

Diploma in IT, Networking and Cloud Elective Module 3 Web Development using JS Frameworks Theory Manual

Table of Contents

| | |
|--|----|
| Introduction to mean stack | 10 |
| MEAN Stack | 10 |
| How Does the MEAN Stack Work? | 10 |
| MEAN Stack Components | 11 |
| Benefits of using Mean Stack | 12 |
| Disadvantages of Mean Stack | 13 |
| How secure is the MEAN stack? | 14 |
| Why choose Mean Stack? | 14 |
| Introduction to Angular JS and UI benefits in Angular JS | 14 |
| Introduction to Angular JS | 14 |
| Angular Version History | 15 |
| AngularJS Architecture | 15 |
| Concepts of AngularJS | 16 |
| Advantages of AngularJS | 17 |
| Disadvantages of AngularJS | 18 |
| General features | 18 |
| UI benefits in Angular JS | 18 |
| Usage of Angular JS with HTML | 21 |
| Event Handling in Angular JS | 24 |
| AngularJS Events | 24 |

| | |
|--|----|
| Introduction Angular & difference b/w angular-1 and angular-5..... | 27 |
| Introduction Angular | 27 |
| Top 12 Angular Features: | 28 |
| Difference b/w Angular-1 to Angular-5 | 35 |
| Typescript Datatype and Operator | 38 |
| Transcript Datatype | 38 |
| Number | 39 |
| Boolean..... | 40 |
| String..... | 40 |
| Void | 41 |
| Null | 41 |
| Typescript Operator | 42 |
| Arithmetic Operators | 43 |
| Comparison (Relational) Operators | 45 |
| Logical Operators..... | 47 |
| Bitwise Operators..... | 48 |
| Assignment Operators | 50 |
| Ternary/Conditional Operator | 52 |
| Concatenation Operator..... | 53 |
| Type Operators | 53 |
| Type Script String & Tuple, Oops..... | 54 |
| Typescript String..... | 54 |

| | |
|--|----|
| Typescript Tuple | 60 |
| Accessing tuple Elements | 61 |
| Operations on Tuple | 62 |
| Update or Modify the Tuple Elements..... | 63 |
| Clear the fields of a Tuple | 64 |
| Destructuring the Tuple..... | 64 |
| Passing Tuple to Functions..... | 65 |
| Typescript OOPS..... | 66 |
| Object and Classes | 68 |
| Creating classes | 68 |
| Creating Instance objects | 71 |
| Accessing Attributes and Functions..... | 71 |
| Inheritance & Interface | 74 |
| Single Interface Inheritance..... | 75 |
| Multiple Interface Inheritance | 75 |
| Angular js Child Component | 77 |
| Advantages of Components: | 77 |
| When not to use Components: | 78 |
| Creating and configuring a Component..... | 78 |
| Component-based application architecture | 78 |
| Angular js Data Binding, Event Binding,2 Way Binding | 81 |
| Angular js Data Binding | 81 |

| | |
|---|-----|
| How To Bind Events In AngularJS | 83 |
| Two Way Data Binding | 87 |
| Angular JS Pipe | 90 |
| What is Pipe? | 90 |
| Why Pipe? | 90 |
| Built-In Pipes | 91 |
| Custom Filter: | 91 |
| Pipe Chaining | 91 |
| Angular js Routing | 92 |
| Introduction to ngRoute Module | 92 |
| Component of ngRoute Module | 93 |
| Pass parameters to route URL | 100 |
| Load local views | 101 |
| Angular js Services | 106 |
| What are AngularJS Services? | 106 |
| When to use Services in AngularJS? | 106 |
| Types of AngularJS Services | 106 |
| Built-in Services in AngularJS | 106 |
| Customs Services in AngularJS | 108 |
| Angular js Http Request | 109 |
| Methods | 112 |
| Property | 112 |

| | |
|---|------------|
| Node Introduction | 113 |
| What is Node.js? | 113 |
| Features of Node.js | 113 |
| Who Uses Node.js? | 114 |
| Concepts | 114 |
| Where to Use Node.js? | 115 |
| Where Not to Use Node.js? | 115 |
| Application of NodeJS: | 116 |
| Using NPM: | 119 |
| Blocking & Nonblocking code..... | 119 |
| Blocking: | 120 |
| Non-Blocking: | 121 |
| Comparison between Blocking and Non-Blocking in Node.js:..... | 122 |
| Create Web App using express | 188 |
| Node.js - Express Framework | 188 |
| Express Overview | 188 |
| Installing Express | 188 |
| Hello world Example | 189 |
| Request & Response | 190 |
| Basic Routing | 191 |
| Introduction to Middleware Concepts | 195 |
| ExpressJS – Middleware..... | 195 |

| | |
|--|-----|
| Third Party Middleware | 198 |
| Express Routing | 200 |
| ExpressJS – Routing | 200 |
| Routers | 201 |
| Express JS Template Engine | 204 |
| ExpressJS – Templating | 204 |
| Important Features of Pug | 205 |
| Comments | 206 |
| Attributes | 207 |
| Passing Values to Templates | 207 |
| Conditionals | 208 |
| Include and Components | 210 |
| Handling Query string parameter | 212 |
| ExpressJS - URL Building | 212 |
| Cookies parser & body parser | 214 |
| ExpressJS – Cookies | 214 |
| Adding Cookies with Expiration Time | 215 |
| Deleting Existing Cookies | 216 |
| Body-parser middleware in Node.js | 216 |
| Session Handling | 221 |
| ExpressJS – Sessions | 221 |
| Express-mailer | 223 |

| | |
|--|------------|
| Installation..... | 223 |
| Usage..... | 223 |
| Sending an email | 224 |
| Database operation with MySQL | 225 |
| Setup Express js for Node.js MySQL | 225 |
| MySQL Installation | 225 |
| <input type="checkbox"/> Database | 228 |
| <input type="checkbox"/> Collection | 229 |
| <input type="checkbox"/> Document..... | 231 |
| <input type="checkbox"/> insertOne() | 232 |
| <input type="checkbox"/> insertMany() | 233 |
| JOIN | 242 |
| Import & export mongo database, Backup restore | 243 |
| Database operation with Node +Mongodb | 244 |

Learning Outcome

- Introduction to mean stack
- Introduction to Angular JS and UI benefits in Angular JS
- Usage of Angular JS with HTML
- Event Handling in Angular JS
- Introduction Angular js & difference b/w angular-1 and angular-5
- Type Script Datatype & Operator
- Type Script String & Tuple, Oops
- Object & Class
- Inheritance & Interface
- Angular js Child Component
- Angular js Data Binding, Event Binding, 2 Way Binding
- Angular js Pipe, Pipe Chaining
- Angular js Routing
- Angular js Services
- Angular js Http Request
- Node Introduction
- Blocking & Nonblocking code
- Create Module & export, import
- Introduction package.json file
- File Handling
- Create Event Driven Programming
- Socket Programming
- Create Own web server
- Create Web App using express
- Introduction Middleware Concepts
- Express Routing
- Express JS Template Engine
- Handling Query string parameter
- Cookies parser & body parser,
- Session Handling (express session), Mailer
- database operation with MySQL

- Introduction Mongo
- Features of Mongo
- Create database, collection, Document & Data
- Simple Query, Insert select data, filter data
- Capped collection, Update & delete Collection
- Index and Relationship, Aggregation & grouping, JOIN
- Import & export mongo database, Backup restore
- database operation with Node +Mongodb

Introduction to mean stack

MEAN Stack

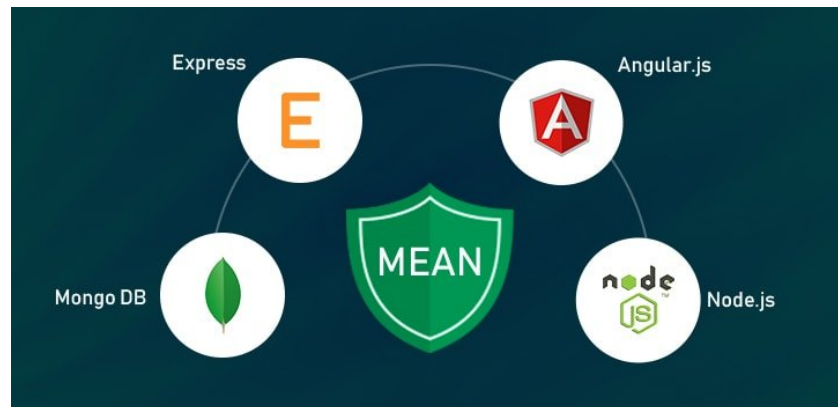


Image 1: MEAN Stack

Reference: <https://broadwayinfosys.com/blog/wp-content/uploads/2019/03/mean-stacks-main.jpg>

The MEAN stack is JavaScript-based framework for developing web applications. MEAN is named after **M**ongoDB, **E**xpress, **A**ngular, and **N**ode, the four key technologies that make up the layers of the stack.

- **M**ongoDB - document database
- **E**xpress(.js) - Node.js web framework
- **A**ngular(.js) - a client-side JavaScript framework
- **N**ode(.js) - the premier JavaScript web server

There are variations to the MEAN stack such as MERN (replacing Angular.js with React.js) and MEVN (using Vue.js). The MEAN stack is one of the most popular technology concepts for building web applications.

How Does the MEAN Stack Work?

MEAN Stack Architecture

The MEAN architecture is designed to make building web applications in JavaScript and handling JSON incredibly easy.

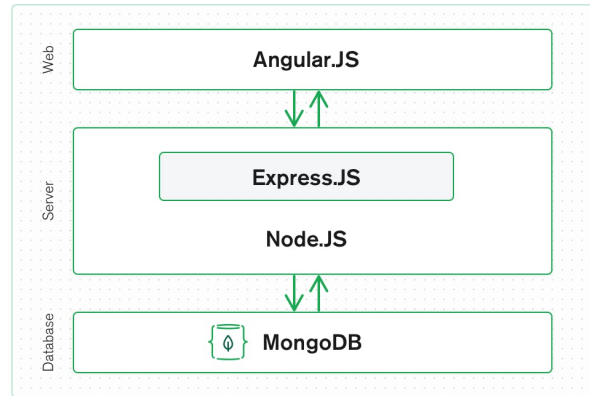


Image 2: MEAN Stack Architecture
Reference: <https://www.mongodb.com/mean-stack>

MEAN Stack Components

Angular.js Front End

At the very top of the MEAN stack is Angular.js, the self-styled “A JavaScript MVW Framework” (MVW stands for “Model View and Whatever”).

Angular.js allows you to extend your HTML tags with metadata in order to create dynamic, interactive web experiences much more powerfully than, say, building them yourself with static HTML and JavaScript (or jQuery).

Angular has all of the bells and whistles you’d expect from a front-end JavaScript framework, including form validation, localization, and communication with your back-end service.

Express.js and Node.js Server Tier

The next level down is Express.js, running on a Node.js server. Express.js calls itself a “fast, unopinionated, minimalist web framework for Node.js,” and that is indeed exactly what it is.

Express.js has powerful models for URL routing (matching an incoming URL with a server function), and handling HTTP requests and responses. By making XMLHttpRequests (XHRs), GETs, or POSTs from your Angular.js front end, you can connect to Express.js functions that power your application.

Those functions in turn use MongoDB's Node.js drivers, either via callbacks or using Promises, to access and update data in your MongoDB database.

MongoDB Database Tier

MongoDB is a NoSQL database system. It is a cross-platform, documented oriented database which works on the concepts and the documents. It saves data in a binary format generally using JSON format which makes it easier to pass data between the client and the server.

If your application stores any data (user profiles, content, comments, uploads, events, etc.), then you're going to want a database that's just as easy to work with as Angular, Express, and Node.

That's where MongoDB comes in: JSON documents created in your Angular.js front end can be sent to the Express.js server, where they can be processed and (assuming they're valid) stored directly in MongoDB for later retrieval.

Again, if you want to easily get the best of MongoDB, you'll want to look at [MongoDB Atlas](#).

Benefits of using Mean Stack

- It is often updated as it has open-source components.
- Developing web applications with Node.js is easy as it provides a good variety of JavaScript modules.
- Mean stack gives the developer a freedom to write code in only one language (JavaScript) for both client and server side that makes it a simple and fast language.
- Precisely flowing data among the layers of JSON does not require rewriting or reformatting as Mean uses common JSON format of data everywhere.
- Transfer of code from a particular framework to another framework is easy with Mean. Mean stack has become a leading end technology recently.
- Mean is highly flexible. Testing application on the cloud platform is easier after the successful completion of its development process. You can add extra information by just adding an extra field to your form.

- Development of apps with mean stack is less costly as it just requires developers who are proficient at javascript.
- Mean assists in the current period to develop real-time trending applications. Since it utilizes single-page applications (SPAs), There is no need to constantly update web pages for each server request.
- Mean saves a lot of time in developing applications because it has an infinite set of module libraries for Node.js which are ready for use.
- There is no time waste in creating modules from scratch. Time management is a huge benefit while in another hand, It also creates quality world-class applications.
- Mean supports MVC (Model View Controller) architecture.
- Mean Stack is very easy and flexible to understand which helps the developers to customize it as per requirements.
- Mean stack provides developers the feature to make website interactive by working on its appearances.
- In the back-end development too, the developers making the website more functional and is also provided by the mean stack.

Mean stack is improving daily and it is also easy to use. The leading development companies use Mean Stack for developing top mobile apps as Mean Stack is listed as the best technology for developing mobile apps. For most innovative web applications this is the most suitable technology.

Disadvantages of Mean Stack

JavaScript is a modern language, and it wasn't initially designed to build backend servers. As the foundation of the MEAN stack is JavaScript, it adds a backend server that comes with concurrency and issues with performance that scales up due to JavaScript nature. As the development opportunities are rapid, the server logic would suffer from potential code and bad practices. The tool won't add concrete JS coding in the final terms, which is appropriate for the Stack. Which ultimately is suitable for one application might not work better for another.

- Once you develop the first site using a mean stack, it's hard to go back to the old approach.
- We don't find any specific general JS coding guidelines.
- It offers poor isolation of the server, therefore, chances of losing data.

How secure is the MEAN stack?

We have a team of developers who highly recommend using Mean Stack with MongoDB Atlas. As Atlas comes with built-in credentials and end-to-end encryptions, it's the best foundation for securing MongoDB. Additionally, Mean Stack comes with concrete three-tier separation using best practices and correct network isolation. There are times when it prevents end-users from accessing the business logic and database layer. The application designed using this tool is the default to avoid malicious user interaction from putting your application at risk.

Why choose Mean Stack?

- Scalable and Flexible
- Excellent Speed
- One Stack, one language
- Free of Cost
- User-Friendly
- Avoids Rewriting

Introduction to Angular JS and UI benefits in Angular JS

Introduction to Angular JS

AngularJS is an open-source Model-View-Controller framework (MVC) that is similar to the JavaScript framework. AngularJS is probably one of the most popular modern-day web frameworks available today. This framework is used for developing mostly Single Page applications. This framework has been developed by a group of developers from Google itself.

It can be added to an HTML page with a `<script>` tag. It enables us to create single-page applications that only require HTML, CSS, and JavaScript on the client side.

AngularJS extends HTML attributes with Directives and Data binding to HTML with Expressions.

Angular Version History

- Google developed this web application framework in 2009. It is officially called AngularJS. Some people call this version Angular 1.0. This version came out on October 20, 2010.
- After Angular 1.0, in 2016, Angular 2 was released. It was written from scratch and is fully different from Angular 1 or JS.
- The third update, Angular 4, was launched in the year 2017. It is not a complete rewrite of the original version. Instead, it is the updated version of Angular 2.
- Angular 5 was released on 1st November 2017. The updates in this version help developers to create apps fast, as it removes unnecessary codes.
- Angular 6 was released on 3rd May 2018, and the version of Angular 7 was out in October 2018.
- Angular Team released Angular 8 on May 28, 2019. It features differential loading of all application code, web workers, supports TypeScript 3.4, dynamic imports for lazy routes, and Angular Ivy as an opt-in preview.
- Angular Team released Angular 9 on Feb 06, 2020. By default, version 9 moves all the applications to use the Ivy compiler and runtime.
- Angular 10 was released on Jun 24, 2020, with four months difference from the previous release.
- The latest angular version from Google is Angular 11. It was made available in the market from 14th November 2020.

AngularJS Architecture

Angular.js follows the MVC architecture, the diagram of the MVC framework as shown below:

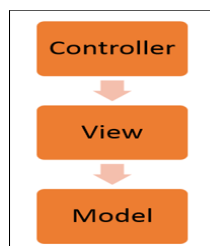


Image 3: Angular JS Architecture

Reference: https://www.guru99.com/images/AngularJS/010416_0549_AngularJSin1.png

- The **Controller** represents the layer that has the business logic. User events trigger the functions which are stored inside your controller. The user events are part of the controller.
- **Views** are used to represent the presentation layer which is provided to the end users
- **Models** are used to represent your data. The data in your model can be as simple as just having primitive declarations. For example, if you are maintaining a student application, your data model could just have a student id and a name. Or it can also be complex by having a structured data model. If you are maintaining a car ownership application, you can have structures to define the vehicle itself in terms of its engine capacity, seating capacity, etc.

Concepts of AngularJS

Few things you need to know to before starting with AngularJS:

- **Modules** - A module can be defined as a container that consists of various application parts. The module is a set of functions defined in a JavaScript file. Module divides an application into small and reusable components.
- **Directives** -Directives indicate the compiler to associate a behavior to the DOM element or modify it. Angular JS contains several directives such as ng-app, ng-controller, ng-view, ng-if, etc.
- **Expressions** -Angular JS expressions are expressed with {{ }} which indicate a data binding in HTML. These expressions can be added into the HTML templates. Expressions do not support control flow statements while support the filters.
- **Controller**- It is a JavaScript object constructor function that controls the AngularJS applications.
- **Scope**- It is a JavaScript object that acts as a bridge between the Controller and the View. It is the source of data in AngularJS. Each data manipulation and assignment takes place with the help of the Scope object.
- **Data Binding**- It coordinates the model and views any changes in either of these two.

- **Validations**- Validations take place with the help of AngularJS forms and controls.
- **Filters** -These let you display the formatting of data on DOM in Angular and extend the behavior of directives and binding expressions. Filters format the values or apply specific.
- **Services** -These are singletons that are used by directives, controllers, or other services.
- **Routing** -The service `$routeProvider` handles the operations of Routing. It divides the map into various views. It helps split the Single Page Applications into different views.
- **Dependency Injection**-It is a design pattern used to handle the dependencies of various components of a software. It lets you develop loosely-structured architectures.
- **Testing** -The codes developed by Dependency Injections are tested. Some of the popular testing frameworks like Jasmine and Karma are two widely-used technologies.

Advantages of AngularJS

- It provides the capability to create Single Page Application in a very clean and maintainable way.
- It provides data binding capability to HTML. Thus, it gives user a rich and responsive experience.
- AngularJS code is unit testable.
- AngularJS uses dependency injection and make use of separation of concerns.
- AngularJS provides reusable components.
- With AngularJS, the developers can achieve more functionality with short code.
- In AngularJS, views are pure html pages, and controllers written in JavaScript do the business processing.

On the top of everything, AngularJS applications can run on all major browsers and smart phones, including Android and iOS-based phones/tablets.

Disadvantages of AngularJS

- Not Secure – Being JavaScript only framework, application written in AngularJS are not safe. Server-side authentication and authorization is must to keep an application secure.
- Not degradable – If the user of your application disables JavaScript, then nothing would be visible, except the basic page.
- Complex at times – At times AngularJS becomes complex to handles as there are multiple ways to do the same thing. This creates confusion and requires considerable efforts.

General features

- We can create rich internet application by using Angular JS.
- Angular JS Allow us to write the client-side code using Javascript in an MVC way.
- AngularJS provides cross-browser compliant and it automatically handles JavaScript code suitable for each browser.
- The AngularJS is Completely free and opensource it is used by thousands of developers all around the world.
- AngularJS is used to build high-performance, large scale, and easy-to-maintain web applications.
- AngularJS Directives: AngularJS is Javascript based framework and it may divide into three major part.
 - **ng-app**: By using this directive we can link AngularJS application to HTML.
 - **ng-model**: By using this directive we can bind the value of AngularJS application data to HTML input controls.
 - **ng-bind**: By using this directive we can bind AngularJS application data to HTML tags.

UI benefits in Angular JS

The AngularJS framework employs basic HTML and offers extensions through directives that allow for the website to become more dynamic. Its capabilities to automatically synchronize with models and views make development on AngularJS an easy process. It exclusively follows DOM methodology, whose primary focus lies in improving application testability & performance. Therefore, the main features of AngularJS

comprise; two-way data binding, templates, MVC structure, dependency injections, directives, and testing features.

Let's take a look at some of the reasons for why AngularJS development companies leverage this framework for front-end product engineering

Provides immense support to XAML Development

XAML is an XML based markup language which is used to instantiate object graphs and set values. It allows you to define various types of objects with properties. It makes it easy to layout complex, ever changing UIs. It also supports for data-binding which allows a symbiosis between the presentation layer and your data without creating hard dependencies between its components. It also enables a developer to conduct any number of testing scenarios. AngularJS translates very well with XAML principles. It also allows a parallel workflow between different aspects that include the markup for the UI itself as well as the underlying logic that fetches and processes data.

Provides a simplistic approach to data binding

Data binding is very easy in the Angular world. The framework eliminates the need to derive from an existing object or place all your properties and dependencies cards on the table. AngularJS uses dirty tracking to enable this. Though several existing frameworks have evolved, the process of mapping everything explicitly to an interim object to data-bind to Angular is significantly easier and faster.

Reduces application side-effects by allowing developers to express declarative UI

Declarative UI has several advantages. A structured UI enables easy understanding and manipulation of the application. Using jQuery forces the developer to know a lot about the document structure which often creates two issues: first, a fairly unstable code working as “glue” that tightly grips the changes in the UI, and second, there is an uncertainty because it is hard to judge by studying the markup just how the UI would function. Placing markup directly in HTML, one can separate the presentation logic from imperative logic and keep it in one place. Understanding of the extended markup provided by Angular i.e. code snippets makes the whereabouts of the data amply clear. The addition, tools like directives and filters not only make the UI intent even more clear but also give clarity on how the information is being shaped.

Simplifies testing

AngularJS product engineering adeptly embraces Test-Driven Development, Behavior-Driven Development, or any of the driven-development methodologies of building an application. It helps in saving time and change the way an application is structured. Angular can help to test everything UI behavior to business logic with its ability to mock dependencies.

Gives Design development workflow a new meaning

Even though HTML and CSS support design, but AngularJS allows a developer to add a markup without completely breaking an application. It often depends on a certain id or structure to locate an element and perform tasks. Developers can rearrange portions of code by moving the elements around and the corresponding code that does the binding and filtering job moves with it.

Makes Single Page Applications easy

The growing popularity of Single Page Applications among AngularJS development companies is not unfounded as they cater to a very specific need. With more and more functionality finding its way to the web, developers are progressively realizing the potential of the browser as a distributed computing node. SPA boasts of more responsive design and can provide an experience of a native app on the web. AngularJS is an apt infrastructure that supports routing, templates, and even journaling making it viable accomplice of SPA.

Here are some additional advantages to leverage for awesome front-end product engineering:

- **Improved Plug & Play Features** – AngularJS makes it easy to add components from an existing application to a new application. It only needs a copy-paste command to make your existing assets available in a new environment.
- **Quicker Development Turnaround Time** – AngularJS supported by the MVC architecture ensures faster development, testing, and maintenance. The quicker turnaround time allows developers to enhance their productivity.

- **Superior Dependency Handling** – One of the biggest USPs of AngularJS is its “dependency injection”. Testing and Single Page Application have been enormously simplified because of the dependency injection feature.
- **Makes Parallel Development Possible** – Apart from faster development, AngularJS in collaboration with MVC architecture allows the developer to perform parallel application development which makes it stand-out amongst the competition.
- **Gives More Control to the Developers** – AngularJS directives give superior control to developers, who get a free hand to experiment with HTML and attributes. The directives allow them more independence to help them create more responsive platforms.
- **State Management Made Easy** – AngularJS helps you manage any application state easily whether it is illusioned or disillusioned. It is highly conducive when it comes to managing properties, permissions, and other major concerns across the application.

Usage of Angular JS with HTML

AngularJS is a JavaScript framework. It can be added to an HTML page with a <script> tag. AngularJS extends HTML attributes with Directives, and binds data to HTML with Expressions.

AngularJS is a JavaScript framework written in JavaScript. AngularJS is distributed as a JavaScript file, and can be added to a web page with a script tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>
```

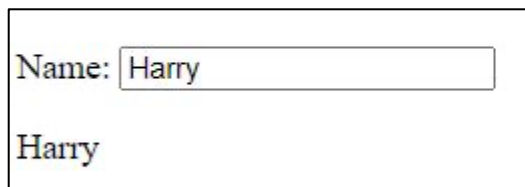
AngularJS Extends HTML

AngularJS extends HTML with ng-directives.

- The ng-app directive defines an AngularJS application.
- The ng-model directive binds the value of HTML controls (input, select, textarea) to application data.
- The ng-bind directive binds application data to the HTML view.

AngularJS Example

```
1 <!DOCTYPE html>
2 <html>
3 <script src="https://ajax.googleapis.com/ajax/libs/
  angularjs/1.6.9/angular.min.js"></script>
4 <body>
5
6 <div ng-app="">
7   <p>Name: <input type="text" ng-model="name"></p>
8   <p ng-bind="name"></p>
9 </div>
10
11 </body>
12 </html>
```



Name:

Harry

Explanation:

AngularJS starts automatically when the web page has loaded.

- The ng-app directive tells AngularJS that the <div> element is the "owner" of an AngularJS application.
- The ng-model directive binds the value of the input field to the application variable name.
- The ng-bind directive binds the content of the <p> element to the application variable name.

AngularJS Directives

As you have already seen, AngularJS directives are HTML attributes with an ng prefix.

The ng-init directive initializes AngularJS application variables.

```

1 <!DOCTYPE html>
2 <html>
3 <script src="https://ajax.googleapis.com/ajax/libs/
  angularjs/1.6.9/angular.min.js"></script>
4 <body>
5
6 <div ng-app="" ng-init="firstName='Harry'">
7
8 <p>The name is <span ng-bind="firstName"></span></p>
9
10 </div>
11
12 </body>
13 </html>
14

```

The name is Harry

AngularJS Expressions

AngularJS expressions are written inside double braces: {{ expression }}.

```

1 <!DOCTYPE html>
2 <html>
3 <script src="https://ajax.googleapis.com/ajax/libs/
  angularjs/1.6.9/angular.min.js"></script>
4 <body>
5
6 <div ng-app="">
7
8 <p>Input something in the input box:</p>
9 <p>Name: <input type="text" ng-model="name"></p>
10 <p>{{name}}</p>
11
12 </div>
13
14 </body>
15 </html>

```

AngularJS expressions bind AngularJS data to HTML the same way as the ng-bind directive.

The ng-bind directive tells AngularJS to replace the content of an HTML element with the value of a given variable, or expression.

If the value of the given variable, or expression, changes, the content of the specified HTML element will be changed as well.

