.  html,
2.  body {
3.    height: 100%;
4.  }
5.
6.  #chatContainer {
7.    height: 80vh !important;
8.    overflow-y: auto;
9.    padding-left: 2vw;
10.   padding-top: 2vh;
11. }
12. .chatform input{
13.     margin-bottom: 2vh;
14.     margin-left: 1vh;
15.     width: 90%;
16.
17. }
18. .chatform{
19.   background-color:beige;
20.   width: 50%;
21.
22. }
```

```
23.
24. .container {
25.   border: 2px solid #dedede;
26.   background-color: #f1f1f1;
27.   border-radius: 5px;
28.   padding: 10px;
29.   margin: 10px 0;
30. }
31.
32. .container::after {
33.   content: "";
34.   clear: both;
35.   display: table;
36. }
37.
38. .time-right {
39.   float: right;
40.   color: #aaa;
41. }
```

The final project structure:



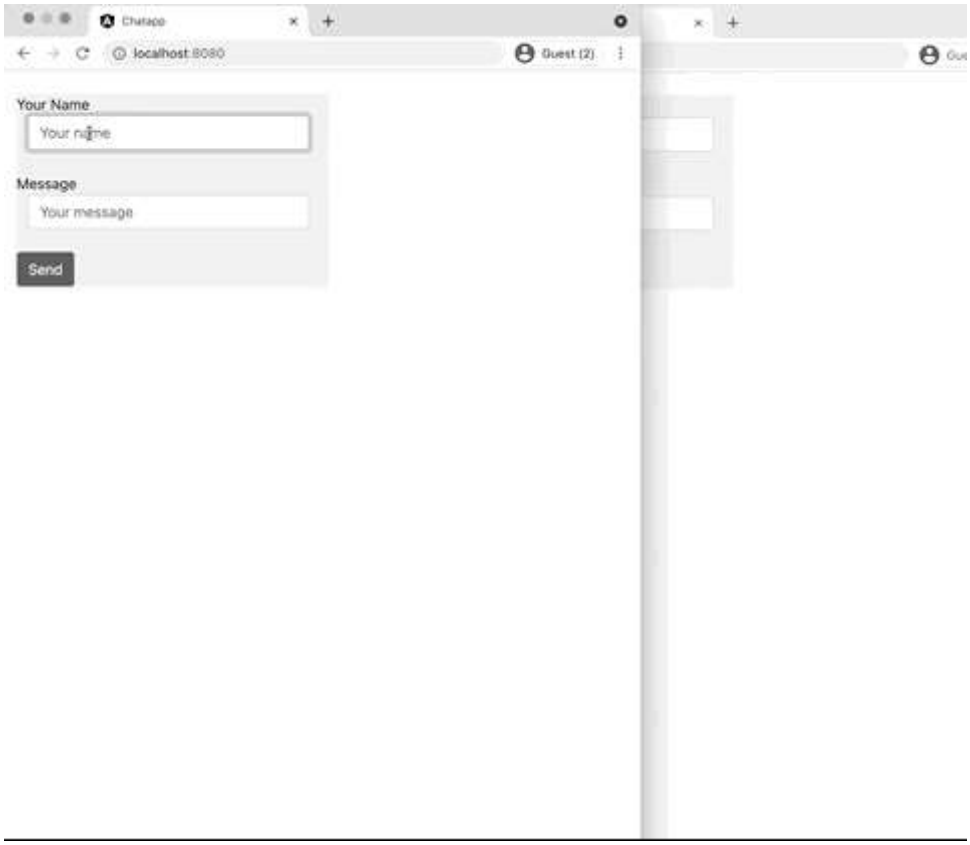
- build your client-side:

```
1. ng build
```

- and run your server:

```
node server.js
```

And here is the expected output:



Socket.io Emit Cheatsheet

```
io.on('connect', onConnect);

function onConnect(socket){

  // sending to the client
  socket.emit('hello', 'can you hear me?', 1, 2, 'abc');

  // sending to all clients except sender
  socket.broadcast.emit('broadcast', 'hello friends!');

