



Building Scalable Web Apps with **Node.js** and **Express**

Design and Develop a Robust, Scalable,
High-Performance Web Application
Using Node.js, Express.js, TypeScript,
and Redis



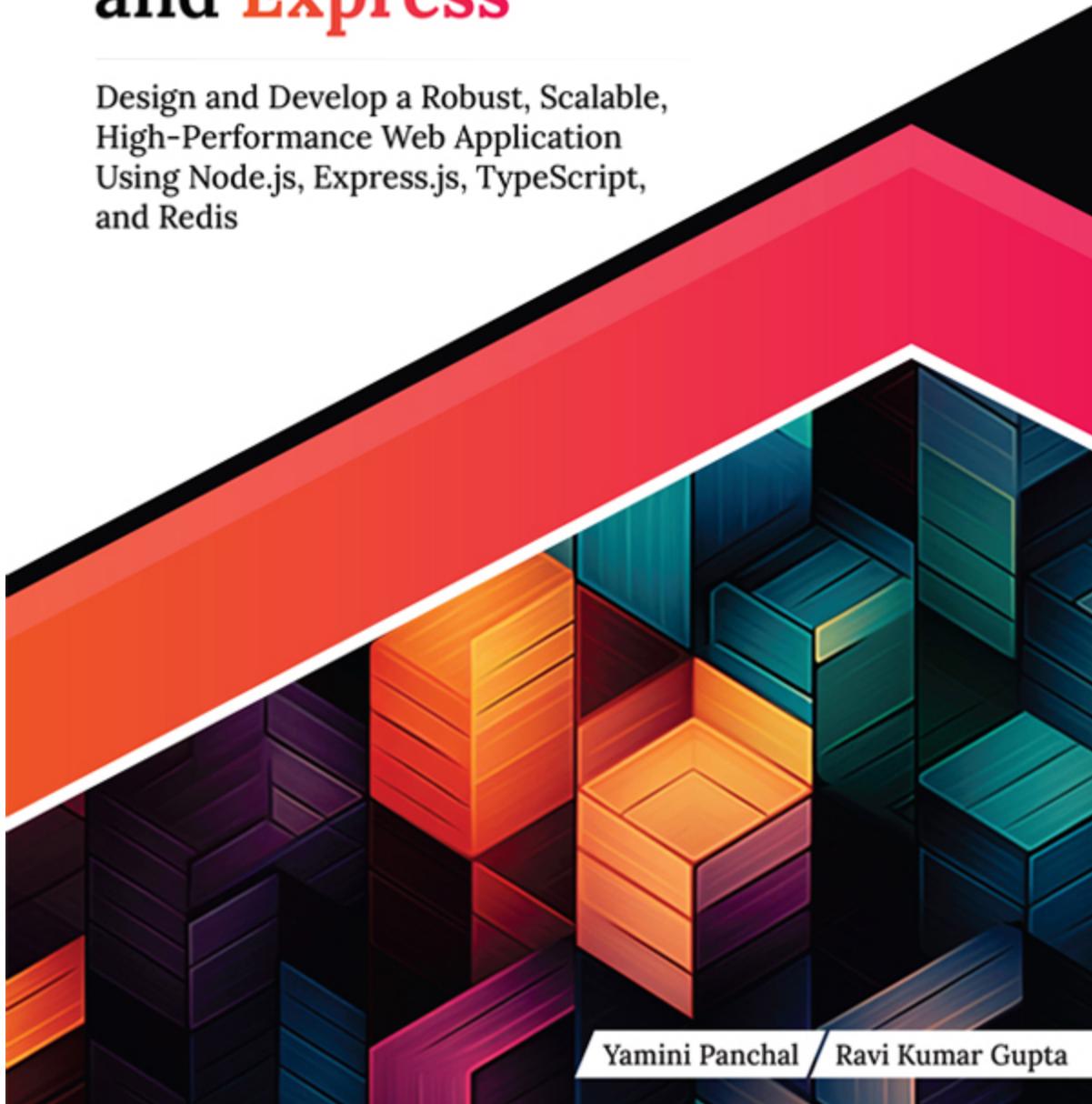
Yamini Panchal

Ravi Kumar Gupta



Building Scalable Web Apps with **Node.js** and **Express**

Design and Develop a Robust, Scalable,
High-Performance Web Application
Using Node.js, Express.js, TypeScript,
and Redis



Yamini Panchal

Ravi Kumar Gupta

Building Scalable Web Apps with Node.js and Express

Design and Develop a Robust, Scalable, High-Performance Web Application Using Node.js, Express.js, TypeScript, and Redis

Yamini Panchal
Ravi Kumar Gupta



www.orangeava.com

[OceanofPDF.com](http://www.oceanofpdf.com)

Copyright © 2024 Orange Education Pvt Ltd, AVA™

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First published: June 2024

Published by: Orange Education Pvt Ltd, AVA™

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN: 978-81-97223-81-5

www.orangeava.com

OceanofPDF.com

About the Authors

Yamini Panchal holds a Bachelor of Engineering in Computer Science from Gujarat Technological University and has over 8 years of experience in the IT industry. She has worked on diverse domain-based web development applications, including IoT, Telecommunication, Healthcare, and Cloud Services.

