# Node JS

- Many web application technologies like JSP, Spring MVC, etc follow Multithreaded Request Response architecture to handle multiple concurrent requests.
- Node JS adopts Single threaded event loop model.

Node JS Architecture - SIngle threaded event loop.

- Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.
- The main heart of Node JS processing model is “Event Loop”.

Single threaded event loop model processing steps:

- Clients Send request to Web Server.
- Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
- Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
- Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them.
- Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
- Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.
- If yes, then pick up one Client Request from Event Queue
    - Starts process that Client Request
    - If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
    - If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
        - Checks Threads availability from Internal Thread Pool
        - Picks up one Thread and assign this Client Request to that thread.
        - That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop.
    - Event Loop in turn, sends that Response to the respective Client.

Request/Response model processing steps:

- Client sends request to web server
- Web server internally maintains a limited thread pool to provide services to the client requests.
- Web server is in infinite loop and waiting for client incoming requests.
- Web server receives those requests.
    - Web Server pickup one Client Request.
    - Pickup one Thread from Thread pool
    - Assign this Thread to Client Request
    - This Thread will take care of reading Client request, processing Client request, performing any Blocking IO Operations (if required) and preparing Response.
    - This Thread sends prepared response back to the Web Server
- Web Server in-turn sends this response to the respective Client.

Drawbacks:

- If more clients requests require Blocking IO Operations, then almost all threads are busy in preparing their responses. Then remaining clients Requests should wait for longer time.

What is event driven programming?

- Event-driven programming is building our application based on and respond to events. When an event occurs, like click or keypress, we are running a callback function which is registered to the element for that event.
- Event driven programming follows mainly a publish-subscribe pattern.

What is callback?

- A callback is a function that is to be executed after another function has finished executing.
- In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called higher-order functions. Any function that is passed as an argument is called a callback function.

Event Loop pseudo code:

```
public class EventLoop {
        while(true){
                if(Event Queue receives a JS Function Call){
                        ClientRequest req =
EventQueue.getClientRequest();
        If(req requires BlokingIO or takes more computation
time)
            Assign request to Thread T1
        Else
           Process and Prepare response
```

<table>
<tr><td></td><td>}<br>}<br>}</td></tr>
<tr><td>What is Node JS?<br>● server side scripting based on Google's V8 JavaScript engine. It is used to build scalable programs especially web applications that are computationally simple but are frequently accessed.<br>● Can be used in developing I/O intensive web applications like video streaming sites. Real-time web application, n/w applications, general purpose and distributed systems.</td><td>Why use Node JS?<br>● Node.js makes building scalable network programs easy. Some of its advantages include:<br>○ Generally fast<br>○ Almost never blocks<br>○ Offers a unified programming language and data type.<br>○ Everything is asynchronous<br>○ Yields great concurrency.</td></tr>
<tr><td>Features of Node JS?<br>● Node.js is a single-threaded but highly scalable system that utilizes JavaScript as its scripting language. It uses asynchronous, event-driven I/O instead of separate processes or threads. It is able to achieve high output via single-threaded event loop and non-blocking I/O.</td><td>Difference between setImmediate and setTimeout and process.nextTick()?<br>● In Node JS version 0.10 or higher, setImmediate(fn), will be used in place of setTimeout(fn, 0)<br>● setImmediate() is designed to execute a script once the current poll (event loop) phase completes.<br>● setTimeout() schedules a script to be run after a minimum threshold in ms has elapsed.<br>● process.nextTick() technically not part of the event loop. Any time you call process.nextTick() in a given phase, all callbacks passed to process.nextTick() will be resolved before the event loop continues.</td></tr>
<tr><td rowspan="2">What is package.json?<br>● This file holds various metadata information about the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.<br>● Some of the fields are: name, name, description, author and dependencies.<br>● When someone installs our project through npm, all the dependencies listed will be installed as well. Additionally, if someone runs npm install in the root directory of our project, it will install all the dependencies to ./node_modules directory.</td><td>How to update NPM to a new version in Node.js?<br>npm install -g npm@latest<br>npm install -g npm@next</td></tr>
<tr><td>Most popular modules of Node.js?<br>● express<br>● async<br>● browserify<br>● socket.io<br>● bower<br>● gulp<br>● grunt</td></tr>
<tr><td>What is express?<br>● It's a web framework that lets you structure a web application to handle multiple different http requests at a specific url.<br>● Express helps you respond to requests with route support so that you may write responses to specific URLs<br>● Sample structure of Node express app:<br>○ controllers/ — defines your app routes and their logic. You main route might be index.js but you might also have a route called for example '/user' so you might want to make a JS file that just handles that.<br>○ helpers/ — code and functionality to be shared by different parts of the project<br>○ middlewares/ — Express middlewares which</td><td>Async:<br>● is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript.<br><br>Bower:<br>● Bower can manage components that contain HTML, CSS, JavaScript, fonts or even image files. Bower doesn't concatenate or minify code or do anything else - it just installs the right versions of the packages you need and their dependencies.<br><br>Gulp:<br>● Toolkit for automating painful or time-consuming tasks in your development workflow, so you can stop messing around and build something.</td></tr>
</table>