  // sending to all clients in 'game' room except sender
  socket.to('game').emit('nice game', "let's play a game");

```

```
14.   // sending to all clients in 'game1' and/or in 'game2' room, except
      sender
15.   socket.to('game1').to('game2').emit('nice game', "let's play a game
      (too)");
16.
17.   // sending to all clients in 'game' room, including sender
18.   io.in('game').emit('big-announcement', 'the game will start soon');
19.
20.   // sending to all clients in namespace 'myNamespace', including
      sender
21.   io.of('myNamespace').emit('bigger-announcement', 'the tournament
      will start soon');
22.
23.   // sending to a specific room in a specific namespace, including
      sender
24.   io.of('myNamespace').to('room').emit('event', 'message');
25.
26.   // sending to individual socketid (private message)
27.   io.to(`${socketId}`).emit('hey', 'I just met you');
28.
29.   // WARNING: `socket.to(socket.id).emit()` will NOT work, as it will
      send to everyone in the room
30.   // named `socket.id` but the sender. Please use the classic
      `socket.emit()` instead.
31.
32.   // sending with acknowledgement
33.   socket.emit('question', 'do you think so?', function (answer) {});
34.
35.   // sending without compression
36.   socket.compress(false).emit('uncompressed', "that's rough");
37.
38.   // sending a message that might be dropped if the client is not
      ready to receive messages
39.   socket.volatile.emit('maybe', 'do you really need it?');
40.
41.   // specifying whether the data to send has binary data
```

```
    socket.binary(false).emit('what', 'I have no binaries!');

    // sending to all clients on this node (when using multiple nodes)
    io.local.emit('hi', 'my lovely babies');

    // sending to all connected clients
    io.emit('an event sent to all connected clients');

};
```

Source: https://socket.io/docs/emit-cheatsheet/

Text to Speech Google Cloud Service

Google Cloud Text-to-Speech enables developers to synthesize natural-sounding speech with 30 voices, available in multiple languages and variants. It applies DeepMind's groundbreaking research in WaveNet and Google's powerful neural networks to deliver high fidelity audio. With this easy-to-use API, you can create lifelike interactions with your users, across many applications and devices.

Advantages of using Cloud Services:

- Easy to integrate
- Heavily tested
- Pay as you go
- No need to reinvent the wheel
- Developing a reliable and accurate text to speech service (for instance) might take months.

> Develop an application that converts a text to a speech using Google Cloud Services.

This application has only the server-side that uses Google cloud service to convert a text to an MP3 file.

Configurations

- Set up authentication:

    - Go to the **Create service account key** page in the GCP Console.
        - https://console.cloud.google.com/apis/credentials/serviceaccountkey
    - From the **Service account** drop-down list, select **New service account**.
    - Enter a name into the **Service account name** field.
    - Don't select a value from the **Role** drop-down list. No role is required to access this service.
    - Click **Create**. A note appears, warning that this service account has no role.
    - Click **Create without role**. A JSON file that contains your key downloads to your computer (name it fit2095.json).
- Set the environment variable GOOGLE_APPLICATION_CREDENTIALS to the file path of the JSON file that contains your service account key. This variable only applies to your current shell session, so if you open a new session, set the variable again.
    - Examples
        - Linux or macOS: `export GOOGLE_APPLICATION_CREDENTIALS="/home/user/Downloads/service-account-file.json"`
        - Windows: `$env:GOOGLE_APPLICATION_CREDENTIALS="C:\Users\username\Downloads\[FILE_NAME].json"`
- Enable the Cloud Text-to-Speech API.
    - From the menu, select APIs & Services->Library
    - Search for text to speech service
    - Click on Enable (or ignore if the service is already enabled)
- Install and initialize the Cloud SDK.
    - Download and install Google SDK by following all the steps on this page:
        - https://cloud.google.com/sdk/docs/

Server Side

- Create a new folder named text2speech
- Install Google Cloud client library