Currently a Technical Lead at Azilen Technologies, she specializes in open-source development with a focus on Node.js, constructing server-side applications and APIs. She has expertise in both SQL and NoSQL databases.

Yamini has contributed to robotics-based applications using Node.js, TypeScript, React.js, and AWS services, as well as developed a WebRTC-based application for audio-video conferencing. She excels in building scalable backend services with Node.js and leveraging AWS services in microservice architectures. She enjoys embracing new technologies, reading, writing, and fostering innovation.

Ravi Kumar Gupta is an accomplished author and open-source software evangelist with a strong technology background. He holds an MS in Software Systems from BITS Pilani and a B.Tech from LNMIIT, Jaipur.

Currently, he works as a Solution Architect at Orbiwise and contributes to the NoiseApp Team. Ravi excels in coding with Python, TypeScript, Node.js, and Java, enhancing OrbiWAN's performance and efficiency.

Previously, he served as a Solution Architect at Azilen and a lead consultant at CIGNEX Datamatics. At TCS, he was a core member of the open-source group, working on Liferay and other UI technologies. Throughout his career, he has built enterprise solutions using the latest technologies and open-source tools.

Ravi enjoys writing, learning, and discussing new technologies. His interest in search engines began with a college project on a crawler. He has co-authored books on Test-Driven JavaScript Development and Mastering Elastic Stack and writes for his blog at TechD of Computer World

(<http://techdc.blogspot.in>). He has also been a Liferay trainer at TCS and CIGNEX and reviewed Learning Bootstrap for Packt Publishing.

OceanofPDF.com

About the Technical Reviewer

Bhargav Bachina is a distinguished figure in the IT industry, boasting a remarkable 12-year journey marked by innovation and leadership in software architecture. His career is a testament to his profound expertise across technological stacks, from front-end and back-end development to the complexities of cloud computing. Bhargav's proficiency in Java, JavaScript, Python, and Node.js has made him a versatile and adept navigator in the world of software development.

As a visionary software architect, Bhargav is renowned for creating comprehensive end-to-end solutions. His commitment to excellence and passion for technology have led to the development of cutting-edge web and mobile platforms, establishing him as a thought leader in the field. Currently the CTO of an educational startup, Bhargav is about to launch an innovative project, marking another milestone in his career. His work as a fractional CTO has made him a sought-after mentor in the startup community, with many emerging startups seeking his expertise on LinkedIn.

Beyond his technical and leadership roles, Bhargav is a prolific writer. His journey as a writer began five years ago on Medium, where he has penned over 700 articles. These writings have reached an audience of over 8 million globally, resonating deeply within the tech community. His articles have been pivotal in guiding and enlightening many, earning him a significant following of 22k on Medium. His influence extends to LinkedIn, where his contributions are frequently lauded for their impact.

Bhargav's commitment to sharing knowledge is further evidenced by his active presence on GitHub. With around 431 repositories, he has become a resource for many in the tech community, evidenced by the regular stars and forks his repositories receive. Bhargav Bachina's journey is more than a career narrative; it's a source of inspiration and a roadmap for aspiring IT professionals worldwide.

To learn more about Bhargav, please visit the following sites:

Medium: <https://medium.com/@bhargavbachina>

GitHub: <https://github.com/bbachi>

YouTube: <https://www.youtube.com/@bachinalabs>

OceanofPDF.com

Acknowledgements

My deepest gratitude to my wife, Kriti. Despite being busy with our two little ones, she continually encouraged and motivated me throughout the writing process. I am forever grateful for her boundless love and patience. A ton of thanks to my co-author, Yamini, for her tremendous support. Finally, my sincere thanks to the Orange Team, editors, and reviewers, whose patience, cooperation, and expertise were vital for this project. My heartfelt thanks go out to everyone involved in this project. Much appreciated!