process the incoming requests before handling them down to the routes
    - models/ — represents data, implements business logic and handles storage
    - public/ — contains all static files like images, styles and javascript
    - views/ — provides templates which are rendered and served by your routes
    - tests/ — tests everything which is in the other folders
    - app.js — initializes the app and glues everything together
    - package.json — remembers all packages that your app depends on and their versions

Grunt:

- Grunt is basically a build / task manager written on top of NodeJS. I would call it the NodeJS stack equivalent of ANT for Java. Here are some common scenarios you would want to use grunt under

Why is Node.js single threaded?

- Node.js is single-threaded for async processing. By doing async processing on a single-thread under typical web loads, more performance and scalability can be achieved as opposed to the typical thread-based implementation.

Explain callback in Node.js?

- A callback function is called at the completion of a given task. This allows other code to be run in the meantime and prevents any blocking. Being an asynchronous platform, Node.js heavily relies on callback. All APIs of Node are written to support callbacks.

What is callback hell in Node.js?

- Callback hell is the result of heavily nested callbacks that make the code not only unreadable but also difficult to maintain.

How to prevent/fix callback hell?

- There are 3 ways to prevent/fix callback hell:
    - Handle every single error
    - Keep your code shallow
    - Modularize - split the callbacks into smaller, independent functions that can be called with some parameters then joining them to achieve desired results.
- We can also you promises, **Generators and Async** functions to fix callback hell.

Explain the role of REPL in Node.JS?

- REPL is read eval print loop
- Used to execute ad-hoc JS statements.
- Allows entry to JS directly into a shell prompt and evaluate the results.

Types of API functions in Node JS?

- There are 2 types of functions in Node.js:
    - Blocking functions
    - Non-blocking functions

Blocking functions:

- In a blocking operation, all other code is blocked from executing until an I/O event that is being waited on occurs. Blocking functions are executed synchronously
- E.g. fs.readFileSync()

Non-blocking functions:

- In a non-blocking operation, multiple I/O calls can be performed without the execution of the program being halted. Non blocking functions execute asynchronously.
- E.g. fs.readFile()

What is the first argument typically passed to Node.js callback handler?

- Typically the first argument to any callback handler is an optional error object. The argument is null or undefined if there is no error.

```
function callback(err, results) {
  // usually we'll check for the error before handling results
  if(err) {
    // handle error somehow and return
  }
  // no error, perform standard callback handling
}
```

Differences between Node.JS and AJAX?

- Node.js and Ajax (Asynchronous JavaScript and XML) are the advanced implementation of JavaScript. They all serve completely different purposes.
- Ajax is primarily designed for dynamically updating a particular section of a page's content, without having to

What are the functionalities of NPM in Node.js?

- NPM provides 2 functionalities:
    - Online repository for Node.js packages.
    - Command line utility for installing packages, version management and dependency management of Node.js packages.

Chaining in Node JS?

- Chaining is a mechanism whereby the output of one

update the entire page.
- Node.js is used for developing client-server applications.

Streams in Node JS? Different types of streams?
- Streams are objects that allow reading of data from the source and writing of data to the destination as a continuous process.
- There are 4 types of streams:
  - To facilitate the reading operation
  - To facilitate the writing operation
  - To facilitate both reading and writing operations
  - Is a form of duplex stream that performs computations based on the available input.

What are globals in Node.js?
3 Keywords in Node js constitute as Globals:
- Global – it represents the Global namespace object and acts as a container for all other objects.
- Process – It is one of the global objects but can turn a synchronous function into an async callback. It can be accessed from anywhere in the code and it primarily gives back information about the application or the environment.
- Buffer – it is a class in Node.js to handle binary data.

