

Guru Tegh Bahadur Institute of Technology

Rajouri Garden, New Delhi, Delhi 110064

WEB DEVELOPMENT USING MEAN STACK

Practical File

(FSD-320P)



SESSION 2023 – 2024

BATCH 2021 – 2025

SUBMITTED TO :

Ms. Gurdeep Kaur

CSE Department

SUBMITTED BY :

Gurpreet Singh

B.Tech CSE – 1

03713202721

INDEX

S.NO	Name of the Practical	Page NO	Date	Signature
1.	Introduction to MEAN Stack	1 - 3	14/02/2024	
2.	Create a static website using HTM / CSS / Javascript and deploy it on a web server.	4 - 20	21/02/2024	
3.	Install and setup Node.js and Express.js.	21 - 25		
4.	Use Angular CLI to create a new component and implement data binding techniques such as string interpolation, property binding, two – way data binding and event binding.	26 - 37		
5.	Implement component interaction using @input, @output decorators and @view Child decorator to get hold of DOM.	38 - 43		
6.	Use built in directives such as ng For, ng If, ng Switch, ng Class, ng Style to manipulate the DOM elements.	44 - 57		
7.	Use HTTP client module to perform HTTP requests by server, handle responses and errors.	58 - 59		
8.	Implement routing in an Angular app, including redirection, wild card route, relative paths and routing guards.	60 - 69		
9.	Create a basic server using Node.js and Express.js and handle requests and responses.	70 - 72		
10.	Connect to Node.js server to a MongoDB database, and perform CRUD(Create, Read, Update and Delete) operations using Mongoose library.	73 - 80		

DATE: 14/02/2024

PRACTICAL - 01

Aim: INTRODUCTION TO MEAN STACK.

Theory: The MEAN stack is a collection of JavaScript-based technologies used for building dynamic web applications. The acronym "MEAN" stands for MongoDB, Express.js, AngularJS (or Angular), and Node.js. Each component of the MEAN stack serves a specific purpose in the development process, and when combined, they offer a powerful and efficient platform for building modern web applications.

- 1. MongoDB:** MongoDB is a NoSQL database system that stores data in a flexible, JSON-like format called BSON (Binary JSON). It is designed to be highly scalable and flexible, making it suitable for handling large volumes of data and accommodating changes in data structure over time. MongoDB uses a document-oriented model, which means that data is stored in documents that can vary in structure and can be nested within each other. This flexibility allows developers to work with data in a way that closely mirrors the structure of the application itself, making it easier to develop and maintain complex applications.

MongoDB is particularly well-suited for applications that require frequent updates or have evolving data requirements, such as social networking sites, content management systems, and e-commerce platforms. Its scalability and performance make it ideal for handling large volumes of data and supporting high-traffic applications. MongoDB also offers features such as replication and sharding, which allow developers to distribute data across multiple servers and ensure high availability and fault tolerance.

- 2. Express.js:** Express.js is a lightweight and flexible web application framework for Node.js. It provides a minimalist set of features for building web servers and APIs, allowing developers to quickly create robust and scalable web applications. Express.js simplifies common tasks such as routing, middleware management, and request handling, making it easy to build and maintain complex web applications.

Express.js follows the "middleware" pattern, which allows developers to define reusable functions that can be applied to incoming HTTP requests. This makes it easy to add functionality such as authentication, logging, and error handling to an application without repeating code. Express.js also provides a powerful routing system that allows developers to define routes for handling different types of requests and mapping them to specific controller functions.

Express.js is often used in conjunction with other Node.js modules and libraries to build full-stack web applications. Its simplicity and flexibility make it a popular choice for developers who want to build fast and scalable web applications using JavaScript.

- 3. Angular (or AngularJS):** Angular is a popular front-end framework for building dynamic web applications. Originally developed by Google, Angular provides a comprehensive set of features for building interactive user interfaces, including data binding, templating, routing, and dependency injection. Angular uses a component-based architecture, which allows developers to encapsulate different parts of the user interface into reusable components that can be composed together to build complex applications.

Angular provides two-way data binding, which means that changes to the application state are automatically reflected in the user interface, and vice versa. This simplifies the process of managing application state and synchronizing data between the client and server. Angular also supports the development of single-page applications (SPAs), where all of the application logic runs in the browser and communicates with the server via APIs.

Angular has a rich ecosystem of tools and libraries that streamline the development process, including Angular CLI for project scaffolding and code generation, RxJS for reactive programming, and Angular Material for building responsive and accessible user interfaces. Angular is widely used for building modern web applications, including enterprise applications, dashboards, and progressive web apps (PWAs).

- 4. Node.js:** Node.js is a server-side JavaScript runtime environment built on the V8 JavaScript engine. It allows developers to run JavaScript code outside of the browser, making it possible to build full-stack web applications entirely in JavaScript. Node.js provides a non-blocking, event-driven architecture that is well-suited for building scalable and high-performance web servers and APIs.

Node.js has a large ecosystem of modules and libraries available through the npm (Node Package Manager) registry, which makes it easy to add functionality to an application by simply installing and importing modules. Node.js is particularly well-suited for building real-time web applications, such as chat applications, online gaming platforms, and collaboration tools, where responsiveness and scalability are critical.

One of the key advantages of using Node.js is its ability to handle a large number of concurrent connections with minimal resource consumption. This makes it well-suited for building applications that require high throughput and low latency, such as streaming services, IoT (Internet of Things) platforms, and online marketplaces.

The MEAN stack offers a multitude of advantages for web developers, including consistency, efficiency, scalability, and flexibility.

- One of the primary advantages of the MEAN stack is its consistency, as all components of the stack utilize JavaScript. This unified language throughout the development

process reduces complexity and promotes code reuse, enabling developers to build applications more efficiently.

- Additionally, the MEAN stack's lightweight architecture and modular design contribute to its efficiency, allowing developers to rapidly develop and iterate on their applications. This agility enables developers to bring their ideas to life more quickly, delivering value to end-users in a timely manner.
- Scalability is another key advantage of the MEAN stack, with each component designed to scale horizontally to handle large volumes of traffic and data. Whether deploying applications on-premises or in the cloud, the MEAN stack provides the flexibility needed to support applications of varying sizes and complexities.
- Flexibility is a defining characteristic of the MEAN stack, allowing developers to choose from a variety of deployment options to suit their specific needs. Whether deploying applications on traditional servers or leveraging cloud-based infrastructure, the MEAN stack provides the flexibility needed to adapt to evolving requirements and business needs.

The MEAN stack provides a powerful and efficient platform for building modern web applications using JavaScript on both the client and server sides. By combining MongoDB for data storage, Express.js for server-side development, Angular for client-side development, and Node.js for server-side runtime, developers can create highly scalable, responsive, and maintainable web applications that meet the demands of today's dynamic digital landscape. Whether you're building a simple blog or a complex enterprise application, the MEAN stack offers the tools and technologies you need to bring your ideas to life.

DATE: 21/02/2024

PRACTICAL - 02

Aim: Create a static website using HTML / CSS / JavaScript and deploy it on a web server.

Theory: A responsive blog page is a web page that adjusts its layout and design dynamically based on the size and characteristics of the device or screen on which it is being viewed. The goal of responsive design is to ensure that the blog page remains visually appealing and functional across a wide range of devices, including desktop computers, laptops, tablets, and smartphones.

Html code.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Trend Blogger</title>
  <!-- Box-icon -->
  <link href='https://unpkg.com/boxicons@2.1.4/css/boxicons.min.css' rel='stylesheet'>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <header>
    <div class="nav container">
      <a href="#" class="logo"><span>Blog Page</span></a>
      <a href="#" class="login">Sign Up</a>
    </div>
  </header>
  <section class="home" id="home">
    <div class="home-text container">
      <h2 class="home-title">Blogger</h2>
      <span class="home-subtitle">Your source of great content</span>
    </div>
  </section>
  <section class="about container" id="about">
    <div class="contentBx">
      <h2 class="titleText">Blogs in one Place.</h2>
      <p class="title-text">
        Blogs are online platforms where individuals or organizations share content, typically in a
        chronological format with the most recent posts displayed first.
      <br>
      <br>Blogs often incorporate text, images, videos, and hyperlinks to provide engaging and
      informative content. Readers can interact with blog posts through comments, likes, and shares,
    </p>
  </div>
</section>
</body>
</html>
```

Gurpreet Singh

fostering a sense of community and dialogue. Whether for entertainment, education, or promotion, blogs play a significant role in disseminating information and connecting people with shared interests across the internet.

```

    </p>
    <a href="#" class="btn2">Read more</a>
  </div>
  <div class="imgBx">
    
  </div>
</section>
<div class="post-filter container">
  <span class="filter-item active-filter" data-filter="all">All</span>
  <span class="filter-item" data-filter="tech">Tech</span>
  <span class="filter-item" data-filter="food">Food</span>
  <span class="filter-item" data-filter="news">News</span>
</div>
<div class="post container">
  <!-- Post 1 -->
  <div class="post-box tech">
    
    <h2 class="category">Tech</h2>
    <a href="#" class="post-title">Artificial Intelligence Microchip</a>
    <span class="post-date">12 Feb 2023</span>
    <p class="post-description">An AI chip focuses on powering AI functions. AI workloads require
    massive amounts of processing power that general-purpose chips, like CPUs, typically can't deliver at the
    requisite scale. </p>
    <div class="profile">
      
      <span class="profile-name">Gaurav Choudhary</span>
    </div>
  </div>
  <!-- Post 2 -->
  <div class="post-box food">
    
    <h2 class="category">Tech</h2>
    <a href="#" class="post-title">Benefits of having tech watch for competitive intelligence </a>
    <span class="post-date">13 Jan 2024</span>
    <p class="post-description">The benefits that implementing a technology watching and
    competitive intelligence system provide to a company can be summarized in an improvement of the
    business overall position, adding value to the products and services and reducing the risk towards an
    eventual business failure.</p>
    <div class="profile">
      
      <span class="profile-name">Lucy Warehem</span>
    </div>
  </div>
  <!-- Post 3 -->
  <div class="post-box food">

```

```


<h2 class="category">Food</h2>
<a href="#" class="post-title"> Tropical Fruit Forum - International Tropical Fruit Growers</a>
<span class="post-date">29 Dec 2023</span>
<p class="post-description">Tropical Fruit Forum offers access to not only a wide variety of fruit
trees and seeds but also a very broad knowledge base of every kind of exotic fruit plant you can
imagine.</p>
  <div class="profile">
    
    <span class="profile-name">Jason Roy</span>
  </div>
</div>
<!-- Post 4 -->
<div class="post-box news">
  
  <h2 class="category">Tech</h2>
  <a href="#" class="post-title">Pace of Technology Innovation</a>
  <span class="post-date">1 Mar 2024</span>
  <p class="post-description">Pace of innovation is the speed at which technological innovation or
advancement is occurring, with the most apparent instances being too slow or too rapid.</p>
    <div class="profile">
      
      <span class="profile-name">Robinson</span>
    </div>
  </div>
<!-- Post 5 -->
<div class="post-box tech">
  
  <h2 class="category">Tech</h2>
  <a href="#" class="post-title">YouTube Official Blog.</a>
  <span class="post-date">09 Feb 2024</span>
  <p class="post-description">YouTube is an American online video sharing and social media
platform owned by Google. Accessible worldwide, it was launched on February 14, 2005, by Steve Chen,
Chad Hurley, and Jawed Karim, three former employees of PayPal.</p>
    <div class="profile">
      
      <span class="profile-name">Suhani Shah</span>
    </div>
  </div>
<!-- Post 6 -->
<div class="post-box news">
  
  <h2 class="category">News</h2>
  <a href="#" class="post-title">Success RCB won WPL</a>
  <span class="post-date">17 Mar 2024</span>
  <p class="post-description">Royal Challengers Bangalore (RCB) won the 2024 Women's Premier
League (WPL) title, defeating Delhi Capitals (DC) by eight wickets in the final at the Arun Jaitley Stadium
in Delhi on Sunday.</p>

```



```

    <div class="profile">
      
      <span class="profile-name">Rahul Sharma</span>
    </div>
  </div>
  <!-- Post 7 -->
  <div class="post-box tech">
    
    <h2 class="category">Tech</h2>
    <a href="#" class="post-title">Evolutionary Robotics: From Algorithms to Implementations</a>
    <span class="post-date">15 Mar 2024</span>
    <p class="post-description"> Evolutionary robotics and computational intelligence, and how
different techniques are applied to robotic system design. </p>
    <div class="profile">
      
      <span class="profile-name">Avleen Luthra</span>
    </div>
  </div>
  <!-- Post 1 -->
  <div class="post-box news">
    
    <h2 class="category">News</h2>
    <a href="#" class="post-title">Donald Trump was assassinated</a>
    <span class="post-date">25 Feb 2024</span>
    <p class="post-description">Donald Trump assassination attempt prediction as 2024 election gets
closer: from Laura Loomer to Alex Jones.</p>
    <div class="profile">
      
      <span class="profile-name">James Anderson</span>
    </div>
  </div>
  <!-- Post 9 -->
  <div class="post-box food">
    
    <h2 class="category">Food</h2>
    <a href="#" class="post-title">Sweet Peas and Saffron</a>
    <span class="post-date">12 Dec 2023</span>
    <p class="post-description">Denise Bustard started Sweet Peas & Saffron in 2012 as a way to
document her kitchen experiments, and shared all things sweet, savory and indulgent.</p>
    <div class="profile">
      