```
npm install --save @google-cloud/text-to-speech
```

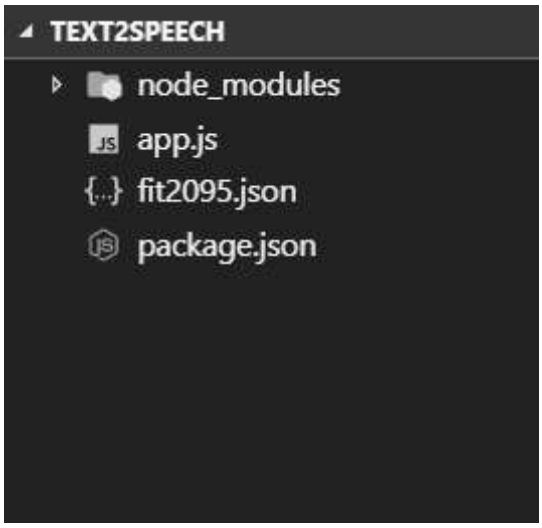
- Create a file named server.js and paste this code in it:

```
const fs = require("fs");

// Imports the Google Cloud client library
const textToSpeech = require("@google-cloud/text-to-speech");
```

```
5.
6.  // Creates a client
7.  const client = new textToSpeech.TextToSpeechClient();
8.
9.  // The text to synthesize
10. const text = "Hello from FIT2095 Week 11 lecture";
11.
12. // Construct the request
13. const request = {
14.   input: { text: text },
15.   // Select the language and SSML Voice Gender (optional)
16.   voice: { languageCode: "en-US", ssmlGender: "NEUTRAL" },
17.   // Select the type of audio encoding
18.   audioConfig: { audioEncoding: "MP3" },
19. };
20.
21. // Performs the Text-to-Speech request
22. client.synthesizeSpeech(request, (err, response) => {
23.   if (err) {
24.     console.error("ERROR:", err);
25.     return;
26.   }
27.
28.   // Write the binary audio content to a local file
29.   fs.writeFile("output.mp3", response.audioContent, "binary", err => {
30.     if (err) {
31.       console.error("ERROR:", err);
32.       return;
33.     }
34.     console.log("Audio content written to file: output.mp3");
35.   });
36. });
```

> Dont forget to set the environment variable GOOGLE_APPLICATION_CREDENTIALS as discussed earlier.

- The project structure will be:



- run your app by:

```
node server.js
```

You will get a new file named output.mp3 added to your project. Open it and test the results.

MODULE

Week 12: MEAN Stack: Authentication and Cloud Services

Nawfal Ali
Updated 13 October 2021

How secure are MEAN stack applications?

MEAN stack applications are becoming popular due to their lightweight, massive ecosystem of middleware plugins and dependencies, and easy to deploy to most cloud vendors. However, these applications might be attacked by common vulnerabilities that are introduced either by using MEAN stack components in their default configurations or due to common developer mistakes.

In this section, we will examine one case

Query selector injection (MongoDB)

MongoDB database is vulnerable to an attack called Query Selector Injection that uses Query selector logic operators to change queries.

Let's have an example:

If a client sends a login request (POST) to a server that consists of a username and password, the intruder can intercept the login request and changed the POST pass parameter to password[$ne]=null where $ne is the MongoDB query operator 'not equal to'.

When Express processes the x-www-form-urlencoded request body, this will be parsed into the object {username: 'admin', password: {$ne: null} and passed to the MongoDB query:

```
db.collection('users').findOne({username:user,password:pass},function(
err,result){
    if(result!==null){
        res.redirect('/')
    }else{
        res.redirect('/login');
    }
```

```
});
```

Passing the above object to MongoDB, the query will match the first username that matches ‘admin’ with a password that is not equal to null, allowing us to authenticate to the application as an administrator without providing a valid password.

Mitigation

The simplest way to mitigate query selector injection is to cast the values to a String before running the query.