```
1 <!DOCTYPE html>
2 <html>
3 <script src="https://ajax.googleapis.com/ajax/libs/
  angularjs/1.6.9/angular.min.js"></script>
4 <body>
5
6 <div ng-app="" ng-init="myText='Hello World!'">
7
8 <p ng-bind="myText"></p>
9
10 </div>
11
12 </body>
13 </html>
```

Event Handling in Angular JS

AngularJS Events

AngularJS includes certain directives which can be used to provide custom behavior on various DOM events, such as click, dblclick, mouseenter etc.

The following table lists AngularJS event directives.

| Event Directive |
|-----------------|
| ng-blur |
| ng-change |
| ng-click |
| ng-dblclick |
| ng-focus |
| ng-keydown |
| ng-keyup |
| ng-keypress |
| ng-mousedown |
| ng-mouseenter |
| ng-mouseleave |
| ng-mousemove |
| ng-mouseover |
| ng-mouseup |

Image 4: Angular JS Event Directive

Reference: https://www.guru99.com/images/AngularJS/010416_0549_AngularJSin1.png

- **ng-mousemove**: Movement of mouse leads to the execution of event.
- **ng-mouseup**: Movement of mouse upwards leads to the execution of event.
- **ng-mousedown**: Movement of mouse downwards leads to the execution of event.
- **ng-mouseenter**: Click of the mouse button leads to the execution of event.
- **ng-mouseover**: Hovering of the mouse leads to the execution of event.
- **ng-cut**: Cut operation leads to the execution of the event.
- **ng-copy**: Copy operation leads to the execution of the event.
- **ng-keypress**: Press of key leads to the execution of the event.
- **ng-keyup**: Press of upward arrow key leads to the execution of the event.
- **ng-keydown**: Press of downward arrow key leads to the execution of the event.
- **ng-click**: Single click leads to the execution of the event.
- **ng-dblclick**: Double click leads to the execution of the event.

Let's take a look at some of the important event directives.

ng-click

The ng-click directive is used to provide event handler for click event.

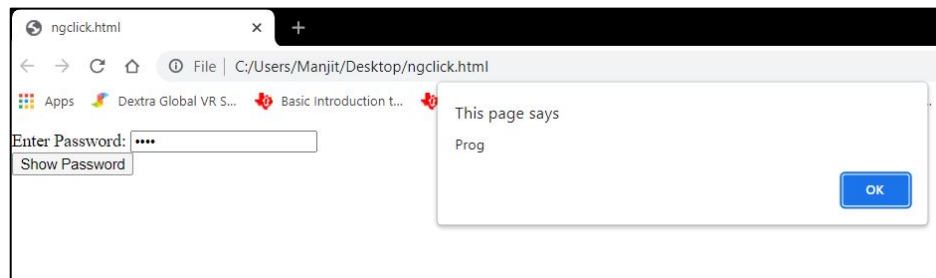
```
<!DOCTYPE html>
<html>
<head>
  <!--<script src="~/Scripts/angular.js"></script>-->
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.16/angular.min.js"
"></script>

</head>
<body ng-app="myApp">
  <div ng-controller="myController">
    Enter Password: <input type="password" ng-model="password" /> <br />

    <button ng-click="DisplayMessage(password)">Show Password</button>
  </div>
  <script>
    var myApp = angular.module('myApp', []);

    myApp.controller("myController", function ($scope, $window) {

      $scope.DisplayMessage = function (value) {
        $window.alert(value)
      }
    });
  </script>
</body>
</html>
```



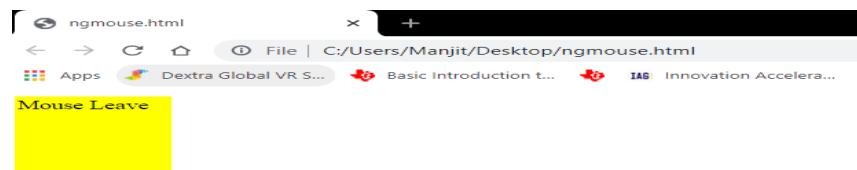
In the above example, ng-click directive is used to call a DisplayMessage() function with the 'password' parameter when a user clicks a button. A 'password' is a model property defined using ng-model directive in the input box. The DisplayMessage() function is attached to a \$scope object in myController, so it will be accessible from button click as button comes under myController. The \$window service is used to display an alert.

Mouse Events

The following example demonstrates important mouse event directives - ng-mouseenter and ng-mouseleave.

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.16/angular.min.js"></script>
<style>
    .redDiv {
        width: 100px;
        height: 100px;
        background-color: red;
        padding: 2px 2px 2px 2px;
    }

    .yellowDiv {
        width: 100px;
        height: 100px;
        background-color: yellow;
        padding: 2px 2px 2px 2px;
    }
</style>
</head>
<body ng-app>
    <div ng-class="{redDiv: enter, yellowDiv: leave}" ng-mouseenter="
        enter=true;leave=false;" ng-mouseleave="leave=true;enter=false">
        Mouse <span ng-show="enter">Enter</span> <span ng-show="leave">Leave</span>
    </div>
</body>
</html>
```



Introduction Angular & difference b/w angular-1 and angular-5

Introduction Angular

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.

Angular has many improvements over AngularJS. It has lots of innovations, which makes it easy to learn and develop enterprise-scale applications. You can build extendable, Maintainable, Testable and Standardized Applications using Angular.

Features of Angular

- Two-Way Data Binding - This is the coolest feature of the Angular. Data binding is automatic and fast. changes made in the View is automatically updated in the component class and vice versa
- Powerful Routing Support - The Angular Powerful routing engine loads the page asynchronously on the same page enabling us to create a Single Page Applications.
- Expressive HTML - Angular enables us to use programming constructs like if conditions, for loops, etc to render and control how the HTML pages.
- Modular by Design - Angular follows the modular design. You can create Angular modules to better organize and manage our codebase
- Built-in Back End Support - Angular has built-in support to communicate with the back-end servers and execute any business logic or retrieve data
- Active Community - Angular is Supported by google and has a very good active community of supporters. This makes a lot of difference as your queries are quickly resolved.

Angular has changed massively from the AngularJS. Angular completely redesigned from scratch. There are many concepts AngularJS that have changed in Angular.

Top 12 Angular Features:

1. Cross-platform

It is imperative to have this factor first on the Angular features list because Angular plays a prominent role in developing Progressive Web Applications (PWA). With PWA by your side, audiences can enjoy an app-like experience using contemporary web capabilities. Most importantly, with this feature, you can deploy a local or a progressive app.

2. Efficient MVC architecture

The Model-View-Controller (MVC) architecture is among the leading Angular 8 features. MVC enhances the worth of the framework for client-side application development and takes care of other features like data binding and scopes. As compared to other frameworks, MVC blends in all the necessary elements of the application to waive off extra coding.

3. Sectional structure

The best part about the Angular framework is it helps in organizing code into different modules as and when you build them. Due to this feature, there is a division of overall functionality into reusable code. Furthermore, this helps in the division of tasks among the Angular developers and permits web applications to accomplish lazy loading.

4. The eminence of Angular CLI

Angular's Command Line Interface (CLI) is a boon for web development. CLI helps in automating certain processes with simple commands. You may add or remove defined functionalities with a combination of these simple commands. In addition, it enables running units' tests and end-to-end tests swiftly. All these AngularJS features enhance the code quality considerably.

5. Data binding

With data binding by your side, a user can easily manipulate web page elements through a web browser. It utilizes dynamic HTML and waives off intricate scripting or programming. Data binding plays a vital role when coming up with web pages equipped with interactive components, like games, tutorials, etc. Also, when a web page has way too much data, it enables a better display.

Besides, a model is a place where you make edits in the application. Those changes are reflected in UI elements. When the model changes, the developer supposedly has to make manual edits in the DOM elements, and then the attributes start reflecting. It is a cumbersome task, but Angular eliminates it with two-way data-binding. This process helps in synchronizing the DOM and the model, and vice versa.

6. Set of Directives

AngularJS expands the functionality of HTML with a set of inbuilt attributes, also known as directives. The imperative functionality of these directives is to boost the competence of HTML. With this, it becomes immensely appropriate for dynamic client-side applications. The best part is these directives can be self-initiated using AngularJS.

7. Prominence of TypeScript

TypeScript is the superscript of JavaScript. The main advantage of using TypeScript is you can detect and correct errors in the code while writing. It also supports AngularJS security features like primitive and interface. Interestingly, Angular is written using TypeScript and boasts all these features.

8. Declarative UI

One of the AngularJS key features is declarative UI. In Angular, you can skip using JavaScript to outline the UI of your web application and instead use HTML as it is less complicated than JavaScript. HTML is a blessing for Angular applications as it imports declarative and intuitive properties of the UI components. With properties like these, you don't have to initiate manual program flows. Instead, you can simply describe the page layout and the path of the data. Further, Angular declarative UI takes care of the components as per the layout. With this, ample time and effort are saved in front-end development.

9. Extensive documentation and support

Google supports Angular because of the stability of the framework. Most importantly, Google has enriched the Angular community with documentation and tools to build functionalities and resolve issues. Due to these reasons, a developer is never helpless in the Angular community. The answers are readily available either in the documents or in the online forums.

10. The prowess of Ivy Renderer

A renderer helps in translating a code written in TypeScript and HTML into standard JavaScript instructions. With this, it helps the browser to interpret. Renderer makes your components and templates understandable for the browser to display them aptly.

Furthermore, with its tree shaking technique, Ivy Renderer removes unused code, making the web application smaller and boosting its loading speed. All in all, this feature reduces the size of the Angular framework and makes the bundle a bit smaller.

11. Test-friendliness

JavaScript is an interpreted language, which makes it imperative for developers to test the code capability. But while using AngularJS, developers are unhindered by this requirement. The framework has features like Dependency Injection (DI) that supports testing. DI simplifies the entire process where the testers must insert the test data in the controller and parallelly check the output. It's that simple!

12. Timely upgradations

If you don't upgrade your framework from time to time, you may lag in this race of the digital world. New settings pour in, and if they are absent, your application will be tagged as outdated. To eliminate this problem, Angular improves its Component Development Kit (CDK) on a timely basis. It not only enhances but also takes care of the angular version upgrades.



Image 5: Angular Features

Reference: <https://qph.fs.quoracdn.net/main-qimg-3d2a60f972904b5dd34a3f99a31dc0c2-lq>

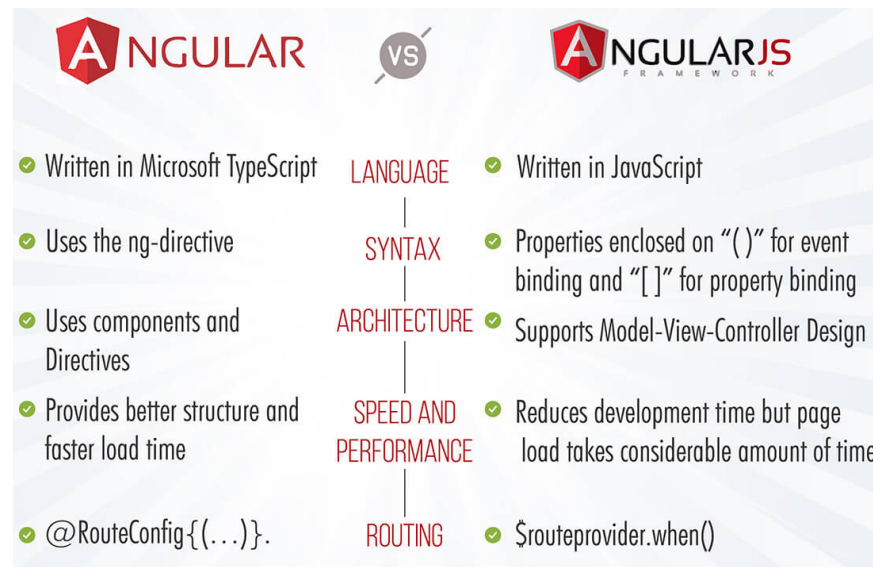


Image 6: Difference between Angular Vs AngularJS

Reference: <https://qph.fs.quoracdn.net/main-qimg-3d2a60f972904b5dd34a3f99a31dc0c2-lq>

Likewise, let's take a look at the more detailed distinction between AngularJS and Angular:

1. Programming Language

One vital difference between Angular and AngularJS is that Angular is based on Typescript Language and Angular JS is Javascript based. Angular can also be used in other languages such as ES5, ES6, and Dart to write codes. On the other hand, AngularJS is solely Javascript-based. Still, developers can use the features such as declarative template language using HTML, which is intuitive.

2. Expression Syntax

While both Angular and AngularJS use directives, Angular has a standard directive, and Angular has a pack of them. When it comes to data binding, Angular is more intuitive than AngularJS. Angular uses () and [] to bind data and attributes between view and model. In

the case of AngularJS, the ng-model is used to bind data. `{{ }}` expressions are applied for two-way binding among view and model.

3. Web Architecture

AngularJS supports Model-View-Controller design that acts as the main component in managing data, rules, logic and expresses how the application behaves. In addition, the Model-view-controller design processes the data presented in the model to produce the output.

On the other hand, Angular uses components and directives with templates. Angular directives are divided into two kinds, structural directives and attribute directives. Structural directives alter the DOM's layout by changing its elements. Meanwhile, attributes directive changes DOM's behavior and appearance.

4. Speed and Performance

Since Angular provides a better structure, the framework offers improved performance and structure. In addition, it's easier to create and maintain big applications. To date, Angular 4 is the fastest version of the framework.

Likewise, Angular JS also reduces development effort and time with the two-way binding features. However, creating more processing on the client-side of programs can slow down page load time.

5. Routing

AngularJS uses `$routeProvider.when()` to provide routing information and configuration. On the other hand, the Angular framework uses a simple path. Developers can use URLs to imitate directives to get to the client view or use `@RouteConfig{(...)}` for routing information, giving this framework an edge over AngularJS.

6. Mobile Friendliness

When it comes to mobile-friendliness, AngularJS does support mobile while Angular is great for mobile. Due to its dynamic web and single-page scripts, Angular is exceptionally mobile-friendly. On the other hand, AngularJS's simple architecture and code cannot support mobile applications.

7. Dependency Injection

AngularJS doesn't employ dependency injection and uses directives instead. Angular uses unidirectional change detection and hierarchical dependency injection to boost the framework's performance.

8. Management

Angular projects are more straightforward to manage than AngularJs due to how structured it is. AngularJS can be difficult to work with as the size of the source code increases. On the other hand, Angular code has a better structure. It is easy to scale and manage as the application becomes more extensive. This is a great advantage when developing big applications.

9. Testing and Tools

AngularJS includes a ready unit testing feature called IDE and Webstorm, a third-party JavaScript tool. These tools can be used by developers in building applications and finding defects in their designs. On the other hand, Angular uses Command Line Interface (CLI) for developing and serving angular applications. It helps developers in reducing development time, creating and testing accessibility, and more.

10. SEO

When it comes to web optimization strategies, Angular has inbuilt extensions for rendering server-side applications. This lets developers synchronize both server-side and client content, which is excellent for SEO. In the past, Google did not crawl properties of pages made in JavaScript. But in 2014, Google updated its AI and announced the

inclusion of JavaScript. Today, AngularJS can also be SEO friendly but requires processes like avoiding using # in URLs, using a limited number of embedded resources, and more.

Difference b/w Angular-1 to Angular-5

| Angular 1 | Angular 5 |
|--|---|
| Released by Google in the year 2010. | Released by Google in the year 2017. |
| JavaScript-based framework for creating SPA. | Angular 5 comes with build optimizer which is a part of the platform's command like a tool. |
| Still supported but no longer will be developed. | |
| The architecture of AngularJS is based on MVC. | |
| AngularJS was not developed with a mobile base in mind. | |
| AngularJS code can write by using only ES5, ES6, and Dart. | Compiler Improvements |
| Based on controllers whose scope is now over. | New Router Lifecycle Event |
| Factory, service, provider, value and constant are used for services | Optimization with HTTP Client Feature Internationalized Date & Currency |
| Run on only client-side | |
| ng-app and angular bootstrap function are used to initialize | |

Angular JS

AngularJS is the official name, but some developers also refer to this as Angular 1. It is a front-end and open-source web application framework based on JavaScript. It uses HTML as a template in this framework. In AngularJS, the data and expressions are merged to create an expressive environment for developing web applications quickly. It simplifies both the testing and development of applications by providing a framework for client-side

model–view–controller (MVC) and Model–View–ViewModel (MVVM) architectures, along with components commonly used in rich Internet applications.

It uses the controller approach where the view communicates using a \$scope.

Angular 5

Angular 5 is more advanced and has more enhanced features than Angular 4. The best feature of Angular 5 is that it aids developers in removing unnecessary codes from their applications.

Other improved features are a code-sharing feature, less time for assembling dynamic web applications, and so on. Moreover, it has DOM support, and its compiler helps with incremental compilation.

| Parameters | AngularJS | Angular 2 | Angular 4 |
|-------------------|---|--|---|
| Origin | It is an open-source front-end framework used for developing dynamic web apps | Angular 2 is an open-source front-end web application framework | Angular 4, an enhanced version of AngularJS, is an open-source web application platform |
| Architecture | AngularJS follows the Model-View-Controller architecture | Angular 2 is based-on the component-service architecture | Angular 4 is based on structural-directives |
| Language used | AngularJS uses JavaScript | Angular 2 uses Microsoft's TypeScript | Angular 4 uses the latest versions of TypeScript |
| Expression syntax | In AngularJS, we need to remember the accurate ng directive to add images, property, and events | Angular 2 follows parentheses () for event binding and square brackets [] for property binding | Angular 4 follows advanced syntax, i.e., ngIf and ngFor directives |
| Performance | Slow performance | Faster performance | Improved performance by dependency injection |
| Routing | The \$routeProvider.when() method is used for routing configuration | @RouterConfig({...}) is used by Angular 2 for routing configuration | It uses two routing methods: RouterModule.forRoot() and RouterModule.forChild() |

Image 7: Difference between Angular 1 TO 4

Reference: <https://qph.fs.quoracdn.net/main-qimg-3d2a60f972904b5dd34a3f99a31dc0c2-lq>

Typescript Datatype and Operator

Transcript Datatype

Whenever a variable is created, the intention is to assign some value to that variable but what type of value can be assigned to that variable is dependent upon the datatype of that Variable. In typeScript, type System represents different types of datatypes which are supported by TypeScript. The data type classification is as given below:

Built-in Datatypes: TypeScript has some pre-defined data-types-

| Built-in Datatype | Keyword | Description |
|-------------------|---------|--|
| Number | number | It is used to represent both Integer as well as Floating-Point numbers |
| Boolean | boolean | Represents true and false |
| String | string | It is used to represent a sequence of characters |
| Void | void | Generally used on function return-types |
| Null | null | It is used when an object does not have any value |

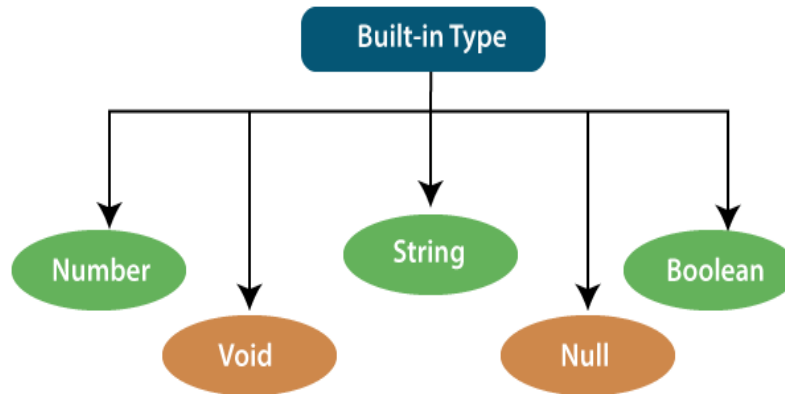


Image : Built-in Datatype

Reference: <https://static.javatpoint.com/tutorial/typescript/images/typescript-types2.png>

Number

Just like JavaScript, TypeScript supports number data type. All numbers are stored as floating-point numbers. These numbers can be Decimal (base 10), Hexadecimal (base 16) or Octal (base 8).

```

let first:number = 123; // number

let second: number = 0x37CF; // hexadecimal

let third:number=0o377 ;    // octal

let fourth: number = 0b111001;// binary

console.log(first); // 123

console.log(second); // 14287

console.log(third); // 255

console.log(fourth); // 57
  
```

In the above example, `let first:number = 1;` stores a positive integer as a number. `let second: number = 0x37CF;` stores a hexadecimal as a number which is equivalent to 14287. When you print this number on your browser's console, it prints the equivalent

floating point of the hexadecimal number. `let third:number=0377;` stores an octal number equivalent to 255.

Boolean

Boolean values are supported by both JavaScript and TypeScript and stored as true/false values.

TypeScript Boolean:

```
let isPresent:boolean = true;
```

Note that, the boolean `Boolean` is different from the lower case `boolean` type. The upper-case `Boolean` is an object type whereas lower case `boolean` is a primitive type. It is recommended to use the primitive type `boolean` in your code, because, while JavaScript coerces an object to its primitive type, the TypeScript type system does not. TypeScript treats it like an object type.

So, instead of using upper case function `checkExistence(b: Boolean)`, use the lower case function `checkExistence(b: boolean)` `boolean` type.

String

String is another primitive data type that is used to store text data. String values are surrounded by single quotation marks or double quotation marks.

```
let employeeName:string = 'John Smith';  
  
//OR  
  
let employeeName:string = "John Smith";
```

Void

Similar to languages like Java, void is used where there is no data. For example, if a function does not return any value then you can specify void as return type.

```
function sayHi(): void {  
    console.log('Hi!')  
}  
  
let speech: void = sayHi();  
console.log(speech); //Output: undefined
```

There is no meaning to assign void to a variable, as only null or undefined is assignable to void.

```
let nothing: void = undefined;  
let num: void = 1; // Error
```

Null

TypeScript has a powerful system to deal with null or undefined values. Null and undefined are primitive types and can be used like other types, such as string.

```
let value: string | undefined | null = null;  
console.log(typeof value);  
  
value = 'hello';  
console.log(typeof value);
```

```
value = undefined;  
console.log(typeof value);
```

Output:

```
object  
string  
undefined
```

Typescript Operator

An Operator is a symbol which operates on a value or data. It represents a specific action on working with data. The data on which operators operates is called operand. It can be used with one or more than one values to produce a single value. All of the standard JavaScript operators are available with the TypeScript program.



Image : Operators in Typescript

Reference: <https://discoversdkcdn.azureedge.net/postscontent/typescript/operators%20in%20typescript.png>

Example

```
10 + 10 = 20;
```

In the above example, the values '10' and '20' are known as an operand, whereas '+' and '=' are known as operators.

In TypeScript, an operator can be classified into the following ways:

- Arithmetic operators
- Comparison (Relational) operators
- Logical operators
- Bitwise operators
- Assignment operators
- Ternary/conditional operator
- Concatenation operator
- Type Operator

Arithmetic Operators

Arithmetic operators take numeric values as their operands, performs an action, and then returns a single numeric value. The most common arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

| Operator | Operator_Name | Description | Example |
|----------|---------------|--|---|
| + | Addition | It returns an addition of the values. | <pre>let a = 20; let b = 30; let c = a + b; console.log(c); //</pre> Output 30 |
| - | Subtraction | It returns the difference of the values. | <pre>let a = 30; let b = 20; let c = a - b; console.log(c); //</pre> Output |

| | | | |
|----|----------------|---|---|
| | | | 10 |
| * | Multiplication | It returns the product of the values. | <pre>let a = 30; let b = 20; let c = a * b; console.log(c); //</pre> <p>Output</p> <p>600</p> |
| / | Division | It performs the division operation, and returns the quotient. | <pre>let a = 100; let b = 20; let c = a / b; console.log(c); //</pre> <p>Output</p> <p>5</p> |
| % | Modulus | It performs the division operation and returns the remainder. | <pre>let a = 95; let b = 20; let c = a % b; console.log(c); //</pre> <p>Output</p> <p>15</p> |
| ++ | Increment | It is used to increments the value of the variable by one. | <pre>let a = 55; a++; console.log(a); //</pre> <p>Output</p> |

| | | | |
|----|-----------|--|---|
| | | | 56 |
| -- | Decrement | It is used to decrements the value of the variable by one. | let a = 55; a--; console.log(a); // Output 54 |

Comparison (Relational) Operators

The comparison operators are used to compares the two operands. These operators return a Boolean value true or false. The important comparison operators are given below.

| Operator | Operator_Name | Description | Example |
|----------|--|---|---|
| == | Is equal to | It checks whether the values of the two operands are equal or not. | let a = 10; let b = 20; console.log(a==b); //false console.log(a==10); //true console.log(10=='10'); //true |
| === | Identical (equal and of the same type) | It checks whether the type and values of the two operands are equal or not. | let a = 10; let b = 20; console.log(a===b); //false console.log(a===10); //true |

| | | | |
|--------------|--------------------------|---|---|
| | | | <code>console.log(10===10); //false</code> |
| != | Not equal to | It checks whether the values of the two operands are equal or not. | <code>let a = 10;</code> <code>let b = 20;</code> <code>console.log(a!=b); //true</code> <code>console.log(a!=10); //false</code> <code>console.log(10!='10'); //false</code> |
| !== | Not identical | It checks whether the type and values of the two operands are equal or not. | <code>let a = 10;</code> <code>let b = 20;</code> <code>console.log(a!==b); //true</code> <code>console.log(a!==10); //false</code> <code>console.log(10!==10); //true</code> |
| > | Greater than | It checks whether the value of the left operands is greater than the value of the right operand or not. | <code>let a = 30;</code> <code>let b = 20;</code> <code>console.log(a>b); //true</code> <code>console.log(a>30); //false</code> <code>console.log(20> 20'); //false</code> |
| >= | Greater than or equal to | It checks whether the value of the left operands is greater than or equal to the value of the right operand or not. | <code>let a = 20;</code> <code>let b = 20;</code> <code>console.log(a>=b); //true</code> <code>console.log(a>=30); //false</code> <code>console.log(20>='20'); //true</code> |
| < | Less than | It checks whether the value of the left operands is less | <code>let a = 10;</code> <code>let b = 20;</code> |

| | | | |
|--------------|-----------------------|--|--|
| | | than the value of the right operand or not. | <pre>console.log(a<b); //true console.log(a<10); //false console.log(10<'10'); //false</pre> |
| <= | Less than or equal to | It checks whether the value of the left operands is less than or equal to the value of the right operand or not. | <pre>let a = 10; let b = 20; console.log(a<=b); //true console.log(a<=10); //true console.log(10<='10'); //true</pre> |

Logical Operators

Logical operators are used for combining two or more condition into a single expression and return the Boolean result true or false. The Logical operators are given below.

| Operator | Operator_Name | Description | Example |
|-------------------|---------------|---|---|
| && | Logical AND | It returns true if both the operands(expression) are true, otherwise returns false. | <pre>let a = false; let b = true; console.log(a&&b); //false console.log(b&&>true); //true console.log(b&&10); //10 which is also 'true' console.log(a&&'10'); //false</pre> |
| | Logical OR | It returns true if any of the operands(expression) | <pre>let a = false; let b = true; console.log(a b); //true</pre> |

| | | | |
|---|-------------|--|---|
| | | are true, otherwise returns false. | <pre>console.log(b true); //true console.log(b 10); //true console.log(a '10'); //'10' which is also 'true'</pre> |
| ! | Logical NOT | It returns the inverse result of an operand(expression). | <pre>let a = 20; let b = 30; console.log(!true); //false console.log(!false); //true console.log(!a); //false console.log(!b); //false console.log(!null); //true</pre> |

Bitwise Operators

The bitwise operators perform the bitwise operations on operands. The bitwise operators are as follows.

| Operator | Operator_Name | Description | Example |
|----------|---------------|--|---|
| & | Bitwise AND | It returns the result of a Boolean AND operation on each bit of its integer arguments. | <pre>let a = 2; let b = 3; let c = a & b; console.log(c); //</pre> <p>Output</p> <p>2</p> |
| | Bitwise OR | It returns the result of a Boolean OR operation | <pre>let a = 2; let b = 3;</pre> |

| | | | |
|-----------------|---------------------|--|--|
| | | on each bit of its integer arguments. | <pre>let c = a b; console.log(c); //</pre> <p>Output</p> <p>3</p> |
| ^ | Bitwise XOR | It returns the result of a Boolean Exclusive OR operation on each bit of its integer arguments. | <pre>let a = 2; let b = 3; let c = a ^ b; console.log(c); //</pre> <p>Output</p> <p>1</p> |
| ~ | Bitwise NOT | It inverts each bit in the operands. | <pre>let a = 2; let c = ~ a; console.log(c); //</pre> <p>Output</p> <p>-3</p> |
| >> | Bitwise Right Shift | The left operand's value is moved to the right by the number of bits specified in the right operand. | <pre>let a = 2; let b = 3; let c = a >> b; console.log(c); //</pre> <p>Output</p> <p>0</p> |
| << | Bitwise Left Shift | The left operand's value is moved to the left by the number of bits | <pre>let a = 2; let b = 3;</pre> |

| | | | |
|-----|-------------------------------|--|--|
| | | specified in the right operand. New bits are filled with zeroes on the right side. | <pre>let c = a << b; console.log(c); //</pre> <p>Output</p> <p>16</p> |
| >>> | Bitwise Right Shift with Zero | The left operand's value is moved to the right by the number of bits specified in the right operand and zeroes are added on the left side. | <pre>let a = 3; let b = 4; let c = a >>> b; console.log(c); //</pre> <p>Output</p> <p>0</p> |

Assignment Operators

Assignment operators are used to assign a value to the variable. The left side of the assignment operator is called a variable, and the right side of the assignment operator is called a value. The data-type of the variable and value must be the same otherwise the compiler will throw an error. The assignment operators are as follows.

| Operator | Operator_Name | Description | Example |
|----------|----------------|---|--|
| = | Assign | It assigns values from right side to left side operand. | <pre>let a = 10; let b = 5; console.log("a=b:" +a); //</pre> <p>Output</p> <p>10</p> |
| += | Add and assign | It adds the left operand with the | <pre>let a = 10;</pre> |

| | | | |
|-----------|---------------------|--|--|
| | | right operand and assigns the result to the left side operand. | <pre>let b = 5; let c = a += b; console.log(c); //</pre> <p>Output</p> <p>15</p> |
| -= | Subtract and assign | It subtracts the right operand from the left operand and assigns the result to the left side operand. | <pre>let a = 10; let b = 5; let c = a -= b; console.log(c); //</pre> <p>Output</p> <p>5</p> |
| *= | Multiply and assign | It multiplies the left operand with the right operand and assigns the result to the left side operand. | <pre>let a = 10; let b = 5; let c = a *= b; console.log(c); //</pre> <p>Output</p> <p>50</p> |
| /= | Divide and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | <pre>let a = 10; let b = 5; let c = a /= b; console.log(c); //</pre> <p>Output</p> <p>2</p> |

| | | | |
|-----------------|--------------------|---|---|
| <code>%=</code> | Modulus and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | <pre>let a = 16; let b = 5; let c = a %= b; console.log(c); //</pre> <p>Output</p> <p>1</p> |
|-----------------|--------------------|---|---|

Ternary/Conditional Operator

The conditional operator takes three operands and returns a Boolean value based on the condition, whether it is true or false. Its working is similar to an if-else statement. The conditional operator has right-to-left associativity. The syntax of a conditional operator is given below.

```
expression ? expression-1 : expression-2;
```

expression: It refers to the conditional expression.