      <span class="profile-name">Scarlet</span>
    </div>
  </div>
</div>

<footer>
  <div class="footer-container">

```

```

<div class="sec aboutus">
  <h2>About Us</h2>
  <p>Welcome to Blog page, your go-to destination for reading different blogs are listed above.
Our team is passionate about providing you best blogs in categories listed above.</p>
  <ul class="sci">
    <li><a href="#"><i class="bx bxl-facebook"></i></a></li>
    <li><a href="#"><i class="bx bxl-instagram"></i></a></li>
    <li><a href="#"><i class="bx bxl-twitter"></i></a></li>
    <li><a href="#"><i class="bx bxl-linkedin"></i></a></li>
  </ul>
</div>
<div class="sec quicklinks">
  <h2>Quick Links</h2>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
  </ul>
</div>
<div class="sec contactBx">
  <h2>Contact Info</h2>
  <ul class="info">
    <li>
      <span><i class='bx bxs-map'></i></span>
      <span> London street <br> Great Britain <br> UK </span>
    </li>
    <li>
      <span><i class='bx bx-envelope' ></i></span>
      <p><a href="mailto:codemyhobby9@gmail.com">abcxyz@gmail.com</a></p>
    </li>
  </ul>
</div>
</div>
</footer>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.1/jquery.min.js" integrity="sha512-
aVKKRRi/Q/YV+4mjoKBsE4x3H+BkegoM/em46NNiCqNTmUYADjBbeNefNxYV7giUp0VxICtqdrbqU7iVaeZ
NXA==" crossorigin="anonymous" referrerpolicy="no-referrer"></script>
<script src="script.js"></script>
</body>
</html>

```

CSS Code

```

*{
  font-family: 'Poppins', sans-serif;
  margin: 0;
  padding: 0;

```

```
    scroll-behavior: smooth;
    scroll-padding-top: 2rem;
    box-sizing: border-box;
  }
:root{
  --container-color: #1a1e21;
  --second-color: rgba(77, 228, 255);
  --text-color: #172317;
  --bg-color: #fff;
}
::selection{
  color: var(--bg-color);
  background: var(--second-color);
}
a{
  text-decoration: none;
}
li{
  list-style: none;
}
img{
  width: 100%;
}
section{
  padding: 3rem 0 2rem;
}
.container{
  max-width: 1068px;
  margin: auto;
  width: 100%;
}
a{
  color: #fff;
}
header{
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  z-index: 200;
}
header.shadow{
  background: var(--bg-color);
  box-shadow: 0 1px 4px hsl(0 4% 14% / 10%);
  transition: .5s;
}
header.shadow .logo{
  color: var(--text-color);
}
.nav{
  display: flex;
```

Gurpreet Singh

```
    align-items: center;
    justify-content: space-between;
    padding: 18px 0;
}
.logo{
    font-size: 1.5rem;
    font-weight: 600;
    color: var(--bg-color);
}
.logo span{
    color: var(--second-color);
}
.login{
    padding: 8px 14px;
    text-transform: uppercase;
    font-weight: 500;
    border-radius: 4px;
    background: var(--second-color);
    color: var(--bg-color);
}
.login:hover{
    background: hsl(199, 98%, 56%);
    transition: .5s;
}
.home{
    width: 100%;
    min-height: 440px;
    background: url("images/banner2.png");
    display: grid;
    justify-content: center;
    align-items: center;
}
.home-text{
    color: var(--bg-color);
    text-align: center;
}
.home-title{
    font-size: 3.5rem;
}
.home-subtitle{
    font-size: 1rem;
    font-weight: 400;
}
.about{
    position: relative;
    width: 100%;
    display: flex !important;
    justify-content: center;
    align-items: center;
}
.about .contentBx{
```

```
    max-width: 50%;
    width: 50%;
    text-align: left;
    padding-right: 40px;
}
.titleText{
    font-weight: 600;
    color: #111;
    font-size: 2rem;
    margin-bottom: 10px;
}
.title-text{
    color: #111;
    font-size: 1em;
}
.about .imgBx{
    position: relative;
    min-width: 50%;
    width: 50%;
    min-height: 500px;
}
.btn2{
    position: relative;
    display: inline-block;
    margin-top: 30px;
    padding: 10px 30px;
    background: #fff;
    border: .8px solid #111;
    color: #333;
    text-decoration: none;
    transition: 0.5s;
}
.btn2:hover{
    background-color: var(--second-color);
    border: none;
    color: #fff;
}
.post-filter{
    display: flex;
    justify-content: center;
    align-items: center;
    column-gap: 1.5rem;
    margin-top: 2rem !important;
}
.filter-item{
    font-size: 0.9rem;
    font-weight: 500;
    cursor: pointer;
}
.active-filter{
    background: var(--second-color);
```

```
    color: var(--bg-color);
    padding: 4px 10px;
    border-radius: 4px;
}
.post{
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(280px, auto));
    justify-content: center;
    gap: 1.5rem;
}
.post-box{
    background: var(--bg-color);
    box-shadow: 0 4px 14px hsl(35deg 25% 15% / 10%);
    padding: 15px;
    border-radius: 0.5rem;
}
.post-img{
    width: 100%;
    height: 200px;
    object-fit: cover;
    object-position: center;
    border-radius: 0.5rem;
}
.category{
    font-size: 0.9rem;
    font-weight: 500;
    text-transform: uppercase;
    color: var(--second-color);
}
.post-title{
    font-size: 1.3rem;
    font-weight: 600;
    color: var(--text-color);
    display: -webkit-box;
    -webkit-line-clamp: 2;
    -webkit-box-orient: vertical;
    overflow: hidden;
}
.post-date{
    display: flex;
    font-size: 0.875rem;
    text-transform: uppercase;
    margin-top: 4px;
    font-weight: 400;
}
.post-description{
    font-size: 0.9rem;
    line-height: 1.5rem;
    margin: 5px 0 10px;
    display: -webkit-box;
    -webkit-line-clamp: 2;
```

```
-webkit-box-orient: vertical;
overflow: hidden;
}
.profile{
  display: flex;
  align-items: center;
  gap: 10px;
}
.profile-img{
  width: 35px;
  height: 35px;
  border-radius: 50%;
  object-fit: cover;
  object-position: center;
  border: 2px solid var(--second-color);
}
.profile-name{
  font-size: .8rem;
  font-weight: 500;
}
footer{
  position: relative;
  width: 100%;
  height: auto;
  padding: 50px 100px;
  margin-top: 3rem;
  background: #111;
  display: flex;
  font-family: sans-serif;
  justify-content: space-between;
}
.footer-container{
  display: flex;
  justify-content: space-between;
  flex-wrap: wrap;
  flex-direction: row;
}
.footer-container .sec{
  margin-right: 30px;
}
.footer-container .sec.aboutus{
  width: 40%;
}
.footer-container h2{
  position: relative;
  color: #fff;
  margin-bottom: 15px;
}
.footer-container h2::before{
  content: ";
  position: absolute;
```

```
    bottom: -5px;
    left: 0;
    width: 50px;
    height: 2px;
    background: rgb(77, 228, 255);
}
footer p{
    color: #fff;
}
.sci{
    margin: 20px;
    display: flex;
}
.sci li{
    list-style: none;
}
.sci li a{
    display: inline-block;
    width: 40px;
    height: 40px;
    background: #222;
    display: flex;
    justify-content: center;
    align-items: center;
    margin-right: 10px;
    text-decoration: none;
    border-radius: 4px;
    transition: .5s;
}
.sci li a:hover{
    background: rgb(77, 228, 255);
}
.sci i a .bx{
    color: #fff;
    font-size: 20px;
}
.quicklinks{
    position: relative;
    width: 25%;
}
.quicklinks ul li{
    list-style: none;
}
.quicklinks ul li a{
    color: #999;
    text-decoration: none;
    margin-bottom: 10px;
    display: inline-block;
    transition: .3s;
}
.quicklinks ul li a:hover{
    color: #fff;
}
```

Gurpreet Singh


```
    color: #fff;
  }
.footer-container .contactBx{
  width: calc(35% - 60px);
  margin-right: 0 !important;
}
.contactBx .info{
  position: relative;
}
.contactBx .info li{
  display: flex !important;
  margin-bottom: 16px;
}
.contactBx .info li span:nth-child(1){
  color: #fff;
  font-size: 20px;
  margin-right: 10px;
}
.contactBx .info li span{
  color: #999;
}
.contactBx .info li a{
  color: #999;
  text-decoration: none;
  transition: .5s;
}
.contactBx .info li a:hover{
  color: #fff;
}
@media (max-width: 1060px){
  .container{
    margin: 0 auto;
    width: 95%;
  }
  .home-text{
    width: 100%
  }
}
@media (max-width: 768px){
  .nav{
    padding: 10px 0;
  }

  section{
    padding: 2rem 0 !important;
  }
  .header-content{
    margin-top: 3rem !important;
  }
  .home{
    min-height: 380px;
  }
}
```

```
}
.home-title{
  font-size: 3rem;
}
.header-title{
  font-size: 2rem;
}
.header-img{
  height: 370px;
}
.about{
  flex-direction: column;
}
.about .contentBx{
  min-width: 100%;
  width: 100%;
  text-align: center;
  padding-right: 0px;
}
.about .contentBx,
.about .imgBx{
  min-width: 100%;
  width: 100%;
  padding-right: 0px;
  text-align: center;
}
.about .imgBx{
  min-height: 250px;
}
.btn2{
  margin-bottom: 30px;
}
.post-header{
  height: 435px;
}
.post-header{
  margin-top: 9rem !important;
}
}
@media (max-width: 570px){
  .post-header{
    height: 390px;
  }
  .header-title{
    width: 100%;
  }
  .header-img{
    height: 340px;
  }
}
}
@media (max-width: 396px){
  Gurpreet Singh
```

```
.home-title{
  font-size: 2rem;
}
.home-subtitle{
  font-size: 0.9rem;
}
.home{
  min-height: 300px;
}
.post-box{
  padding: 10px;
}
.header-title{
  font-size: 1.4rem;
}
.header-img{
  height: 240px;
}
.post-header{
  height: 335px;
}
.header-img{
  height: 340px;
}
}
/* Footer Media Query */
@media (max-width: 991px){
  footer{
    padding: 40px;
    font-size: 20px;
  }
  footer .footer-container{
    flex-direction: column;
  }
  footer .footer-container .sec{
    margin-right: 0;
    margin-bottom: 40px;
  }
  footer .footer-container .sec.aboutus{
    width: 100% !important;
  }
  footer .footer-container .quicklinks{
    width: 100%;
  }
  footer .footer-container .contactBx{
    width: 100%;
  }
}}
```

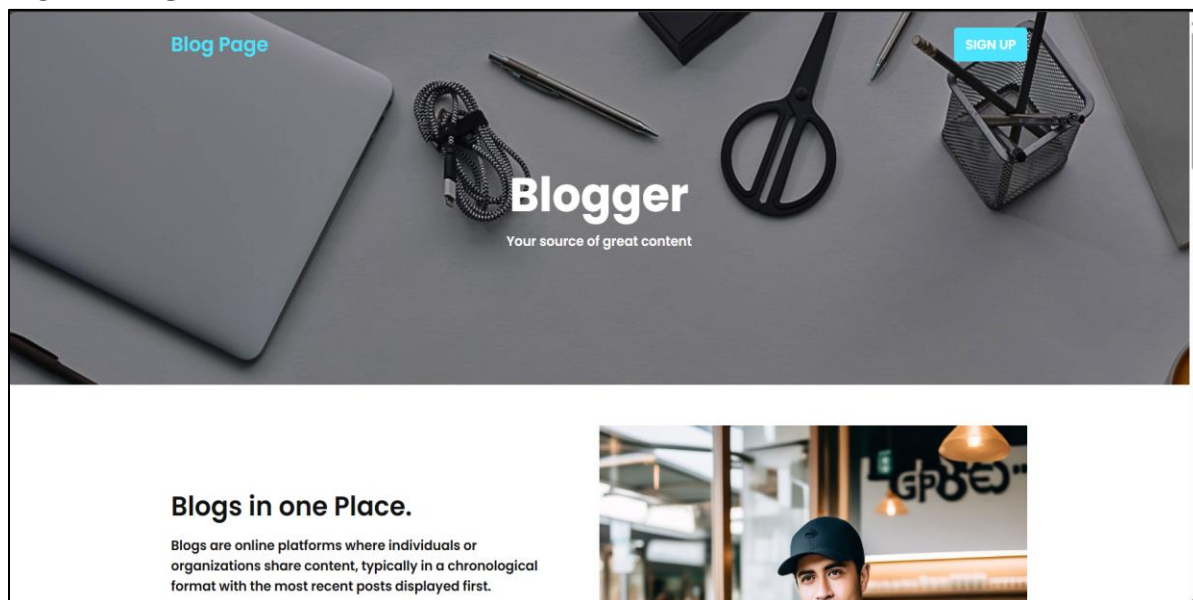
Javascript code