- **Ravi Kumar Gupta**

I wish to convey my heartfelt gratitude to my mother, who has been my pillar of strength and my greatest source of inspiration, shaping me into the person I am today. Additionally, I extend my profound appreciation to my co-author, Ravi, for extending this invaluable opportunity to me. I owe a debt of gratitude beyond words to him. His mentorship has been the cornerstone of this project, without which it would have remained a distant dream. Ravi's guidance and encouragement have propelled me forward, and I am immensely grateful for his unwavering faith in my abilities. I would also like to extend my appreciation to the Orange AVA Team, editors, reviewers, and the publishing team for their expertise, patience, guidance, and dedication in helping bring this book to life. Your insights and suggestions have greatly enriched the final product, and I am grateful for your professionalism and support.

Last but not least, I want to thank the readers who have embraced my work. Your passion and enthusiasm inspire me to continue writing, and I am deeply grateful for your support. Thank you all who directly or indirectly supported me for being part of this incredible journey.

- **Yamini Panchal**

Preface

From cumbersome setups for writing APIs to launching projects in just five minutes with Node.js and Express.js, developers have come a long way to witness revolutionary simplification in web development. If you are new to Node.js, this book offers a clear pathway to master backend development using Node.js, TypeScript, and Express.js. If you are already familiar with JavaScript or Node.js and aim to build scalable and efficient APIs, this book will elevate your skills, equipping you with the techniques needed for modern backend architecture.

The book begins with the fundamentals of Node.js, TypeScript, and Express.js, offering guidance on setting up your development environment and crafting your first API. It focuses on creating backend APIs for a Project Management System, using this consistent example to illustrate database design and module-wise implementation.

The book also delves into advanced topics such as caching, message queues, testing, and deployment using AWS cloud services, providing a comprehensive guide to modern backend development practices.

What the book covers

The book is divided into 11 chapters. The first three chapters introduce you to Node.js, TypeScript, and Express.js. The middle chapters begin with developing API for all modules, followed by chapters covering caching, testing, and deployment. The final chapter recaps what we learned throughout the book. Details about what each chapter covers are as follows:

Chapter 1. Introduction to Node.js: This chapter starts the journey into Node.js, covering the basics and the pros and cons. It guides readers through setting up Node.js on various operating systems, introduces event-driven programming and architectures, and concludes with creating a basic HTTP(s) server.

Chapter 2. Introduction to TypeScript: This chapter provides a comprehensive overview of TypeScript, discussing its advantages and potential pitfalls. It includes steps for installing necessary packages and

explores object-oriented programming concepts while developing a basic application with TypeScript.

Chapter 3. Overview of Express.js: This chapter introduces Express.js, discussing its benefits and limitations. It starts with defining what the Express.js framework is and moves into building a basic application with an API. It concludes with discussing the core features and best practices of Express.js.

Chapter 4. Planning the App: This chapter lays the foundation of the primary example covered throughout the book - the Project Management System. This chapter outlines the roadmap and helps in setting up the project structure. It focuses on designing the database entities and routing, which are crucial in planning the application's architecture and flow.

Chapter 5. REST API for User Module: This chapter completes the APIs for User module. The essential functions such as user signup, login, password recovery along with CRUD operations are covered here. This chapter delves into critical aspects of authentication and authorization, explaining how roles and rights are implemented to secure the application.

Chapter 6. REST API for Project and Task Modules: This chapter continues building APIs for Project and Task modules. It includes the creation and management of projects and tasks, ensuring they maintain relationships with the user entity. Functions such as retrieve, update, list, and delete for both projects and tasks are detailed, enhancing the overall functionalities of the application.

Chapter 7. API Caching: This chapter delves into the basics of caching and the use of Redis as a caching Server. It guides you through the setup of the Redis server, discussing the pros and cons of using Redis and the caching mechanism in general. The chapter also builds a Caching utility to enhance the application's performance.

Chapter 8. Notification Module: This chapter explores the necessity of a Notification module and showcases its implementation using Message Queues. It starts with a basic queue implementation using Redis. The chapter then transitions to a more standardized approach by introducing the Bull package, a robust queueing system.

Chapter 9. Testing API: This chapter guides readers with an overview of testing and how Mocha framework can be used for API testing. It covers the

basics of writing test cases, setting up the testing environment, and executing tests using Mocha along with Chai, which is an assertion library.

Chapter 10. Building and Deploying Application: This chapter covers the crucial steps required to build a production-ready application and deployment using AWS Cloud Services. The chapter also delves into the topic of Code Obfuscation while discussing common techniques used in code obfuscation.