- expression-1: If the condition is true, expression-1 will be returned.
- expression-2: If the condition is false, expression-2 will be returned.

Example

```
let num = 16;
let result = (num > 0) ? "True":"False"
console.log(result);
```

Output:

```
True
```

Concatenation Operator

The concatenation (+) operator is an operator which is used to append the two string. In concatenation operation, we cannot add a space between the strings. We can concatenate multiple strings in a single statement. The following example helps us to understand the concatenation operator in TypeScript.

Example

```
let message = "Welcome to " + "EduNet Foundation";
console.log("Result of String Operator: " +message);
```

Output:

```
Result of String Operator: Welcome to EduNet Foundation
```

Type Operators

There are a collection of operators available which can assist you when working with objects in TypeScript. Operators such as typeof, instanceof, in, and delete are the examples of Type operator. The detail explanation of these operators is given below.

| Operator_Name | Description | Example |
|---------------|---|--|
| In | It is used to check for the existence of a property on an object. | <pre>let Bike = {make: 'Honda', model: 'CLIQ', year: 2018}; console.log('make' in Bike); //</pre> <p>Output:</p> <p>True</p> |
| Delete | It is used to delete the properties from the objects. | <pre>let Bike = { Company1: 'Honda', Company2: 'Hero', Company3: 'Royal Enfield'</pre> |

| | | |
|-------------------|--|---|
| | | <pre>}; delete Bike.Company1; console.log(Bike); //</pre> <p>Output:</p> <pre>{ Company2: 'Hero', Company3: 'Royal Enfield' }</pre> |
| Typeof | It returns the data type of the operand. | <pre>let message = "Welcome to " + "JavaTpoint"; console.log(typeof message); //</pre> <p>Output:</p> <pre>String</pre> |
| Instanceof | It is used to check if the object is of a specified type or not. | <pre>let arr = [1, 2, 3]; console.log(arr instanceof Array); // true console.log(arr instanceof String); // false</pre> |

Type Script String & Tuple, Oops

Typescript String

In TypeScript, the string is an object which represents the sequence of character values. It is a primitive data type which is used to store text data. The string values are surrounded by single quotation mark or double quotation mark. An array of characters works the same as a string.

Syntax

```
let var_name = new String(string);
```

Example

```
let uname = new String("Hello Edunet");  
console.log("Message: " +uname);  
console.log("Length: "+uname.length);
```

Output:

```
Message: Hello Edunet  
Length: 11
```

There are three ways in which we can create a string.

Single quoted strings

It enclosed the string in a single quotation mark, which is given below.

Example

```
var studentName: String = 'Peter';
```

Double quoted strings

It enclosed the string in double quotation marks, which is given below.

Example:

```
var studentName: String = "Peter";
```

Back-ticks strings

It is used to write an expression. We can use it to embed the expressions inside the string. It is also known as Template string. TypeScript supports Template string from ES6 version.

Example:


```
let empName:string = "Rohit Sharma";
let compName:string = "Edunet";
// Pre-ES6
let empDetail1: string = empName + " works in the " + compName + " company.";
// Post-ES6
let empDetail2: string = `${empName} works in the ${compName} company.`;
console.log("Before ES6: " +empDetail1);
console.log("After ES6: " +empDetail2);
```

Output:

```
Before ES6: Rohit Sharma works in the Edunet company.
After ES6: Rohit Sharma works in the Edunet company.
```

Multi-Line String

ES6 provides us to write the multi-line string. We can understand it from the below example.

Example

```
let multi = 'hello ' +
  'world ' +
  'my ' +
  'name ' +
  'is ' +
  'Rohit';
```

If we want that each line in the string contains "new line" characters, then we have to add "\n" at the end of each string.

Example

```
let multi = 'hello\n ' +
  'Students\n ' +
  'my\n ' +
  'name\n ' +
  'is\n ' +
  'Rohit Sharma';
console.log(multi);
```

Output:

```
hello
Students
my
name
is
Rohit Sharma
```

String Literal Type

A string literal is a sequence of characters enclosed in double quotation marks (" "). It is used to represent a sequence of character which forms a null-terminated string. It allows us to specify the exact string value specified in the "string literal type." It uses "pipe" or " | " symbol between different string value.

Syntax

```
Type variableName = "value1" | "value2" | "value3"; // upto N number of values
```

String Methods

The list of string methods with their description is given below.

| | charAt() | It returns the character of the given index. |
|--|---------------|--|
| | concat() | It returns the combined result of two or more string. |
| | endsWith() | It is used to check whether a string ends with another string. |
| | includes() | It checks whether the string contains another string or not. |
| | indexOf() | It returns the index of the first occurrence of the specified substring from a string, otherwise returns -1. |
| | lastIndexOf() | It returns the index of the last occurrence of a value in the string. |
| | match() | It is used to match a regular expression against the given string. |
| | replace() | It replaces the matched substring with the new substring. |
| | search() | It searches for a match between a regular expression and string. |
| | slice() | It returns a section of a string. |
| | split() | It splits the string into substrings and returns an array. |
| | substring() | It returns a string between the two given indexes. |

| | | |
|--|---------------|---|
| | toLowerCase() | It converts the all characters of a string into lower case. |
| | toUpperCase() | It converts the all characters of a string into upper case. |
| | trim() | It is used to trims the white space from the beginning and end of the string. |
| | trimLeft() | It is used to trims the white space from the left side of the string. |
| | trimRight() | It is used to trims the white space from the right side of the string. |
| | valueOf() | It returns a primitive value of the specified object. |

Example

```
//String Initialization
let str1: string = 'Hello';
let str2: string = 'Edunet';

//String Concatenation
console.log("Combined Result: " +str1.concat(str2));

//String charAt
console.log("Character At 4: " +str2.charAt(4));

//String indexOf
console.log("Index of T: " +str2.indexOf('T'));

//String replace
console.log("After Replacement: " +str1.replace('Hello', 'Welcome to'));

//String uppercase
```

```
console.log("UpperCase: " +str2.toUpperCase());
```

Output:

Combined Result: HelloJavaTpoint

Character At 4: T

Index of T: 4

After Replacement: Welcome to

UpperCase: EDUNET

Typescript Tuple

We know that an array holds multiple values of the same data type. But sometimes, we may need to store a collection of values of different data types in a single variable. Arrays will not provide this feature, but TypeScript has a data type called Tuple to achieve this purpose. A Tuple is an array which store multiple fields belong to different data types. It is similar to the structures in the C programming language.

A tuple is a data type which can be used like any other variables. It represents the heterogeneous collection of values and can also be passed as parameters in a function call.

In abstract mathematics, the term tuple is used to denote a multi-dimensional coordinate system. JavaScript does not have tuple as data type, but tuples are available in TypeScript. The order of elements in a tuple is important.

Syntax

```
let tuple_name = [val1,val2,val3, ...val n];
```

Example

```
let arrTuple = [101, "Edunet", 105, "Abhishek"];
```

```
console.log(arrTuple);
```

Output:

```
[101, 'Edunet', 105, 'Abhishek']
```

We can also declare and initialize a tuple separately by initially declaring the tuple as an empty tuple in Typescript.

Example

```
let arrTuple = [];  
arrTuple[0] = 101  
arrTuple[1] = 105
```

Accessing tuple Elements

We can read or access the fields of a tuple by using the index, which is the same as an array. In Tuple, the index starts from zero.

Example:

```
let empTuple = ["Rohit Sharma", 25, "Edunet"];  
console.log("Name of the Employee is : "+empTuple [0]);  
console.log("Age of the Employee is : "+empTuple [1]);  
console.log(empTuple [0]+" is working in "+empTuple [2]);
```

Output:

```
Name of the Employee is: Rohit Sharma  
Age of the Employee is: 25  
Rohit Sharma is working in Edunet
```

Operations on Tuple

A tuple has two operations:

- Push()
- Pop()

Push()

The push operation is used to add an element to the tuple.

Example:

```
let empTuple = ["Rohit Sharma", 25, "Edunet"];

console.log("Items: "+empTuple);

console.log("Length of Tuple Items before push: "+empTuple.length); // returns the
tuple size

empTuple.push(10001); // append value to the tuple

console.log("Length of Tuple Items after push: "+empTuple.length);

console.log("Items: "+empTuple);
```

Output:

```
Items: Rohit Sharma, 25, Edunet

Length of Tuple Items before push: 3

Length of Tuple Items after push: 4

Items: Rohit Sharma, 25, Edunet, 10001
```

Pop()

The pop operation is used to remove an element from the tuple.

Example

```
let empTuple = ["Rohit Sharma", 25, "Edunet", 10001];
console.log("Items: "+empTuple);
console.log("Length of Tuple Items before pop: "+empTuple.length); // returns the tuple size
empTuple.pop(); // removed value to the tuple
console.log("Length of Tuple Items after pop: "+empTuple.length);
console.log("Items: "+empTuple);
```

Output:

```
Items: Rohit Sharma,25, Edunet, 10001
Length of Tuple Items before pop: 4
Length of Tuple Items after pop: 3
Items: Rohit Sharma, 25, Edunet
```

Update or Modify the Tuple Elements

Tuples are mutable, which means we can update or change the values of tuple elements. To modify the fields of a Tuple, we need to use the index of the fields and assignment operator. We can understand it with the following example.

Example:

```
let empTuple = ["Rohit Sharma", 25, "Edunet"];
empTuple[1] = 30;
console.log("Name of the Employee is: "+empTuple [0]);
console.log("Age of the Employee is: "+empTuple [1]);
```



```
console.log(empTuple [0]+" is working in "+empTuple [2]);
```

Output:

Name of the Employee is: Rohit Sharma

Age of the Employee is: 30

Rohit Sharma is working in Edunet

Clear the fields of a Tuple

We cannot delete the tuple variable, but its fields could be cleared. To clear the fields of a tuple, assign it with an empty set of tuple field, which is shown in the following example.

Example:

```
let empTuple = ["Rohit Sharma", 25, "Edunet"];
empTuple = [];
console.log(empTuple);
```

Output:

```
[]
```

Destructuring the Tuple

Destructuring allows us to break up the structure of an entity. TypeScript used destructuring in the context of a tuple.

Example:

```
let empTuple = ["Rohit Sharma", 25, "Edunet"];
let [emp, student] = empTuple;
```

```
console.log(emp);
console.log(student);
```

Output:

```
Rohit Sharma
25
```

Passing Tuple to Functions

We can pass a tuple to functions, which can be shown in the below example.

Example:

```
//Tuple Declaration
let empTuple = ["EduNet", 101, "Abhishek"];

//Passing tuples in function
function display(tuple_values:any[]) {
  for(let i = 0;i<empTuple.length;i++) {
    console.log(empTuple[i]);
  }
}

//Calling tuple in function
display(empTuple);
```

Output:

```
EduNet
101
```

Abhishek

Typescript OOPS

Object Oriented Programming or OOP is a programming paradigm that has four principles which are:

- Inheritance,
- Abstraction,
- Polymorphism,
- And Encapsulation.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). ... In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

At the heart of OOP is the concept of an object which may refer to an abstract or concrete object in our world so this makes it easier for programmers to model actual problems with computers languages before trying to find the solutions.

An object can have data (which form the properties or attributes of the real-world object) and code in a form of methods (which represent the behavior in the equivalent real-world object). Think of a car for example, it has a color, weight and speed and can move forward or backward.

You can very easily create a "Car" object with an OOP language such as TypeScript to represent this car in the computer memory.

Class-based programming, or more commonly class-orientation, is a style of Object-oriented programming(OOP) in which inheritance occurs via defining classes of objects, instead of inheritance occurring via the objects alone (compare prototype-based programming).

Unlike JavaScript which has a prototype-based OOP, TypeScript is a class-based OOP language.

In a programming language, a class has the same meaning in the sense that it represents a category of objects or a type but it also has a concrete form as an extensible template of code for creating objects via instantiation.

Let's refer back to our "Car" object. Before, you can create the object, you need to create the "Car" class which contains the data fields and methods (procedures which are attached to the class) that each car has, next, you instantiate the class to create one or more objects (cars).

We refer to Inheritance in OOP when a class A inherits the properties of another class B. A is also said to extend B. This is a familiar behavior in nature where the children of humans or other creatures inherit the traits of their parents. ** Thanks to inheritance, we can reuse the fields and methods of the existing class which facilitates reusability, one of the goals of the OOP paradigm.

Inheritance is a relationship between two classes where the class that is used as the basis for inheritance is referred to as a *superclass or base class. While the class that inherits from a base class is referred to as a subclass.

In TypeScript, we use the extends keyword for defining an inheritance.

Inheritance implies Polymorphism (Another fundamental principle of OOP). Think of that, when a class B and C inherit the methods of a class A. They can customize or change the inherited methods as necessary. For example, both a Plane and a Car inherit a move() method from a Vehicle but the move behavior of a Plane is actually the flying instead of moving using wheels. So when we create the Plane class that extends the Vehicle class, we need to override the move() method to implement a flying behavior instead of the regular movement.

Overriding the inherited (parent) method and re-implementing its behavior is what refers to Polymorphism. In fact, the meaning of polymorphism from the Greek origin is when something occurs in many different forms.

What about Encapsulation?

Encapsulation represents another principle of Object-oriented programming. The concept refers to the grouping of data variables and methods. The class in OOP languages ****enables encapsulation via providing the way to group data and methods.

Encapsulation is also used to hide data and methods that are meant to be internal and only required for the inner working of the object. In this meaning, encapsulation is equivalent to Abstraction, another fundamental principle of OOP.

TypeScript provides the programmer with access modifiers or keywords like public, protected and private to specify the degree of visibility of the class members to the outside.

Object and Classes

TypeScript is object-oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn't support classes. Typescript gets this feature from ES6.

Creating classes

Use the class keyword to declare a class in TypeScript. The syntax for the same is given below –

Syntax

```
class class_name {  
    //class scope  
}
```

The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.

A class definition can include the following –

- Fields – A field is any variable declared in a class. Fields represent data pertaining to objects

- Constructors – Responsible for allocating memory for the objects of the class
- Functions – Functions represent actions an object can take. They are also at times referred to as methods

These components put together are termed as the data members of the class.

Consider a class Person in typescript.

```
class Person {  
}
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10  
var Person = (function () {  
    function Person() {  
    }  
    return Person;  
})();
```

Example: Declaring a class

```
class Car {  
    //field  
    engine:string;  
  
    //constructor  
    constructor(engine:string) {  
        this.engine = engine
```

```

    }

    //function
    disp():void {
        console.log("Engine is : "+this.engine)
    }
}

```

The example declares a class Car. The class has a field named engine. The var keyword is not used while declaring a field. The example above declares a constructor for the class.

A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword. A constructor is a function and hence can be parameterized.

The this keyword refers to the current instance of the class. Here, the parameter name and the name of the class's field are the same. Hence to avoid ambiguity, the class's field is prefixed with the this keyword.

disp() is a simple function definition. Note that the function keyword is not used here.

On compiling, it will generate following JavaScript code.

```

//Generated by typescript 1.8.10
var Car = (function () {
    //constructor
    function Car(engine) {
        this.engine = engine;
    }
}

```

```
//function
Car.prototype.disp = function () {
    console.log("Engine is : " + this.engine);
};
return Car;
})();
```

Creating Instance objects

To create an instance of the class, use the new keyword followed by the class name. The syntax for the same is given below –

Syntax

```
var object_name = new class_name([ arguments ])
```

The new keyword is responsible for instantiation.

The right-hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

Example: Instantiating a class

```
var obj = new Car("Engine 1")
```

Accessing Attributes and Functions

A class's attributes and functions can be accessed through the object. Use the ' .' dot notation (called as the period) to access the data members of a class.

```
//accessing an attribute
obj.field_name
```

```
//accessing a function
```

```
obj.function_name()
```

Example: Putting them together

```
class Car {
    //field
    engine:string;

    //constructor
    constructor(engine:string) {
        this.engine = engine
    }

    //function
    disp():void {
        console.log("Function displays Engine is : "+this.engine)
    }
}

//create an object
var obj = new Car("XXSY1")
```

```
//access the field
console.log("Reading attribute value Engine as : "+obj.engine)

//access the function
obj.disp()
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var Car = (function () {
    //constructor
    function Car(engine) {
        this.engine = engine;
    }

    //function
    Car.prototype.disp = function () {
        console.log("Function displays Engine is : " + this.engine);
    };
    return Car;
})();

//create an object
```

```
var obj = new Car("XXSY1");

//access the field

console.log("Reading attribute value Engine as : " + obj.engine);

//access the function

obj.disp();
```

The output of the above code is as follows –

Reading attribute value Engine as : XXSY1

Function displays Engine is : XXSY1

Inheritance & Interface

An interface can be extended by other interfaces. In other words, an interface can inherit from other interface. Typescript allows an interface to inherit from multiple interfaces.



Image : Inheritance & Interface

Reference: https://www.tutorialspoint.com/typescript/images/interface_and_objects.jpg

Use the extends keyword to implement inheritance among interfaces.

Single Interface Inheritance

Syntax:

```
Child_interface_name extends super_interface_name
```

Multiple Interface Inheritance

Syntax:

```
Child_interface_name extends super_interface1_name,  
super_interface2_name,...,super_interfaceN_name
```

Example: Simple Interface Inheritance

```
interface Person {  
    age:number  
}  
  
interface Musician extends Person {  
    instrument:string  
}  
  
var drummer = <Musician>{};  
drummer.age = 27  
drummer.instrument = "Drums"
```

```
console.log("Age: "+drummer.age) console.log("Instrument: "+drummer.instrument)
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var drummer = {};
drummer.age = 27;
drummer.instrument = "Drums";
console.log("Age: " + drummer.age);
console.log("Instrument: " + drummer.instrument);
```

Its output is as follows –

```
Age: 27
Instrument: Drums
```

Example: Multiple Interface Inheritance

```
interface IParent1 {
    v1:number
}

interface IParent2 {
    v2:number
}
```

```
interface Child extends IParent1, IParent2 { }
var lobj:Child = { v1:12, v2:23}
console.log("value 1: "+this.v1+" value 2: "+this.v2)
```

The object lobj is of the type interface leaf. The interface leaf by the virtue of inheritance now has two attributes- v1 and v2 respectively. Hence, the object lobj must now contain these attributes.

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var lobj = { v1: 12, v2: 23 };
console.log("value 1: " + this.v1 + " value 2: " + this.v2);
```

The output of the above code is as follows –

```
value 1: 12 value 2: 23
```

Angular js Child Component

In AngularJS, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure.

This makes it easier to write an app in a way that's similar to using Web Components or using the new Angular's style of application architecture.

Advantages of Components:

- simpler configuration than plain directives
- promote sane defaults and best practices

- optimized for component-based architecture
- writing component directives will make it easier to upgrade to Angular

When not to use Components:

- for directives that need to perform actions in compile and pre-link functions, because they aren't available
- when you need advanced directive definition options like priority, terminal, multi-element
- when you want a directive that is triggered by an attribute or CSS class, rather than an element

Creating and configuring a Component

Components can be registered using the `.component()` method of an AngularJS module (returned by `angular.module()`). The method takes two arguments:

- The name of the Component (as string).
- The Component config object. (Note that, unlike the `.directive()` method, this method does not take a factory function.)

Component-based application architecture

As already mentioned, the component helper makes it easier to structure your application with a component-based architecture. But what makes a component beyond the options that the component helper has?

- Components only control their own View and Data: Components should never modify any data or DOM that is out of their own scope. Normally, in AngularJS it is possible to modify data anywhere in the application through scope inheritance and watches. This is practical, but can also lead to problems when it is not clear which part of the application is responsible for modifying the data. That is why component directives use an isolate scope, so a whole class of scope manipulation is not possible.
- Components have a well-defined public API - Inputs and Outputs: However, scope isolation only goes so far, because AngularJS uses two-way binding. So if you pass an object to a component like this - `bindings: {item: '='}`, and modify one

of its properties, the change will be reflected in the parent component. For components however, only the component that owns the data should modify it, to make it easy to reason about what data is changed, and when. For that reason, components should follow a few simple conventions

Example of a component tree

The following example expands on the simple component example and incorporates the concepts we introduced above:

Instead of an ngController, we now have a heroList component that holds the data of different heroes, and creates a heroDetail for each of them.

The heroDetail component now contains new functionality:

- a delete button that calls the bound onDelete function of the heroList component
- an input to change the hero location, in the form of a reusable editableField component. Instead of manipulating the hero object itself, it sends a changeset upwards to the heroDetail, which sends it upwards to the heroList component, which updates the original data.

Components as route templates

Components are also useful as route templates (e.g. when using ngRoute). In a component-based application, every view is a component:

```
var myMod = angular.module('myMod', ['ngRoute']);

myMod.component('home', {
  template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',
  controller: function() {
    this.user = {name: 'world'};
  }
})
```



```
});
myMod.config(function($routeProvider) {
  $routeProvider.when('/', {
    template: '<home></home>'
  });
});
```

When using `$routeProvider`, you can often avoid some boilerplate, by passing the resolved route dependencies directly to the component. Since 1.5, `ngRoute` automatically assigns the resolves to the route scope property `$resolve` (you can also configure the property name via `resolveAs`). When using components, you can take advantage of this and pass resolves directly into your component without creating an extra route controller:

```
var myMod = angular.module('myMod', ['ngRoute']);
myMod.component('home', {
  template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',
  bindings: {
    user: '<'
  }
});
myMod.config(function($routeProvider) {
  $routeProvider.when('/', {
    template: '<home user="$resolve.user"></home>',
    resolve: {
```

```
user: function($http) { return $http.get('...'); }  
  
}  
  
});  
  
});
```

Angular js Data Binding, Event Binding, 2 Way Binding

Angular js Data Binding

Data-binding in AngularJS apps is the automatic synchronization of data between the model and view components. The way that AngularJS implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

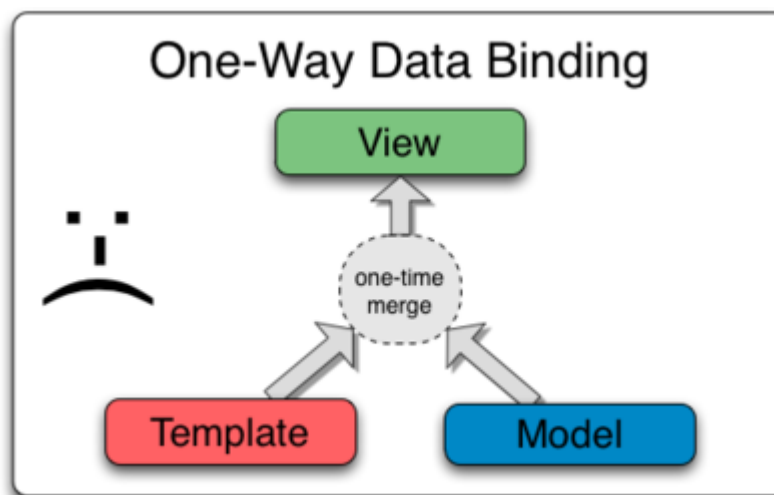


Image : Angular js Data Binding
Reference: https://docs.angularjs.org/img/One_Way_Data_Binding.png

Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model. This means that

the developer has to write code that constantly syncs the view with the model and the model with the view.

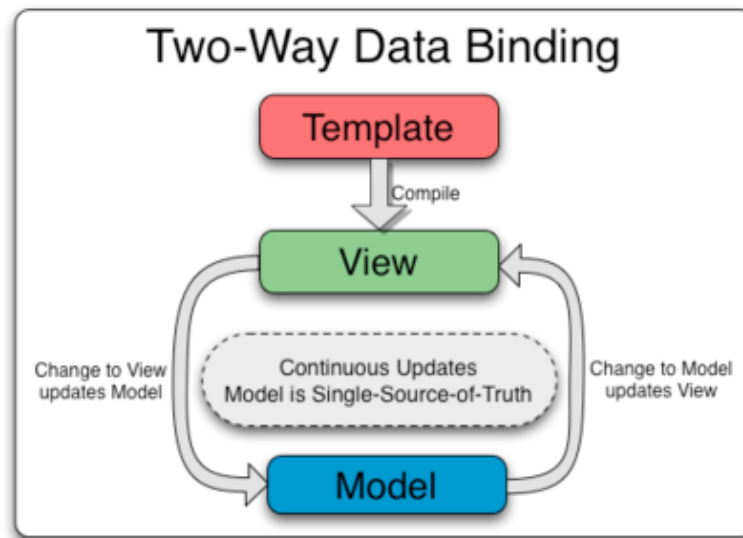


Image : Angular js 2-way Data Binding

Reference: <https://www.javatpoint.com/js/angularjs/images/two-way-data-binding.png>

AngularJS templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

How To Bind Events In AngularJS

We want the user to be able to take an action, and cause something to happen on the page. Users will enter text into input boxes, pick items from lists and click buttons. These types of user actions result in a flow of data from an element to a component. Listening for certain events such as keystrokes, mouse movements, and clicks are done with Angular event binding.

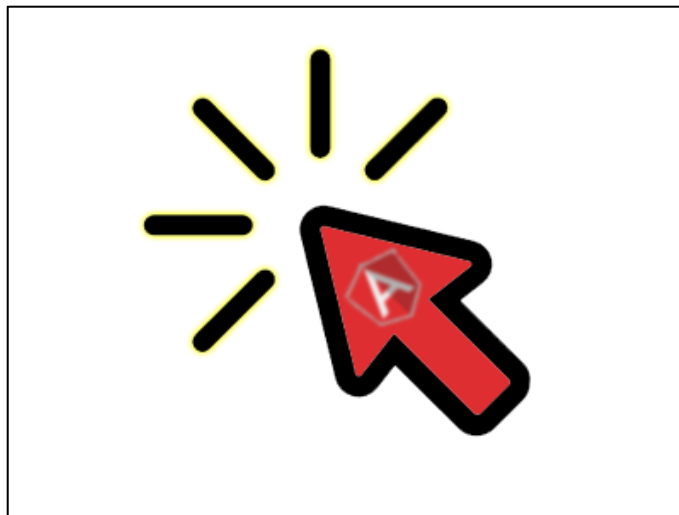


Image : How To Bind Events In AngularJS

Reference: <https://vegibit.com/wp-content/uploads/2018/11/How-To-Bind-Events-In-AngularJS.png>

The most common action a user may take is a click event. Some of the common JavaScript events include onclick, onmouseover, onmouseout, onchange, onkeydown, onkeyup, and many more. We can set up a click event in Angular using a special syntax.

`(click)="methodToRun()"`

In the virtual-machines.component.html Angular Template, if you would like to respond to a click event you can use this form of binding.