What is tracing in Node.js?
- Tracing provides a mechanism to collect tracing information generated by V8, Node core and user space code in a log file. Tracing can be enabled by passing the --trace-events-enabled flag when starting a Node.js application.

EventEmitter in Node.js?
- All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object.
- When the EventEmitter object emits an event, all of the functions attached to that specific event are called synchronously.

Security mechanisms available in Node.js?
We can secure our Node.js application in the following ways:

- Authentication — Authentication is one of the primary security stages at which user is identified as permitted to access the application at all. Authentication verifies the user's identity through one or several checks. In Node.js, authentication can be either session-based or token-based. In session-based authentication, the user's credentials are compared to the user account stored on the server and, in the event of successful validation, a session is started for the user. Whenever the session

stream is connected to another stream creating a chain of multiple stream operations.

Exit codes in Node JS?
- Exit codes are specific codes that are used to end a "process" (a global object used to represent a node process).
- Examples of exit codes include:
  - Unused
  - Uncaught Fatal Exception
  - Fatal Error
  - Non-function Internal Exception Handler
  - Internal Exception handler Run-Time Failure
  - Internal JavaScript Evaluation Failure

Difference b/w Angular JS and Node JS?
- Angular.JS is a web application development framework while Node.js is a runtime system.

What is the purpose of module.exports in Node.JS?
- A module encapsulates related code into a single unit of code. This can be interpreted as moving all related functions into a file and then exposes the functions in this file to the outer world. So that We can import them in another file.

How to debug in Node.js?
- Node.js includes a debugging utility called debugger. To enable it start the Node.js with the debug argument followed by the path to the script to debug.
- Inserting the statement debugger; into the source code of a script will enable a breakpoint at that position in the code:

Crypto in Node.js?
- The crypto module in Node.js provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

Use of Timers in Node.js?
- The Timers module in Node.js contains functions that execute code after a set period of time. Timers do not need to be imported via require(), since all the methods are available globally to emulate the browser JavaScript API.
- The Node.js API provides several ways of scheduling code to execute at some point after the present moment. The functions below may seem familiar, since they are available in most browsers, but Node.js actually provides its own implementation of these methods.
- Node.js Timer provides setTimeout(), setImmediate() and setInterval.

Passport in Node.JS?

expires, the user needs to log in again. In token-based authentication, the user's credentials are applied to generate a string called a token which is then associated with the user's requests to the server.

- Error Handling — Usually, the error message contains the explanation of what's actually gone wrong for the user to understand the reason. At the same time, when the error is related to the application code syntax, it can be set to display the entire log content on the frontend. For an experienced hacker, the log content can reveal a lot of sensitive internal information about the application code structure and tools used within the software.

- Request Validation — Another aspect which has to be considered, while building a secure Node.js application, is a validation of requests or, in other words, a check of the incoming data for possible inconsistencies. It may seem that invalid requests do not directly affect the security of a Node.js application, however, they may influence its performance and robustness. Validating the incoming data types and formats and rejecting requests not conforming to the set rules can be an additional measure of securing your Node.js application.

- Node.js Security Tools and Best Practices — We can use tools like helmet (protects our application by setting HTTP headers), csurf (validates tokens in incoming requests and rejects the invalid ones), node rate limiter (controls the rate of repeated requests. This function can protect you from brute force attacks) and cors (enables cross-origin resource sharing).

- Passport.js is a simple, unobtrusive Node.js authentication middleware for Node.js. Passport.js can be dropped into any Express.js-based web application.
- Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as strategies, are packaged as individual modules. Applications can choose which strategies to employ, without creating unnecessary dependencies.
- By default, if authentication fails, Passport will respond with a 401 Unauthorized status, and any additional route handlers will not be invoked. If authentication succeeds, the next handler will be invoked and the req.user property will be set to the authenticated user.