```
// nav background
let header = document.querySelector("header");


window.addEventListener("scroll", () => {
  header.classList.toggle("shadow", window.scrollY > 0)
})
//Filter
$(document).ready(function () {
  $(".filter-item").click(function () {
    const value = $(this).attr("data-filter");
    if (value == "all"){
      $(".post-box").show("1000")
    } else{
      $(".post-box")
        .not("." + value)
        .hide(1000);
      $(".post-box")
        .filter("." + value)
        .show("1000")
    }
  });
  $(".filter-item").click(function () {
    $(this).addClass("active-filter").siblings().removeClass("active-filter")
  });
});
```

WEBSITES SNIPPETS

HOME PAGE




ALL BLOGS



Artificial Intelligence Microchip
12 FEB 2023
An AI chip focuses on powering AI functions. AI workloads require massive...


Gaurav Choudhary

Tech Food News




Benefits of having tech watch for competitive...
13 JAN 2024
The benefits that implementing a technology watching and competitive...

Lucy Wareham




Tropical Fruit Forum - International Tropical Fruit...
29 DEC 2023
Tropical Fruit Forum offers access to not only a wide variety of fruit trees and see...

Jason Roy




Pace of Technology Innovation
1 MAR 2024
Pace of innovation is the speed at which technological innovation or advances...

Robinson




YouTube Official Blog.
09 FEB 2024
YouTube is an American online video sharing and social media platform owne...

Suhani Shah




Success RCB won WPL
17 MAR 2024
Royal Challengers Bangalore (RCB) won the 2024 Women's Premier League (WPL)...

Rahul Sharma




Evolutionary Robotics: From Algorithms to...
15 MAR 2024
Evolutionary robotics and computational intelligence, and how different technique...

Avleen Luthra



Donald Trump was assassinated
25 FEB 2024
Donald Trump assassination attempt prediction as 2024 election gets closer...

James Anderson




Sweet Peas and Saffron
12 DEC 2023
Denise Bustard started Sweet Peas & Saffron in 2012 as a way to document her...

Scarlet


TECH BLOGS

All Tech Food News



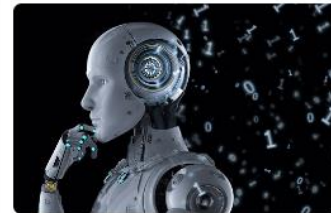
Artificial Intelligence Microchip
12 FEB 2023
An AI chip focuses on powering AI functions. AI workloads require massive...

Gaurav Choudhary



YouTube Official Blog.
09 FEB 2024
YouTube is an American online video sharing and social media platform owne...

Suhani Shah




Evolutionary Robotics: From Algorithms to...
15 MAR 2024
Evolutionary robotics and computational intelligence, and how different technique...

Avleen Luthra

FOOD BLOGS

All Tech **Food** News





TECH

Benefits of having tech watch for competitive...

13 JAN 2024

The benefits that implementing a technology watching and competitive...

 Lucy Warehem





FOOD

Tropical Fruit Forum – International Tropical Fruit...

29 DEC 2023

Tropical Fruit Forum offers access to not only a wide variety of fruit trees and see...

 Jason Roy




FOOD

Sweet Peas and Saffron


12 DEC 2023

Denise Bustard started Sweet Peas & Saffron in 2012 as a way to document her...

 Scarlet

NEWS BLOGS

All Tech Food **News**





TECH

Pace of Technology Innovation

1 MAR 2024

Pace of innovation is the speed at which technological innovation or advanceme...

 Robinson





NEWS

Success RCB won WPL

17 MAR 2024

Royal Challengers Bangalore (RCB) won the 2024 Women's Premier League (WPL...

 Rahul Sharma




NEWS

Donald Trump was assassinated

25 FEB 2024

Donald Trump assassination attempt prediction as 2024 election gets closer...

 James Anderson

Date:

PRACTICAL – 3

Aim: : Installation of Node.JS and Express.JS

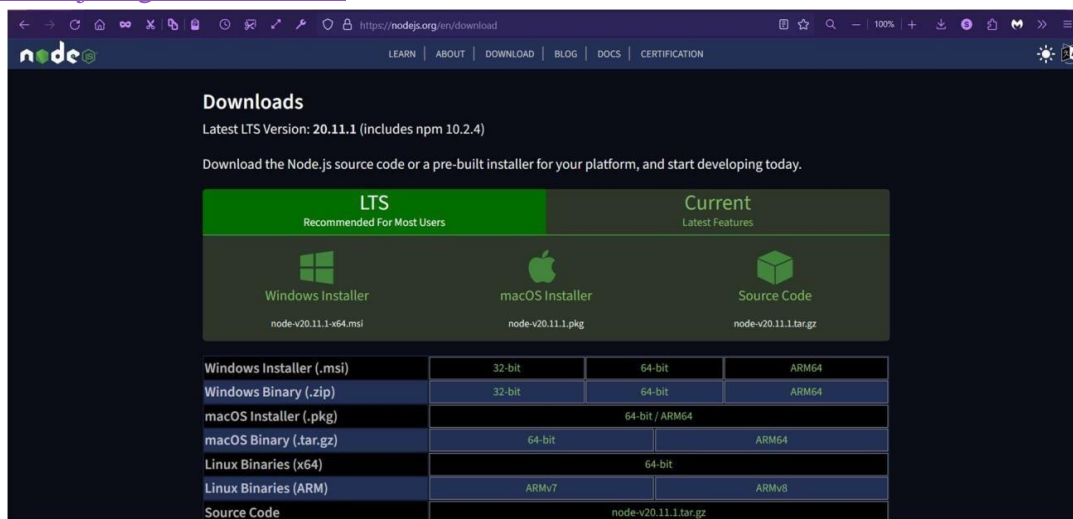
Theory: The Node can be installed in multiple ways on a computer. The approach used by you depends on the existing development environment in the system. There are different package installers for different environments. You can install Node by grabbing a copy of the source code and compiling the application. Another way of installing Node is by cloning the GIT repository in all three environments and then installing it on the system.

Installing Node on Windows (WINDOWS 10):

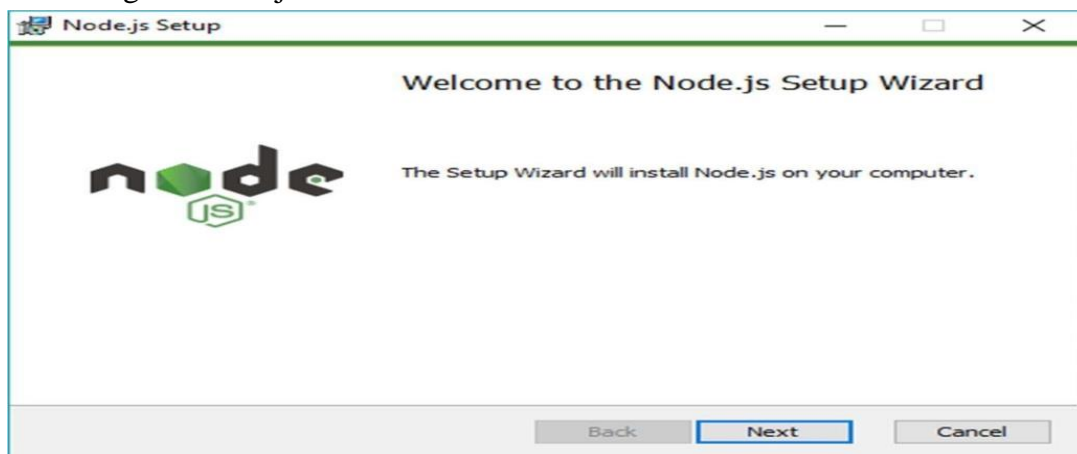
You have to follow the following steps to install the Node.js on your Windows:

Step 1: Downloading the Node.js ‘.msi’ installer the first step to install Node.js on Windows is to download the installer. Visit the official Node.js website i.e.)

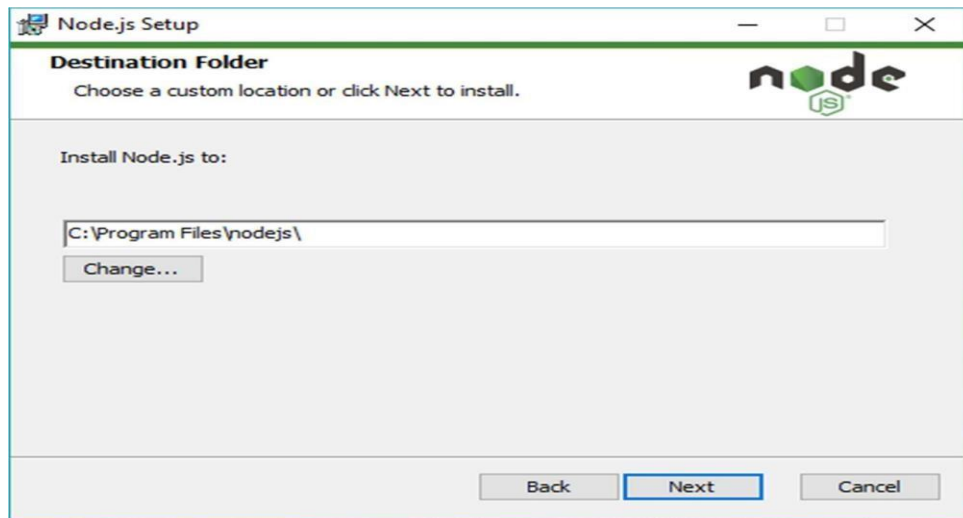
<https://nodejs.org/en/download/>



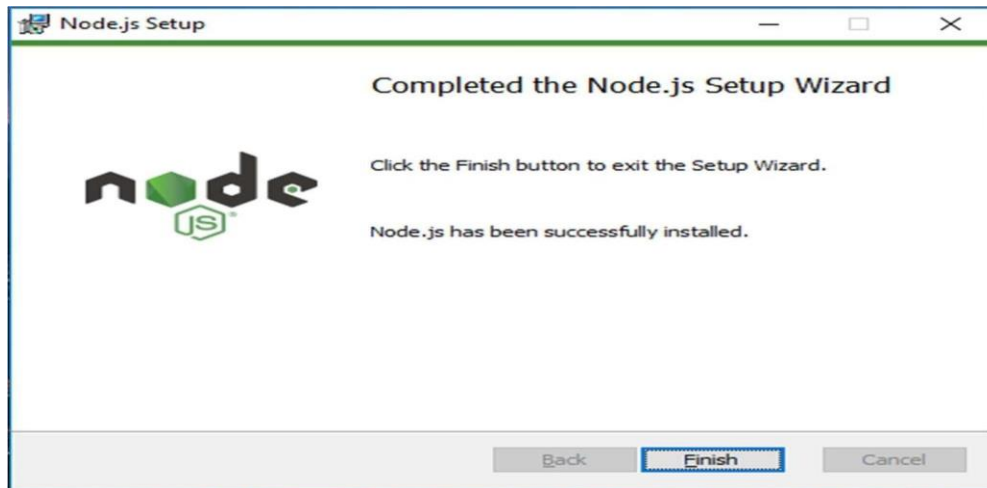
Step 2: Running the Node.js installer



Step 3: After clicking “Next”, End-User License Agreement (EULA) will open.



Step 4: Complete the Node.js Setup Wizard.



Step 5: Verify that Node.js was properly installed or not.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

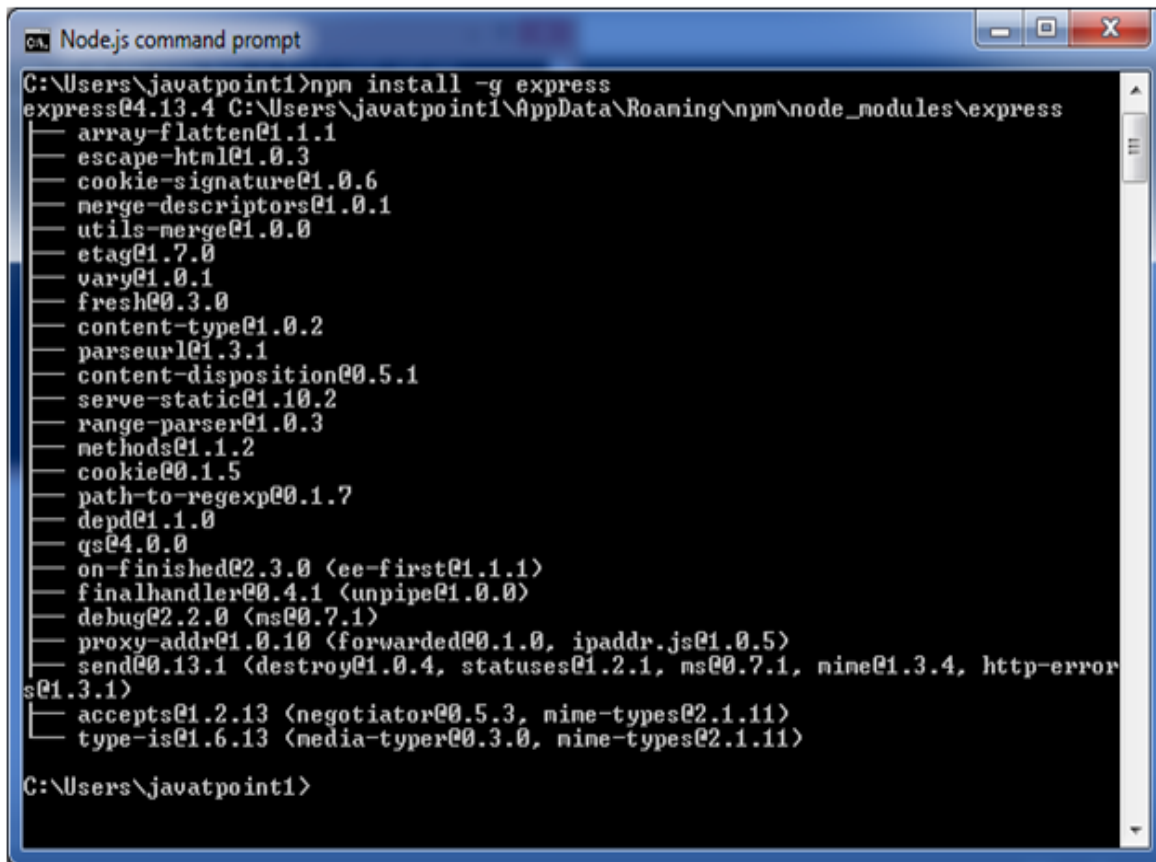
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\guddu> node -v
v18.18.1
PS C:\Users\guddu>
```


Installation of Express JS after Node JS

Firstly, you have to install the express framework globally to create web application using Node terminal. Use the following command to install express framework globally.