```
<button  
  [disabled]="!allowNewVm"  
  class="btn btn-lg"
```

```
(click)="onCreateVM()"
>Add a VM</button>
```

The (click) says, “hey, we’re listening for the user to click”. The =”onCreateVM()” portion of the markup says, “run this piece of code when clicked”.

Configure The .ts file to respond

In our template, we now have a click event listener set up. When it is click, we want something to happen. To set this up, first we will create a new variable to hold an initial state of data. This will be the vmCreated variable we see here in virtual-machines.component.ts.

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

@Component({
  selector: 'app-virtual-machines',
  templateUrl: './virtual-machines.component.html',
  styleUrls: ['./virtual-machines.component.css'],
})
export class VirtualMachinesComponent implements OnInit {

  allowNewVm = false;
  vmCreated = 'Initial State: Add a VM?';

  constructor() {
    setTimeout(() => {
```

```

    this.allowNewVm = true
  }, 1500);
}

ngOnInit() {
}
}

```

In addition to the new variable, we also need the new method or function that is supposed to run when the button is clicked. We had named that function `onCreateVM` in the template. That means we will add that named function to the TypeScript file like so:

```

import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-virtual-machines',
  templateUrl: './virtual-machines.component.html',
  styleUrls: ['./virtual-machines.component.css'],
})
export class VirtualMachinesComponent implements OnInit {

  allowNewVm = false;

  vmCreated = 'Initial State: Add a VM?';
}

```

```

constructor() {
  setTimeout(() => {
    this.allowNewVm = true
  }, 1500);
}

ngOnInit() {
}

onCreateVM() {
  this.vmCreated = 'Button Clicked: New VM spun up!';
}
}

```

Reference The Data in The Template

Finally, we output the contents of the vmCreated variable via string interpolation. On page load, it should output the initial value of the variable. When the user clicks the button, that value should change which will also update the user interface in real-time.

```
<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a VM</button>
```

```
the allowNewVm variable is currently <b [innerText]="allowNewVm"></b>
```

```
<hr>
<h5>{{ vmCreated }}</h5>
<app-virtual-machine></app-virtual-machine>
```

Reference The Data in The Template

Finally, we output the contents of the vmCreated variable via string interpolation. On page load, it should output the initial value of the variable. When the user clicks the button, that value should change which will also update the user interface in real-time.

```
<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a
VM</button>
```

the allowNewVm variable is currently <b [innerText]="allowNewVm">

```
<hr>
<h5>{{ vmCreated }}</h5>
<app-virtual-machine></app-virtual-machine>
```

(input)="onMethodToRun(\$event)"

Speaking of user input, let's set up an <input> field that will allow the user to provide a name for the VM before creating it. That \$event is special. It is like a reserved variable name you can use in the template during event binding. It is a way to fetch the data from that input when it fires.

Two Way Data Binding

In the code above, we kind of took the long route to set up data binding. An easier approach is to simply use the [(ngModel)] directive.


```

<form>

  <div class="form-group">

    <label for="vmname">VM Name</label>

    <input
      type="text"
      class="form-control"
      [(ngModel)]="vmName"
      name="vmName"
      id="vmname"
    >

    <small class="form-text text-muted">Enter the name for a new VM.</small>

  </div>

</form>

<p>Add the <b>{{ vmName }}</b> virtual machine?</p>

<button [disabled]="!allowNewVm" class="btn btn-lg" (click)="onCreateVM()">Add a
VM</button>

the allowNewVm variable is currently <b [innerText]="allowNewVm"></b>

<hr>

<h5>{{ vmCreated }}</h5>

```

`<app-virtual-machine></app-virtual-machine>` You'll note that we also set the name attribute using `name="vmName"`. The reason for this is because if you leave it out, you may see an error like "ERROR Error: If ngModel is used within a form tag, either the name attribute must be set or the formcontrol must be defined as 'standalone' in ngModelOptions."

Testing out the data binding using `[(ngModel)]="vmName"` does appear to be working well. Also note that we no longer need the `onSetVmName(event)` method in `virtual-machines.component.ts` since `ngModel` is handling that for us automatically.

Angular JS Pipe

What is Pipe?

The pipe symbol (|) is used for applying filters in AngularJS. A filter is a function that is invoked for handling model transformations. Its basically just a global function that doesn't require registration of functions on a scope, and offers more convenient syntax to regular function calls.

A pipe class must implement the PipeTransform interface. For example, if the name is "myPipe", use a template binding expression such as the following:

```
{{ exp | myPipe }}
```

The result of the expression is passed to the pipe's transform() method.

A pipe must belong to an NgModule in order for it to be available to a template. To make it a member of an NgModule, list it in the declarations field of the NgModule metadata.

Angular Pipes are used to transform data on a template, without writing a boilerplate code in a component.

Why Pipe?

A pipe takes in data as input and transforms it to the desired output. It is like a filter in Angular 1 (AngularJS).

Generally, If we need to transform data, we write the code in the component, For example, we want to transform today's date into a format like '16 Apr 2018' or '16-04-2018', We need to write separate code in the component.

So instead of writing separate boilerplate code, we can use the built-in pipe called DatePipe which will take input and transform it into the desired date format.

We can use the pipe on a template using Pipe Operator | .

As shown below,

```
{{today | date : 'fullDate'}}
```

This interpolation give output in Monday, April 16, 2018 date format.

here,

- today is the component variable, which specifies the current date.
- date represent DataPipe
- fullDate is an optional parameter or argument which specifies the date format.

Angular comes with a set of built-in pipes such as DatePipe, UpperCasePipe, LowerCasePipe. Other than this, We can also create our own custom pipe.

Built-In Pipes

Angular comes with a collection of built-in pipes such as:

- i. DatePipe
- ii. UpperCasePipe
- iii. LowerCasePipe
- iv. CurrencyPipe
- v. DecimalPipe
- vi. PercentPipe

Built-in Angular Pipes are defined in @angular/common package.

Custom Filter:

Sometimes the built-in filters in Angular cannot meet the needs or requirements for filtering output. In such a case, an AngularJS custom filter can be created, which can pass the output in the required manner.

Similarly, for numbers, you can use other filters. During this tutorial, we will see the different standard built-in filters available in Angular.

Pipe Chaining

What Is Pipe Channing?

The chaining Pipe is used to perform the multiple operations within the single expression. This chaining operation will be chained using the pipe (|).

In the following example, to display the birthday in the upper case- will need to use the inbuilt date-pipe and upper-case-pipe.

In the following example –

```
{{ birthday | date | uppercase}}
```

<!-- The output is - MONDAY, MARCH 10, 1984 -->

What Is Parameterizing Pipe?

A pipe can accept any number of optional parameters to achieve output. The parameter value can be any valid template expressions. To add optional parameters follow the pipe name with a colon (:). Its looks like- currency: 'INR'

In the following example –

```
<h2>The birthday is - {{ birthday | date:"MM/dd/yy" }} </h2>
```

<!-- Output - The birthday is - 10/03/1984 -->

Angular js Routing

Routing is a core feature in AngularJS. This feature is useful in building SPA (Single Page Application) with multiple views. In SPA application, all views are different Html files and we use Routing to load different part of application and its help to divide application logically and make it manageable. In other words, Routing helps us to divide our application in logical views and bind them with different controllers.

Introduction to ngRoute Module

This module provides routing in AngularJS application and also provides deep linking services and directives. To achieve routing in AngularJS, we need to include the library file of ngRoute.

Component of ngRoute Module

There are four main components of ngRoute module:

- i. **ngView:** ngView is directive and creates new scope. It is used to load html templates.
- ii. **\$routeProvider:** It is used to configure routes.
- iii. **\$route:** It is used to make deep linking URLs between controllers and view. \$route watches \$location.url() and tries to map the path to an existing route defined by \$routeProvider
- iv. **\$routeParams:** This is Angular service which allows us to retrieve the current set of route parameters.

\$routeProvider used for configuring the routes in AngularJS application. It is depends on ngRoute module. All application routes are defined via \$routeProvider and it is the provider of the \$route Service. It is very easy to wire up controllers, view templates and browser URL location using \$route service. This service also help us to implement deep linking that utilize the browser back and forward navigation (browser's history).

Hello World Example

Routing in AngularJS is used to load different templates at runtime. In the following example we will elaborate it more step by step.

Step 1: Create basic structure of application (SPA).

In this step, I have created demo.html and it includes all the required AngularJS library and bootstrap library. Here I have also created structure for application. Also defined two links: page1 and page2. Each link loads respective template.

Project structure

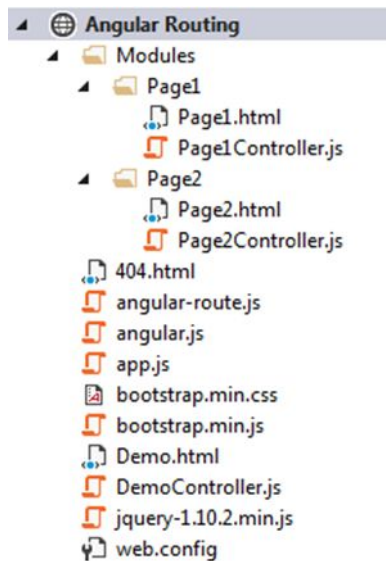


Image: Project structure

Reference: <https://csharpcorner-mindcrackerinc.netdna-ssl.com/UploadFile/ff2f08/routing-in-angularjs/Images/Project%20structure.jpg>

Demo.html

```
<!DOCTYPEhtml>
```

```
<htmldata-ng-apphtmldata-ng-app="AngularApp">
```

```
<head>
```

```
<metacontentmetacontent="IE=edge, chrome=1" http-equiv="X-UA-Compatible" />
```

```
<title>AngularJS - Routing</title>
```

```
<scriptsrcscriptsrc="angular.js">
```

```
</script>
```

```
<scriptsrcscriptsrc="angular-route.js">
```

```

</script>
<scriptsrcscriptsrc="app.js">
  </script>
  <scriptsrcscriptsrc="jquery-1.10.2.min.js">
    </script>
    <scriptsrcscriptsrc="DemoController.js">
      </script>
      <scriptsrcscriptsrc="Modules/Page1/Page1Controller.js">
        </script>
        <scriptsrcscriptsrc="Modules/Page2/Page2Controller.js">
          </script>
          <scriptsrcscriptsrc="bootstrap.min.js">
            </script>
            <linkhreflinkhref="bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div>
    <div ng-controller="demoController" class="container">
      <p><b>Hello World - Routing Example</b></p>
      <divclassdivclass="row">
        <divclassdivclass="col-md-3">
          <ulclassulclass="nav">

```



```

        <li>
            <a href="#/page1"> Page 1 </a>
        </li>
        <li>
            <a href="#/page2"> Page 2 </a>
        </li>
    </ul>
</div>
<div class="col-md-9">
    <div ng-view>
</div>
</div>
</div>
</div>
</div>
</div>
</body>
</html>

```

Here I have divided screen in two sections: Left contains the menu and in the right pane respective template will be loaded.

The ngView directive is responsible to render the template of current route into the main layout file passed by \$route service. I have also defined ng-app directive once. The ngView become place holder for views. Every view render by the route is loaded into this section.

Step 2: Add Routing

In the Demo.html, I have included app.js file which hold the definition of AngularJS application. The \$routeProvider definition contain by the module is called "ngRoute". In app.js file, I have defined an angular app using "angular. Module" method. After creating module, we need to configure the routes. The "config" method is used to configure \$routeProvider. Using "when" and "otherwise" method of \$routeProvider, we can define the route for our AngularJS application.

app.js

```
var app = angular.module("AngularApp", ['ngRoute']);

app.config(['$routeProvider',
    function ($routeProvider)
    {
        $routeProvider.
        when('/page1',
            {
                templateUrl: 'Modules/Page1/page1.html',
                controller: 'Page1Controller'
            })
        .
        when('/page2',
            {
                templateUrl: 'Modules/Page2/page2.html',
                controller: 'Page2Controller'
            })
    })
```

```

        .
        otherwise(
        {
            redirectTo: '/page1'
        });
    }
});

```

In the above code, I have defined two routes: "page1" and "page2" and mapped them with template view "Modules/Page1/page1.html" and "Modules/Page2/page2.html" respectively. I have set default page using "otherwise" method.

Step 3: Define HTML Template and controller

In this example, I have added two html templates: page1.html and page2.html and created two controllers files: page1controller.js and page2controller.js. The following are the definition of both HTML template and controller.

Page1.html

```

<div ng-controller="Page1Controller">
    <h2>Page 1</h2> Hi, {{myName}} </div>

```

Page1Controller.js

```

app.controller("Page1Controller", ['$scope', function ($scope)
{
    $scope.myName = "Tejas Trivedi";
}]);

```

Page2.html

```
<div ng-controller="Page2Controller">
  <h2>Page 2</h2> Hi, {{myName}} </div>
```

Page1Controller.js

```
app.controller("Page2Controller", ['$scope', function ($scope)
{
  $scope.myName = "Jignesh Trivedi";
}]);
```

Output

Hello World - Routing Example

Page 1

Page 2

Page 1

Hi, Tejas Trivedi

Hello World - Routing Example

Page 1

Page 2

Page 2

Hi, Jignesh Trivedi

Pass parameters to route URL

We can also define parameters in the route URL. Parameter name should define by colon (:) after the route path. The `$routeParams` service allow us to retrieve the route parameters in controller.

This feature is useful when we want to use same view for two or more different purpose and it can be identified by the parameter value or we want to display detail of any master record based on master id and master id passed as parameter. In Angular, route parameter can be defined using parameter name in URL. For example,

app.js

```
.when('/page3/:id',
{
  templateUrl: 'Modules/Page3/page3.html',
  controller: 'Page3Controller'
})
```

We can read parameter value in controller by using `$routeParams`. Note that do not forget to inject `$routeParams` service in controller.

page3Controller.js

```
app.controller("Page3Controller", ['$scope', '$routeParams', function ($scope,
$routeParams)
{
  $scope.myName = "Jignesh Trivedi";
  $scope.id = $routeParams.id;
}]);
```

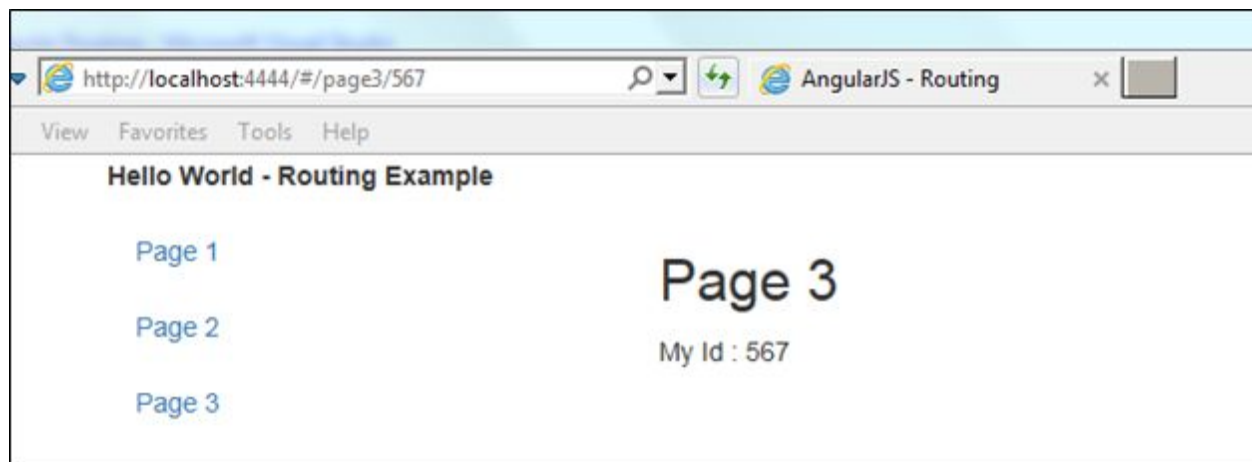
page3.html

```
<div ng-controller="Page2Controller">
  <h2>Page 3</h2> My Id : {{id}} </div>
```

Demo.html

```
<div class="col-md-3">
  <ul class="nav">
    <li><a href="#/page1"> Page 1 </a></li>
    <li><a href="#/page2"> Page 2 </a></li>
    <li><a href="#/page3/567"> Page 3 </a></li>
  </ul>
</div>
```

Output



Load local views

This is not always required to load view from the different html files because view templates are very small and we might want to keep them within main html file instead of creating separate html files. The ng-template can be used to define small templates in the main html file.

Syntax

```
<script type="text/ng-template" id="page4.html">

  //define definition of html here...

</script>
```

Here I have defined a template "page4.html" and "page5.html" inside the script tag. AngularJS will automatically load these templates in ng-view when "page4.html" or "page5.html" is referred in route.

DemoLocalView.html defines the structure of the application. The appLocalView.js is very similar to previous example. Here I have used same controller js files.

appLocalView.js

```
var app = angular.module("AngularApp", ['ngRoute']);

app.config(['$routeProvider',

  function ($routeProvider)

  {

    $routeProvider.

    when('/page4',

      {

        templateUrl: 'page4.html',

        controller: 'Page1Controller'

      })

    .

    when('/page5',

      {
```

```

        templateUrl: 'page5.html',
        controller: 'Page2Controller'
    })
    .
    otherwise(
    {
        redirectTo: '/page4'
    });
}
]);

```

DemoLocalView.html

```

<!DOCTYPE html>
<html data-ng-app="AngularApp">
  <head>
    <meta content="IE=edge, chrome=1" http-equiv="X-UA-Compatible" />
    <title>AngularJS - Routing</title>
    <script src="angular.js"></script>
    <script src="angular-route.js"></script>
    <script src="appLocalView.js"></script>
    <script src="jquery-1.10.2.min.js"></script>
    <script src="Modules/Page1/Page1Controller.js"></script>

```



```

<script src="Modules/Page2/Page2Controller.js"></script>
<script src="bootstrap.min.js"></script>
<link href="bootstrap.min.css" rel="stylesheet" /> </head>
<body>
  <div>
    <div class="container">
      <p><b>Load local view Example</b></p>
      <div class="row">
        <div class="col-md-3">
          <ul class="nav">
            <li><a href="#page4"> Page 4 </a></li>
            <li><a href="#page5"> Page 5 </a></li>
          </ul>
        </div>
        <div class="col-md-9">
          <div ng-view></div>
        </div>
      </div>
      <script type="text/ng-template" id="page4.html">
        <h2> Page 4 </h2> Hi, {{myName}} </script>
      <script type="text/ng-template" id="page5.html">
        <h2> Page 5 </h2> Hi, {{myName}} </script>

```

```
</div>  
</div>  
</body>  
</html>
```

Output

Load local view Example

Page 4

Page 5

Page 5

Hi, Jignesh Trivedi

Angular js Services

What are AngularJS Services?

AngularJs service is a function, which can use for the business layer of an application. It is like a constructor function that will invoke only once at runtime with new. Services in AngularJS are stateless and singleton objects because we get only one object in it regardless of which application interface created it.

We can use it to provide functionality in our web application. Each service performs a specific task.

When to use Services in AngularJS?

We can use AngularJS services when we want to create things that act as an application interface. It can be used for all those purposes for which constructor is used.

Types of AngularJS Services

There are two types of services in angular:

- Built-in services – There are approximately 30 built-in services in angular.
- Custom services – In angular if the user wants to create its own service he/she can do so.

Built-in Services in AngularJS

They are pre-built services in AngularJS. These services get registered automatically at runtime with the dependency injector. Therefore, by using dependency injector we can easily incorporate these built-in services in our angular application.

The various built-in services are as follows:

- i. **\$http:** It is a service to communicate with a remote server. It makes an ajax call to the server.
- ii. **\$interval:** It is a wrapper in angular for window.setInterval.
- iii. **\$timeout:** It is the same as setTimeout function in javascript. To set a time delay on the execution of a function \$timeout is used.
- iv. **\$anchorscroll:** The page which specifies by an anchor in \$location.hash() scrolls using \$anchorscroll.

- v. **\$animate:** It consists of many DOM (Document Object Model) utility methods that provide support for animation hooks.
- vi. **\$animateCss:** It will perform animation only when ngAnimate includes, by default.
- vii. **\$cacheFactory:** It is a factory that constructs cache objects. It puts the key-value pair and retrieves the key-value pair. Also, it can provide access to other services.
- viii. **\$templateCache:** Whenever a template is used for the first time, it is loaded in template cache. It helps in quick retrieval.
- ix. **\$compile:** We can compile HTML string or DOM in the template by it. Also, it produces a template function which will use to link template and scope together.
- x. **\$controller:** By using a \$controller, we can instantiate Angular controller components.
- xi. **\$document:** J-query wrapped the reference to the window. Document element is specified by it.
- xii. **\$exceptionHandler:** Any uncaught exception in an angular expression is sent to this service.
- xiii. **\$filter:** Use to format the data for displaying to the user.
- xiv. **\$httpParamSerializer:** It converts an object to a string.
- xv. **\$httpParamSerializerJQLike:** We can sort Params in alphabetic order. It follows j-query's param() method logic.
- xvi. **\$xhrFactory:** XMLHttpRequest objects is created using factory function.
- xvii. **\$httpBackend:** Browser incompatibilities can be handled using it.
- xviii. **\$interpolate:** Use for data binding by HTML \$compile service.
- xix. **\$locale :** For various angular components, localization rules can be provided using \$locale.
- xx. **\$location:** URL in the address bar of a browser is parsed by it and then the URL is made available to your application. Changes to the URL in the address bar is reflected in \$location service and vice-versa.
- xxi. **\$log:**It is a console logger.
- xxii. **\$parse:** Use to convert Angular expression into a function using \$parse.
- xxiii. **\$q:** A function can run asynchronously using \$q and its return value, which will use when they finish the processing.
- xxiv. **\$rootElement:** It is a root element of the angular application.

- xxv. **\$rootScopeUse** in an angular application.
- xxvi. **\$sceDelegate**: Use in the backend by \$sce.

Customs Services in AngularJS

We can create our own service by connecting it with a module in AngularJS. And to use it add it as a dependency while defining controller.

Angular js Http Request

AngularJS provides \$http control which works as a service to read data from the server. \$http is an AngularJS service for reading data from remote servers. The \$http is a core AngularJS service that is used to communicate with the remote HTTP service via browser's XMLHttpRequest object or via JSONP.

Syntax:

```
$http({  
  method: 'Method_Name',  
  url: '/someUrl'  
}).then(function successCallback(response) {  
  //when the response is available, this callback will be called asynchronously  
}, function errorCallback(response) {  
  // this method will called when server returns response with an error status.  
});
```

The \$http service is function that takes a configured object to generate a HTTP request and return the response. This response contains data, status code, header, configuration object and status text. In \$http the first function executes on successful callback and the second function xecutes on error.

Example 1:

```
<html>
```

```

<head>

  <title>Angular JS Includes</title>

  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.15/angular.min.js">

    </script>

    <script src="angular.min.js">

      </script>

</head>

<body>

  <h2>AngularJS Ajax Demo</h2>

  <div ng-app="app" ng-controller="Employee"> <span>{{Message}}</span><br/>
<span>{{Status}}</span><br/> <span>{{Headers}}</span><br/>
<span>{{Config}}</span><br/> <span>{{StatusText}}</span><br/> </div>

  <script>

    var obj = angular.module('app', []);

    obj.controller('Employee', function ($scope, $http)

    {

      $http(

        {

          method: 'GET',

          url: 'index.html'

        }).then(function successCallback(response)

        {

```

```

        $scope.Message = response.data;
        $scope.Status = response.status;
        $scope.Headers = response.headers;
        $scope.Config = response.config;
        $scope.StatusText = response.statusText;
    }, function errorCallback(response)
    {
        alert("Unsuccessful call!");
    });
});
</script>
</body>
</html>

```

Index.HTML

This is \$Ajax Example.

Output

AngularJS Ajax Demo

This is \$Ajax Example.
200

```
{ "method": "GET", "transformRequest": [null], "transformResponse": [null], "url": "index.html", "headers": { "Accept": "application/json, text/plain, */*" } }
```

In the above example we use a \$ajax service and define the “url” and “method” property of \$ajax. On successful Callback method we are showing data of “index.html” page and on error we define the alert message of failure.

Methods

In the above example we used the .get shortcut method for \$ajax service . There are also other shortcut methods.

- \$http.get
- \$http.post
- \$http.head
- \$http.put
- \$http.delete
- \$http.patch
- \$http.jsonp

Property

The response from the server is retrieved as an object and this object contains the following properties:

| Property | Description |
|-------------|---|
| .config | The configuration object that was used to generate the request. |
| .status | Status number defining the HTTP status. |
| .data | The response body transformed with the transform functions. |
| .headers | Function to use to get header information. |
| .statusText | HTTP status text of the response. |

Node Introduction

What is Node.js?

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36. The definition of Node.js as supplied by its official documentation is as follows –

- Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Features of Node.js

Following are some of the important features that make Node.js the first choice of software architects.

- Asynchronous and Event Driven – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- Very Fast – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- Single Threaded but Highly Scalable – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single

threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

- No Buffering – Node.js applications never buffer any data. These applications simply output the data in chunks.
- License – Node.js is released under the MIT license.