Node.js built in modules:

- assert: provides a set of assertion tests
- buffer: handle binary data
- child_process: run a child process
- Cluster: split a single node process into multiple processes
- crypto: handle openssl crypto functions
- dns: dns lookup and name resolution functions.
- events: handle events
- fs: handle file system
- http: make node js act as http server
- https: make node.js act as https server
- net: create servers and clients
- os: info about the OS
- path: handle file paths
- querystring: URL query strings
- readline: readable streams one line at the time
- stream: streaming data
- string_decoder: decode buffer objects into strings
- timers: execute a func after given number of milliseconds
- url: parse url strings
- util: access utility functions
- v8: info about v8 engine
- vm: compile JS code in Virtual machine
- zlib: compress or decompress files

**AngularJS vs Angular 2 vs Angular 4**

| | AngularJS | Angular 2 | Angular 4 |
|---|---|---|---|
| | • open-source, JavaScript-based, front-end web application framework for dynamic web app development. | • Open-source typescript based front-end web application platform. | • Improved compared to Angular 2 and is backward compatible. |

| | | | |
|---|---|---|---|
| Architecture | • Based on MVC design.<br>• The model is the central comp that expresses the apps behavior & manages its datalogic and rules. | • Controllers and scopes are replaced by components and directives with a template. | • Structural derivatives like ngIf and ngFor have been improved, and you can use if/else design syntax in your templates. |
| Language | • JS | • Typescript superset of ES6 | • Compatible with the most recent versions of TypeScript. |
| Expression Syntax | • To bind an image/property or event with AngularJS, you have to remember the right ng directive | • Focuses on "()" for event binding and "[]" for property binding. | • In addition to expression syntax, improved design syntax for *ngIf and *ngFor |
| Routing | • Uses $routeprovider.when() to configure routing. | • Uses @Routerconfig{(...)} to configure routing. | • Uses RouterModule.-forRoot() and RouterModule.-forChild() to configure routing. |
| Advantages | • Unit testing ready<br>• Great MVC data binding makes app development fast<br>• HTML as a declarative lang makes it very intuitive<br>• It is comprehensive solution for rapid front-end development since it does not need any other framework or plugins. | • Typescripting allows code optimization using the OOPS concept.<br>• Mobile-oriented<br>• Improved dependency injection and modularity<br>• Offers simpler routing | • Fat development process<br>• Ideal for single page web applications with an extended interface.<br>• Fully typescript support helps in building bulky applications<br>• Tests are easy to write |
| Disadvantages: | • Big and complicated due to multiple ways of doing the same thing.<br>• Implementations scale poorly<br>• If JS disabled, nothing but the basic page is visible.<br>• Lagging UI if there are more than 200 watchers. | • More complicated to set up compared to AngularJS<br>• Inefficient only if you need to create simple, small web apps | • Slow when displaying enormous amount of data. |
| | | | |

JavaScript vs TypeScript:

- Typescript is a modern age Javascript development language. It is a statically compiled language to write clear and simple Javascript code. It can be run on Node js or any browser which supports ECMAScript 3 or newer versions.
- Typescript provides optional static typing, classes, and interface. For a large JavaScript project adopting Typescript can bring you more robust software and easily deployable with a regular JavaScript application.

Why TS?

- TypeScript supports JS libraries & API Documentation
- It is a superset of Javascript
- It is optionally typed scripting language
- TypeScript Code can be converted into plain JavaScript Code

- Better code structuring and object-oriented programming techniques
- Allows better development time tool support
- It can extend the language beyond the standard decorators, async/await

## Angular 7

Architecture overview:

- Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript.
- The basic building blocks of an Angular application are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.
    - Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
    - Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.
- Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.
    - The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
    - The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).
- An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

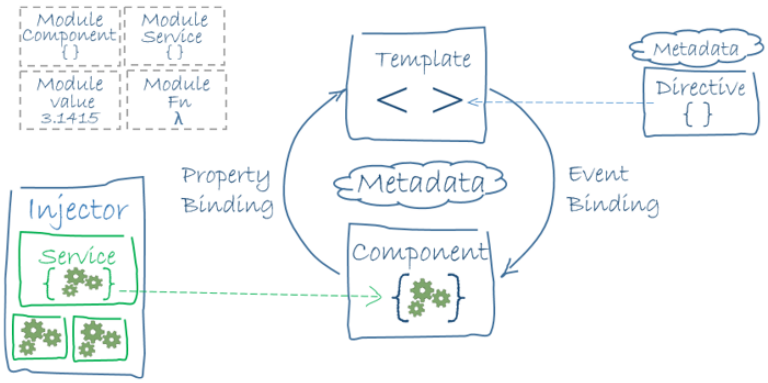
Modules:

- An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.
- Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.
- Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.
- Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

NgModule metadata:

- An NgModule is defined by a class decorated with @NgModule(). The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.
- declarations: The components, directives, and pipes that belong to this NgModule.
- exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
- imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
- providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
- bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
```




Components:

- Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.
- The @Component() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.
- A component controls a patch of screen called a view.
- You define a component's application logic—what it does to support the view—inside a class.

component metadata:

- The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view.
- selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
- templateUrl: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
- providers: An array of providers for services that the component requires. In the example, this tells Angular how to provide the HeroService instance that the component's constructor uses to get the list of heroes to display.