1. `npm install -g express`



```
CA Node.js command prompt
C:\Users\javatpoint1>npm install -g express
express@4.13.4 C:\Users\javatpoint1\AppData\Roaming\npm\node_modules\express
├── array-flatten@1.1.1
├── escape-html@1.0.3
├── cookie-signature@1.0.6
├── merge-descriptors@1.0.1
├── utils-merge@1.0.0
├── etag@1.7.0
├── vary@1.0.1
├── fresh@0.3.0
├── content-type@1.0.2
├── parseurl@1.3.1
├── content-disposition@0.5.1
├── serve-static@1.10.2
├── range-parser@1.0.3
├── methods@1.1.2
├── cookie@0.1.5
├── path-to-regexp@0.1.7
├── depd@1.1.0
├── qs@4.0.0
├── on-finished@2.3.0 <ee-first@1.1.1>
├── finalhandler@0.4.1 <unpipe@1.0.0>
├── debug@2.2.0 <ms@0.7.1>
├── proxy-addr@1.0.10 <forwarded@0.1.0, ipaddr.js@1.0.5>
├── send@0.13.1 <destroy@1.0.4, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-error
s@1.3.1>
├── accepts@1.2.13 <negotiator@0.5.3, mime-types@2.1.11>
└── type-is@1.6.13 <media-typer@0.3.0, mime-types@2.1.11>

C:\Users\javatpoint1>
```

Use the following command to install express:

1. `npm install express --save`

```

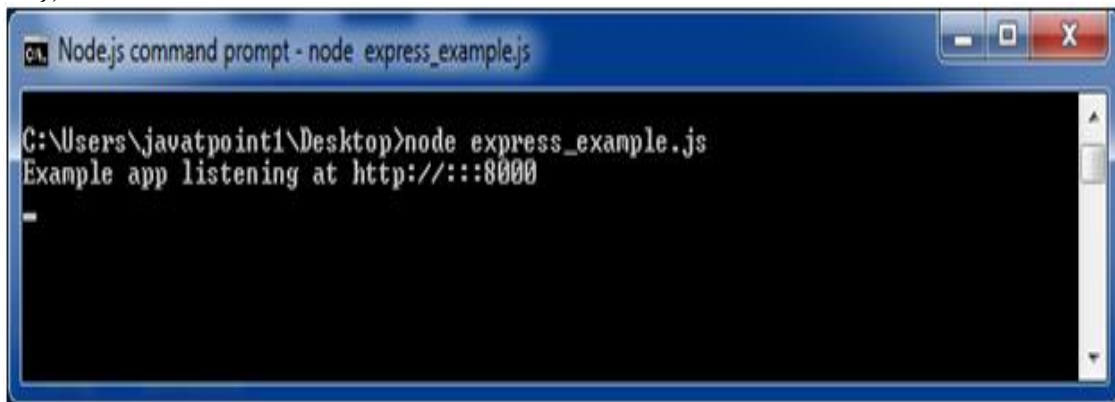
C:\Users\javatpoint1>npm install express --save
express@4.13.4 node_modules\express
├── array-flatten@1.1.1
├── escape-html@1.0.3
├── cookie-signature@1.0.6
├── utils-merge@1.0.0
├── serve-static@1.10.2
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── content-type@1.0.2
├── methods@1.1.2
├── etag@1.7.0
├── merge-descriptors@1.0.1
├── content-disposition@0.5.1
├── fresh@0.3.0
├── cookie@0.1.5
├── range-parser@1.0.3
├── parseurl@1.3.1
├── depd@1.1.0
├── qs@4.0.0
├── finalhandler@0.4.1 (unpipe@1.0.0)
├── on-finished@2.3.0 (ee-first@1.1.1)
├── debug@2.2.0 (ms@0.7.1)
├── proxy-addr@1.0.10 (forwarded@0.1.0, ipaddr.js@1.0.5)
├── type-is@1.6.13 (media-typer@0.3.0, mime-types@2.1.11)
├── accepts@1.2.13 (negotiator@0.5.3, mime-types@2.1.11)
└── send@0.13.1 (destroy@1.0.4, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)
C:\Users\javatpoint1>_

```

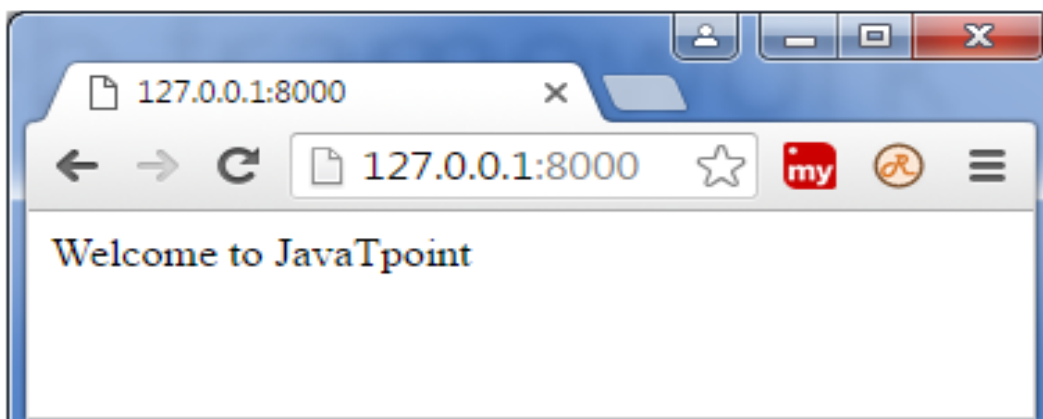
The above command install express in node_module directory and create a directory named express inside the node_module. You should install some other important modules along with express. Following is the list:

- **body-parser:** This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser:** It is used to 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.

1. npm install body-parser --save
2. File: express_example.js
3. var **express** = **require**('express');
4. var **app** = **express**();
5. app.get('/', function (req, res) {
6. res.send('Welcome to JavaTpoint');
7. })
8. var **server** = **app**.listen(8000, function () {
9. var **host** = **server**.address().address
10. var **port** = **server**.address().port
11. console.log("Example app listening at http://%s:%s", host, port)
12. })



Open <http://127.0.0.1:8000/> in your browser to see the result.



Date:

PRACTICAL – 4

Aim: : Use Angular CLI to create a new component and implement data binding techniques such as string interpolation, property binding, two-way data binding, and event binding.

Theory: Open command prompt and create new Angular application using below command

Cd /go/to/workspace

ng new databind-app

cd databind-app

Create a **test** component using Angular CLI as mentioned below –

ng generate component test

The above create a new component and the output is as follows –

CREATE src/app/test/test.component.scss (0 bytes) CREATE
src/app/test/test.component.html (19 bytes) CREATE src/app/test/test.component.spec.ts
(614 bytes)

CREATE src/app/test/test.component.ts (262 bytes) UPDATE src/app/app.module.ts (545
bytes)

Run the application using below command –

ng serve.

One-way data binding is a one-way interaction between component and its template. If you perform any changes in your component, then it will reflect the HTML elements. It supports the following types –

String Interpolation

In general, **String interpolation** is the process of formatting or manipulating strings. In Angular, **Interpolation** is used to display data from component to view (DOM). It is denoted by the expression of `{{ }}` and also known as mustache syntax.

Let's create a simple string property in component and bind the data to view.

Add the below code in **test.component.ts** file as follows –

```
export class TestComponent implements OnInit {
  appName = "My first app in Angular 8";
```

```
}
```

Move to test.component.html file and add the below code –

```
<h1>{{appName}}</h1>
```

Add the test component in your **app.component.html** file by replacing the existing content as follows –

```
<app-test></app-test>
```

Finally, start your application (if not done already) using the below command –

```
ng serve
```

You could see the following output on your screen –



Event binding

Events are actions like mouse click, double click, hover or any keyboard and mouse actions. If a user interacts with an application and performs some actions, then event will be raised. It is denoted by either parenthesis () or **on-**. We have different ways to bind an event to DOM element. Let's understand one by one in brief.

Component to view binding

Let's understand how simple button click even handling works.

Add the following code in **test.component.ts** file as follows –

```
export class TestComponent {  
  showData($event: any){  
    console.log("button is clicked!"); if($event) {  
      console.log($event.target);  
      console.log($event.target.value);  
    }  
  }  
}
```

```

    }
  }
}

```

event has all the information about event and the target element. Here, the target is button. \$event.target property will have the target information.

We have two approaches to call the component method to view (**test.component.html**). First one is defined below –

```
<h2>Event Binding</h2>
```

```
<button (click)="showData($event)">Click here</button>
```

Alternatively, you can use **prefix - on** using canonical form as shown below –

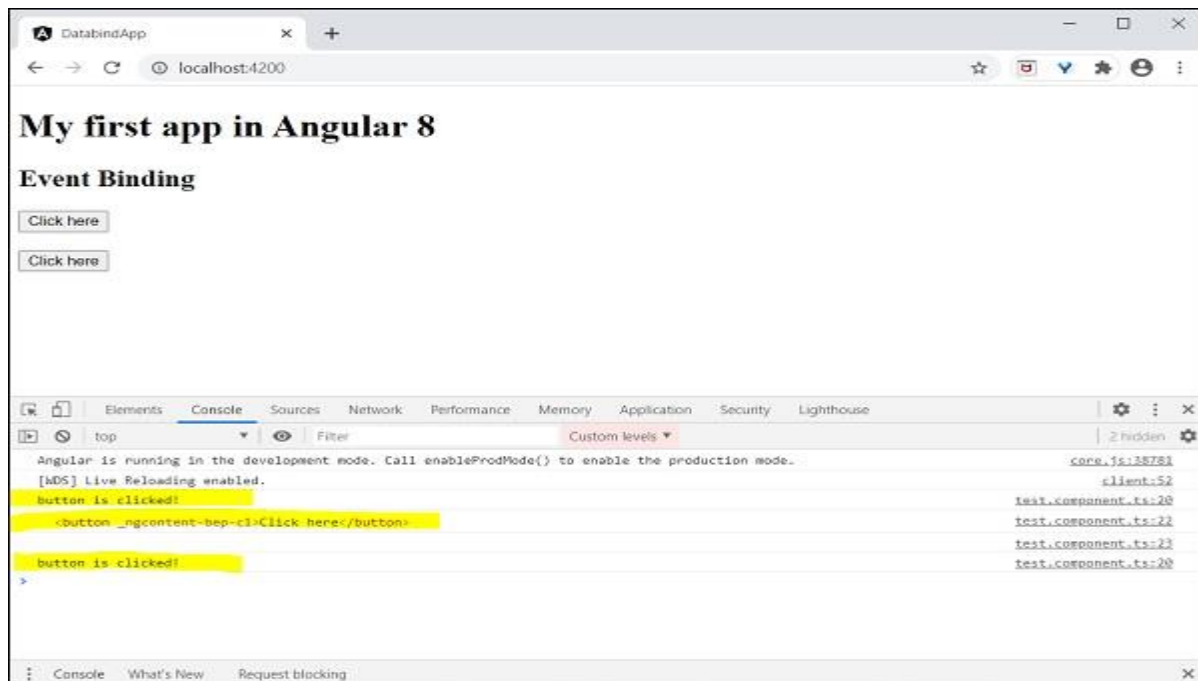
```
<button on-click = "showData()">Click here</button>
```

Here, we have not used **\$event** as it is optional.

Finally, start your application (if not done already) using the below command –

ng serve

Now, run your application and you could see the below response –



Here, when the user clicks on the button, event binding understands to button click action and call component `showData()` method so we can conclude it is one-way binding.

Property binding is used to bind the data from property of a component to DOM elements. It is denoted by `[]`.

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
export class TestComponent {  
  userName:string = "Peter";  
}
```

Add the below changes in view `test.component.html`,

```
<input type="text" [value]="userName">
```

Here,

userName property is bind to an attribute of a DOM element **<input>** tag.

Finally, start your application (if not done already) using the below command –

`ng serve`



Attribute binding is used to bind the data from component to HTML attributes. The syntax is as follows –

```
<HTMLTag [attr.ATTR]="Component data">
```

For example,

```
<td [attr.colspan]="columnSpan"> ... </td>
```

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
export class TestComponent {
  userName:string = "Peter";
}
```

Add the below changes in view **test.component.html**,

```
<input type="text" [value]="userName">
```

Here,

userName property is bind to an attribute of a DOM element <input> tag.

Finally, start your application (if not done already) using the below command – ng serve



Class binding is used to bind the data from component to HTML class property. The syntax is as follows –


```
<HTMLTag [class]="component variable holding class name">
```

Class Binding provides additional functionality. If the component data is boolean, then the class will bind only when it is true. Multiple class can be provided by string (“foo bar”) as well as Array of string. Many more options are available.

For example,

```
<p [class]="myClasses">
```

Let’s understand with a simple example.

Add the below code in test.component.ts file,

```
export class TestComponent {
  myCSSClass = "red";
  applyCSSClass = false;
}
```

Add the below changes in view **test.component.html**.

```
<p [class]="myCSSClass">This paragraph class comes from *myClass* property </p>
<p [class.blue]="applyCSSClass">This paragraph class does not apply</p>
```

Add the below content in test.component.css.

```
.red {
  color: red;
}
.blue {
  color: blue;
}
```

Finally, start your application (if not done already) using the below command –

```
ng serve
```

The final output will be as shown below –



Style binding is used to bind the data from component into HTML style property. The syntax is as follows –

```
<HTMLTag [style.STYLE]="component data">
```

For example,

```
<p [style.color]="myParaColor"> ... </p>
```

Let's understand with a simple example.

Add the below code in **test.component.ts** file.

```
myColor = 'brown';
```

Add the below changes in view **test.component.html**.

```
<p [style.color]="myColor">Text color is styled using style binding</p>
```

Finally, start your application (if not done already) using the below command –

```
ng serve
```

The final output will be as shown below –



Two-way data binding is a two-way interaction, data flows in both ways (from component to views and views to component). Simple example is **ngModel**. If you do any changes in your property (or model) then, it reflects in your view and vice versa. It is the combination of property and event binding.

NgModel

NgModel is a standalone directive. **ngModel** directive binds form control to property and property to form control. The syntax of **ngModel** is as follows –

```
<HTML [(ngModel)]="model.name" />
```

For example,

```
<input type="text" [(ngModel)]="model.name" />
```

Let's try to use **ngModel** in our test application.

Configure **FormsModule** in **AppModule** (src/app/app.module.ts)

```
import { FormsModule } from '@angular/forms'; @NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ]
})
export class AppModule { }
```

FormModule do the necessary setup to enable two-way data binding.