```
db.collection("users").findOne(
  {
    username: String(user),
    password: String(pass),
  },
  function(err, result) {
    if (result !== null) {
      res.redirect("/");
    } else {
      res.redirect("/login");
    }
  }
);
```

This website contains a very long list of Web Application and Network Vulnerabilities CLICK HERE.

MEAN Stack Authentication

> The main source of this section is: http://www.passportjs.org/docs/

In modern web applications, authentication can take a variety of forms. Traditionally, users log in by providing a username and password. With the rise of social networking, single sign-on using an OAuth provider such as Facebook or Twitter has become a popular authentication method. Services that expose an API often require token-based credentials to protect access.

There are several authentication libraries that can be used to provide user management and authentication such as PassportJS and Everyauth.

> Note: you can build your own database and a hashing algorithm (for passwords) or you can use User Management as a Service Cloud Services such as Okta and onelogin.

In this section, we will use PassportJS as an example for Auth libraries.

What is Passport.js?

Passport is authentication middleware for Node.js. As it's extremely flexible and modular, Passport can be unobtrusively dropped into any Express-based web application. A comprehensive set of strategies supports authentication using a username and password, Facebook, Twitter, and more. Find out more about Passport here.

Features

- 300+ authentication strategies
- Single sign-on with OpenID and OAuth
- Easily handle success and failure
- Supports persistent sessions
- Dynamic scope and permissions
- Pick and choose required strategies
- Implement custom strategies
- Does not mount routes in application
- Lightweight code base

Now, we will examine four strategies provided by PassportJS, which are: local, Open ID, Facebook, and Google.

Username & Password (Local)

The most widely used way for websites to authenticate users is via a username and password. Support for this mechanism is provided by the passport-local module.

Install

```
npm install passport-local
```

```
var passport = require('passport')
  , LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function(err, user) {
      if (err) { return done(err); }
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (!user.validPassword(password)) {
        return done(null, false, { message: 'Incorrect password.' });
      }
      return done(null, user);
    });
  }
));
```

The verify callback for local authentication accepts username and password arguments, which are submitted to the application via a login form.

Parameters

By default, LocalStrategy expects to find credentials in parameters named username and password. If your site prefers to name these fields differently, options are available to change the defaults.

```
passport.use(new LocalStrategy({
    usernameField: 'email',
    passwordField: 'passwd'
  },
```

```
  function(username, password, done) {
    // ...
  }
));
```

Route

The login form is submitted to the server via the POST method. Using authenticate() with the local strategy will handle the login request.

OpenID

OpenID is an open standard for federated authentication. When visiting a website, users present their OpenID to sign in. The user then authenticates with their chosen OpenID provider, which issues an assertion to confirm the user's identity. The website verifies this assertion in order to sign the user in.

Install

```
npm install passport-openid
```

Configuration

When using OpenID, a return URL and realm must be specified. The returnURL is the URL to which the user will be redirected after authenticating with their OpenID provider. realm indicates the part of URL-space for which authentication is valid. Typically this will be the root URL of the website.

```
var passport = require('passport')
  , OpenIDStrategy = require('passport-openid').Strategy;

passport.use(new OpenIDStrategy({
    returnURL: 'http://www.example.com/auth/openid/return',
    realm: 'http://www.example.com/'
  },
  function(identifier, done) {
    User.findOrCreate({ openId: identifier }, function(err, user) {
```

```
      done(err, user);
    });
  }
));
```

Facebook

The Facebook strategy allows users to log in to a web application using their Facebook account. Internally, Facebook authentication works using OAuth 2.0.

Install

```
npm install passport-facebook
```

Configuration

In order to use Facebook authentication, you must first create an app at Facebook Developers. When created, an app is assigned an App ID and App Secret. Your application must also implement a redirect URL, to which Facebook will redirect users after they have approved access for your application.

```
var passport = require('passport')
  , FacebookStrategy = require('passport-facebook').Strategy;

passport.use(new FacebookStrategy({
    clientID: FACEBOOK_APP_ID,
    clientSecret: FACEBOOK_APP_SECRET,
    callbackURL: "http://www.example.com/auth/facebook/callback"
  },
  function(accessToken, refreshToken, profile, done) {
    User.findOrCreate(..., function(err, user) {
      if (err) { return done(err); }
      done(null, user);
    });
  }
));
```

Routes

Two routes are required for Facebook authentication. The first route redirects the user to Facebook. The second route is the URL to which Facebook will redirect the user after they have logged in.