```
 imports:      [ BrowserModule ],
 providers:    [ Logger ],
 declarations: [ AppComponent ],
 exports:      [ AppComponent ],
 bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

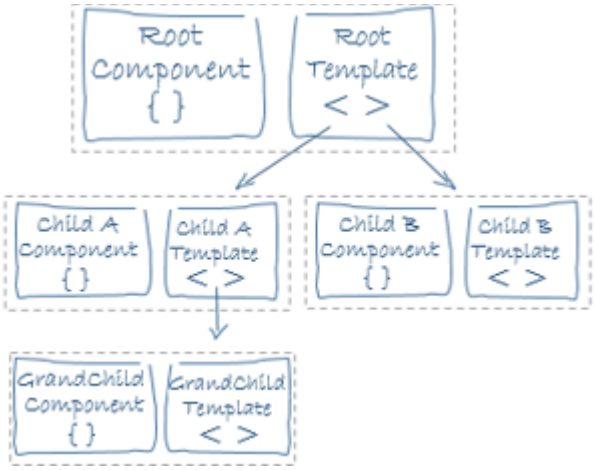
Component example:

```
@Component({
 selector:    'app-hero-list',
 templateUrl: './hero-list.component.html',
 providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

Decorators:

- Decorators are functions that modify JavaScript classes. Angular defines a number of decorators that attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

E.g. @Component, @NgModule




Templates, Directives and Data binding:

- A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the DOM. There are two types of data binding:

Services and Dependency Injection:

- For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the @Injectable() decorator. The decorator provides the metadata that allows your

    - Event binding lets your app respond to user input in the target environment by updating your application data.
    - Property binding lets you interpolate values that are computed from your application data into the HTML.
- Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, are also reflected in your program data.
- Your templates can use pipes to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

Template syntax:

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero"
[hero]="selectedHero"></app-hero-detail>
```

Architecture summary:

- Together a component and template defines an angular view.
    - A decorator on a component class adds the metadata, including a pointer to the associated template.
    - Directives and binding markup in a component's template modify views based on program data and logic.
- The dependency injector provides services to a component, such as the router service that lets you define navigation among views.

Data Binding:

- Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced jQuery programmer can attest.
- Angular supports two-way data binding, a mechanism for coordinating the parts of a template with the parts of a component. Add binding markup to the template

service to be injected into client components as a dependency.

- Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

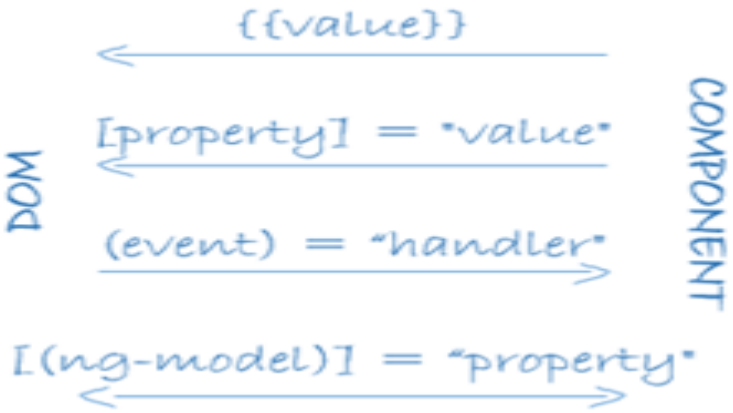
Routing:

- The Angular Router NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:
    - Enter a URL in the address bar and the browser navigates to a corresponding page.
    - Click links on the page and the browser navigates to a new page.
    - Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.
- The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.
- If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.
- The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.
- To define navigation rules, you associate navigation paths with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

Pipes:

- Angular pipes let you declare display-value transformations in your template HTML. A class with the @Pipe decorator defines a function that transforms input values to output values for display in a view.
- Angular defines various pipes, such as the date pipe and currency pipe.
- To specify a value transformation in an HTML template, use the pipe operator (|).
  {{interpolated_value | pipe_name}}

HTML to tell Angular how to connect both sides.