Who Uses Node.js?

Following is the link on github wiki containing an exhaustive list of projects, application and companies which are using Node.js. This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.

Projects, Applications, and Companies Using Node

Concepts

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.

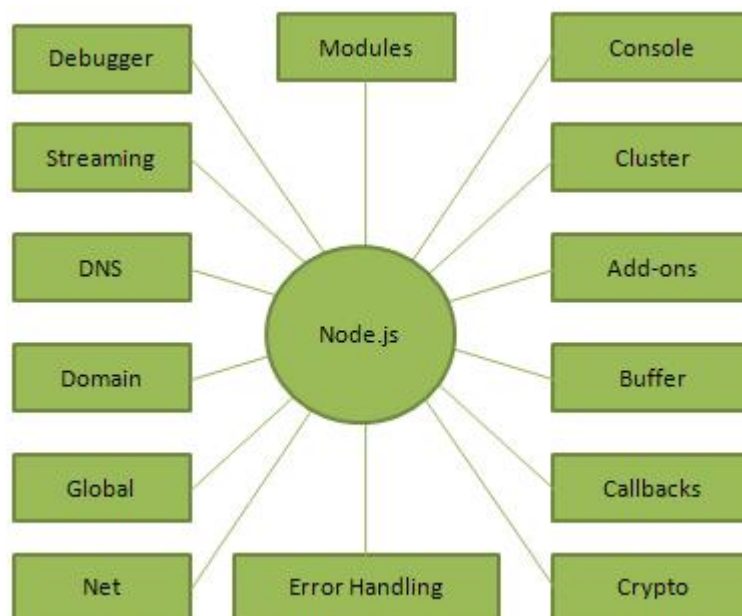


Image: Node.js Concepts

Reference: https://www.tutorialspoint.com/nodejs/images/nodejs_concepts.jpg

Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications.

Advantages of NodeJS:

Here are the benefits of using Node.js Easy Scalability: Developers prefer to use Node.js because it is easily scaling the application in both horizontal and vertical directions. We can also add extra resources during the scalability of the application.

- **Real-time web apps:** If you are building a web app you can also use PHP, and it will take the same amount of time when you use Node.js, But if I am talking about building chat apps or gaming apps Node.js is much more preferable because of faster synchronization. Also, the event loop avoids HTTP overloaded for Node.js development.
- **Fast Suite:** NodeJs runs on the V8 engine developed by Google. Event loop in NodeJs handles all asynchronous operation so NodeJs acts like a fast suite and all the operations can be done quickly like reading or writing in the database, network connection, or file system
- **Easy to learn and code:** NodeJs is easy to learn and code because it uses JavaScript. If you are a front-end developer and have a good grasp of JavaScript you can easily learn and build the application on NodeJS
- **Advantage of Caching:** It provides the caching of a single module. Whenever there is any request for the first module, it gets cached in the application memory, so you don't need to re-execute the code.
- **Data Streaming:** In NodeJs HTTP request and response are considered as two separate events. They are data stream so when you process a file at the time of loading it will reduce the overall time and will make it faster when the data is presented in the form of transmissions. It also allows you to stream audio and video files at lightning speed.
- **Hosting:** PaaS (Platform as a Service) and Heroku are the hosting platforms for NodeJS application deployment which is easy to use without facing any issue.
- **Corporate Support:** Most of the well-known companies like Walmart, Paypal, Microsoft, Yahoo are using NodeJS for building the applications. NodeJS uses JavaScript, so most of the companies are combining front-end and backend Teams together into a single unit.

Application of NodeJS:

NodeJS should be preferred to build:

- Real-Time Chats,
- Complex Single-Page applications,
- Real-time collaboration tools,
- Streaming apps
- JSON APIs based application

Installing Node and using It:

Using Website:

1. You can visit the link [Download Node](#) and download LTS version.
2. After installing the node you can check your node version in command prompt using command.

```
~ $node --version
```

After that, you can just create a folder and add a file here for example app.js. To run this file you need to execute command...

```
first app $node app.js
```

3. **Node Modules:** There are some built-in modules that you can use to create your applications. Some popular modules are- OS, fs, events, HTTP, URL and then you can include these modules in your file using these lines.

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

4. Here is an example of how to include an HTTP module to build the server...

```
var http = require('http');
// Create a server object:
http.createServer(function (req, res) {
  // Write a response to the client
  res.write('GeeksForGeeks');
  // End the response
  res.end();
// The server object listens on port 8080
}).listen(8080);
```

This will listen to the server on port 8080. Once you will run your file in command prompt it will execute your file and listen to the server on this port. You can also create your own module and include it in your file.

Using NPM:

NPM is a Node Package Manager that provides packages to download and use. It contains all the files and modules that you require in your application. To install any package you need to execute a command...

```
npm install
```

This is an example of using the Events module.

```
var events = require('events');
var EventEmitter = new events.EventEmitter();
// Create an event handler:
var myEventHandler = function () {

    console.log('Welcome to GeeksforGeeks');
}
// Assign the event handler to an event:
eventEmitter.on('geeks', myEventHandler);
// Fire the 'geeks' event:
eventEmitter.emit('geeks');
```

So this is how you can start with node and build your own applications. There are some frameworks of the node which you can use to build your applications. Some popular frameworks of node are...Express.js, Socket.io, Koa.js, Meteor.js, Sail.js.

Blocking & Nonblocking code

There are two types of execution of our code, synchronous and asynchronous. The code is executed in sequence synchronously, and execution is awaited when the function is called. In asynchronous execution, the line is not necessarily followed, and

completion of the operation is not necessary. Based on this, we have methods that are blocking and non-blocking in Node.js.

Blocking:

It refers to the blocking of further operation until the current operation finishes. Blocking methods are executed synchronously. Synchronously means that the program is executed line by line. The program waits until the called function or the operation returns.

Example: Following example uses the `readFileSync()` function to read files and demonstrate Blocking in Node.js

Index.js

```
const fs = require('fs');
const filepath = 'text.txt';
// Reads a file in a synchronous and blocking way
const data = fs.readFileSync(filepath, {encoding: 'utf8'});

// Prints the content of file
console.log(data);
// This section calculates the sum of numbers from 1 to 10
let sum = 0;
for(let i=1; i<=10; i++){
    sum = sum + i;
}
// Prints the sum
```

```
console.log('Sum: ', sum);
```

Run the index.js file using the following command:

```
node index.js
```

Output:

```
This is from text file.  
Sum: 55
```

Non-Blocking:

It refers to the program that does not block the execution of further operations. Non-Blocking methods are executed asynchronously. Asynchronously means that the program may not necessarily execute line by line. The program calls the function and move to the next operation and does not wait for it to return.

Example: Following example uses the `readFile()` function to read files and demonstrate Non-Blocking in Node.js

Index.js

```
const fs = require('fs');  
  
const filepath = 'text.txt';  
  
// Reads a file in a asynchronous and non-blocking way  
fs.readFile(filepath, {encoding: 'utf8'}, (err, data) => {  
    // Prints the content of file  
    console.log(data);  
});  
  
// This section calculates the sum of numbers from 1 to 10  
let sum = 0;
```

```
for(let i=1; i<=10; i++){
    sum = sum + i;
}
// Prints the sum
console.log('Sum: ', sum);
```

Run the index.js file using the following command

```
node index.js
```

Output:

```
Sum: 55
This is from text file.
```

Note: In the non-blocking program the sum actually prints before the content of the file. This is because the program does not wait for the `readFile()` function to return and move to the next operation. And when the `readFile()` function returns it prints the content.

Comparison between Blocking and Non-Blocking in Node.js:

Non-blocking execution is usually faster if we compare blocking and non-blocking code in node.js, allowing concurrency. It will enable the event loop to execute JavaScript callback functions after completing other operations.

Suppose, if a program takes 40ns to execute and 30ns of that work can be done asynchronously, that frees up 30ns per request handle in which the system can handle other requests.

It is not usually a good idea to mix blocking and non-blocking code in node.js:

```
const fs = require('fs');
fs.readFile('/uploads/sample.txt', (err, data) => {
```

```

if (err) throw err;

console.log(data);

});

fs.unlinkSync('/uploads/sample.txt');

```

In this code, 'fs.unlinkSync()' is likely to run first which would delete the sample.txt file before it is read. To avoid this, code can be written in clear non-blocking way:

```

const fs = require('fs');
fs.readFile('/uploads/sample.txt', (readFileErr, data) => {
  if (readFileErr) throw readFileErr;
  console.log(data);
  fs.unlink('/uploads/sample.txt', (unlinkErr) => {
    if (unlinkErr) throw unlinkErr;
  });
});

```

In this, fs.unlink() is within the callback of fs.readFile(), the order of operations will be definite.

Learning Outcomes

After completing this Elective module, a student will be able to:

1. Create Module & export, import
2. Introduction package.json file
3. File Handling
4. Create Event Driven Programming
5. Socket Programming
6. Create Own web server

AngularJS | Modules

The AngularJS module defines the functionality of the application which is applied on the entire HTML page. It helps to link many components. So it is just a group of related components. It is a container which consists of different parts like controllers, services, directives.

Note: This modules should be made in a normal HTML files like index.html and no need to create a new project in VisualStudio for this section.

How to create a Module:

```
var app = angular.module("Module-name", []);
```

In this [] we can add a list of components needed but we are not including any components in this case. This created module is bound with any tag like div, body, etc by adding it to the list of modules.

```
<div ng-app = "module-name">

    The code in which the module is required.

</div>
```

Adding a Controller:

```
app.controller("Controller-name", function($scope) {

    $scope.variable-name= "";

});
```

Here, we can add any number of variables in controller and use them in the html files, body of the tag in which the controller is added to that tag by writing:

```
<body>
<div ng-app="Module-name">
  <div ng-controller="Controller-name">
    {{variable-name}}
  </div>
<!-- This wont get printed since its
not part of the div in which
controller is included -->
{{variable-name}}
</div>
</body>
```

Module and Controllers in Files: While we can make modules and controllers in the same file along with the HTML file which requiring it however we may want to use this module in some other file. Hence this will lead to redundancy so we will prefer to create Module, Controller and HTML file separately. The Module and Controller are to be stored by using .js files and in order to use them in the HHTML file we have to include them in this way:

Example:

DemoComponent.js

```
// Here the Component name is DemoComponent
```

```
// so saving the file as DemoComponent.js
app.controller('DemoController', function($scope) {

    $scope.list = ['A', 'E', 'I', 'O', 'U'];
    $scope.choice = 'Your choice is: ADIT IBM ';

    $scope.ch = function(choice) {
        $scope.choice = "Your choice is: " + choice;
    };

    $scope.c = function() {
        $scope.choice = "Your choice is: " + $scope.mychoice;
    };
});
```

Module-name: **DemoApp.js**

```
var app = angular.module('DemoApp', []);
```


index.html file

```
<!DOCTYPE html>
<html>
<head>
  <title>
    Modules and Controllers in Files
  </title>
</head>
<body ng-app="DemoApp">
  <h1>
    Using controllers in Module
  </h1>
  <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
  </script>

  <script src="DemoApp.js"></script>
  <script src="DemoController"></script>
  <div ng-app="DemoApp" ng-controller="DemoController">
```

```

Vowels List : <button ng-click="ch('A')" >A</button>

<button ng-click="ch('E')" >E</button>

<button ng-click="ch('I')" >I</button>

<button ng-click="ch('O')" >O</button>

<button ng-click="ch('U')" >U</button>

<p>{{ choice }}</p>

Vowels List :

<select ng-options="option for option in list"
      ng-model="mychoice" ng-change="c()">

</select>

<p>{{ choice }}</p>

</div>

</body>

</html>

```

Output:

Using controllers in Module

Vowels List :

Your choice is: E

Vowels List :

Your choice is: E

Note: It makes sure the module and component files are in the same folder otherwise provide the path in which they are saved and run.

Directives in a Module: To add a directive in module follow the steps:

Creating a module like we did earlier:

```
var app = angular.module("DemoApp", []);
```

Creating a directive:

```
app.directive("Directive-name", function() {
    return {
        template : "string or some code which is to be executed"
    };
});
```

Example:

```
<!DOCTYPE html>

<html>

<head>

  <title>

    Modules and Controllers in Files

  </title>

  <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">

  </script>

</head>

<body>

  <div ng-app="GFG" w3-test-directive></div>

  <script>

var gfg_app = angular.module("GFG", []);

gfg_app.directive("w3TestDirective", function() {

  return {

    template : "Welcome to ADIT IBM!"

  };

});

  </script>
```

```
</body>
```

```
</html>
```

Output:

Welcome to ADIT IBM!

Import Module

AngularJS supports modular approach. Modules are used to separate logic such as services, controllers, application etc. from the code and maintain the code clean. We define modules in separate js files and name them as per the module.js file. In the following example, we are going to create two modules –

- **Application Module** – used to initialize an application with controller(s).
- **Controller Module** – used to define the controller.

Application Module

Here is a file named *mainApp.js* that contains the following code –

```
var mainApp = angular.module("mainApp", []);
```

Here, we declare an application **mainApp** module using `angular.module` function and pass an empty array to it. This array generally contains dependent modules.

Controller Module

studentController.js

```
mainApp.controller("studentController", function($scope) {
    $scope.student = {
        firstName: "Mahesh",
```

```

    lastName: "Parashar",
    fees:500,

    subjects:[
        {name:'Physics',marks:70},
        {name:'Chemistry',marks:80},
        {name:'Math',marks:65},
        {name:'English',marks:75},
        {name:'Hindi',marks:67}
    ],
    fullName: function() {
        var studentObject;
        studentObject = $scope.student;
        return studentObject.firstName + " " + studentObject.lastName;
    }
};
});

```

Here, we declare a controller **studentController** module using `mainApp.controller` function.

Use Modules

```

<div ng-app = "mainApp" ng-controller = "studentController">
...

```

```
<script src = "mainApp.js"></script>

<script src = "studentController.js"></script>

</div>
```

Here, we use application module using ng-app directive, and controller using ngcontroller directive. We import the mainApp.js and studentController.js in the main HTML page.

Example

The following example shows use of all the above mentioned modules.

testAngularJS.htm

```
<html>

<head>

  <title>Angular JS Modules</title>

  <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

  <script src = "/angularjs/src/module/mainApp.js"></script>

  <script src = "/angularjs/src/module/studentController.js"></script>

</head>

<style>

  table, th , td {

    border: 1px solid grey;

    border-collapse: collapse;
```

```
padding: 5px;
}
table tr:nth-child(odd) {
background-color: #f2f2f2;
}
table tr:nth-child(even) {
background-color: #ffffff;
}
</style>
</head>

<body>
<h2>AngularJS Sample Application</h2>
<div ng-app = "mainApp" ng-controller = "studentController">

<table border = "0">
<tr>
<td>Enter first name:</td>
<td><input type = "text" ng-model = "student.firstName"></td>
</tr>
<tr>
<td>Enter last name: </td>
```



```

        <td><input type = "text" ng-model = "student.lastName"></td>
    </tr>
    <tr>
        <td>Name: </td>
        <td>{{student.fullName()}}</td>
    </tr>
    <tr>
        <td>Subject:</td>

        <td>
            <table>
                <tr>
                    <th>Name</th>
                    <th>Marks</th>
                </tr>
                <tr ng-repeat = "subject in student.subjects">
                    <td>{{ subject.name }}</td>
                    <td>{{ subject.marks }}</td>
                </tr>
            </table>
        </td>
    </tr>

```

```

        </table>

    </div>

</body>
</html>

```

mainApp.js

```
var mainApp = angular.module("mainApp", []);
```

studentController.js

```

mainApp.controller("studentController", function($scope) {

    $scope.student = {

        firstName: "Mahesh",

        lastName: "Parashar",

        fees:500,


        subjects:[

            {name:'Physics',marks:70},

            {name:'Chemistry',marks:80},

            {name:'Math',marks:65},

```

```

        {name:'English',marks:75},
        {name:'Hindi',marks:67}
    ],
    fullName: function() {
        var studentObject;
        studentObject = $scope.student;
        return studentObject.firstName + " " + studentObject.lastName;
    }
};
});

```

Output

Open the file *textAngularJS.htm* in a web browser. See the result.

AngularJS Sample Application

| Enter first name: | Mahesh | | |
|-------------------|--|------|-------|
| Enter last name: | Parashar | | |
| Name: | Mahesh Parashar | | |
| Subject: | <table> <tr> <th>Name</th><th>Marks</th></tr> </table> | Name | Marks |
| Name | Marks | | |

| | | |
|--|-----------|----|
| | Physics | 70 |
| | Chemistry | 80 |
| | Math | 65 |
| | English | 75 |
| | Hindi | 67 |

Introduction package.json file

What Is package.json?

Learn all about a Node project's *package.json*, npm's configuration file housed in the root directory of your project. In this tutorial, learn how to manage metadata in an initial *package.json* file such as name, version, description, and keywords, as well as dependencies and devDependencies. Learn how to run, develop, and optionally publish your project to NPM.

Goal

Understand what a package.json file is, how it relates to your project, and what common properties we need to know about.

Get to know package.json

If you've worked with Node.js before, you have likely encountered a package.json file. It is a [JSON](#) file that lives in the root directory of your project. Your package.json holds important information about the project. It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.

An example package.json might look like this:

```
{
  "name": "my-project",
  "version": "1.5.0",
  "description": "Express server project using compression",
  "main": "src/index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon",
    "lint": "eslint **/*.js"
  },
  "dependencies": {
    "express": "^4.16.4",
    "compression": "~1.7.4"
  },
  "devDependencies": {
    "eslint": "^5.16.0",
    "nodemon": "^1.18.11"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/osiolabs/example.git"
  }
}
```

```

    },
    "author": "Jon Church",
    "contributors": [{
      "name": "Amber Matz",
      "email": "example@example.com",
      "url": "https://www.osiolabs.com/#team"
    }],
    "keywords": ["server", "osiolabs", "express", "compression"]
  }

```

What is the purpose of package.json?

Your project's package.json is the central place to configure and describe how to interact with and run your application. It is used by the npm CLI (and yarn) to identify your project and understand how to handle the project's dependencies. It's the package.json file that enables npm to start your project, run scripts, install dependencies, publish to the NPM registry, and many other useful tasks. The npm CLI is also the best way to manage your package.json because it helps generate and update your package.json file throughout a project's life.

Your package.json fills several roles in the lifecycle of your project, some of which only apply for packages published to NPM. If you're not publishing your project to the NPM registry or otherwise making it publicly available to others, your package.json is still essential to the development flow.

Your project also must include a package.json before any packages can be installed from NPM. This is probably the top reason why you need one in your project.

Common fields in package.json

Let's look at some of the most common and important fields that can be in a package.json, to better understand how to use and manage this essential file. Some are required for publishing to NPM, while others help the npm CLI run the application or install dependencies.

There are more fields than the ones we cover, and you can read about the rest in the [documentation](#), but these are the essential package.json properties to understand.

name

```
"name": "my-project"
```

The name field defines the name of the package. When publishing to the NPM registry, this is the name the package will be listed under. It must be no more than 214 characters, only lowercase letters, and it must be URL-safe (hyphens and underscores allowed, but no spaces or other characters disallowed in URLs).

If publishing your package to NPM, the name property is required and must be unique. You'll receive an error if trying to publish a package under a name that is currently used on the NPM registry. If you aren't developing a package which you'll eventually publish, the name does not have to be unique.

version

```
"version": "1.5.0",
```

The version field is very important for any published package, and required before publishing. It is the current version of the software that the package.json is describing.

You are not required to use [SemVer](#), but it is the standard used in the Node.js ecosystem and highly recommended. For an unpublished package, this property isn't strictly required. Typically, the version number is bumped according to SemVer before publishing new versions to NPM.

This workflow isn't typically used when a package is not being relied upon as a dependency, or the package isn't being published to NPM. But if a package is being used as a dependency, keeping the version field up to date is very important to make

sure others are using the proper version of a package. [Learn more about semantic versioning](#).

license

This is a very important but often overlooked property. The license field lets us define what license applies to the code the *package.json* is describing. Again, this is very important when publishing a project to the NPM registry, as the license may limit the use of your software by some developers or organizations. Having a clear license in place helps clearly define what terms the software is able to be used under.

The value of this field will usually be the license's identifier code -- a string like "MIT" or "ISC" for the [MIT](#) license and [ISC](#) license respectively. If you don't wish to provide a license, or explicitly do not want to grant use of a private or unpublished package, you can put "UNLICENSED" as the license. [Choose a License](#) is a helpful resource if you're not sure which license to use.

author and contributors

```
"author": "Jon Church jon@example.com https://www.osiolabs.com/#team",
"contributors": [{
  "name": "Amber Matz",
  "email": "example@example.com",
  "url": "https://www.osiolabs.com/#team"
}],
```

The author and contributor fields function similarly. They are both "people" fields which can be either a string in the format of "Name <email> <url>" , or an object with fields name, email, url. The email and url are both optional.

Author is for a single person, and contributors is an array of people.

These fields are a useful way to list contacts for a public project, as well as share credit with contributors.

description

The description field is used by the NPM registry for published packages, to describe the package in search results and on the npmjs.com website.

This string is used to help surface packages when users search the NPM registry. This should be a short summary of what the package is for.

It can also be useful as simple documentation for your project, even if you aren't publishing it to the NPM registry.

keywords

```
"keywords": ["server", "osiolabs", "express", "compression"]
```

The keywords field is an array of strings, and serves a similar purpose to the description. This field is indexed by the NPM registry to help find packages when someone searches for them. Each value in the array is one keyword associated with your package.

This field doesn't have much use if you're not publishing to the NPM registry, and you can feel free to omit it.

main

```
"main": "src/index.js",
```

The main field is a functional property of your *package.json*. This defines the entry point into your project, and commonly the file used to start the project.

If your package (let's say its name is *foo-lib*) is installed by a user, then when a user does `require('foo-lib')`, it is the `module.exports` property of the file listed in the main field that is returned by `require`.

This is commonly an *index.js* file in the root of your project, but it can be any file you choose to use as the main entry-point to your package.

scripts

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon"
}
```

The scripts field is another functional piece of metadata in your *package.json*. The scripts property takes an object with its keys being scripts we can run with `npm run <scriptName>`, and the value is the actual command which is run. These are typically terminal commands, which we put into the scripts field so we can both document them and reuse them easily.

Scripts are powerful tools that the npm CLI can use to run tasks for your project. They can do the job of most task runners used during development. [Learn more about npm scripts](#).

repository

```
"repository": {
  "type": "git",
  "url": "https://github.com/osiolabs/example.git"
}
```

You can record the repository the code for a project lives in by providing the repository field. This field is an object which defines the url where the source code is located, and what type of version control system it uses. For open source projects, this is likely GitHub or Bitbucket with Git as the version control system.

An important note is that the URL field is meant to point to where the version control can be accessed from, not just the released code base.

dependencies

```
"dependencies": {
```

```
"express": "^4.16.4",
"compression": "~1.7.4"
}
```

This is one of the most important fields in your *package.json*, and likely the entire reason you need one. All of the dependencies your project uses (the external code that the project relies on) are listed here. When a package is installed using the npm CLI, it is downloaded to your *node_modules/* folder and an entry is added to your dependencies property, noting the name of the package and the installed version.

The dependency field is an object with package names as keys, and a version or version range as a value. From this list, npm knows what packages to fetch and install (and at what versions) when npm install is run in the directory. The dependency field of your *package.json* is at the heart of your project, and defines the external packages your project requires.

The carets (^) and tildes (~) you see in the dependency versions are notation for version ranges defined in SemVer. [Learn more about dependency versions and SemVer.](#)

devDependencies

```
"devDependencies": {
  "nodemon": "^1.18.11"
}
```

Similar to the dependencies field, but for packages which are only needed during development, and aren't needed in production.

An example would be using a tool to reload your project during development, like [nodemon](#), which we have no use for once the application is deployed and in production.

The `devDependencies` property lets us explicitly note which dependencies aren't needed in production. When installing your app in a production environment, you can use `npm install --production` to only install what is listed in the dependency field of *package.json*.

Recording a *devDependency* is a great way to document what tools are needed for the app during development. To install a package from npm as a `devDependency`, you can run `npm install --save-dev <package>`.

There is another way the `devDependencies` property is useful to us, and that's by using them in our npm scripts. [Learn more about using devDependencies in npm scripts](#).

Manage your *package.json*

A *package.json* file must be valid JSON. This means any missing commas, unclosed quotes, or other formatting errors will prevent npm from interacting with the *package.json*. If you do introduce an error, the next time you run an npm command you will see an error from npm. It's recommended to use the npm CLI for updating and managing your *package.json* when possible, to avoid accidentally introducing errors to your *package.json*, and to make managing your dependencies easier.

Using `npm init` to [create your package.json](#) will help to ensure you generate a valid file.

Dependencies are best managed by using npm's commands `npm install`, `npm uninstall`, and `npm update`, so your *package.json* and *node_modules/* folder are kept in sync. Manually adding a dependency listing will require you to run `npm install` before the dependency is actually installed to your project.

Because our *package.json* is only where we record dependencies, and our *node_modules/* folder is where the actual code for dependencies is installed, manually updating the dependency field of *package.json* does not immediately reflect the state of our *node_modules/* folder. That's why you want to use npm to help manage dependencies, because it will update both the *package.json* and *node_modules/* folder in tandem.

You can always edit your *package.json* manually in your text editor and make changes. That works well for most fields, so long as you're careful not to introduce any JSON formatting errors. We recommend you use the npm CLI commands wherever you can, however.

Recap

The *package.json* file is the heart of any Node project. It records important metadata about a project which is required before publishing to NPM, and also defines functional attributes of a project that npm uses to install dependencies, run scripts, and identify the entry point to our package.

Not all fields available in *package.json* will apply to you, but we can achieve some powerful benefits by recording information about our application in its *package.json* file. Understanding the role of *package.json* and how it relates to npm is an important part of developing Node.js apps, and increasingly an important part of the JavaScript ecosystem.

How to get file content and other details in AngularJS?

We can get the file content by using some basic angular functions and other details like the name and size of the file in AngularJS. To understand it look into the below example where both HTML and JS files are implemented.

Note: Consider below two files are of same component in angular.

app.module.html:

```
<!-- Script for display  
data in particular format -->  
<!DOCTYPE html>
```

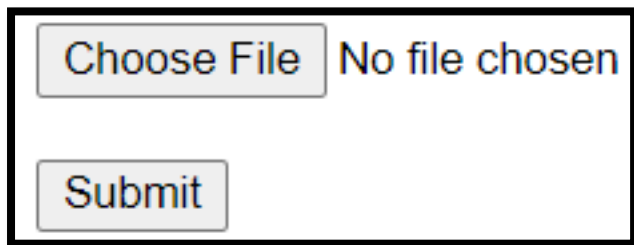
```
<html>

<script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
</script>

<body ng-app="myApp">
  <div ng-controller="MyCtrl">
    <input type="file" id="myFileInput" />
    <button ng-click="submit()"> Submit</button>
    <br /><br />
    <h1>
      Filename: {{ fileName }}
    </h1>
    <h2>
      File size: {{ fileSize }} Bytes
    </h2>
    <h2>
      File Content: {{ fileContent }}
    </h2>
  </div>
</body>
```

```
</html>
```

Output:



In the above HTML file we have simply made a structure to how it should be looked on the webpage. For that, we have used some angular stuff like 'ng-controller' and also doubly curly brackets which we will implement in the below javascript code.

app.module.ts:

```
import { BrowserModule } from
  '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from
  '@angular/platform-browser/animations';
import { FormsModule, ReactiveFormsModule }
  from '@angular/forms';
```