Update **TestComponent** view (**test.component.html**) as mentioned below –

```
<input type="text" [(ngModel)]="userName" />
<p>Two way binding! Hello {{ userName }}!</p>
```

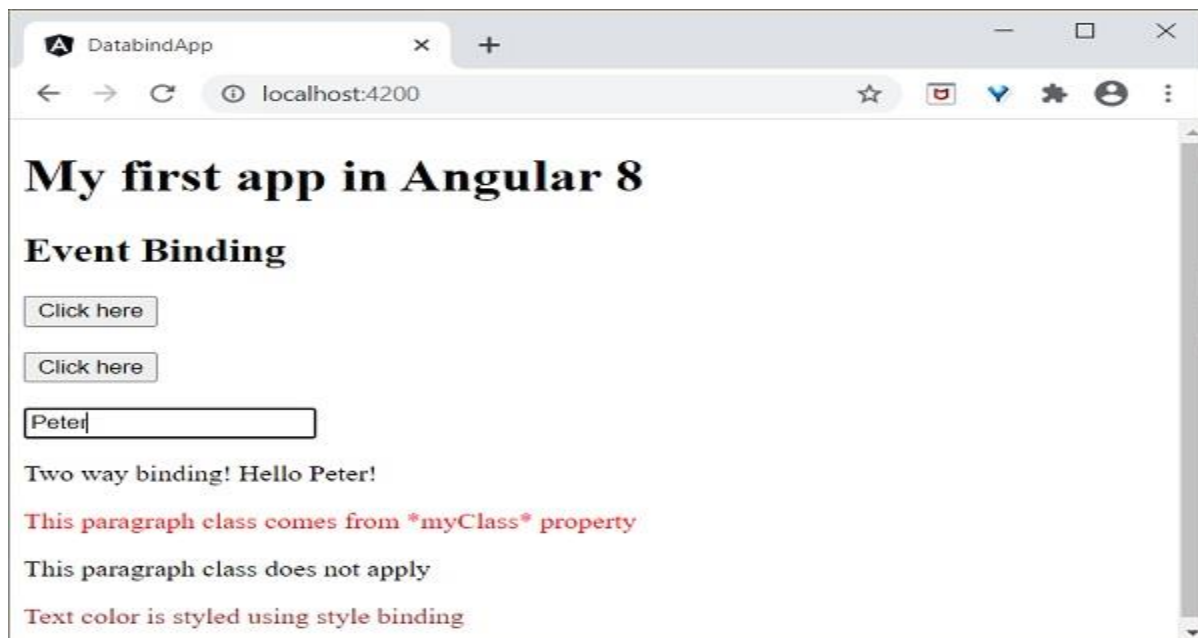
Here,

Property is bind to form control **ngModel** directive and if you enter any text in the textbox, it will bind to the property. After running your application, you could see the below changes –

Finally, start your application (if not done already) using the below command –

ng serve

Now, run your application and you could see the below response –



Now, try to change the input value to **Jack**. As you type, the text below the input gets changed and the final output will be as shown below –



We will learn more about form controls in the upcoming chapters.

Working example

Let us implement all the concept learned in this chapter in our **ExpenseManager** application.

Open command prompt and go to project root folder. `cd /go/to/expense-manager`

Create `ExpenseEntry` interface (`src/app/expense-entry.ts`) and add `id`, `amount`, `category`, `Location`, `spendOn` and `createdOn`.

Import **ExpenseEntry** into **ExpenseEntryComponent**.

```
import { ExpenseEntry } from '../expense-entry';
```

Create a **ExpenseEntry** object, **expenseEntry** as shown below –

```
export class ExpenseEntryComponent implements OnInit {
  title: string;
  expenseEntry: ExpenseEntry;
  constructor() { }
  ngOnInit() {
    this.title = "Expense Entry";
    this.expenseEntry = {
      id: 1,
      item: "Pizza",
      amount: 21,
      category: "Food",

```

```

    location: "Zomato",
    spendOn: new Date(2020, 6, 1, 10, 10, 10), createdOn: new Date(2020, 6, 1, 10, 10,
10),
  };
}
}

```

Update the component template using **expenseEntry** object, **src/app/expense-entry/expense-entry.component.html** as specified below –

```

<!-- Page Content -->
<div class="container">
  <div class="row">
    <div class="col-lg-12 text-center" style="padding-top: 20px;">
      <div class="container" style="padding-left: 0px; padding-right: 0px;">
        <div class="row">
          <div class="col-sm" style="text-align: left;">
            {{ title }}
          </div>
          <div class="col-sm" style="text-align: right;">
            <button type="button" class="btn btn-primary">Edit</button>
          </div>
        </div>
      </div>
    <div class="container box" style="margin-top: 10px;">
      <div class="row">
        <div class="col-2" style="text-align: right;">
          <strong><em>Item:</em></strong>
        </div>
        <div class="col" style="text-align: left;">
          {{ expenseEntry.item }}
        </div>
      </div>
      <div class="row">
        <div class="col-2" style="text-align: right;">
          <strong><em>Amount:</em></strong>
        </div>
        <div class="col" style="text-align: left;">
          {{ expenseEntry.amount }}
        </div>
      </div>
      <div class="row">
        <div class="col-2" style="text-align: right;">
          <strong><em>Category:</em></strong>

```

```

</div>
<div class="col" style="text-align: left;">
  {{ expenseEntry.category }}
</div>
</div>
<div class="row">
  <div class="col-2" style="text-align: right;">
    <strong><em>Location:</em></strong>
  </div>
  <div class="col" style="text-align: left;">
    {{ expenseEntry.location }}
  </div>
</div>
<div class="row">
  <div class="col-2" style="text-align: right;">
    <strong><em>Spend On:</em></strong>
  </div>
  <div class="col" style="text-align: left;">
    {{ expenseEntry.spendOn }}
  </div>
</div>
</div>
</div>
</div>
</div>

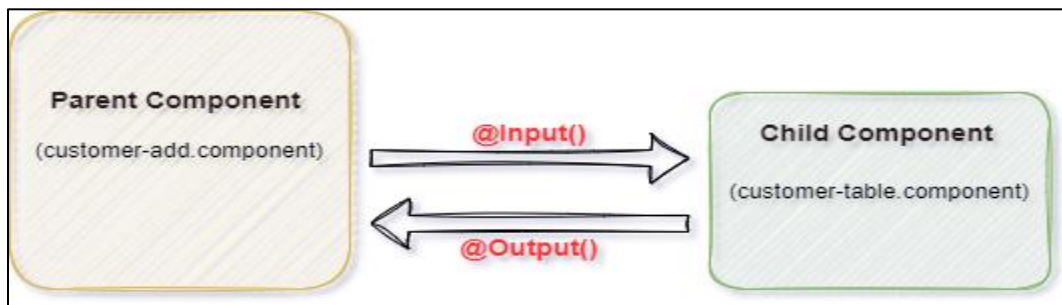
```



Date:**PRACTICAL – 5**

Aim: : Implement component interaction using @Input, @Output decorators, and @ViewChild decorator to get hold of DOM.

Theory: In [Angular](#), @Input() and @Output() are decorators that allow a component to share data with its parent or child components, respectively. @ViewChild() is a decorator that allows a component to access a child component, directive, or DOM element.



Understanding the use of @Input()

@Input() allows a component to receive data from its parent component. For example, a child component might use @Input() to receive data from its parent component like this:

Here, whenever I added a new entry as "Add Customer", the updated information is passed to the child component (e.g. Customer Table) and the child is acting as a term component. And the parent is acting like smart component.

The screenshot shows a web application with a form to 'Enter Customer Name' and an 'Add Customer' button. Below the form is a 'Child Component' table. The table has two columns: 'SI No.' and 'Customer Name'. The table contains two rows of data: (1, Anupam M) and (2, Soumya). The table is highlighted with a red border.

SI No.	Customer Name
1	Anupam M
2	Soumya

customer-add.component acts as a Parent Component

In this example, the parent component contains Add customer sections and a sub-component like a customer table. When we click on Add Customer that information being passed to the child component (customers table) and displayed on the table.


```

<div class="form-group">
  <label>Enter Customer Name:</label>
  <input type="text" class="form-control" [(ngModel)]="customerName">
</div>
<div style="padding-top:10px">
  <input type="button" class="btn btn-primary" value="Add Customer" (click)
    ="addCustomer()"/>
</div>
<hr/>
<app-customer-table [customers]="customerList"></app-customer-table>

```

Child component exposed with property "customers"

```

export class CustomerAddComponent {
  customerName: string;
  customerList: string[];
  constructor() {
    this.customerList = [];
    this.customerName = '';
  }

  addCustomer() {
    this.customerList.push(this.customerName);
  }
}

```

customer-table.component acts as a Child Component,

```

<table class="table table-striped">
  <thead>
    <tr>
      <th>
        Sl No.
      </th>
      <th>
        Customer Name
      </th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let cust of customerList; let indx = index;">
      <td>{{indx+1}}</td>
      <td>{{cust}}</td>
    </tr>
  </tbody>
</table>

```

Child component - .html and .ts file

```

@Component({
  selector: 'app-customer-table',
  templateUrl: './customer-table.component.html',
  styleUrls: ['./customer-table.component.css'],
})
export class CustomerTableComponent {
  @Input('customers')
  customerList: string[];

  constructor() {
    this.customerList = [];
  }
}

```

@Input decorator as "customers"

In the above example, the parent component uses the `@Input()` decorator to bind the "customers" property to a value passed from the parent component.

And the child component uses the `@Input()` decorator to receive the value passed from the parent component.

So, `@Input()` decorator is used to pass the data from the parent to the child component with help of property.

Understanding the use of `@Output()`

`@Output()` allows a component to emit events to its parent component. For example, a child component might use `@Output()` to emit an event when a button is clicked like this:

In the above example, whenever we select data from the customer table which acts as a child, that data should pass to the customer's name textbox which acts as the parent. It's basically sending data from child to parent. To do that, we need to expose an event with an output decorator.

Enter Customer Name:

Suprajata

Add Customer

Child Component

SI No.	Customer Name	Select
1	Anupam	Select
2	Soumya	Select
3	Suprajata	Select

On Select from grid, name is populating on textbox

Enter Customer Name:

Anupam

Add Customer

Child Component

SI No.	Customer Name	Select
1	Anupam	Select
2	Soumya	Select
3	Suprajata	Select



@Output() decorator to listen for an event emitted by the child component, which is then handled in the parent component.



```
export class CustomerAddComponent {
  customerName: string;
  customerList: string[];
  constructor() {
    this.customerList = [];
    this.customerName = '';
  }

  addCustomer() {
    this.customerList.push(this.customerName);
  }

  readData(data: string) {
    this.customerName = data;
  }
}
```

So, @Output() decorator is used to pass the data from child to parent with the help of the event.

Understanding the use of @ViewChild()

@ViewChild() allows a component to access a child component or DOM element. For example, a parent component might use @ViewChild() to access a child component like this:

In order to use the @ViewChild() decorator, we need to reference the child component in the parent component's class, and then we can use to query the child component's properties and methods or manipulate the DOM directly.

```
<div class="form-group">
  <label>Enter Customer Name:</label>
  <input type="text" class="form-control" [(ngModel)]="customerName">
</div>
<div style="padding-top:10px">
  <input type="button" class="btn btn-primary" value="Add Customer" (click)
    ="addCustomer()"/>
</div>

<hr/>
<!-- <app-customer-table [customers]="customerList"
  (onSelectCustomer)="readData($event)"></app-customer-table> -->
<app-customer-table #clist (onSelectCustomer)="readData($event)"></app-customer-table>
```

```

export class CustomerAddComponent {
  customerName: string;
  customerList: string[];

  @ViewChild('clist')
  customerListComp!: CustomerTableComponent;

  constructor() {
    this.customerList = [];
    this.customerName = '';
  }

  addCustomer() {
    //this.customerList.push(this.customerName);
    this.customerListComp.customerList.push(this.customerName);
  }

  readData(data: string) {
    this.customerName = data;
  }
}

```

ViewChild with name "clist"

Another example where @ViewChild() can be used to access the DOM element as like below:

```

<div #errorMessage class="alert alert-primary"></div>
...

```

```

> data-sharing > customer-add > customer-add.component.ts > CustomerAddComponent

```

```

@ViewChild('errorMessage')
alertDiv!: ElementRef;

constructor() {
  this.customerList = [];
  this.customerName = '';
}

addCustomer() {
  //this.customerList.push(this.customerName);
  this.customerListComp.customerList.push(this.customerName);

  this.alertDiv.nativeElement.innerText = 'New Customer Added';

  setTimeout(() => {
    this.alertDiv.nativeElement.innerText = '';
  }, 2000);
}

```

In summary, ViewChild is used to access child components or elements in the template, while @Input and @Output are used to pass data between components. If you need to access a specific child component or element in the template, ViewChild will be more useful, while if you need to pass data between components, @Input and @Output will be more .

Date:**PRACTICAL – 6**

Aim: : Use built-in directives such as ng For, ng If, ng Switch, ng Class, ng Style to manipulate the DOM elements.

Theory: Built-in directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

The different types of Angular directives are as follows:

DIRECTIVE TYPES	DETAILS
<u>Components</u>	Used with a template. This type of directive is the most common directive type.
<u>Attribute directives</u>	Change the appearance or behavior of an element, component, or another directive.
<u>Structural directives</u>	Change the DOM layout by adding and removing DOM elements.

Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many attribute directives are defined through modules such as the [CommonModule](#), [RouterModule](#) and [FormsModule](#).

The most common attribute directives are as follows:

COMMON DIRECTIVES	DETAILS
NgClass	Adds and removes a set of CSS classes.
NgStyle	Adds and removes a set of HTML styles.
NgModel	Adds two-way data binding to an HTML form element.

Built-in directives use only public APIs. They do not have special access to any private APIs that other directives can't access.

Adding and removing classes with [NgClass](#)

Add or remove multiple CSS classes simultaneously with [ngClass](#).

To add or remove a *single* class, use [class binding](#) rather than [NgClass](#).