```
// Redirect the user to Facebook for authentication.  When complete,
// Facebook will redirect the user back to the application at
//     /auth/facebook/callback
app.get('/auth/facebook', passport.authenticate('facebook'));

// Facebook will redirect the user to this URL after approval.  Finish the
// authentication process by attempting to obtain an access token.  If
// access was granted, the user will be logged in.  Otherwise,
// authentication has failed.
app.get('/auth/facebook/callback',
  passport.authenticate('facebook', { successRedirect: '/',
                                      failureRedirect: '/login' }));
```

Google

The Google strategy allows users to sign in to a web application using their Google account. Google used to support OpenID internally, but it now works based on OpenID Connect and supports oAuth 1.0 and oAuth 2.0.

Install

```
npm install passport-google-oauth
```

Configuration

```
var passport = require('passport');
var GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;

// Use the GoogleStrategy within Passport.
//   Strategies in Passport require a `verify` function, which accept
```

```
//   credentials (in this case, an accessToken, refreshToken, and Google
//   profile), and invoke a callback with a user object.
passport.use(new GoogleStrategy({
    clientID: GOOGLE_CLIENT_ID,
    clientSecret: GOOGLE_CLIENT_SECRET,
    callbackURL: "http://www.example.com/auth/google/callback"
  },
  function(accessToken, refreshToken, profile, done) {
       User.findOrCreate({ googleId: profile.id }, function (err, user) {
         return done(err, user);
       });
  }
));
```

Routes

```
// GET /auth/google
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  The first step in Google authentication will involve
//   redirecting the user to google.com.  After authorization, Google
//   will redirect the user back to this application at /auth/google/callback
app.get('/auth/google',
  passport.authenticate('google', { scope: ['https://www.googleapis.com/auth/plus.login'] }));

// GET /auth/google/callback
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  If authentication fails, the user will be redirected back to the
//   login page.  Otherwise, the primary route function function will be called,
```

```
//   which, in this example, will redirect the user to the home page.
app.get('/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/');
  });
```

Cloud Services

In addition to text to speech service we have covered in the previous week, this section demonstrates two cloud services that can be used to build smart and pervasive applications.

Cloud Storage

In order to make it easier for developers to build scalable and pervasive applications, moving their data to the cloud is one of the options. Regardless of the weak point of being less secure, cloud storage has better flexibility and reliability. There are many cloud vendors that offer cloud storages such as:

- Rackspace Cloud Storage
- MS Azure Cloud Services on Windows Azure
- Google Firebase
- Zadara
- Cloudian

Google Firebase

Cloud Storage for Firebase stores your data in Google Cloud Storage, an exabyte scale object storage solution with high availability and global redundancy. Firebase Admin SDK allows you to directly access your Google Cloud Storage buckets from privileged environments. Then you can use Google Cloud Storage APIs to manipulate the objects stored in the buckets.

Google Realtime Database is an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.

Receive Messages

To receive a message from Google firebase storage, you have to listen to events such as ‘child_added’ which will be triggered with each new object added to the storage.

```
var callback = function(snap) {
  var data = snap.val();
  displayMessage(snap.key, data.name, data.text, data.profilePicUrl, data.imageUrl);
};

firebase.database().ref('/messages/').limitToLast(12).on('child_added', callback);
```

In the above code, ‘/messages/’ is the table’s (i.e. collection) name and ‘firebase’ is a reference to the firebase client. The callback function will be able to access the new object using its input paramter ‘snap’.

Send Messages

```
// Saves a new message on the Firebase DB.
function saveMessage(messageText) {
  // Add a new message entry to the Firebase Database.
```