```
import { MatInputModule } from
    '@angular/material/input';
import { MatDialogModule } from
    '@angular/material/dialog';
import { MatFormFieldModule } from
    '@angular/material/form-field';
import { MatIconModule } from
    '@angular/material/icon';
```

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatFormFieldModule,
    MatIconModule,
```



```

        MatDialogModule,
    ],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

app.component.ts:

```

// Code to get file content
// and other data
import { Component, OnInit }
    from '@angular/core';
// Imports
import { FormGroup, FormControl,
    } from '@angular/forms';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {

```

```
constructor() { }  
  
ngOnInit() {  
  
}  
  
var myApp = angular.module('myApp', []);  
myApp.controller('MyCtrl', function ($scope) {  
  
    // Initially declaring empty string  
  
    // and assigning size to zero  
  
    $scope.fileContent = "";  
  
    $scope.fileSize = 0;  
  
    $scope.fileName = "";  
  
    // Implementing submit function  
  
    $scope.submit = function () {  
  
        var file = document.getElementById("myFileInput")  
            .files[0];  
  
        if(file) {  
  
            var Reader = new FileReader();  
  
            Reader.readAsText(file, "UTF-8");  
  
            Reader.onload = function (evt) {
```

```

        // Getting required result
        // of the file
        $scope.fileContent = Reader.result;
        $scope.fileName = document.getElementById(
            "myFileInput").files[0].name;
        $scope.fileSize = document.getElementById(
            "myFileInput").files[0].size;;
    }
    // Printing error if data
    //is not proper
    Reader.onerror = function (evt) {
        $scope.fileContent = "error";
    }
}
}
}
});

```

Output:

Choose File sample.docx

Submit

Filename: sample.docx

File size: 20600 Bytes

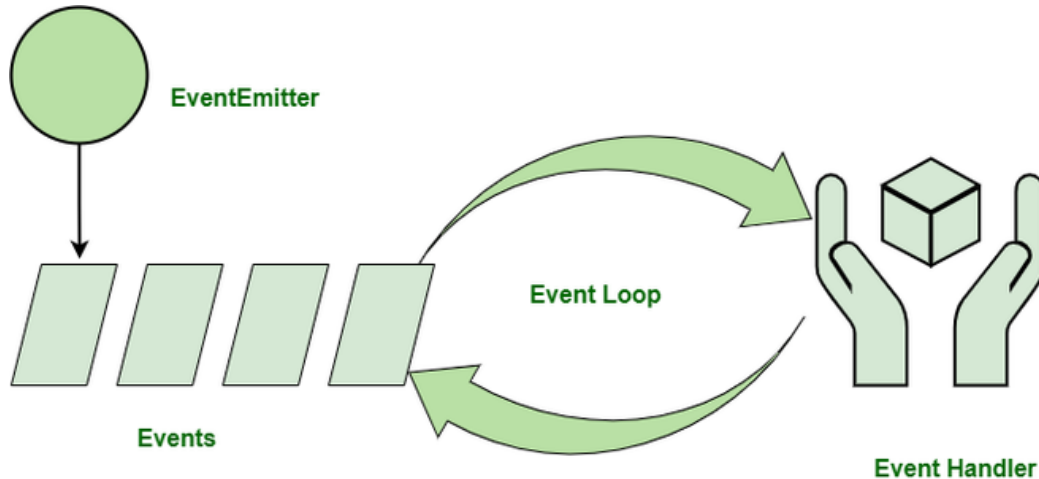
File Content: Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in

Explain Event-Driven Programming in Node.js

Event-Driven Programming in Node.js: Node.js makes extensive use of events which is one of the reasons behind its speed when compared to other similar technologies. Once we start a Node.js server, it initializes the variables and functions and then listens for the occurrence of an event.

Event-driven programming is used to synchronize the occurrence of multiple events and to make the program as simple as possible. The basic components of an Event-Driven Program are:

- A callback function (called an event handler) is called when an event is triggered.
- An event loop that listens for event triggers and calls the corresponding event handler for that event.



Ref: <https://media.geeksforgeeks.org/wp-content/uploads/20211017211104/EDP1drawio.png>

A function that listens for the triggering of an event is said to be an 'Observer'. It gets triggered when an event occurs. Node.js provides a range of events that are already in-built. These 'events' can be accessed via the 'events' module and the EventEmitter class. Most of the in-built modules of Node.js inherit from the EventEmitter class

EventEmitter: The EventEmitter is a Node module that allows objects to communicate with one another. The core of Node's asynchronous event-driven architecture is EventEmitter. Many of Node's built-in modules inherit from EventEmitter.

The idea is simple – emitter objects send out named events, which trigger listeners that have already been registered. Hence, an emitter object has two key characteristics:

- **Emitting name events:** The signal that something has happened is called emitting an event. A status change in the emitting object is often the cause of this condition.
- **Registering and unregistering listener functions:** It refers to the binding and unbinding of the callback functions with their corresponding events.

Event-Driven Programming Principles:

- A suite of functions for handling the events. These can be either blocking or non-blocking, depending on the implementation.
- Binding registered functions to events.
- When a registered event is received, an event loop polls for new events and calls the matching event handler(s).

Implementation: Filename: app.js

```
// Import the 'events' module
const events = require('events');

// Instantiate an EventEmitter object
const eventEmitter = new events.EventEmitter();

// Handler associated with the event
const connectHandler = function connected() {
  console.log('Connection established.');
```



```
  // Trigger the corresponding event
  eventEmitter.emit('data_received');
```



```
}
```

```
// Binds the event with handler
eventEmitter.on('connection', connectHandler);

// Binds the data received
eventEmitter.on(
  'data_received', function () {
    console.log('Data Transfer Successful.');
```

```
  });

// Trigger the connection event
eventEmitter.emit('connection');
console.log("Finish");
```

The above code snippet binds the handler named ‘connectHandler’ with the event ‘connection’. The callback function is triggered when the event is emitted.

Run the *app.js* file using the following command:

```
node app.js
```

Output:

```
Connection established.
Data Transfer Successful.
Finish
```

Advantages of Event-Driven Programming:

- **Flexibility:** It is easier to alter sections of code as and when required.
- **Suitability for graphical interfaces:** It allows the user to select tools (like radio buttons etc.) directly from the toolbar
- **Programming simplicity:** It supports predictive coding, which improves the programmer's coding experience.
- **Easy to find natural dividing lines:** Natural dividing lines for unit testing infrastructure are easy to come by.
- **A good way to model systems:** Useful method for modeling systems that must be asynchronous and reactive.
- **Allows for more interactive programs:** It enables more interactive programming. Event-driven programming is used in almost all recent GUI apps.
- **Using hardware interrupts:** It can be accomplished via hardware interrupts, lowering the computer's power consumption.
- **Allows sensors and other hardware:** It makes it simple for sensors and other hardware to communicate with software.

Disadvantages of Event-Driven Programming:

- **Complex:** Simple programs become unnecessarily complex.
- **Less logical and obvious:** The flow of the program is usually less logical and more obvious
- **Difficult to find error:** Debugging an event-driven program is difficult
- **Confusing:** Too many forms in a program might be confusing and/or frustrating for the programmer.
- **Tight coupling:** The event schema will be tightly coupled with the consumers of the schema.

- **Blocking:** Complex blocking of operations.

Relation between Event-Driven Programming and Object-Oriented

Programming: We can combine Object-oriented Programming (OOP) and Event-driven programming (EDP) and use them together in the same code snippet.

When OOP is used with EDP:

- All OOP fundamentals (encapsulation, inheritance, and polymorphism) are preserved.
- Objects get the ability to post-event notifications and subscribe to event notifications from other objects.

How to distinguish between OOP with and without EDP: The control flow between objects is the distinction between OOP with and without EDP. On a method call in OOP without EDP, control flows from one object to another. The primary function of an object is to call the methods of other objects.

However, on event notification, control in OOP with EDP moves from one object to another. Object subscribes to notifications from other objects, waits for notifications from those objects, performs work based on the notification, and then publishes its own notifications.

How To Create a Real-Time App with Socket.IO, Angular, and Node.js

Introduction

WebSocket is the internet protocol that allows for full-duplex communication between a server and clients. This protocol goes beyond the typical HTTP request and response paradigm. With WebSockets, the server may send data to a client without the client initiating a request, thus allowing for some very interesting applications.

In this tutorial, you will build a real-time document collaboration application (similar to Google Docs). We'll be using the [Socket.IO](#) Node.js server framework and [Angular 7](#) to accomplish this.

You can find the complete [source code for this example project on GitHub](#).

Prerequisites

To complete this tutorial, you will need:

- Node.js installed locally, which you can do by following [How to Install Node.js and Create a Local Development Environment](#).
- A modern web browser that [supports WebSocket](#).

This tutorial was originally written in an environment consisting of Node.js v8.11.4, npm v6.4.1, and Angular v7.0.4.

This tutorial was verified with Node v14.6.0, npm v6.14.7, Angular v10.0.5, and [Socket.IO](#) v2.3.0.

Step 1 — Setting Up the Project Directory and Creating the Socket Server

First, open your terminal and create a new project directory that will hold both our server and client code:

1. `mkdir socket-example`

Next, change into the project directory:

1. `cd socket-example`

Then, create a new directory for the server code:

1. `mkdir socket-server`

into the server directory.

1. `cd socket-server`

Then, initialize a new npm project:

1. `npm init -y`

Now, we will install our package dependencies:

1. `npm install express@4.17.1 socket.io@2.3.0 @types/socket.io@2.1.10 --save`

These packages include Express, [Socket.IO](#), and `@types/socket.io`.

Now that you have completed setting up the project, you can move on to writing code for the server.

First, create a new `src` directory:

1. `mkdir src`

Now, create a new file called `app.js` in the `src` directory, and open it using your favorite text editor:

1. `nano src/app.js`

Start with the require statements for Express and [Socket.IO](#):

```
socket-server/src/app.js

const app = require('express')();
const http = require('http').Server(app);
const io = require('socket.io')(http);
```

As you can tell, we're using Express and [Socket.IO](#) to set up our server. [Socket.IO](#) provides a layer of abstraction over native WebSockets. It comes with some nice features, such as a fallback mechanism for older browsers that do not support WebSockets, and the ability to create *rooms*. We'll see this in action in a minute.

For the purposes of our real-time document collaboration application, we will need a way to store documents. In a production setting, you would want to use a database, but for the scope of this tutorial, we will use an in-memory store of documents:

```
socket-server/src/app.js
```

```
const documents = {};
```

Now, let's define what we want our socket server to actually do:

```
socket-server/src/app.js
io.on("connection", socket => {
  // ...
});
```

Let's break this down. `.on('...')` is an event listener. The first parameter is the name of the event, and the second one is usually a callback executed when the event fires, with the event payload.

The first example we see is when a client connects to the socket server (connection is a reserved event type in [Socket.IO](#)).

We get a socket variable to pass to our callback to initiate communication to either that one socket or to multiple sockets (i.e., broadcasting).

safeJoin

We will set up a local function (safeJoin) that takes care of joining and leaving *rooms*:

```
socket-server/src/app.js
io.on("connection", socket => {
  let previousId;

  const safeJoin = currentId => {
    socket.leave(previousId);

    socket.join(currentId, () => console.log(`Socket ${socket.id} joined room
    ${currentId}`));
  };
});
```

```
    previousId = currentId;  
};  
  
// ...  
});
```

In this case, when a client has joined a room, they are editing a particular document. So if multiple clients are in the same room, they are all editing the same document.

Technically, a socket can be in multiple rooms, but we don't want to let one client edit multiple documents at the same time, so if they switch documents, we need to leave the previous room and join the new room. This little function takes care of that.

There are three event types that our socket is listening for from the client:

- getDoc
- addDoc
- editDoc

And two event types that are emitted by our socket to the client:

- document
- documents

getDoc

Let's work on the first event type - getDoc:

```
socket-server/src/app.js  
  
io.on("connection", socket => {  
  // ...
```

```
socket.on("getDoc", docId => {
  safeJoin(docId);
  socket.emit("document", documents[docId]);
});

// ...

});
```

When the client emits the getDoc event, the socket is going to take the payload (in our case, it's just an id), join a room with that docId, and emit the stored document back to the initiating client only. That's where socket.emit('document', ...) comes into play.

addDoc

Let's work on the second event type - addDoc:

```
socket-server/src/app.js

io.on("connection", socket => {
  // ...

  socket.on("addDoc", doc => {
    documents[doc.id] = doc;
    safeJoin(doc.id);
    io.emit("documents", Object.keys(documents));
    socket.emit("document", doc);
  });
});
```

```
// ...  
});
```

With the addDoc event, the payload is a document object, which, at the moment, consists only of an id generated by the client. We tell our socket to join the room of that ID so that any future edits can be broadcast to anyone in the same room.

Next, we want everyone connected to our server to know that there is a new document to work with, so we broadcast to all clients with the io.emit('documents', ...) function.

Note the difference between socket.emit() and io.emit() - the socket version is for emitting back to only initiating the client, the io version is for emitting to everyone connected to our server.

editDoc

Let's work on the third event type - editDoc:

```
socket-server/src/app.js  
  
io.on("connection", socket => {  
  // ...  
  
  socket.on("editDoc", doc => {  
    documents[doc.id] = doc;  
    socket.to(doc.id).emit("document", doc);  
  });  
  
  // ...
```

```
});
```

With the editDoc event, the payload will be the whole document at its state after any keystroke. We'll replace the existing document in the database and then broadcast the new document to only the clients that are currently viewing that document. We do this by calling `socket.to(doc.id).emit(document, doc)`, which emits to all sockets in that particular room.

Finally, whenever a new connection is made, we broadcast to all the clients to ensure the new connection receives the latest document changes when they connect:

```
socket-server/src/app.js
io.on("connection", socket => {
  // ...

  io.emit("documents", Object.keys(documents));

  console.log(`Socket ${socket.id} has connected`);
});
```

After the socket functions are all set up, pick a port and listen on it:

```
socket-server/src/app.js
http.listen(4444, () => {
  console.log('Listening on port 4444');
});
```

Run the following command in your terminal to start the server:

```
node src/app.js
```


We now have a fully-functioning socket server for document collaboration!

Step 2 — Installing @angular/cli and Creating the Client App

Open a new terminal window and navigate to the project directory.

Run the following commands to install the Angular CLI as a devDependency:

1. `npm install @angular/cli@10.0.4 --save-dev`

Now, use the @angular/cli command to create a new Angular project, with no Angular Routing and with SCSS for styling:

1. `ng new socket-app --routing=false --style=scss`

Then, change into the server directory:

1. `cd socket-app`

Now, we will install our package dependencies:

1. `npm install ngx-socket-io@3.2.0 --save`

ngx-socket-io is an Angular wrapper over [Socket.IO](https://socket.io/) client libraries.

Then, use the @angular/cli command to generate a document model, a document-list component, a document component, and a document service:

1. `ng generate class models/document --type=model`
2. `ng generate component components/document-list`
3. `ng generate component components/document`
4. `ng generate service services/document`

Now that you have completed setting up the project, you can move on to writing code for the client.

App Module

Open app.modules.ts:

1. `nano src/app/app.module.ts`

And import FormsModule, SocketioModule, and SocketioConfig:

```
socket-app/src/app/app.module.ts
```

```
// ... other imports
```

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

```
import { SocketioModule, SocketioConfig } from 'ngx-socket-io';
```

And before your @NgModule declaration, define config:

```
socket-app/src/app/app.module.ts
```

```
const config: SocketioConfig = { url: 'http://localhost:4444', options: {} };
```

You'll notice that this is the port number that we declared earlier in the server's app.js.

Now, add to your imports array, so it looks like:

```
socket-app/src/app/app.module.ts
```

```
@NgModule({
```

```
  // ...
```

```
  imports: [
```

```
    // ...
```

```
    FormsModule,
```

```
    SocketioModule.forRoot(config)
```

```
  ],
```

```
  // ...
```

```
})
```

This will fire off the connection to our socket server as soon as AppModule loads.

Document Model and Document Service

Open document.model.ts:

1. nano src/app/models/document.model.ts

And define id and doc:

```
socket-app/src/app/models/document.model.ts

export class Document {
  id: string;
  doc: string;
}
```

Open document.service.ts:

1. nano src/app/services/document.service.ts

And add the following in the class definition:

```
socket-app/src/app/services/document.service.ts

import { Injectable } from '@angular/core';
import { Socket } from 'ngx-socket-io';
import { Document } from 'src/app/models/document.model';

@Injectable({
  providedIn: 'root'
})
export class DocumentService {
```

```

currentDocument = this.socket.fromEvent<Document>('document');
documents = this.socket.fromEvent<string[]>('documents');

constructor(private socket: Socket) { }

getDocument(id: string) {
  this.socket.emit('getDoc', id);
}

newDocument() {
  this.socket.emit('addDoc', { id: this.docId(), doc: " " });
}

editDocument(document: Document) {
  this.socket.emit('editDoc', document);
}

private docId() {
  let text = "";
  const possible =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';

```

```

for (let i = 0; i < 5; i++) {
  text += possible.charAt(Math.floor(Math.random() * possible.length));
}

return text;
}
}

```

The methods here represent each emit the three event types that the socket server is listening for. The properties `currentDocument` and `documents` represent the events emitted by the socket server, which is consumed on the client as an `Observable`.

You may notice a call to `this.docId()`. This is a little private method that generates a random string to assign as the document id.

Document List Component

Let's put the list of documents in a `sidenav`. Right now, it's only showing the `docId` - a random string of characters.

Open `document-list.component.html`:

1. `nano src/app/components/document-list/document-list.component.html`
- 2.

And replace the contents with the following:

```

socket-app/src/app/components/document-list/document-list.component.html

<div class='sidenav'>

  <span

    (click)='newDoc()'

```

```
>
  New Document
</span>
<span
  [class.selected]='docId === currentDoc'
  (click)='loadDoc(docId)'
  *ngFor='let docId of documents | async'
>
  {{ docId }}
</span>
</div>
```

Open document-list.component.scss:

1. nano src/app/components/document-list/document-list.component.scss

And add some styles:

```
socket-app/src/app/components/document-list/document-list.component.scss

.sidenav {
  background-color: #111111;
  height: 100%;
  left: 0;
  overflow-x: hidden;
  padding-top: 20px;
  position: fixed;
```

```

top: 0;
width: 220px;

span {
  color: #818181;
  display: block;
  font-family: 'Roboto', Tahoma, Geneva, Verdana, sans-serif;
  font-size: 25px;
  padding: 6px 8px 6px 16px;
  text-decoration: none;

  &.selected {
    color: #e1e1e1;
  }

  &:hover {
    color: #f1f1f1;
    cursor: pointer;
  }
}
}

```

Open document-list.component.ts:

1. nano src/app/components/document-list/document-list.component.ts

And add the following in the class definition:

```
socket-app/src/app/components/document-list/document-list.component.ts
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Observable, Subscription } from 'rxjs';

import { DocumentService } from 'src/app/services/document.service';

@Component({
  selector: 'app-document-list',
  templateUrl: './document-list.component.html',
  styleUrls: ['./document-list.component.scss']
})
export class DocumentListComponent implements OnInit, OnDestroy {
  documents: Observable<string[]>;
  currentDoc: string;
  private _docSub: Subscription;

  constructor(private documentService: DocumentService) { }

  ngOnInit() {
    this.documents = this.documentService.documents;
```



```

    this._docSub = this.documentService.currentDocument.subscribe(doc =>
this.currentDoc = doc.id);
}

ngOnDestroy() {
    this._docSub.unsubscribe();
}

loadDoc(id: string) {
    this.documentService.getDocument(id);
}

newDoc() {
    this.documentService.newDocument();
}
}

```

Let's start with the properties. `documents` will be a stream of all available documents. `currentDocId` is the id of the currently selected document. The document list needs to know what document we're on, so we can highlight that doc id in the sidebar. `_docSub` is a reference to the Subscription that gives us the current or selected doc. We need this so we can unsubscribe in the `ngOnDestroy` lifecycle method.

You'll notice the methods `loadDoc()` and `newDoc()` don't return or assign anything. Remember, these fire off events to the socket server, which turns around and fires an event back to our Observables. The returned values for getting an existing document or adding a new document are realized from the Observable patterns above.

Document Component

This will be the document editing surface.

Open document.component.html:

1. nano src/app/components/document/document.component.html

And replace the contents with the following:

```
socket-app/src/app/components/document/document.component.html
<textarea
  [(ngModel)]='document.doc'
  (keyup)='editDoc()'
  placeholder='Start typing...'
></textarea>
```

Open document.component.scss:

1. nano src/app/components/document/document.component.scss

And change some styles on the default HTML textarea:

```
socket-app/src/app/components/document/document.component.scss
textarea {
  border: none;
  font-size: 18pt;
  height: 100%;
  padding: 20px 0 20px 15px;
  position: fixed;
```

```

resize: none;

right: 0;

top: 0;

width: calc(100% - 235px);

}

```

Open document.component.ts:

1. src/app/components/document/document.component.ts

And add the following in the class definition:

```

socket-app/src/app/components/document/document.component.ts

import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { startWith } from 'rxjs/operators';

import { Document } from 'src/app/models/document.model';
import { DocumentService } from 'src/app/services/document.service';

@Component({
  selector: 'app-document',
  templateUrl: './document.component.html',
  styleUrls: ['./document.component.scss']
})
export class DocumentComponent implements OnInit, OnDestroy {

```

```

document: Document;
private _docSub: Subscription;

constructor(private documentService: DocumentService) { }

ngOnInit() {
  this._docSub = this.documentService.currentDocument.pipe(
    startWith({ id: '', doc: 'Select an existing document or create a new one to get
started' })
  ).subscribe(document => this.document = document);
}

ngOnDestroy() {
  this._docSub.unsubscribe();
}

editDoc() {
  this.documentService.editDocument(this.document);
}
}

```

Similar to the pattern we used in the DocumentListComponent above, we're going to subscribe to the changes for our current document, and fire off an event to the socket server whenever we change the current document. This means that we will see all the

changes if any other client is editing the same document we are, and vice versa. We use the RxJS `startWith` operator to give a little message to our users when they first open the app.

AppComponent

Open `app.component.html`:

1. `nano src/app.component.html`

And compose the two custom components by replacing the contents with the following:

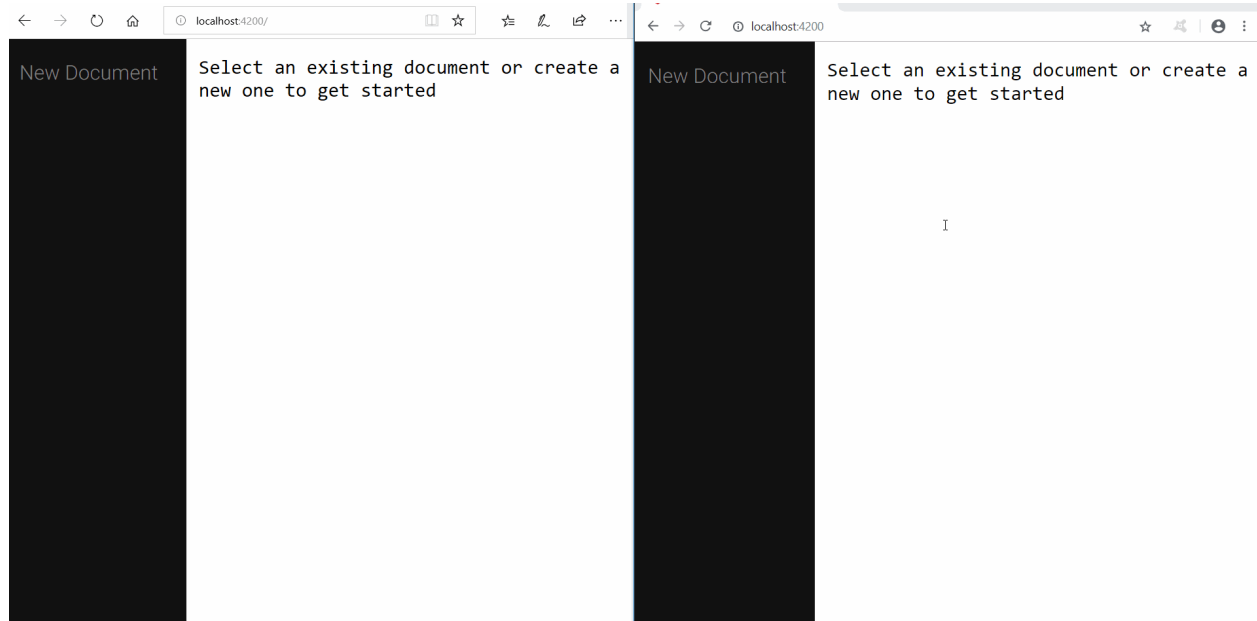
```
socket-app/src/app.component.html
<app-document-list></app-document-list>
<app-document></app-document>
```

Step 3 — Viewing the App in Action

With our socket server still running in a terminal window, let's open a new terminal window and start our Angular app:

1. `ng serve`

Open more than one instance of `http://localhost:4200` in separate browser tabs and see it in action.



Now, you can create new documents and see them update in both browser windows. You can make a change in one browser window and see the change reflected in the other browser window.

Conclusion

In this tutorial, you have completed an initial exploration into using WebSocket. You used it to build a real-time document collaboration application. It supports multiple browser sessions to connect to a server and update and modify multiple documents.

If you'd like to learn more about Angular, check out [our Angular topic page](#) for exercises and programming projects.

If you'd like to learn more about [Socket.IO](#), check out [Integrating Vue.js and Socket.IO](#).

Further WebSocket projects include real-time chat applications. See [How To Build a Realtime Chat App with React and GraphQL](#).

Node.js Web Server

In this section, we will learn how to create a simple Node.js web server and handle HTTP requests.

To access web pages of any web application, you need a [web server](#). The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

server.js

```
var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) { // 2 - creating server
    //handle incoming requests here..
});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')
```

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and

response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

```
C:\> node server.js
Node.js web server at port 5000 is running..
```

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

Handle HTTP Request

The http.createServer() method includes [request](#) and [response](#) parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

server.js

```
var http = require('http'); // Import Node.js core module

var server = http.createServer(function (req, res) { //create web server
  if (req.url == '/') { //check the URL of the current request
    // set response header
    res.writeHead(200, { 'Content-Type': 'text/html' });

    // set response content
    res.write('<html><body><p>This is home Page.</p></body></html>');
    res.end();
  }
});
```



```

    }

    else if (req.url == "/student") {

        res.writeHead(200, { 'Content-Type': 'text/html' });

        res.write('<html><body><p>This is student Page.</p></body></html>');

        res.end();

    }

    else if (req.url == "/admin") {

        res.writeHead(200, { 'Content-Type': 'text/html' });

        res.write('<html><body><p>This is admin Page.</p></body></html>');

        res.end();

    }

    else

        res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

```

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

```
C:\> node server.js
```

```
Node.js web server at port 5000 is running..
```

To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