Import [CommonModule](#) in the component

To use [NgClass](#), import [CommonModule](#) and add it to the component's imports list.

src/app/app.component.ts (CommonModule import)

```
content_copyimport { CommonModule } from '@angular/common';
```

```
/* ... */
```

```
@Component({
  standalone: true,
```

```
/* ... */
```

```
imports: [
```

```
  CommonModule, // <-- import into the component
```

```
/* ... */
```

```
],
```

```
})
```

```
export class AppComponent implements OnInit {
```

```
  /* ... */
```

```
}
```

Using [NgClass](#) with an expression

On the element you'd like to style, add [\[ngClass\]](#) and set it equal to an expression. In this case, `isSpecial` is a boolean set to true in `app.component.ts`. Because `isSpecial` is true, [ngClass](#) applies the class of `special` to the `<div>`.

src/app/app.component.html

```
content_copy<!-- toggle the "special" class on/off with a property -->
```

```
<div [ngClass]="isSpecial ? 'special' : ''>This div is special</div>
```

Using [NgClass](#) with a method

1. To use [NgClass](#) with a method, add the method to the component class. In the following example, `setCurrentClasses()` sets the property `currentClasses` with an object that adds or removes three classes based on the true or false state of three other component properties.

Each key of the object is a CSS class name. If a key is true, [ngClass](#) adds the class. If a key is false, [ngClass](#) removes the class.

src/app/app.component.ts

```
content_copycurrentClasses: Record<string, boolean> = {};

/* ... */

setCurrentClasses() {

    // CSS classes: added/removed per current state of component properties

    this.currentClasses = {

        saveable: this.canSave,

        modified: !this.isUnchanged,

        special: this.isSpecial,

    };

}
```

2. In the template, add the [ngClass](#) property binding to `currentClasses` to set the element's classes:

src/app/app.component.html

```
content_copy<div [ngClass]="currentClasses">This div is initially saveable,
unchanged, and special.</div>
```

For this use case, Angular applies the classes on initialization and in case of changes. The full example calls `setCurrentClasses()` initially with `ngOnInit()` and when the dependent properties change through a button click. These steps are not necessary to implement [ngClass](#). For more information, see the [live example](#) / [download example](#) `app.component.ts` and `app.component.html`.

Setting inline styles with [NgStyle](#)

Import [CommonModule](#) in the component

To use [NgStyle](#), import [CommonModule](#) and add it to the component's imports list.

src/app/app.component.ts (CommonModule import)

```
content_copyimport { CommonModule } from '@angular/common';

/* ... */
@Component({
  standalone: true,
  /* ... */
  imports: [
    CommonModule, // <-- import into the component
    /* ... */
  ],
})
export class AppComponent implements OnInit {
  /* ... */
}
```

Using [NgStyle](#) in your component

Use [NgStyle](#) to set multiple inline styles simultaneously, based on the state of the component.

1. To use [NgStyle](#), add a method to the component class.
In the following example, setCurrentStyles() sets the property currentStyles with an object that defines three styles, based on the state of three other component properties.
src/app/app.component.ts

```
content_copycurrentStyles: Record<string, string> = {};

/* ... */

setCurrentStyles() {
  // CSS styles: set per current state of component properties

  this.currentStyles = {
    'font-style': this.canSave ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold' : 'normal',
```

```
'font-size': this.isSpecial ? '24px' : '12px',
};
}
```

- To set the element's styles, add an [ngStyle](#) property binding to currentStyles.
src/app/app.component.html

```
content_copy<div [ngStyle]="currentStyles">
```

This div is initially italic, normal weight, and extra large (24px).

```
</div>
```

For this use case, Angular applies the styles upon initialization and in case of changes. To do this, the full example calls setCurrentStyles() initially with ngOnInit() and when the dependent properties change through a button click. However, these steps are not necessary to implement [ngStyle](#) on its own. See the [live example](#) / [download example](#) app.component.ts and app.component.html for this optional implementation.

Displaying and updating properties with [ngModel](#)

Use the [NgModel](#) directive to display a data property and update that property when the user makes changes.

- Import [FormsModule](#) and add it to the AppComponent's imports list.
src/app/app.component.ts (FormsModule import)

```
content_copyimport {FormsModule} from '@angular/forms'; // <--- JavaScript
import from Angular

/* ... */
@Component({
  standalone: true,
  /* ... */
  imports: [
    CommonModule, // <-- import into the component
    FormsModule, // <--- import into the component
    /* ... */
  ],
})
export class AppComponent implements OnInit {
  /* ... */
}
```

2. Add an `[(ngModel)]` binding on an HTML `<form>` element and set it equal to the property, here name.
src/app/app.component.html (NgModel example)

```
content_copy<label for="example-ngModel">[(ngModel)]:</label>

<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

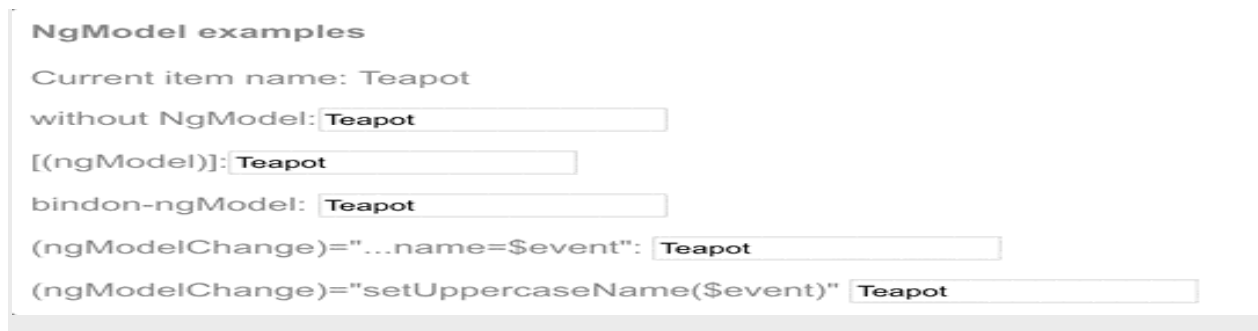
This `[(ngModel)]` syntax can only set a data-bound property.

To customize your configuration, write the expanded form, which separates the property and event binding. Use [property binding](#) to set the property and [event binding](#) to respond to changes. The following example changes the `<input>` value to uppercase:

src/app/app.component.html

```
content_copy<input [ngModel]="currentItem.name"
(ngModelChange)="setUppercaseName($event)" id="example-uppercase">
```

Here are all variations in action, including the uppercase version:



[NgModel](#) and value accessors

The [NgModel](#) directive works for an element supported by a [ControlValueAccessor](#). Angular provides *value accessors* for all of the basic HTML form elements. For more information, see [Forms](#).

To apply `[(ngModel)]` to a non-form built-in element or a third-party custom component, you have to write a value accessor. For more information, see the API documentation on [DefaultValueAccessor](#).

When you write an Angular component, you don't need a value accessor or [NgModel](#) if you name the value and event properties according to Angular's [two-way binding syntax](#).

Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section introduces the most common built-in structural directives:

COMMON BUILT-IN STRUCTURAL DIRECTIVES	DETAILS
NgIf	Conditionally creates or disposes of subviews from the template.
NgFor	Repeat a node for each item in a list.
NgSwitch	A set of directives that switch among alternative views.

Import [CommonModule](#) in the component

To use built-in structural directives, import [CommonModule](#) and add it to the component's imports list.

src/app/app.component.ts (CommonModule import)

```
content_copyimport { CommonModule } from '@angular/common';

/* ... */

@Component({
  standalone: true,
  /* ... */
  imports: [
    CommonModule, // <-- import into the component
    /* ... */
  ],
})

export class AppComponent implements OnInit {
```

```
/* ... */
}
```

Adding or removing an element with [NgIf](#)

Add or remove an element by applying an [NgIf](#) directive to a host element.

When [NgIf](#) is false, Angular removes an element and its descendants from the DOM. Angular then disposes of their components, which frees up memory and resources.

To add or remove an element, bind [*ngIf](#) to a condition expression such as `isActive` in the following example.

src/app/app.component.html

```
content_copy<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

When the `isActive` expression returns a truthy value, [NgIf](#) adds the `ItemDetailComponent` to the DOM. When the expression is falsy, [NgIf](#) removes the `ItemDetailComponent` from the DOM and disposes of the component and all of its subcomponents.

For more information on [NgIf](#) and `NgIfElse`, see the [NgIf API documentation](#).

Guarding against null

By default, [NgIf](#) prevents display of an element bound to a null value.

To use [NgIf](#) to guard a `<div>`, add [*ngIf="yourProperty"](#) to the `<div>`. In the following example, the `currentCustomer` name appears because there is a `currentCustomer`.

src/app/app.component.html

```
content_copy<div *ngIf="currentCustomer">Hello, {{currentCustomer.name}}</div>
```

However, if the property is null, Angular does not display the `<div>`. In this example, Angular does not display the `nullCustomer` because it is null.

src/app/app.component.html

```
content_copy<div *ngIf="nullCustomer">Hello,
<span>{{nullCustomer}}</span></div>
```

Listing items with [NgFor](#)

Use the [NgFor](#) directive to present a list of items.

1. Define a block of HTML that determines how Angular renders a single item.
2. To list your items, assign the shorthand let item of items to [*ngFor](#).

src/app/app.component.html

```
content_copy<div *ngFor="let item of items">{{ item.name }}</div>
```

The string "let item of items" instructs Angular to do the following:

- Store each item in the items array in the local item looping variable
- Make each item available to the templated HTML for each iteration
- Translate "let item of items" into an [<ng-template>](#) around the host element
- Repeat the [<ng-template>](#) for each item in the list

For more information see the [Structural directive shorthand](#) section of [Structural directives](#).

Repeating a component view

To repeat a component element, apply [*ngFor](#) to the selector. In the following example, the selector is <app-item-detail>.

src/app/app.component.html

```
content_copy<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

Reference a template input variable, such as item, in the following locations:

- Within the [ngFor](#) host element
- Within the host element descendants to access the item's properties

The following example references item first in an interpolation and then passes in a binding to the item property of the <app-item-detail> component.

src/app/app.component.html

```
content_copy<div *ngFor="let item of items">{{ item.name }}</div>
```

```
<!-- ... -->
```

```
<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

For more information about template input variables, see [Structural directive shorthand](#).

Getting the index of `*ngFor`

Get the index of `*ngFor` in a template input variable and use it in the template.

In the `*ngFor`, add a semicolon and let `i=index` to the shorthand. The following example gets the index in a variable named `i` and displays it with the item name.

src/app/app.component.html

```
content_copy<div *ngFor="let item of items; let i=index">{{i + 1}} -  
  {{item.name}} </div>
```

The index property of the `NgFor` directive context returns the zero-based index of the item in each iteration.

Angular translates this instruction into an `<ng-template>` around the host element, then uses this template repeatedly to create a new set of elements and bindings for each item in the list. For more information about shorthand, see the [Structural Directives](#) guide.

Repeating elements when a condition is true

To repeat a block of HTML when a particular condition is true, put the `*ngIf` on a container element that wraps an `*ngFor` element.

For more information see [one structural directive per element](#).

Tracking items with `*ngFor` `trackBy`

Reduce the number of calls your application makes to the server by tracking changes to an item list. With the `*ngFor` `trackBy` property, Angular can change and re-render only those items that have changed, rather than reloading the entire list of items.

1. Add a method to the component that returns the value `NgFor` should track. In this example, the value to track is the item's id. If the browser has already rendered id, Angular keeps track of it and doesn't re-query the server for the same id.

src/app/app.component.ts

```
content_copytrackByItems(index: number, item: Item): number {  
  return item.id;  
}
```

2. In the shorthand expression, set `trackBy` to the `trackByItems()` method.

src/app/app.component.html

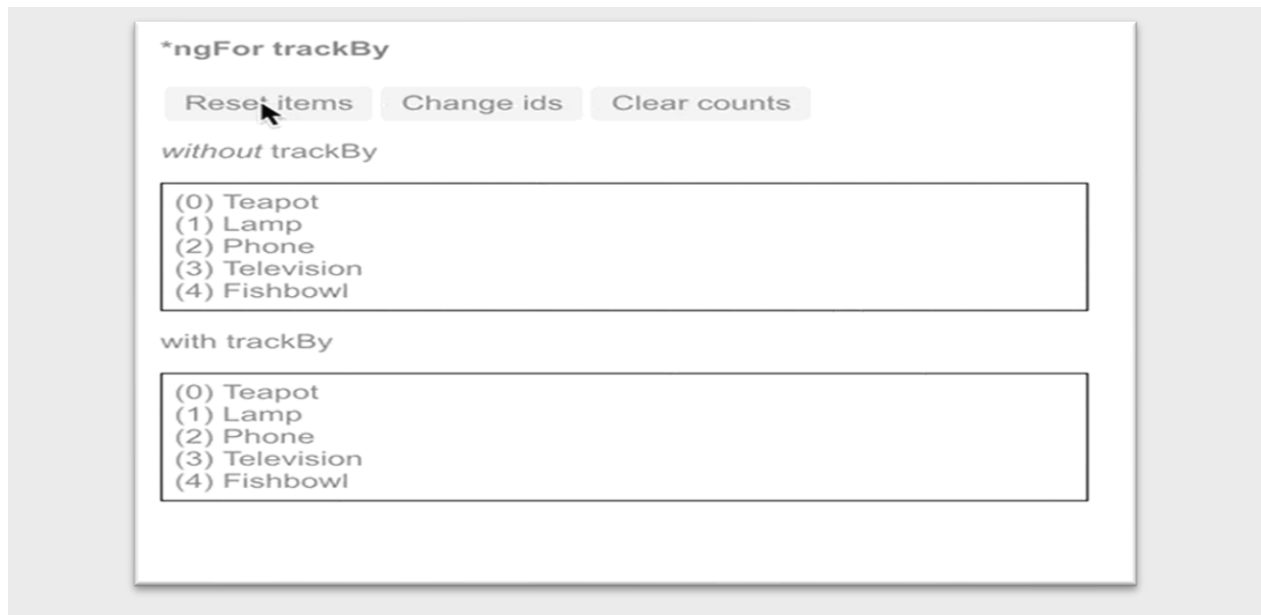
```
content_copy<div *ngFor="let item of items; trackBy: trackByItems">
```

```
{{item.id}}) {{item.name}}
```

```
</div>
```

Change ids creates new items with new item.ids. In the following illustration of the trackBy effect, **Reset items** creates new items with the same item.ids.