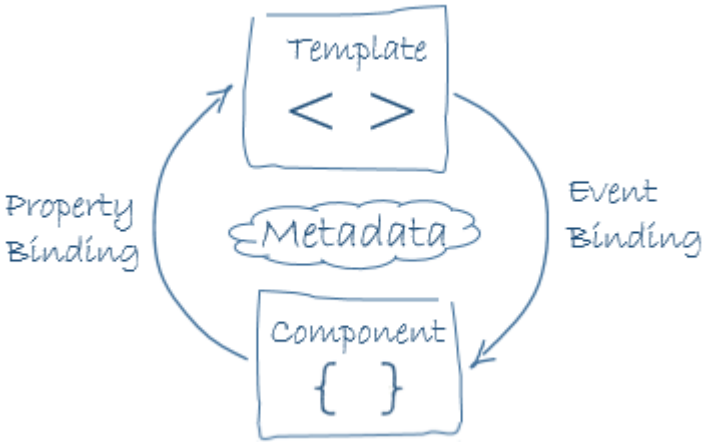


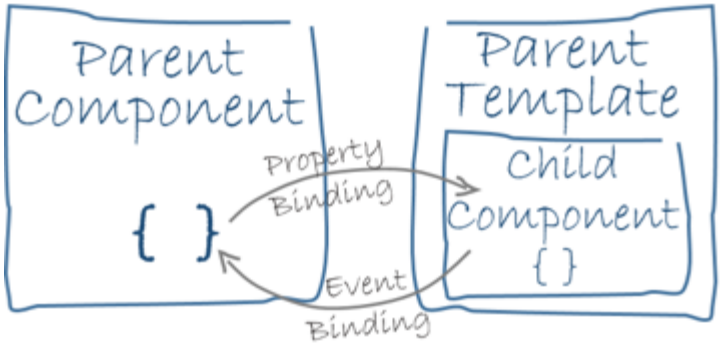

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

- The {{hero.name}} interpolation displays the component's hero.name property value within the <li> element.
- The [hero] property binding passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
- The (click) event binding calls the component's selectHero method when the user clicks a hero's name.

```
<input [(ngModel)]="hero.name">
```

- In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.




```
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

Directives:

- Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives. A directive is a class with a @Directive() decorator.
- A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the @Component() decorator, which extends the @Directive() decorator with template-oriented features.

Structural directives:

- Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

```
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- *ngFor is an iterative; it tells Angular to stamp out one <li> per hero in the heroes list.
- *ngIf is a conditional; it includes the HeroDetail component only if a selected hero exists.

Attribute directives:

- Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.
- The ngModel directive, which implements two-way data binding, is an example of an attribute directive. ngModel modifies the behavior of an existing element (typically <input>) by setting its display value property and responding to change events.

```
<input [(ngModel)]="hero.name">
```

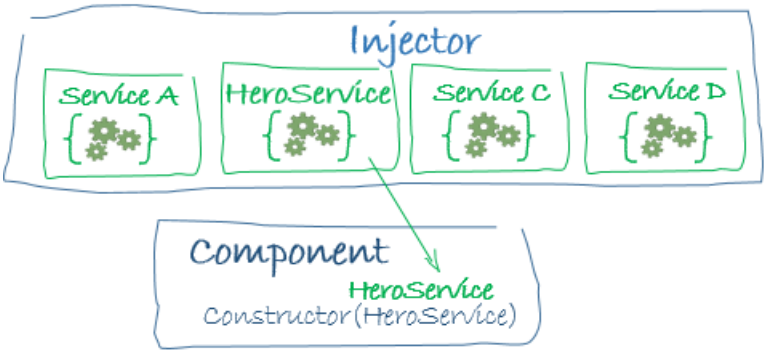
Service Example:

- A service is typically a class with a narrow, well-defined purpose.
- Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.
- Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).
- A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component.
- Angular doesn't enforce these principles. Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}

export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Dependency Injection:

- DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.
- To define a class as a service in Angular, use the @Injectable() decorator to provide the metadata that allows Angular to inject it into a component as a dependency.
- Similarly, use the @Injectable() decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) has a dependency.
    - The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.
    - An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible.
    - A provider is an object that tells an injector how to obtain or create a dependency.
- When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types.