```
curl -i http://localhost:5000
```

You should see the following response.

HTTP/1.1 200 OK

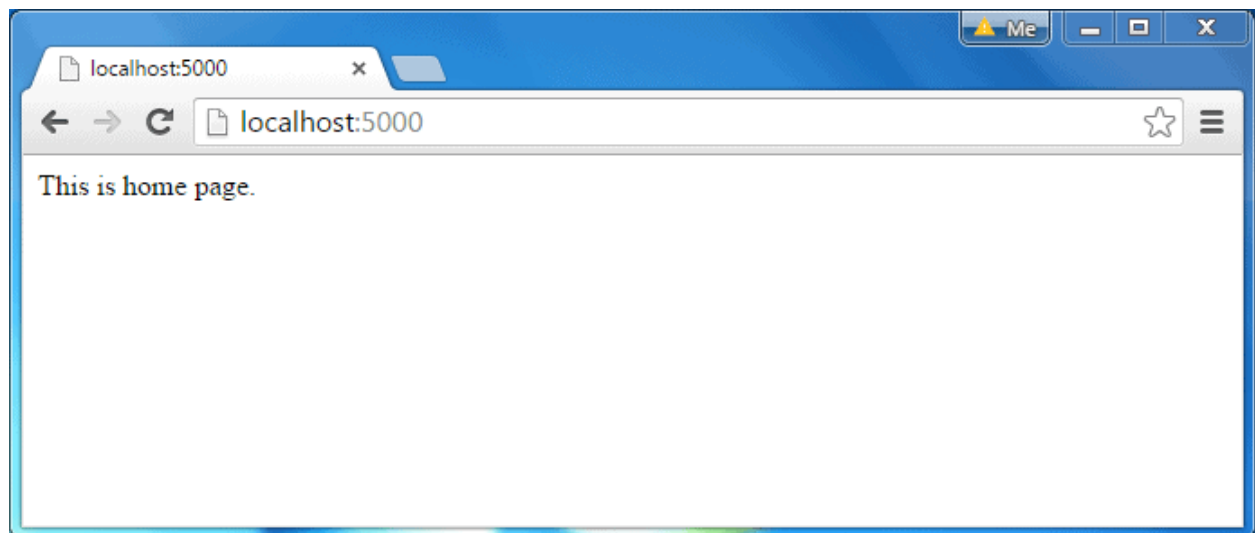
Content-Type: text/plain

Date: Tue, 8 Sep 2015 03:05:08 GMT

Connection: keep-alive

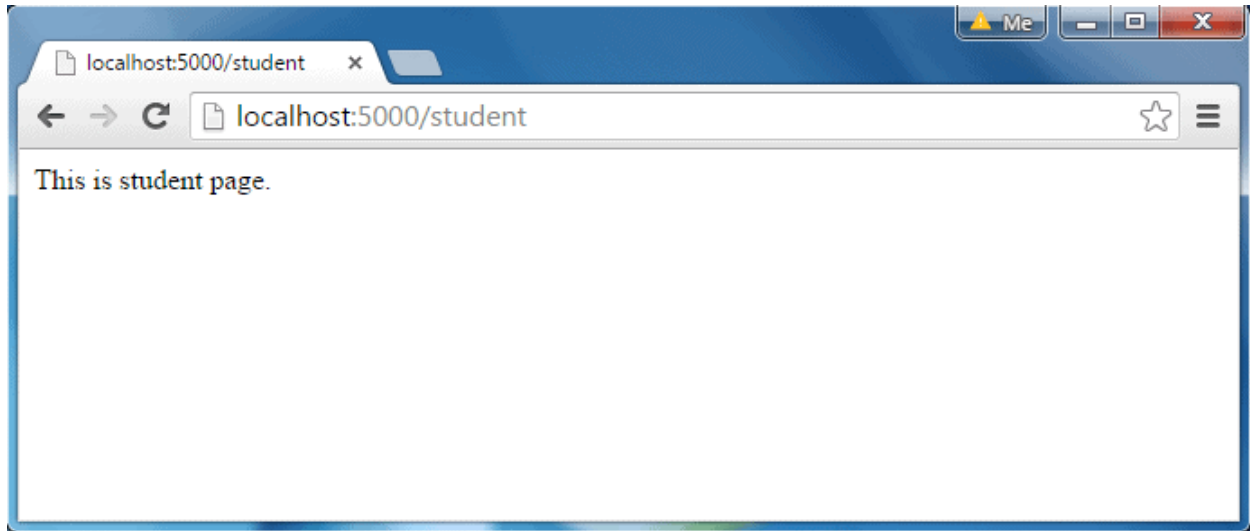
This is home page.

For Windows users, point your browser to *http://localhost:5000* and see the following result.



Node.js Web Server Response

The same way, point your browser to *http://localhost:5000/student* and see the following result.



Node.js Web Server Response

It will display "Invalid Request" for all requests other than the above URLs.

Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

server.js

```
var http = require('http');
var server = http.createServer(function (req, res) {
  if (req.url == '/data') { //check the URL of the current request
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.write(JSON.stringify({ message: "Hello World" }));
    res.end();
  }
});
```

```
server.listen(5000);  
console.log('Node.js web server at port 5000 is running..')
```

So, this way you can create a simple web server that serves different responses.

Create Web App using express

Node.js - Express Framework

Express Overview

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

\$ npm install express --save

The above command saves the installation locally in the node_modules directory and creates a directory express inside node_modules. You should install the following important modules along with express –

- body-parser – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- cookie-parser – Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- multer – This is a node.js middleware for handling multipart/form-data.

\$ npm install body-parser --save

\$ npm install cookie-parser --save

\$ npm install multer --save

Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with Hello World! for requests to the homepage. For every other path, it will respond with a 404 Not Found.

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

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

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

\$ node server.js

You will see the following output –

Example app listening at http://0.0.0.0:8081

Open <http://127.0.0.1:8081/> in any browser to see the following result.



Image : Output

Reference: https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

Request & Response

Express application uses a callback function whose parameters are request and response objects.

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

Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print req and res objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to handle more types of HTTP requests.

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

// This responds with "Hello World" on the homepage
app.get('/', function (req, res) {
  console.log("Got a GET request for the homepage");
  res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
  console.log("Got a POST request for the homepage");
  res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
```



```

    console.log("Got a DELETE request for /del_user");
    res.send('Hello DELETE');
  })

  // This responds a GET request for the /list_user page.
  app.get('/list_user', function (req, res) {
    console.log("Got a GET request for /list_user");
    res.send('Page Listing');
  })

  // This responds a GET request for abcd, abxcd, ab123cd, and so on
  app.get('/ab*cd', function(req, res) {
    console.log("Got a GET request for /ab*cd");
    res.send('Page Pattern Match');
  })

  var server = app.listen(8081, function () {
    var host = server.address().address
    var port = server.address().port

    console.log("Example app listening at http://%s:%s", host, port)
  })

```

Save the above code in a file named server.js and run it with the following command.

\$ node server.js

You will see the following output –

Example app listening at http://0.0.0.0:8081

Now you can try different requests at http://127.0.0.1:8081 to see the output generated by server.js. Following are a few screens shots showing different responses for different URLs.

Screen showing again http://127.0.0.1:8081/list_user



Image : Output

Reference: https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

Screen showing again <http://127.0.0.1:8081/abcd>



Image : Output

Reference: https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

Screen showing again <http://127.0.0.1:8081/abcdefg>



Image : Output

Reference: https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

Introduction to Middleware Concepts

ExpressJS – Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. These functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.

Here is a simple example of a middleware function in action –

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

//Simple request time logger
app.use(function(req, res, next){
  console.log("A new request received at " + Date.now());

  //This function call is very important. It tells that more processing is
  //required for the current request and is in the next middleware
  function route handler.
  next();
});

app.listen(3000);
```

The above middleware is called for every request on the server. So after every request, we will get the following message in the console –

A new request received at 1467267512545

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of `app.use()`. For Example,

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

//Middleware function to log request protocol
app.use('/things', function(req, res, next){
  console.log("A request for things received at " + Date.now());
  next();
});

// Route handler that sends the response
app.get('/things', function(req, res){
  res.send('Things');
});

app.listen(3000);
```

Now whenever you request any subroute of `/things`, only then it will log the time.

Order of Middleware Calls

One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.

For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

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

//First middleware before response is sent
app.use(function(req, res, next){
  console.log("Start");
  next();
});

//Route handler
app.get('/', function(req, res, next){
  res.send("Middle");
  next();
});

app.use('/', function(req, res){
  console.log('End');
});

app.listen(3000);
```

When we visit '/' after running this code, we receive the response as Middle and on our console –

Start

End

The following diagram summarizes what we have learnt about middleware –

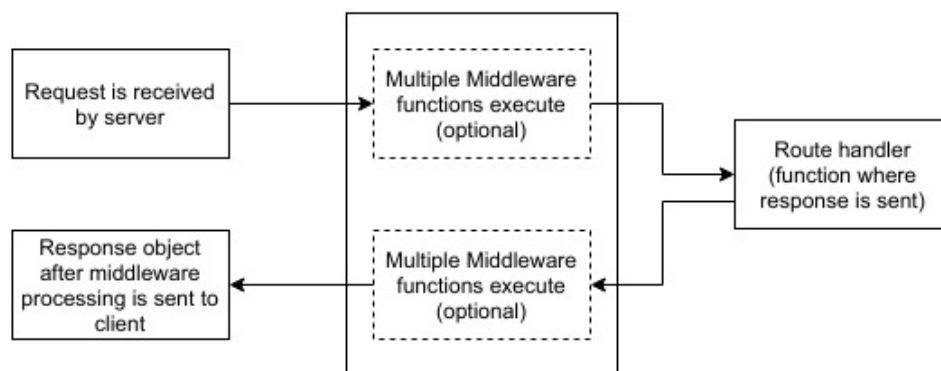


Image : Express Middleware

Reference: https://www.tutorialspoint.com/expressjs/expressjs_middleware.htm

Third Party Middleware

A list of Third party middleware for Express is available here. Following are some of the most commonly used middleware; we will also learn how to use/mount these –

body-parser

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using `npm install --save body-parser` and to mount it, include the following lines in your `index.js` –

```

var bodyParser = require('body-parser');

//To parse URL encoded data
app.use(bodyParser.urlencoded({ extended: false }));

//To parse json data
  
```

```
app.use(bodyParser.json())
```

cookie-parser

It parses Cookie header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using `npm install --save cookie-parser` and to mount it, include the following lines in your index.js –

```
var cookieParser = require('cookie-parser');  
app.use(cookieParser())
```


Express Routing

ExpressJS – Routing

Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.

The following function is used to define routes in an Express application –

app.method(path, handler)

This METHOD can be applied to any one of the HTTP verbs – get, set, put, delete. An alternate method also exists, which executes independent of the request type.

Path is the route at which the request will run.

Handler is a callback function that executes when a matching request type is found on the relevant route. For example,

```
var express = require('express');  
var app = express();  
app.get('/hello', function(req, res){  
  res.send("Hello World!");  
});  
app.listen(3000);
```

If we run our application and go to localhost:3000/hello, the server receives a get request at route "/hello", our Express app executes the callback function attached to this route and sends "Hello World!" as the response.

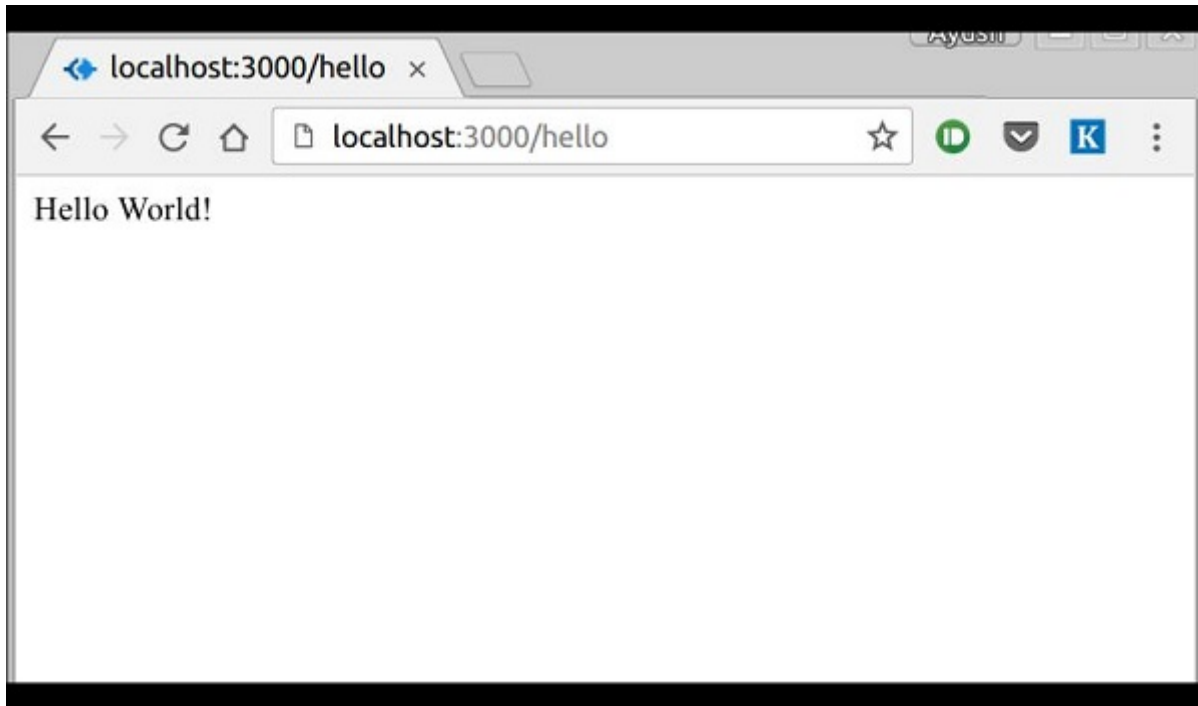


Image : Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_routing.htm

Routers

Defining routes like above is very tedious to maintain. To separate the routes from our main index.js file, we will use Express.Router. Create a new file called things.js and type the following in it.

```
var express = require('express');  
var router = express.Router();  
  
router.get('/', function(req, res){  
  res.send('GET route on things.');
```

```
});
```

```
router.post('/', function(req, res){
  res.send('POST route on things.');
```

```
});
```

//export this router to use in our index.js

```
module.exports = router;
```

Now to use this router in our index.js, type in the following before the app.listen function call.

```
var express = require('Express');
```

```
var app = express();
```

```
var things = require('./things.js');
```

//both index.js and things.js should be in same directory

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

```
app.listen(3000);
```

The app.use function call on route '/things' attaches the things router with this route. Now whatever requests our app gets at the '/things', will be handled by our things.js router. The '/' route in things.js is actually a subroute of '/things'. Visit localhost:3000/things/ and you will see the following output.

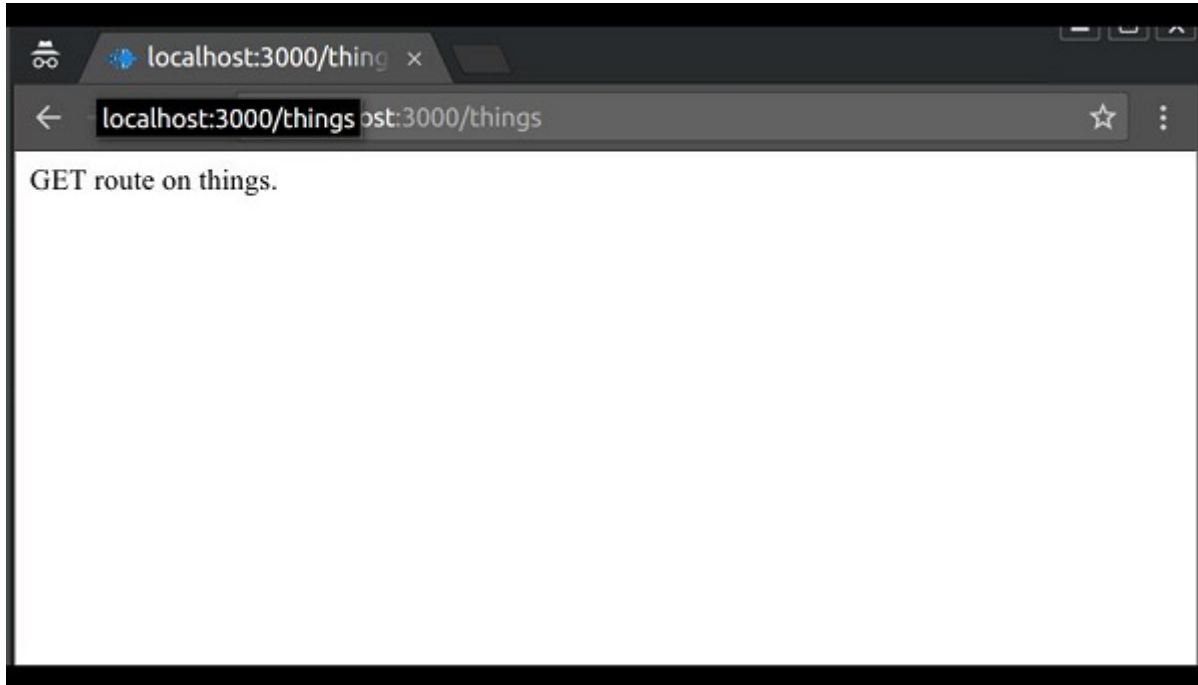


Image : Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_routing.htm

Routers are very helpful in separating concerns and keep relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your index.js file.

Express JS Template Engine

ExpressJS – Templating

Pug is a templating engine for Express. Templating engines are used to remove the cluttering of our server code with HTML, concatenating strings wildly to existing HTML templates. Pug is a very powerful templating engine which has a variety of features including filters, includes, inheritance, interpolation, etc. There is a lot of ground to cover on this.

To use Pug with Express, we need to install it,

npm install --save pug

Now that Pug is installed, set it as the templating engine for your app. You don't need to 'require' it. Add the following code to your index.js file.

```
app.set('view engine', 'pug');
app.set('views', './views');
```

Now create a new directory called views. Inside that create a file called first_view.pug, and enter the following data in it.

```
doctype html
html
  head
    title = "Hello Pug"
  body
    p.greetings#people Hello World!
```

To run this page, add the following route to your app –

```
app.get('/first_template', function(req, res){
  res.render('first_view');
```

```
});
```

You will get the output as – Hello World! Pug converts this very simple looking markup to html. We don't need to keep track of closing our tags, no need to use class and id keywords, rather use '.' and '#' to define them. The above code first gets converted to –

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Pug</title>
  </head>

  <body>
    <p class = "greetings" id = "people">Hello World!</p>
  </body>
</html>
```

Pug is capable of doing much more than simplifying HTML markup.

Important Features of Pug

Let us now explore a few important features of Pug.

Simple Tags

Tags are nested according to their indentation. Like in the above example, <title> was indented within the <head> tag, so it was inside it. But the <body> tag was on the same indentation, so it was a sibling of the <head> tag.

We don't need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us.

To put text inside of a tag, we have 3 methods –

- **Space seperated**

h1 Welcome to Pug

- **Piped text**

div

| To insert multiline text,

| You can use the pipe operator.

- **Block of text**

div.

But that gets tedious if you have a lot of text.

You can use "." at the end of tag to denote block of text.

To put tags inside this block, simply enter tag in a new line and indent it accordingly.

Comments

Pug uses the same syntax as JavaScript(//) for creating comments. These comments are converted to the html comments(<!--comment-->). For example,

//This is a Pug comment

This comment gets converted to the following.

<!--This is a Pug comment-->

Attributes

To define attributes, we use a comma separated list of attributes, in parenthesis. Class and ID attributes have special representations. The following line of code covers defining attributes, classes and id for a given html tag.

div.container.column.main#division(width = "100", height = "100")

This line of code, gets converted to the following. –

<div class = "container column main" id = "division" width = "100" height = "100"></div>

Passing Values to Templates

When we render a Pug template, we can actually pass it a value from our route handler, which we can then use in our template. Create a new route handler with the following.

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

app.get('/dynamic_view', function(req, res){
  res.render('dynamic', {
    name: "Wikipedia",
    url:"https://www.wikipedia.org"
  });
});

app.listen(3000);
```


And create a new view file in views directory, called dynamic.pug, with the following code –

```
html
head
title=name
body
h1=name
a(href = url) URL
```

Open localhost:3000/dynamic_view in your browser;

We can also use these passed variables within text. To insert passed variables in between text of a tag, we use `#{variableName}` syntax. For example, in the above example, if we wanted to put Greetings from TutorialsPoint, then we could have done the following.

```
html
head
title = name
body
h1 Greetings from #{name}
a(href = url) URL
```

This method of using values is called interpolation.

Conditionals

We can use conditional statements and looping constructs as well.

Consider the following –

If a User is logged in, the page should display "Hi, User" and if not, then the "Login/Sign Up" link. To achieve this, we can define a simple template like –

```
html
head
title Simple template
body
if(user)
h1 Hi, #{user.name}
else
a(href = "/sign_up") Sign Up
```

When we render this using our routes, we can pass an object as in the following program –

```
res.render('/dynamic',{
  user: {name: "Ayush", age: "20"}
});
```

You will receive a message – Hi, Ayush. But if we don't pass any object or pass one with no user key, then we will get a signup link.

Include and Components

Pug provides a very intuitive way to create components for a web page. For example, if you see a news website, the header with logo and categories is always fixed. Instead of copying that to every view we create, we can use the include feature. Following example shows how we can use this feature –

Create 3 views with the following code –

HEADER.PUG

div.header.

I'm the header for this website.

CONTENT.PUG

```
html
head
title Simple template
body
include ./header.pug
h3 I'm the main content
include ./footer.pug
```

FOOTER.PUG

```
div.footer.
I'm the footer for this website.
```

Create a route for this as follows –

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

```
var app = express();

app.get('/components', function(req, res){
  res.render('content');
});

app.listen(3000);
```

Go to localhost:3000/components, you will receive the following output –

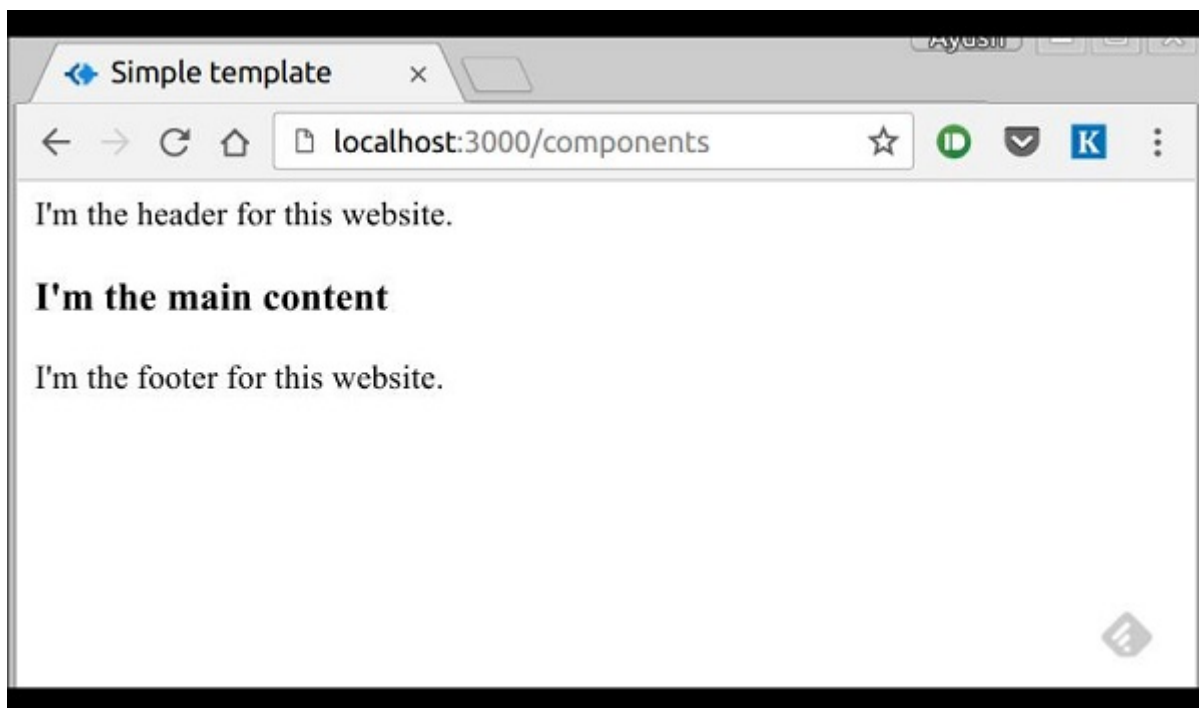


Image : Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_templating.htm

include can also be used to include plaintext, css and JavaScript.

Handling Query string parameter

ExpressJS - URL Building

We can now define routes, but those are static or fixed. To use the dynamic routes, we SHOULD provide different types of routes. Using dynamic routes allows us to pass parameters and process based on them.

Here is an example of a dynamic route –

```
var express = require('express');  
var app = express();  
  
app.get('/:id', function(req, res){  
  res.send('The id you specified is ' + req.params.id);  
});  
app.listen(3000);
```

To test this go to <http://localhost:3000/123>. The following response will be displayed.

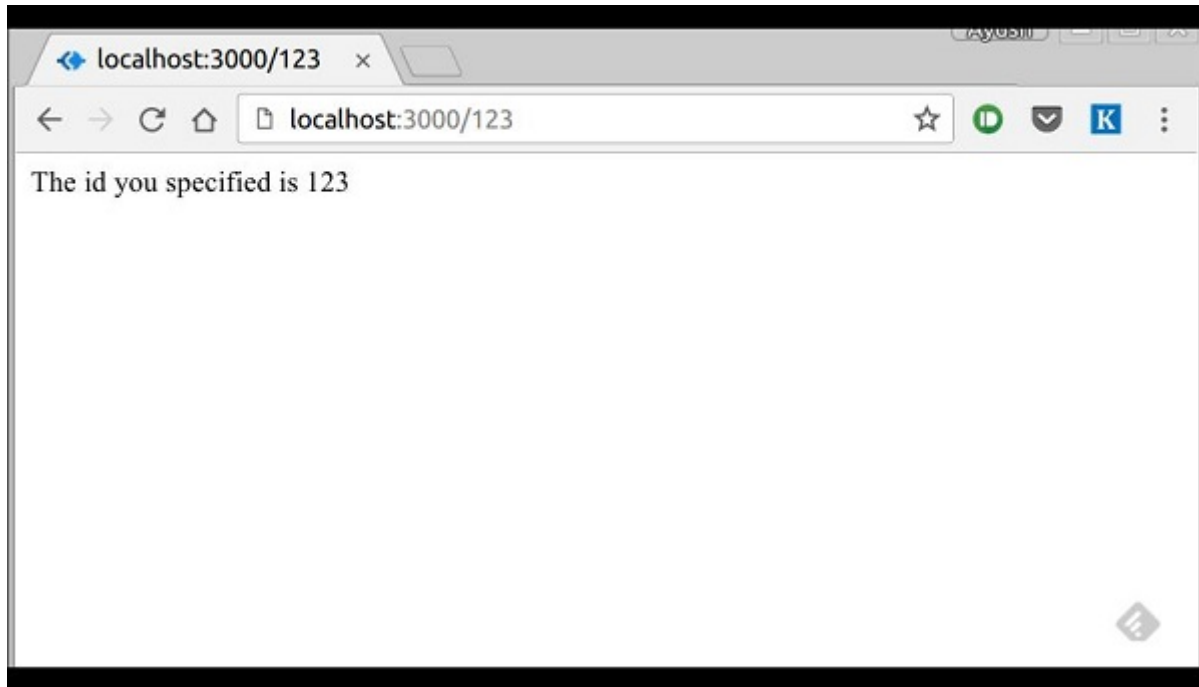


Image : Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_url_building.htm

You can replace '123' in the URL with anything else and the change will reflect in the response.

Cookies parser & body parser

ExpressJS – Cookies

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user's actions.

The following are the numerous uses of the HTTP Cookies –

- Session management
- Personalization(Recommendation systems)
- User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code –

npm install --save cookie-parser

Now to use cookies with Express, we will require the cookie-parser. cookie-parser is a middleware which parses cookies attached to the client request object. To use it, we will require it in our index.js file; this can be used the same way as we use other middleware. Here, we will use the following code.

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

cookie-parser parses Cookie header and populates req.cookies with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like –

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

app.get('/', function(req, res){
  res.cookie('name', 'express').send('cookie set'); //Sets name = express
```

```
});  
  
app.listen(3000);
```

To check if your cookie is set or not, just go to your browser, fire up the console, and enter –

```
console.log(document.cookie);
```

You will get the output like (you may have more cookies set maybe due to extensions in your browser) –

```
"name = express"
```

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

```
console.log('Cookies: ', req.cookies);
```

Next time you send a request to this route, you will receive the following output.

```
Cookies: { name: 'express' }
```

Adding Cookies with Expiration Time

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

```
//Expires after 360000 ms from the time it is set.
```

```
res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

Another way to set expiration time is using 'maxAge' property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

```
//This cookie also expires after 360000 ms from the time it is set.
```

```
res.cookie(name, 'value', {maxAge: 360000});
```


Deleting Existing Cookies

To delete a cookie, use the `clearCookie` function. For example, if you need to clear a cookie named `foo`, use the following code.