- With no trackBy, both buttons trigger complete DOM element replacement.
- With trackBy, only changing the id triggers element replacement.



Hosting a directive without a DOM element

The Angular [<ng-container>](#) is a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM.

Use [<ng-container>](#) when there's no single element to host the directive.

Here's a conditional paragraph using [<ng-container>](#).

src/app/app.component.html (ngif-ngcontainer)

```
content_copy<p>
```

```
I turned the corner
```

```
<ng-container *ngIf="hero">
```

```
and saw {{hero.name}}. I waved
```

```
</ng-container>
```


and continued on my way.

</p>

1. Import the [ngModel](#) directive from [FormsModule](#).
2. Add [FormsModule](#) to the imports section of the relevant Angular module.
3. To conditionally exclude an <option>, wrap the <option> in an [<ng-container>](#).
src/app/app.component.html (select-ngcontainer)

content_copy<div>

Pick your favorite hero

(<label for="showSad"><input id="showSad" type="checkbox" checked
(change)="showSad = !showSad">show sad</label>)

</div>

<select [(ngModel)]="hero">

<ng-container *ngFor="let h of heroes">

<ng-container *ngIf="showSad || h.emotion !== 'sad'">

<option [ngValue]="h">{{ h.name }} ({{ h.emotion }})</option>

</ng-container>

</ng-container>

</select>

Switching cases with [NgSwitch](#)

Like the JavaScript switch statement, [NgSwitch](#) displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

[NgSwitch](#) is a set of three directives:

[NGSWITCH](#) DIRECTIVES

DETAILS

[NgSwitch](#)

An attribute directive that changes the behavior of its companion directives.

NGSWITCH DIRECTIVES

DETAILS

NgSwitchCase

Structural directive that adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.

NgSwitchDefault

Structural directive that adds its element to the DOM when there is no selected NgSwitchCase.

1. On an element, such as a <div>, add [ngSwitch] bound to an expression that returns the switch value, such as feature. Though the feature value in this example is a string, the switch value can be of any type.
2. Bind to *ngSwitchCase and *ngSwitchDefault on the elements for the cases.
src/app/app.component.html

```
content_copy<div [ngSwitch]="currentItem.feature">

  <app-stout-item *ngSwitchCase="stout" [item]="currentItem"></app-
stout-item>

  <app-device-item *ngSwitchCase="slim" [item]="currentItem"></app-
device-item>

  <app-lost-item *ngSwitchCase="vintage" [item]="currentItem"></app-
lost-item>

  <app-best-item *ngSwitchCase="bright" [item]="currentItem"></app-
best-item>

  <!-- ... -->

  <app-unknown-item *ngSwitchDefault [item]="currentItem"></app-
unknown-item>

</div>
```

3. In the parent component, define currentItem, to use it in the [ngSwitch] expression.
src/app/app.component.ts

```
content_copycurrentItem!: Item;
```

4. In each child component, add an item input property which is bound to the currentItem of the parent component. The following two snippets show the parent

component and one of the child components. The other child components are identical to StoutItemComponent.

In each child component, here StoutItemComponent

```
content_copyexport class StoutItemComponent {  
    @Input() item!: Item;  
}
```

Switch directives also work with built-in HTML elements and web components. For example, you could replace the <app-best-item> switch case with a <div> as follows.

src/app/app.component.html

```
content_copy<div *ngSwitchCase="bright"> Are you as bright as  
{{ currentItem.name }}?</div>
```

Date:

PRACTICAL – 7

Aim: : Use Http client module to perform HTTP requests to a server, handle responses, and errors.

Theory: HTTP client - Handle request errors

If the request fails on the server, [HttpClient](#) returns an *error* object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

When an error occurs, you can obtain details of what failed to inform your user. In some cases, you might also automatically [retry the request](#).

An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

- The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error *responses*.
- Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors have status set to 0 and the error property contains a ProgressEvent object, whose type might provide further information.

[HttpClient](#) captures both kinds of errors in its [HttpErrorResponse](#). Inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined ConfigService.

app/config/config.service.ts (handleError)

```
content_copyprivate handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong.
    console.error(
```

```

        `Backend returned code ${error.status}, body was: `, error.error);
    }
    // Return an observable with a user-facing error message.
    return throwError(() => new Error('Something bad happened; please try again
    later.'));
}

```

The handler returns an RxJS ErrorObservable with a user-friendly error message. The following code updates the getConfig() method, using a [pipe](#) to send all observables returned by the [HttpClient.get\(\)](#) call to the error handler.

```

app/config/config.service.ts (getConfig v.3 with error handler)
content_copygetConfig() {
    return this.http.get<Config>(this.configUrl)
        .pipe(
            catchError(this.handleError)
        );
}

```

Retrying a failed request

Sometimes the error is transient and goes away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again can produce a successful result.

The [RxJS library](#) offers several *retry* operators. For example, the `retry()` operator automatically re-subscribes to a failed Observable a specified number of times. *Re-subscribing* to the result of an [HttpClient](#) method call has the effect of reissuing the HTTP request.

The following example shows how to pipe a failed request to the `retry()` operator before passing it to the error handler.

```

app/config/config.service.ts (getConfig with retry)
content_copygetConfig() {
    return this.http.get<Config>(this.configUrl)
        .pipe(
            retry(3), // retry a failed request up to 3 times
            catchError(this.handleError) // then handle the error
        );
}

```

Date:**PRACTICAL – 8**

Aim: : Implement Routing in an Angular app, including redirection, wild card route, relative paths, and routing guards.

Theory: Route**INTERFACE**

A configuration object that defines a single route. A set of routes are collected in a [Routes](#) array to define a [Router](#) configuration. The router attempts to match segments of a given URL against each route, using the configuration options defined in this object.

[See more...](#)

```
interface Route {
  title?: string | Type<Resolve<string>> | ResolveFn<string>
  path?: string
  pathMatch?: 'prefix' | 'full'
  matcher?: UrlMatcher
  component?: Type<any>
  loadComponent?: () => Type<unknown> | Observable<Type<unknown>> |
  DefaultExport<Type<unknown>>> | Promise<Type<unknown>> |
  DefaultExport<Type<unknown>>>
  redirectTo?: string
  outlet?: string
  canActivate?: Array<CanActivateFn | DeprecatedGuard>
  canMatch?: Array<CanMatchFn | DeprecatedGuard>
  canActivateChild?: Array<CanActivateChildFn | DeprecatedGuard>
  canDeactivate?: Array<CanDeactivateFn<any> | DeprecatedGuard>
  canLoad?: Array<CanLoadFn | DeprecatedGuard>
  data?: Data
  resolve?: ResolveData
  children?: Routes
  loadChildren?: LoadChildren
  runGuardsAndResolvers?: RunGuardsAndResolvers
  providers?: Array<Provider | EnvironmentProviders>
}
```

Description

Supports static, parameterized, redirect, and wildcard routes, as well as custom route data and resolve methods.

For detailed usage information, see the [Routing Guide](#).

Further information is available in the [Usage Notes...](#)

Properties	
Property	Description
title?: string Type < Resolve <string>> ResolveFn <string>	Used to define a page title for the route. This can be a static string or an Injectable that implements Resolve . See also: TitleStrategy
path?: string	The path to match against. Cannot be used together with a custom matcher function. A URL string that uses router matching notation. Can be a wild card (**) that matches any URL (see Usage Notes below). Default is "/" (the root path).
pathMatch?: 'prefix' 'full'	The path-matching strategy, one of 'prefix' or 'full'. Default is 'prefix'. By default, the router checks URL elements from the left to see if the URL matches a given path and stops when there is a config match. Importantly there must still be a config match for each segment of the URL. For example, '/team/11/user' matches the prefix 'team/:id' if one of the route's children matches the segment 'user'. That is, the URL '/team/11/user' matches the config {path: 'team/:id', children: [{path: ':user', component: User}]} but does not match when there are no children as in {path: 'team/:id', component: Team}. The path-match strategy 'full' matches against the entire URL. It is important to do this when redirecting empty-path routes. Otherwise, because an empty path is a prefix of any URL, the router would apply the redirect even when navigating to the redirect destination, creating an endless loop.

Property	Description
matcher?: UrlMatcher	A custom URL-matching function. Cannot be used together with path.
component?: Type <any>	The component to instantiate when the path matches. Can be empty if child routes specify components.
loadComponent?: () => Type <unknown> Observable< Type <unknown> DefaultExport < Type <unknown>>> Promise< Type <unknown> DefaultExport < Type <unknown>>>	An object specifying a lazy-loaded component.
redirectTo?: string	A URL to redirect to when the path matches. Absolute if the URL begins with a slash (/), otherwise relative to the path URL. When not present, router does not redirect.
outlet?: string	Name of a RouterOutlet object where the component can be placed when the path matches.
canActivate?: Array< CanActivateFn DeprecatedGuard >	An array of CanActivateFn or DI tokens used to look up CanActivate () handlers, in order to determine if the current user is allowed to activate the component. By default, any user can activate. When using a function rather than DI tokens, the function can call inject to get any required dependencies. This inject call must be done in a synchronous context.
canMatch?: Array< CanMatchFn DeprecatedGuard >	An array of CanMatchFn or DI tokens used to look up CanMatch () handlers, in order to determine if the current user is allowed to

Property	Description
	<p>match the Route. By default, any route can match.</p> <p>When using a function rather than DI tokens, the function can call <code>inject</code> to get any required dependencies. This inject call must be done in a synchronous context.</p>
<code>canActivateChild?:</code> <code>Array<CanActivateChildFn DeprecatedGuard></code>	<p>An array of CanActivateChildFn or DI tokens used to look up CanActivateChild() handlers, in order to determine if the current user is allowed to activate a child of the component. By default, any user can activate a child.</p> <p>When using a function rather than DI tokens, the function can call <code>inject</code> to get any required dependencies. This inject call must be done in a synchronous context.</p>
<code>canDeactivate?:</code> <code>Array<CanDeactivateFn<any> DeprecatedGuard></code>	<p>An array of CanDeactivateFn or DI tokens used to look up CanDeactivate() handlers, in order to determine if the current user is allowed to deactivate the component. By default, any user can deactivate.</p> <p>When using a function rather than DI tokens, the function can call <code>inject</code> to get any required dependencies. This inject call must be done in a synchronous context.</p>
<code>canLoad?: Array<CanLoadFn DeprecatedGuard></code>	<p>DEPRECATED</p> <p>An array of CanLoadFn or DI tokens used to look up CanLoad() handlers, in order to determine if the current user is allowed to load the component. By default, any user can load.</p> <p>When using a function rather than DI tokens, the function can call <code>inject</code> to get any required dependencies. This inject call must be done in a synchronous context.</p> <p>Deprecated Use <code>canMatch</code> instead</p>

Property	Description
data?: Data	Additional developer-defined data provided to the component via ActivatedRoute . By default, no additional data is passed.
resolve?: ResolveData	A map of DI tokens used to look up data resolvers. See Resolve .
children?: Routes	An array of child Route objects that specifies a nested route configuration.
loadChildren?: LoadChildren	An object specifying lazy-loaded child routes.
runGuardsAndResolvers?: RunGuardsAndResolvers	<p>A policy for when to run guards and resolvers on a route.</p> <p>Guards and/or resolvers will always run when a route is activated or deactivated. When a route is unchanged, the default behavior is the same as paramsChange.</p> <p>paramsChange : Rerun the guards and resolvers when path or path param changes. This does not include query parameters. This option is the default.</p> <p>always : Run on every execution.</p> <p>pathParamsChange : Rerun guards and resolvers when the path params change. This does not compare matrix or query parameters.</p> <p>paramsOrQueryParamsChange : Run when path, matrix, or query parameters change.</p> <p>pathParamsOrQueryParamsChange : Rerun guards and resolvers when the path params change or query params have changed. This does not include matrix parameters.</p> <p>See also: RunGuardsAndResolvers</p>
providers?: Array< Provider EnvironmentProviders >	A Provider array to use for this Route and its children.

Property	Description
	The Router will create a new EnvironmentInjector for this Route and use it for this Route and its children. If this route also has a <code>loadChildren</code> function which returns an NgModuleRef , this injector will be used as the parent of the lazy loaded module.

Usage notes

Simple Configuration

The following route specifies that when navigating to, for example, `/team/11/user/bob`, the router creates the 'Team' component with the 'User' child component in it.

```
content_copy[{\n  path: 'team/:id',\n  component: Team,\n  children: [{\n    path: 'user/:name',\n    component: User\n  }]\n}]
```

Multiple Outlets

The following route creates sibling components with multiple outlets. When navigating to `/team/11(aux:chat/jim)`, the router creates the 'Team' component next to the 'Chat' component. The 'Chat' component is placed into the 'aux' outlet.

```
content_copy[{\n  path: 'team/:id',\n  component: Team\n}, {\n  path: 'chat/:user',\n  component: Chat\n  outlet: 'aux'\n}]
```

Wild Cards

The following route uses wild-card notation to specify a component that is always instantiated regardless of where you navigate to.

```
content_copy[{\n  path: 'team/:id',\n  component: Team\n}, {\n  path: 'chat/:user',\n  component: Chat\n  outlet: 'aux'\n}]
```

```

    path: '**',
    component: WildcardComponent
  ]
}

```

Redirects

The following route uses the `redirectTo` property to ignore a segment of a given URL when looking for a child path.

When navigating to `/team/11/legacy/user/jim`, the router changes the URL segment `/team/11/legacy/user/jim` to `/team/11/user/jim`, and then instantiates the `Team` component with the `User` child component in it.

```

content_copy[
  {
    path: 'team/:id',
    component: Team,
    children: [
      {
        path: 'legacy/user/:name',
        redirectTo: 'user/:name'
      }, {
        path: 'user/:name',
        component: User
      }
    ]
  }
]

```

The redirect path can be relative, as shown in this example, or absolute. If we change the `redirectTo` value in the example to the absolute URL segment `/user/:name`, the result URL is also absolute, `/user/jim`.

Empty Path

Empty-path route configurations can be used to instantiate components that do not 'consume' any URL segments.