Providing Services:

- You must register at least one provider of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific modules or components. You register providers in the metadata of the service (in the @Injectable() decorator), or in the @NgModule() or @Component()

HttpClient:

- Most front-end applications communicate with backend services over the HTTP protocol. Modern browsers support two different APIs for making HTTP requests: the XMLHttpRequest interface and the fetch() API.
- The HttpClient in @angular/common/http offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by

metadata

- By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @Injectable() decorator.

```
@Injectable({
 providedIn: 'root',
})
```

- When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects it into any class that asks for it. Registering the provider in the @Injectable() metadata also allows Angular to optimize an app by removing the service from the compiled app if it isn't used.
- When you register a provider with a specific NgModule, the same instance of a service is available to all components in that NgModule. To register at this level, use the providers property of the @NgModule() decorator.

```
@NgModule({
  providers: [
  BackendService,
  Logger
 ],
 ...
})
```

- When you register a provider at the component level, you get a new instance of the service with each new instance of that component. At the component level, register a service provider in the providers property of the @Component() metadata.

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

Angular Router Configuration:

```
const appRoutes: Routes = [
 { path: 'crisis-center', component: CrisisListComponent },
 { path: 'hero/:id',      component: HeroDetailComponent },
 {
   path: 'heroes',
   component: HeroListComponent,
   data: { title: 'Heroes List' }
 },
 { path: '',
   redirectTo: '/heroes',
   pathMatch: 'full'
 },
 { path: '**', component: PageNotFoundComponent }
```

browsers. Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

- Before you can use the HttpClient, you need to import the Angular HttpClientModule. Most apps do so in the root AppModule.

```
import { NgModule }        from '@angular/core';
import { BrowserModule }    from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
 imports: [
   BrowserModule,
   // import HttpClientModule after BrowserModule.
   HttpClientModule,
 ],
 declarations: [
   AppComponent,
 ],
 bootstrap: [ AppComponent ]
})
export class AppModule {}
```

- Having imported HttpClientModule into the AppModule, you can inject the HttpClient into an application class as shown in the following ConfigService example.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
 constructor(private http: HttpClient) { }
}
```

Why Write a Service?

- This example is so simple that it is tempting to write the Http.get() inside the component itself and skip the service.
- However, data access rarely stays this simple. You typically post-process the data, add error handling, and maybe some retry logic to cope with intermittent connectivity.
- The component quickly becomes cluttered with data access minutia. The component becomes harder to understand, harder to test, and the data access logic can't be re-used or standardized.
- That's why it is a best practice to separate presentation of data from data access by encapsulating data access in a separate service and delegating to that service in the component, even in simple cases like this one.

```
];

@NgModule({
 imports: [
  RouterModule.forRoot(
   appRoutes,
   { enableTracing: true } // <-- debugging purposes only
  )
  // other imports here
 ],
 ...
})
export class AppModule { }
```

Router Summary:
The application has a configured router. The shell component has a RouterOutlet where it can display views produced by the router. It has RouterLinks that users can click to navigate via the router.
Here are the key Router terms and their meanings:

- Route:
  - Displays the application component for the active URL. Manages navigation from one component to the next.
- RouterModule:
  - A separate NgModule that provides the necessary service providers and directives for navigating through application views.
- Routes:
  - Defines an array of Routes, each mapping a URL path to a component.
- Route:
  - Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
- RouterOutlet:
  - The directive (<router-outlet>) that marks where the router displays a view.
- RouterLink:
  - The directive for binding a clickable HTML element to a route. Clicking an element with a routerLink directive that is bound to a string or a link parameters array triggers a navigation.
- RouterLinkActive:
  - The directive for adding/removing classes from an HTML element when an associated routerLink contained on or inside the element becomes active/inactive.
- ActivatedRoute:
  - A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
- RouterState:
  - The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.
- Link parameters array:
  - An array that the router interprets as a routing instruction. You can bind that array to a RouterLink or pass the array as an argument to the Router.navigate method.
- Routing component:
  - An Angular component with a RouterOutlet that displays views based on router navigations.

Note: Information gathered in this document has been collected from various sources on the Internet.