```
var express = require('express');  
var app = express();  
  
app.get('/clear_cookie_foo', function(req, res){  
  res.clearCookie('foo');  
  res.send('cookie foo cleared');  
});  
  
app.listen(3000);
```

Body-parser middleware in Node.js

Body-parser is the Node.js body parsing middleware. It is responsible for parsing the incoming request bodies in a middleware before you handle it.

Installation of body-parser module:

You can visit the [link](#) to Install body-parser module. You can install this package by using this command.

npm install body-parser

After installing body-parser you can check your body-parser version in command prompt using the command.

npm --version body-parser

After that, you can just create a folder and add a file, for example, index.js. To run this file you need to run the following command.

node index.js

Filename: SampleForm.ejs

```
<!DOCTYPE html>

<html>

<head>
  <title>Body-Parser Module Demo</title>
</head>

<body>
  <h1>Demo Form</h1>

  <form action="saveData" method="POST">
    <pre>
      Enter your Email   : <input type="text"
                           name="email"> <br>

      <input type="submit" value="Submit Form">
```

```

    </pre>
  </form>
</body>
</html>

```

Filename: index.js

```

const bodyparser = require('body-parser')
const express = require("express")
const path = require('path')
const app = express()

var PORT = process.env.port || 3000

// View Engine Setup
app.set("views", path.join(__dirname))
app.set("view engine", "ejs")

// Body-parser middleware
app.use(bodyparser.urlencoded({extended:false}))
app.use(bodyparser.json())

app.get("/", function(req, res){
  res.render("SampleForm")

```

```
});

app.post('/saveData', (req, res) => {
  console.log("Using Body-parser: ", req.body.email)
})

app.listen(PORT, function(error){
  if (error) throw error
  console.log("Server created Successfully on PORT", PORT)
})
```

Steps to run the program:

1. Make sure you have installed 'view engine' like I have used "ejs" and also installed express and body-parser module using following commands:

npm install express

npm install ejs

npm install body-parser

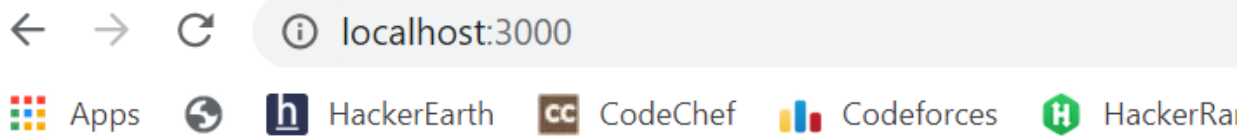
2. Run index.js file using below command:

node index.js

3. Now Open browser and type the below URL and you will see the Demo Form as shown below:

http://localhost:3000/

Now submit the form



Demo Form

Enter your Email :

Image : Output

Reference: <https://www.geeksforgeeks.org/body-parser-middleware-in-node-js/>

4. Now submit the form

Session Handling

ExpressJS – Sessions

HTTP is stateless; in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side. Sessions solve exactly this problem. You assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID.

We will need the Express-session, so install it using the following code.

npm install --save express-session

We will put the session and cookie-parser middleware in place. In this example, we will use the default store for storing sessions, i.e., MemoryStore. Never use this in production environments. The session middleware handles all things for us, i.e., creating the session, setting the session cookie and creating the session object in req object.

Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted). We can add more properties to the session object. In the following example, we will create a view counter for a client.

```
var express = require('express');  
  
var cookieParser = require('cookie-parser');  
  
var session = require('express-session');  
  
  
var app = express();  
  
  
app.use(cookieParser());  
  
app.use(session({secret: "Shh, its a secret!"}));
```

```
app.get('/', function(req, res){
  if(req.session.page_views){
    req.session.page_views++;
    res.send("You visited this page " + req.session.page_views + " times");
  } else {
    req.session.page_views = 1;
    res.send("Welcome to this page for the first time!");
  }
});
app.listen(3000);
```

What the above code does is, when a user visits the site, it creates a new session for the user and assigns them a cookie. Next time the user comes, the cookie is checked and the page_view session variable is updated accordingly.

Now if you run the app and go to localhost:3000, the following output will be displayed.

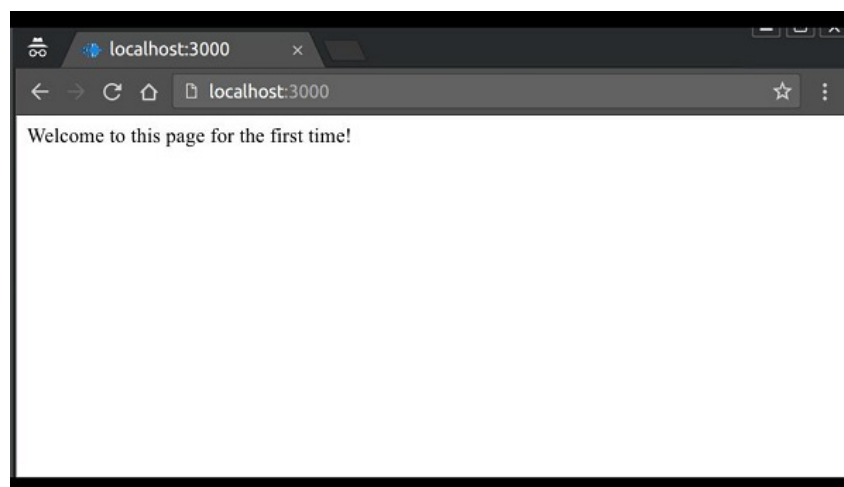


Image: Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

If you revisit the page, the page counter will increase. The page in the following screenshot was refreshed 42 times.

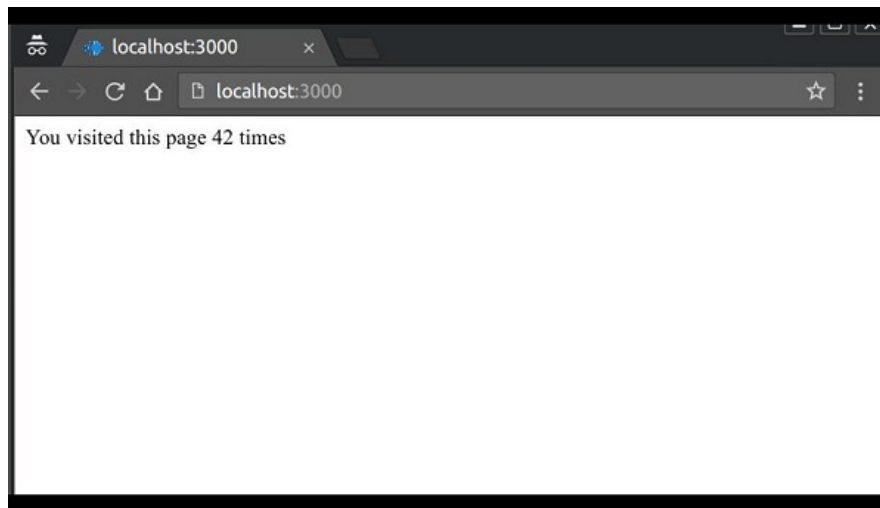


Image: Output

Reference: https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

Express-mailer

Installation

Works with Express 3.x.x

\$ npm install express-mailer

Usage

Express Mailer extends your express application

```
// project/app.js
var app = require('express')(),
    mailer = require('express-mailer');
```



```

mailer.extend(app, {
  from: 'no-reply@example.com',
  host: 'smtp.gmail.com', // hostname
  secureConnection: true, // use SSL
  port: 465, // port for secure SMTP
  transportMethod: 'SMTP', // default is SMTP. Accepts anything that nodemailer accepts
  auth: {
    user: 'gmail.user@gmail.com',
    pass: 'userpass'
  }
});

```

Sending an email

You can send an email by calling `app.mailer.send(template, locals, callback)`. To send an email using the template above you could write:

```

app.get('/', function (req, res, next) {
  app.mailer.send('email', {
    to: 'example@example.com', // REQUIRED. This can be a comma delimited string just
    like a normal email to field.
    subject: 'Test Email', // REQUIRED.
    otherProperty: 'Other Property' // All additional properties are also passed to the
    template as local variables.
  }, function (err) {

```

```

if (err) {
  // handle error

  console.log(err);

  res.send('There was an error sending the email');

  return;
}

res.send('Email Sent');
});
});

```

You can also send an email by calling `mailer` on an applications response object:
`res.mailer.send(template, options, callback)`

Database operation with MySQL

Setup Express js for Node.js MySQL

To set up express, we will use the `express-generator`. You can generate an express js app without any view engine for this Node.js MySQL tutorial with the following command:

`npx express-generator --no-view --git nodejs-mysql`

To quickly check the output execute the following:

`cd nodejs-mysql && npm install && DEBUG=nodejs-mysql:* npm start`

MySQL Installation

`$ npm install mysql`

Example

```
const mysql = require('mysql')

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'dbuser',
  password: 's3kreer7',
  database: 'my_db'
})

connection.connect()

connection.query('SELECT 1 + 1 AS solution', (err, rows, fields) => {
  if (err) throw err

  console.log('The solution is: ', rows[0].solution)
})

connection.end()
```

Learning Outcome

In this section, we will read about:

- Introduction Mongo
- Features of Mongo
- Create database, collection, Document & Data
- Simple Query, Insert select data, filter data
- Capped collection, Update & delete Collectin
- Index and Relationship, Aggregation & grouping, JOIN
- Import & export mongo database, Backup restore
- database operation with Node +Mongodb

Introduction MongoDB

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling. MongoDB is a document-oriented NoSQL database used for high-volume data storage. It contains the data model, which allows to represent hierarchical relationships. It uses JSON-like documents with optional schema instead of using tables and rows in traditional relational databases. Documents containing key-value pairs are the basic units of data in MongoDB. The following table lists the relation between MongoDB and RDBMS terminologies.

| MongoDB (NoSQL Database) | RDBMS (SQL Server, Oracle, etc.) |
|--------------------------|----------------------------------|
| Database | Database |
| Collection | Table |
| Document | Row (Record) |
| Field | Column |

Features of MongoDB

- Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
- The document structure is more in line with how developers construct their classes and objects in their respective programming languages.
- The rows (or documents as called in MongoDB) don't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
- The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
- Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

Create database, collection, Document & Data

• Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

If there is no existing database, the following command is used to create a new database.

use DATABASE_NAME

To use a database with name <mydb>, then use DATABASE statement would be as follows –

```
>use mydb
switched to db mydb
```

To check your currently selected database, use the command **db**

```
>db
mydb
```

To check your databases list, use the command **show dbs**.

```
>show dbs
local 0.78125GB
test 0.23012GB
```

- **Collection**

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose. Basic syntax of createCollection() command is as follows –

```
db.createCollection(name, options)
```

In the command, name is name of collection to be created. Options is a document and is used to specify configuration of collection.

| Parameter | Type | Description |
|-----------|----------|---|
| Name | String | Name of the collection to be created |
| Options | Document | (Optional) Specify options about memory size and indexing |

Options parameter is optional. Following is the list of options–

| Field | Type | Description |
|-------------|---------|---|
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also. |
| autoIndexId | Boolean | (Optional) If true, automatically create index on _id field.s Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also. |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Basic syntax of createCollection() method without options is as follows –

```
>use test
```

```
switched to db test
```

```
>db.createCollection("mycollection")
```

```
{ "ok" : 1 }
```

```
>
```

Check the created collection by using the command show collections.

```
>show collections
```

```
mycollection
```

system.indexes

- **Document**

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

In the RDBMS database, a table can have multiple rows and columns. Similarly in MongoDB, a collection can have multiple documents which are equivalent to the rows. Each document has multiple "fields" which are equivalent to the columns. So in simple terms, each MongoDB document is a record and a collection is a table that can store multiple documents.

The following is an example of JSON based document.



Image 1- JSON based document
Reference <https://www.tutorialsteacher.com/Content/images/mongodb/document.png>

In the above example, a document is contained within the curly braces. It contains multiple fields in "field": "value" format. Above, "_id", "firstName", and "lastName" are field names with their respective values after a colon :. Fields are separated by a comma. A single collection can have multiple such documents separated by a comma.

The following table shows the relationship of RDBMS terminology with MongoDB.

| RDBMS | MongoDB |
|----------|------------|
| Database | Database |
| Table | Collection |

| | |
|-------------|---|
| Tuple/Row | Document |
| column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary Key (Default key <code>_id</code> provided by MongoDB itself) |

Simple Query, Insert select data, filter data

- **insertOne()**

Use the `db.<collection>.insertOne()` method to insert a single document in a collection. `db` points to the current database, `<collection>` is existing or a new collection name.

`db.collection.insertOne(document, [writeConcern])`

Where,

- `document`: A document to insert into the collection.
- `writeConcern`: Optional. A document expressing the write concern to override the default write concern.

The following inserts a document into employees collection.

```
db.employees.insertOne({
  firstName: "John",
  lastName: "King",
  email: "john.king@abc.com"
})
```

In the above example, a document is passed to the `insertOne()` method. Notice that `_id` field haven't specified. So, MongoDB inserts a document to a collection with the auto-generated unique `_id` field. It returns an object with a boolean field `acknowledged` that indicates whether the insert operation is successful or not, and `insertedId` field with the newly inserted `_id` value.

Use the `find()` to list all data of a collection, and the `pretty()` method to format resulted data.

`db.employees.find().pretty()`

- **`insertMany()`**

The `db.<collection>.insertMany()` inserts multiple documents into a collection. It cannot insert a single document. The following adds multiple documents using the `insertMany()` method.

```
db.employees.insertMany(
[
  {
    firstName: "John",
    lastName: "King",
    email: "john.king@abc.com"
  },
  {
    firstName: "Sachin",
    lastName: "T",
    email: "sachin.t@abc.com"
  },
  {
    firstName: "James",
    lastName: "Bond",
    email: "jamesb@abc.com"
  },
],
)
```

Capped collection, Update & delete Collection

- **Capped collection**

Capped collections are fixed-size circular collections that follow the insertion order to support high performance for create, read, and delete operations. By circular, it means that when the fixed size allocated to the collection is exhausted, it will start deleting the oldest document in the collection without providing any explicit commands.

Capped collections restrict updates to the documents if the update results in increased document size. Since capped collections store documents in the order of the disk storage,

it ensures that the document size does not increase the size allocated on the disk. Capped collections are best for storing log information, cache data, or any other high volume data.

To create a capped collection, use the normal `createCollection` command but with **capped** option as **true** and specifying the maximum size of collection in bytes.

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000})
```

In addition to collection size, limit the number of documents in the collection using the **max** parameter

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000,max:1000})
```

To check whether a collection is capped or not, use the following `isCapped` command –

```
>db.cappedLogCollection.isCapped()
```

If there is an existing collection which need to convert into capped use the following code

```
>db.runCommand({"convertToCapped":"posts",size:10000})
```

- **Update & delete Collection**

Use the `db.<collection>.updateOne()` method to update a single document in a collection that matches with the specified filter criteria. It updates the first matching document even if multiple documents match with the criteria.

```
db.collection.updateOne(filter, document, options)
```

Parameters:

- **filter:** The selection criteria for the update, same as `find()` method.
- **document:** A document or pipeline that contains modifications to apply.
- **options:** Optional. May contains options for update behavior. It includes `upsert`, `writeConcern`, `collation`, etc.

In the above syntax, `db` points to the current database, `<collection>` points is an existing collection name. The following updates a single field in a single document in `employees` collection.

```
db.employees.updateOne({_id:1}, { $set: {firstName:'Morgan'}})
```

In the above syntax, the first parameter is the filter criteria specified as a document, `{_id:1}` indicates that find a document whose `_id` is 1. The second parameter is used to specify fields and values to be modified on the matching document in the `{<update-operator>: { field: value, field:value,... } }` format. Use the update operator to specify what action to perform. To set the value of a field, so use `$set` operator to specify fields and updated values in `{field:updated-value}` format. `{ $set: {firstName:'Morgan'}}` modifies the `firstName` to "Morgan" to the first document that matches with the specified criteria `{_id:1}`.

The following syntax updates multiple fields: email and lastName in employees collection.

```
db.employees.updateOne({_id:2}, { $set: {lastName:"Tendulkar", email:"sachin.tendulkar@abc.com"}})
```

- **Delete Collection**

Use the `db.<collection>.deleteOne()` method to delete the first documents that match with the specified filter criteria in a collection.

```
db.collection.deleteOne(filter, options)
```

Parameters:

- `filter`: The selection criteria for the update.
- `options`: Optional. May contains options for update behavior. It includes `writeConcern`, `collation`, and `hint` parameters.

In the above syntax, `db` points to the current database, `<collection>` points is an existing collection name.

The following deletes a single document from the employees collection.

```
db.employees.deleteOne({ salary:7000 })
```

The following deletes all documents from the employees collection that match with the specified criteria.

```
db.employees.deleteMany({ salary:7000 })
```

Index and Relationship, Aggregation & grouping, JOIN

- **Index**

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

```
db.COLLECTION_NAME.createIndex({KEY:1})
```

The key determines the field on the basis of which an index is created and 1 (or - 1) determines the order in which these indexes will be arranged(ascending or descending).

Example –

```
db.mycol.createIndex({"age":1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

The createIndex() method also has a number of optional parameters.

- Background (Boolean)
- Unique (Boolean)
- name (string)

- sparse (Boolean)
- Drop an Index

In order to drop an index, MongoDB provides the `dropIndex()` method.

`db.NAME_OF_COLLECTION.dropIndex({KEY:1})`

The `dropIndex()` methods can only delete one index at a time. In order to delete (or drop)

multiple indexes from the collection, MongoDB provides the `dropIndexes()` method

that takes multiple indexes as its parameters.

`db.NAME_OF_COLLECTION.dropIndexes({KEY1:1, KEY2, 1})`

- **Relationship**

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Following is the sample document structure of **user** document –

```
{
  "_id": ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

Following is the sample document structure of **address** document –

```
{
  "_id": ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

In the **embedded approach**, we will embed the address document inside the user document.

```
> db.users.insert({
  {
    "_id": ObjectId("52ffc33cd85242f436000001"),
    "contact": "987654321",
    "dob": "01-01-1991",
    "name": "Tom Benzamin",
    "address": [
      {
        "building": "22 A, Indiana Apt",
        "pincode": 123456,
        "city": "Los Angeles",
        "state": "California"
      },
      {
        "building": "170 A, Acropolis Apt",
        "pincode": 456789,
        "city": "Chicago",
        "state": "Illinois"
      }
    ]
  }
})
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as

```
>db.users.findOne({"name":"Tom Benzamin"},"address":1})
```

In the above query, **db** and **users** are the database and collection respectively. The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained

separately but the user document will contain a field that will reference the address document's id field.

```
{
  "_id": ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

As shown above, the user document contains the array field `address_ids` which contains `ObjectIds` of corresponding addresses. Using these `ObjectIds`, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the `address_ids` fields from user document and second to fetch these addresses from address collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

- **Aggregation & grouping, JOIN**

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL `count(*)` and `with group by` is an equivalent of MongoDB aggregation.

Basic syntax of **aggregate()** method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

Example

```
{
```



```

    _id: ObjectId(7df78ad8902c)
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by_user: 'Edunet',
    url: 'https://www.edunetfoundation.org',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
  },
  {
    _id: ObjectId(7df78ad8902d)
    title: 'NoSQL Overview',
    description: 'No sql database is very fast',
    by_user: 'Edunet',
    url: 'https://www.edunetfoundation.org',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 10
  },
  {
    _id: ObjectId(7df78ad8902e)
    title: 'Neo4j Overview',
    description: 'Neo4j is no sql database',
    by_user: 'abcj',
    url: 'http://www.abc.com',
    tags: ['neo4j', 'database', 'NoSQL'],
    likes: 750
  },

```

Now from the above collection, To display a list stating how many tutorials are written by each user, then use the following **aggregate()** method –

```

> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}]
{ "_id" : " Edunet ", "num_tutorial" : 2 }
{ "_id" : "abc", "num_tutorial" : 1 }

```

In the above example, we have grouped documents by field **by_user** and on each occurrence of by user previous value of sum is incremented. Following is a list of available aggregation expressions.

| Expression | Description | Example |
|------------|-------------|---------|
|------------|-------------|---------|

| | | |
|------------|--|---|
| \$sum | Sums up the defined value from all documents in the collection. | db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}]) |
| \$avg | Calculates the average of all given values from all documents in the collection. | db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}]) |
| \$min | Gets the minimum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}]) |
| \$max | Gets the maximum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}]) |
| \$push | Inserts the value to an array in the resulting document. | db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push : "\$url"}}}]) |
| \$addToSet | Inserts the value to an array in the resulting document but does not create duplicates. | db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}]) |
| \$first | Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage. | db.mycol.aggregate([{\$group : {_id : "\$by_user", first_url : {\$first : "\$url"}}}]) |
| \$last | Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage. | db.mycol.aggregate([{\$group : {_id : "\$by_user", last_url : {\$last : "\$url"}}}]) |

JOIN

MongoDB is not a relational database, but a left outer join can be performed by using the **\$lookup** stage. The **\$lookup** stage lets you specify which collection you want to join with the current collection, and which fields that should match.

Consider you have a "orders" collection and a "products" collection:

orders

```
[
  { _id: 1, product_id: 154, status: 1 }
]
```

products

```
[
  { _id: 154, name: 'Chocolate Heaven' },
  { _id: 155, name: 'Tasty Lemons' },
  { _id: 156, name: 'Vanilla Dreams' }
]
```

Join the matching "products" document(s) to the "orders" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection('orders').aggregate([
    { $lookup:
      {
        from: 'products',
        localField: 'product_id',
        foreignField: '_id',
        as: 'orderdetails'
      }
    }
  ]).toArray(function(err, res) {
    if (err) throw err;
    console.log(JSON.stringify(res));
    db.close();
  });
});
```

```
});  
});
```

Import & export mongo database, Backup restore

MongoDB provides you with 2 ways:

- mongoexport/mongoimport
- mongodump/mongorestore

mongoexport: Being used to export data from a *Collection* to a file (json, csv,..)

mongoimport: Being used to import data to a *Collection* from a file (json, csv,..)

mongodump: Being used to export all datas of a database to files (Put in a folder), including some files (bson, json)

mongorestore: Being used to import all datas to a database from dump directory (Product of mongodump)

To create backup of database in MongoDB, use **mongodump** command. This command will dump the entire data of your server into the dump directory. There are many options available by which you can limit the amount of data or create backup of your remote server. The basic syntax of **mongodump** command is as follows –

>mongodump

To restore backup data MongoDB's **mongorestore** command is used. This command restores all of the data from the backup directory.

>mongorestore

Database operation with Node +Mongodb

Node.js can be used in database applications. To download and install the official MongoDB driver, open the Command Terminal and execute the following:

C:\Users\Your Name>npm install mongodb

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database MongoDB will create the database if it does not exist, and make a connection to it.

- **Create a database called "mydb":**

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

Save the code above in a file called "demo_create_mongo_db.js" and run the file:

C:\Users\Your Name>node demo_create_mongo_db.js

- **Create a collection called "customers":**

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
```

```

    console.log("Collection created!");
    db.close();
  });
});

```

- **Insert a document in the "customers" collection:**

```

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});

```

- **Delete the document with the address "Mountain 21":**

```

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: 'Mountain 21' };

```

```

    dbo.collection("customers").deleteOne(myquery, function(err, obj) {
        if (err) throw err;
        console.log("1 document deleted");
        db.close();
    });
});

```

- **Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":**

```

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";
MongoClient.connect(url, function(err, db) {
    if (err) throw err;
    var dbo = db.db("mydb");
    var myquery = { address: "Valley 345" };
    var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
    dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
        if (err) throw err;
        console.log("1 document updated");
        db.close();
    });
});

```

References

1. <https://www.mongodb.com/mean-stack>
2. <https://www.guru99.com/angularjs-introduction.html>
3. https://www.w3schools.com/angular/angular_events.asp
4. <https://www.tutorialsteacher.com/typescript/for-loop>
5. <https://docs.angularjs.org/guide/component>
6. <https://angular.io/guide/event-binding>
7. <https://vegibit.com/how-to-bind-events-in-angularjs/>
8. <https://www.javatpoint.com/angularjs-data-binding>
9. https://www.tutorialspoint.com/typescript/typescript_interfaces.htm
10. <https://www.techiediaries.com/object-oriented-programming-concepts/>
11. <https://www.w3schools.blog/inheritance-typescript>
12. <https://docs.angularjs.org/guide/databinding>
13. <https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html>
14. <https://www.ngdevelop.tech/angular/pipes/>
15. <https://angular.io/api/core/Pipe#description>
16. <https://www.code-sample.com/2018/05/angular-5-6-7-chaining-pipes.html>
17. <https://www.c-sharpcorner.com/UploadFile/ff2f08/routing-in-angularjs/>
18. <https://data-flair.training/blogs/angularjs-services/>
19. <https://www.c-sharpcorner.com/UploadFile/f0b2ed/what-is-http-service-in-angularjs815/>
20. https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm
21. <https://www.geeksforgeeks.org/blocking-and-non-blocking-in-node-js/>
22. <https://www.geeksforgeeks.org/introduction-to-node-js/>
23. <https://www.geeksforgeeks.org/blocking-and-non-blocking-in-node-js/>
24. <https://www.codingninjas.com/codestudio/library/blocking-and-non-blocking-in-node-js>
25. https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm
26. https://www.tutorialspoint.com/expressjs/expressjs_middlewares.htm
27. https://www.tutorialspoint.com/expressjs/expressjs_routing.htm
28. https://www.tutorialspoint.com/expressjs/expressjs_templating.htm
29. https://www.tutorialspoint.com/expressjs/expressjs_url_building.htm
30. https://www.tutorialspoint.com/expressjs/expressjs_cookies.htm
31. <https://www.geeksforgeeks.org/body-parser-middleware-in-node-js/>

- 32. https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm
- 33. <https://www.npmjs.com/package/express-mailer>
- 34. <https://expressjs.com/en/guide/database-integration.html#mysql>
- 35. <https://geshan.com.np/blog/2020/11/nodejs-mysql-tutorial/>
- 36. <https://www.tutorialspoint.com/mongodb/index.htm>
- 37. <https://www.guru99.com/mongodb-tutorials.html>
- 38. <https://www.tutorialsteacher.com/mongodb/what-is-mongodb>
- 39. https://www.w3schools.com/nodejs/nodejs_mongodb.asp
- 40. <https://www.javatpoint.com/mongodb-tutorial>
- 41. <https://www.geeksforgeeks.org/mongodb-an-introduction/>