In the following configuration, when navigating to `/team/11`, the router instantiates the `AllUsers` component.

```

content_copy[
  {
    path: 'team/:id',
    component: Team,
    children: [
      {
        path: '',
        component: AllUsers
      }, {

```

```

    path: 'user/:name',
    component: User
  }}
}
```

Empty-path routes can have children. In the following example, when navigating to `/team/11/user/jim`, the router instantiates the wrapper component with the user component in it.

Note that an empty path route inherits its parent's parameters and data.

```

content_copy[ {
  path: 'team/:id',
  component: Team,
  children: [ {
    path: "",
    component: WrapperCmp,
    children: [ {
      path: 'user/:name',
      component: User
    } ]
  } ]
} ]
```

Matching Strategy

The default path-match strategy is 'prefix', which means that the router checks URL elements from the left to see if the URL matches a specified path. For example, `/team/11/user` matches `team/:id`.

```

content_copy[ {
  path: "",
  pathMatch: 'prefix', //default
  redirectTo: 'main'
}, {
  path: 'main',
  component: Main
} ]
```

You can specify the path-match strategy 'full' to make sure that the path covers the whole unconsumed URL. It is important to do this when redirecting empty-path routes. Otherwise, because an empty path is a prefix of any URL, the router would apply the redirect even when navigating to the redirect destination, creating an endless loop.

In the following example, supplying the 'full' pathMatch strategy ensures that the router applies the redirect if and only if navigating to '/'.

```
content_copy[{
  path: "",
  pathMatch: 'full',
  redirectTo: 'main'
}, {
  path: 'main',
  component: Main
}]
```

Componentless Routes

You can share parameters between sibling components. For example, suppose that two sibling components should go next to each other, and both of them require an ID parameter. You can accomplish this using a route that does not specify a component at the top level.

In the following example, 'MainChild' and 'AuxChild' are siblings. When navigating to 'parent/10/(a//aux:b)', the route instantiates the main child and aux child components next to each other. For this to work, the application component must have the primary and aux outlets defined.

```
content_copy[{
  path: 'parent/:id',
  children: [
    { path: 'a', component: MainChild },
    { path: 'b', component: AuxChild, outlet: 'aux' }
  ]
}]
```

The router merges the parameters, data, and resolve of the componentless parent into the parameters, data, and resolve of the children.

This is especially useful when child components are defined with an empty path string, as in the following example. With this configuration, navigating to '/parent/10' creates the main child and aux components.

```
content_copy[{
  path: 'parent/:id',
  children: [
    { path: "", component: MainChild },
    { path: "", component: AuxChild, outlet: 'aux' }
  ]
}]
```

Lazy Loading

Lazy loading speeds up application load time by splitting the application into multiple bundles and loading them on demand. To use lazy loading, provide the `loadChildren` property in the [Route](#) object, instead of the `children` property.

Given the following example route, the router will lazy load the associated module on demand using the browser native import system.

```
content_copy[{  
  path: 'lazy',  
  loadChildren: () => import('./lazy-route/lazy.module').then(mod => mod.LazyModule),  
}];
```

Date:

PRACTICAL – 9

Aim: Create a basic server using Node.js and Express.js, and handle requests and responses.
Setup a basic Node Js server using ExpressJs, Mongoose

Theory:

Step 1: Setting Up the Project First, make sure you have Node.js and MongoDB installed on your machine. Once that's done, follow these steps:

1. Create a new directory for your project and navigate into it using the command line.
2. Initialize a new Node.js project by running the command: `npm init`. Follow the prompts to set up your project's details.
3. Install the required dependencies.

`npm install express mongoose`

Step 2: Creating the Server Now, let's set up the Node.js server using Express.js.

1. Create a new file named `server.js`.
2. Import the necessary dependencies and set up the Express app.

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

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

Step 3: Connecting to MongoDB Next, we'll connect to the MongoDB database using Mongoose.

1. Create a new file named `db.js`.
2. Import Mongoose and establish a connection to your MongoDB database.

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

const MONGO_URI = 'mongodb://localhost/your-database-name';

mongoose.connect(MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
```



```

    })
    .then(() => {
      console.log('Connected to MongoDB');
    })
    .catch((error) => {
      console.error('Error connecting to MongoDB:', error);
    });

```

In your `server.js` file, require the `db.js` file to establish the MongoDB connection

```
require('./db');
```

Step 4: Creating APIs, Controllers, and Middleware Now, we'll create the APIs, controllers, and middleware for your eCommerce application.

1. Create a new directory named `controllers` in your project directory.
2. Inside the `controllers` directory, create a new file named `productController.js`.
3. In `productController.js`, define your product-related APIs.

```

const express = require('express');
const router = express.Router();
const Product = require('../models/Product');

// Get all products
router.get('/products', async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (error) {
    res.status(500).json({ error: 'Internal server error' });
  }
});

// Get a single product by ID
router.get('/products/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);
    if (!product) {
      return res.status(404).json({ error: 'Product not found' });
    }
    res.json(product);
  } catch (error) {

```

```
res.status(500).json({ error: 'Internal server error' });  
}  
});
```

```
module.exports = router;
```

4. Create a new directory named `models` in your project directory.
5. Inside the `models` directory, create a new file named `Product.js`.
6. In `Product.js`, define your `Product` model using Mongoose.

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

```
const productSchema = new mongoose.Schema({  
  name: { type: String, required: true },  
  price: { type: Number, required: true },  
  // Add more fields as per your product requirements  
});
```

```
module.exports = mongoose.model('Product', productSchema);
```

7. In `server.js`, import the product controller and register it as middleware

```
const productController = require('./controllers/productController');  
app.use('/api', productController);
```

Step 5: Running the Server To start the server and test your APIs, run the following command in your project directory.

node server.js

Now you have a basic setup for a Node.js server with MongoDB integration.

Date:

PRACTICAL – 10

Aim: Connect a Node.js server to a MongoDB database, and perform CRUD (Create, Read, Update, and Delete) operations using Mongoose library

Theory: In Node.js, databases are used to store and retrieve data for web applications. They are an essential part of building dynamic and scalable applications. Node.js provides various modules and packages to work with databases such as MySQL, PostgreSQL, MongoDB, and more. They allow developers to store, query, and manipulate data using various operations such as create, read, update, and delete (CRUD).

They are particularly useful in web applications where data needs to be stored and retrieved quickly and efficiently. For example, an eCommerce website may use a database to store product information, user data, and order details. A social media application may use a database to store user profiles, posts, and comments.

In addition to storing data, databases also provide features such as data indexing, data integrity, and data security. These features ensure that data is stored and accessed correctly and securely.

Hence they are a critical component of web application development in Node.js, and developers must have a good understanding of how to work with databases and how to use them efficiently to build robust applications.

Databases and ORMs

Databases and ORMs (Object Relational Mappers) play a vital role in building web applications using Node.js. As described, a database is a collection of data that is organized in a specific manner to enable easy access, management, and updating of information. In a Node.js application, databases are used to store and retrieve data.

An ORM is a programming technique that maps objects to relational database tables. ORMs provide a higher level of abstraction, making it easier for developers to work with databases by allowing them to interact with the database using objects rather than SQL queries. ORMs help to reduce the amount of code needed to interact with databases and provide an additional layer of security by preventing SQL injection attacks.

Node.js supports both SQL and NoSQL databases, including PostgreSQL, MySQL, MongoDB, and Redis. The choice of database depends on the application's needs and requirements. SQL databases are best suited for applications that require complex queries and transactions, while NoSQL databases are suitable for applications that require flexibility and scalability.

Mongoose is a popular ORM for Node.js that provides a schema-based solution to model the application data. Mongoose simplifies the interaction with MongoDB by allowing developers to define schemas and models for their data. The schema defines the structure of the data and the models represent the collection of data in the database.

Using Mongoose

As described above, ORMs (Object-Relational Mapping) are used to simplify the process of interacting with databases, making it easier to perform CRUD (Create, Read, Update, Delete) operations by using object-oriented programming concepts rather than directly writing SQL queries. With ORMs, developers can work with data in a more intuitive and efficient way, increasing productivity and reducing errors.

Mongoose is a popular ORM for MongoDB in Node.js. It provides a schema-based solution to model application data and provides features like validation, middleware, and more. Here's an example of how to use Mongoose in a Node.js application:

First, install Mongoose using npm:

```
npm install mongoose
```

Then, create a connection to the MongoDB database using Mongoose:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/my_database', { useNewUrlParser: true,
useUnifiedTopology: true })
.then(() => console.log('MongoDB connected'))
.catch((err) => console.log(err));
```

This code connects to a local MongoDB database named `my_database` and logs a message to the console when the connection is successful.

Next, define a Mongoose schema for the data that will be stored in the database:

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

const userSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  }
});
```

```

    },
    password: {
      type: String,
      required: true
    },
    createdAt: {
      type: Date,
      default: Date.now
    }
  });

```

```
module.exports = mongoose.model('User', userSchema);
```

This code defines a schema for a user object that includes a name, email, password, and createdAt property. The `required` property specifies that certain fields are mandatory, and the `unique` property ensures that each email address can only be used once.

Finally, use the defined schema to create, read, update, and delete documents in the database:

```

const User = require('./models/user');

// Create a new user
const newUser = new User({
  name: 'John Doe',
  email: 'johndoe@example.com',
  password: 'password123'
});

newUser.save()
  .then(() => console.log('User created'))
  .catch((err) => console.log(err));

// Read all users
User.find()
  .then((users) => console.log(users))
  .catch((err) => console.log(err));

// Update a user
User.findOneAndUpdate({ name: 'John Doe' }, { name: 'Jane Doe' })
  .then(() => console.log('User updated'))
  .catch((err) => console.log(err));

// Delete a user

```

```
User.deleteOne({ name: 'Jane Doe' })  
  .then(() => console.log('User deleted'))  
  .catch((err) => console.log(err));
```

In this example, the User model is imported from the previously defined schema file. A new user is created using the `save()` method, all users are read using the `find()` method, a user is updated using the `findOneAndUpdate()` method, and a user is deleted using the `deleteOne()` method. These methods are all provided by Mongoose and simplify the process of interacting with the database.

Basic Application Development

To create a Node.js application with Mongoose and perform CRUD operations, we will follow these steps:

4. Initialize a new Node.js project.
5. Install the required dependencies (express, mongoose).
6. Set up the MongoDB database connection.
7. Create a Mongoose schema for our data.
8. Create routes to handle CRUD operations.
9. Test our application.

Step 1: Initialize a new Node.js project

To create a new Node.js project, we will use the npm package manager. Open a command prompt or terminal window and navigate to the folder where you want to create your project.

Type the following command to initialize a new Node.js project:

```
npm init
```

This command will prompt you for information about your project, such as the name, version, and author. You can either enter the information or press enter to accept the default values.

Step 2: Install the required dependencies

To install the required dependencies for our application, we will use npm. In the same command prompt or terminal window, type the following command:

```
npm install express mongoose
```

This command will install the Express.js and Mongoose packages in your project.

Step 3: Set up the MongoDB database connection

In order to use Mongoose with MongoDB, we need to set up a connection to our MongoDB database. We can do this by creating a new file called `db.js` in the root directory of our project, and adding the following code:

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

mongoose.connect('mongodb://localhost/my_database', { useNewUrlParser: true });

const db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  console.log('Database connected successfully');
});
```

This code connects to a MongoDB database called “my_database” running on the local machine. If you have a different database name or URL, you can change the connection string accordingly.

Step 4: Create a Mongoose schema for our data

Now that we have set up our database connection, we can create a Mongoose schema to define the structure of our data. In this example, we will create a simple schema for a “User” model.

Create a new file called `user.js` in the root directory of our project, and add the following code:

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

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

const User = mongoose.model('User', userSchema);

module.exports = User;
```

This code defines a Mongoose schema with three fields: “name”, “email”, and “age”. We then create a Mongoose model called “User” using this schema, and export it for use in other parts of our application.

Step 5: Create routes to handle CRUD operations

Now that we have our database connection and schema set up, we can create routes to handle CRUD (create, read, update, delete) operations on our data.

Create a new file called `routes.js` in the root directory of our project, and add the following code:

```
const express = require('express');
const User = require('./user');

const router = express.Router();

// Create a new user
router.post('/users', async (req, res) => {
  const { name, email, age } = req.body;

  try {
    const user = new User({ name, email, age });
    await user.save();
    res.send(user);
  } catch (error) {
    console.error(error);
    res.status(500).send(error);
  }
});

// Get all users
router.get('/users', async (req, res) => {
  try {
    const users = await User.find({ });
    res.send(users);
  } catch (error) {
    console.error(error);
    res.status(500).send(error);
  }
});

// Update a user
router.put('/users/:id', async (req, res) => {
  const { id } = req.params;
  const { name, email, age } = req.body;

  try {
```



```

    const user = await User.findByIdAndUpdate(id, { name, email, age }, { new: true });
    res.send(user);
  } catch (error) {
    console.error(error);
    res.status(500).send(error);
  }
});

// Delete a user
router.delete('/users/:id', async (req, res) => {
  const { id } = req.params;

  try {
    const user = await User.findByIdAndDelete(id);
    res.send(user);
  } catch (error) {
    console.error(error);
    res.status(500).send(error);
  }
});

```

Step 6: Test our application

Now that we have created all the necessary routes, we can test our application. Create a new file called `index.js` in the root directory of our project, and add the following code:

```

const express = require('express');
const bodyParser = require('body-parser');
const db = require('./db');
const routes = require('./routes');

const app = express();

app.use(bodyParser.json());

app.use('/', routes);

app.listen(3000, () => {
  console.log('Server started on port 3000');
});

```

This code sets up an Express.js server, adds the middleware for parsing JSON request bodies, and sets up the routes we created earlier. It also starts the server and logs a message to the console to indicate that it is running.

To test our application, we can use a tool like Postman or curl to send requests to the server. For example, to create a new user, we can send a POST request to `http://localhost:3000/users` with a JSON body containing the user's name, email, and age. To get all users, we can send a GET request to <http://localhost:3000/users>.