```
  return firebase.database().ref('/messages/').push({
    name: getUserName(),
    text: messageText,
    profilePicUrl: getProfilePicUrl()
  }).catch(function(error) {
    console.error('Error writing new message to Firebase Database',
error);
  });
}
```

the above code pushes a JSON object to the cloud storage.

Reference: https://codelabs.developers.google.com/codelabs/firebase-web/#0

Amazon Web Services

Amazon Web Services offers a broad set of global cloud-based products including compute, storage, databases, analytics, networking, mobile, developer tools, management tools, IoT, security and enterprise applications. These services help organizations move faster, lower IT costs, and scale. AWS is trusted by the largest enterprises and the hottest start-ups to power a wide variety of workloads including: web and mobile applications, game development, data processing and warehousing, storage, archive, and many others.

Amazon Rekognition

Amazon Rekognition makes it easy to add image and video analysis to your applications. You just provide an image or video to the Rekognition API, and the service can identify the objects, people, text, scenes, and activities, as well as detect any inappropriate content. Amazon Rekognition also provides highly accurate facial analysis and facial recognition on images and video that you provide. You can detect, analyze, and compare faces for a wide variety of user verification, people counting, and public safety use cases.

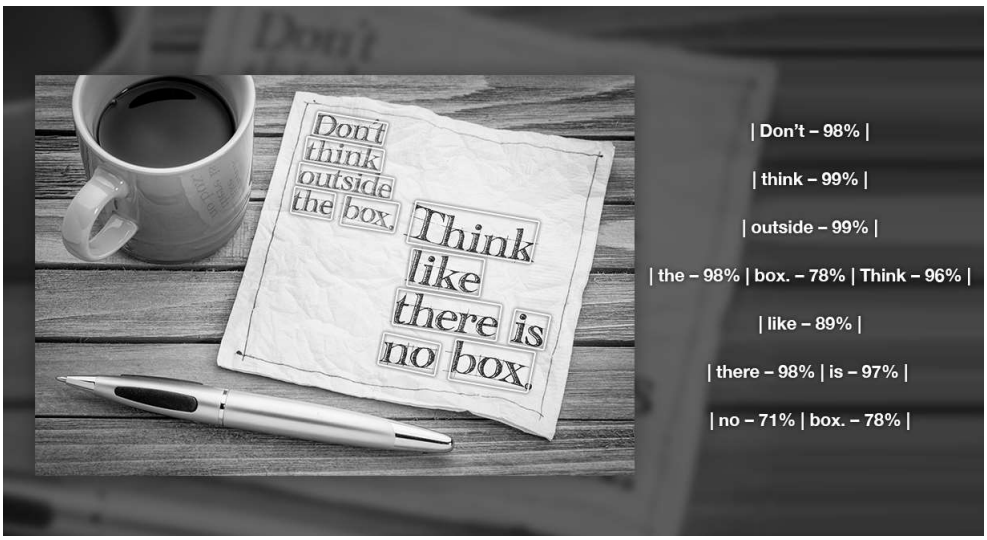
Key Features

- Object, scene, and activity detection

- Facial recognition

- Facial analysis

- Text in images



The following piece of code shows how simple and easy comparing two images is.

```
/* This operation compares the largest face detected in the source image with each face detected in the target image. */

 var params = {
  SimilarityThreshold: 90,
  SourceImage: {
   S3Object: {
    Bucket: "mybucket",
    Name: "mysourceimage"
   }
  },
  TargetImage: {
   S3Object: {
    Bucket: "mybucket",
    Name: "mytargetimage"
   }
  }
 };
 rekognition.compareFaces(params, function(err, data) {
   if (err) console.log(err, err.stack); // an error occurred
   else     console.log(data);           // successful response
   /*
```

```
    data = {
     FaceMatches: [
        {
       Face: {
        BoundingBox: {
         Height: 0.33481481671333313,
         Left: 0.31888890266418457,
         Top: 0.4933333396911621,
         Width: 0.25
        },
        Confidence: 99.9991226196289
       },
       Similarity: 100
      }
     ],
     SourceImageFace: {
      BoundingBox: {
       Height: 0.33481481671333313,
       Left: 0.31888890266418457,
       Top: 0.4933333396911621,
       Width: 0.25
      },
      Confidence: 99.9991226196289
     }
    }
    */
  });
```

References:

- https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Rekognition.html
- https://aws.amazon.com/products/
- https://www.acunetix.com/vulnerabilities/web/mongodb-injection/
- https://medium.com/@SW_Integrity/mongodb-preventing-common-vulnerabilities-in-the-mean-stack-ac27c97198ec