Chapter 11. The Journey Ahead: This final chapter recaps the key lessons from the book and outlines the directions for future development. The *Next Steps* section of the chapter suggests areas of expansion such as front-end development, security measures, reporting, monitoring, integrating machine learning, adopting containerized deployment, and many more. The chapter also discusses the impact and potential of GenAI and LLMs such as ChatGPT.

OceanofPDF.com

Downloading the code bundles and colored images

Please follow the link or scan the QR code to download the
Code Bundles and Images of the book:

<https://github.com/ava-orange-education/Building-Scalable-Web-Apps-with-Node.js-and-Express>



The code bundles and images of the book are also hosted on
<https://rebrand.ly/db7c4e>



In case there's an update to the code, it will be updated on the existing GitHub repository.

Errata

We take immense pride in our work at **Orange Education Pvt Ltd** and follow best practices to ensure the accuracy of our content to provide an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@orangeava.com

Your support, suggestions, and feedback are highly appreciated.

OceanofPDF.com

DID YOU KNOW

Did you know that Orange Education Pvt Ltd offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.orangeava.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: info@orangeava.com for more details.

At www.orangeava.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on AVA™ Books and eBooks.

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at info@orangeava.com with a link to the material.

ARE YOU INTERESTED IN AUTHORING WITH US?

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please write to us at business@orangeava.com. We are on a journey to help developers and tech professionals to gain insights on the present technological advancements and innovations happening across the globe and build a community that believes Knowledge is best acquired by sharing and learning with others. Please reach out to us to learn what our audience demands and how you can be part of this educational reform. We also welcome ideas from tech experts and help them build learning and development content for their domains.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers

can then see and use your unbiased opinion to make purchase decisions. We at Orange Education would love to know what you think about our products, and our authors can learn from your feedback. Thank you!

For more information about Orange Education, please visit www.orangeava.com.

OceanofPDF.com

Table of Contents

1. Introduction to Node.js

Introduction

Structure

Defining Node.js

Applications of Node.js

Single-Page Applications

Real-time Applications

IoT Devices Applications

Data Streaming Application

Pros of Node.js

Cons of Node.js

Installing Node.js

Installing Node For Linux/Ubuntu

Installing Node.js for Windows

Installing Node.js for mac

Event-Driven Mechanism

Example of Event Programming

Example of Synchronous Code

Example of Asynchronous code

Types of Node.js Architectures

Monolithic Architecture

Microservice Architecture

Serverless Architecture

Writing an HTTP Server

Making it HTTPS

Using Cluster Module

Programming Without Cluster Module Example

Programming With Cluster Module Example

Conclusion

Multiple Choice Questions

Answers

Further Reading

2. Introduction to TypeScript

Introduction

Structure

Overview of TypeScript

Advantages of TypeScript

Pitfalls of TypeScript

Installing TypeScript

Global Installation

Project-wise Installation

Building a Basic Application with TypeScript

OOP Concepts in TS

Data Types

Primitive Types

Non-Primitive Types

Class

Inheritance

Access Modifiers

Interface

Abstraction

Encapsulation

Polymorphism

ECMAScript Features

Arrow Functions

Using this Keyword

Using the new Keyword

Blocking Scopes

The let Keyword

The const Keyword

Template Literals

Classes

Promises

Destructuring

Object Destructuring

Array Destructuring

Default Parameters

Modules

Enhanced Object Literals

EsLint

Installing Eslint

Configuring Eslint

Running Eslint

Conclusion

Multiple Choice Questions

Answers

Further Reading

3. Overview of Express.js

Introduction

Structure

Defining Express.js

Advantages of Express.js

Limitations of Express.js

Express.js Installation and Creating a Basic Application

Core Features of Express.js

REST APIs

REST Principles

Building REST API

Routing

Route Methods

Route Paths

Route Parameters

Route Handlers

Middleware

Error Handling

Built-in Error Handling

Custom Error Handling

Async Error Handling

Static File Serving

Templating Engines

Security and Performance Best Practices

Conclusion

Multiple Choice Questions

Answers

Further Readings

4. Planning the App

Introduction

Structure

Overview of the Application

Roadmap

Scope

Defining the Modules

User Module

Project Module

Task Module

Comment Module

Database Design

User Schema Table

Role Schema Table

Project Schema Table

Task Schema Table

Comment Schema Table

Setting Up the Project Structure

Init the project

Installation of Project Dependency

Project Directory Structure

Create Express Server

With Cluster for a Large Project

Connecting the Database

Database Models (Entities)

Role Entity

User Entity

Project Entity

Task Entity

Comment Entity

Routes

Role Routes

User Routes

Project Routes

Task Routes

Comment Routes

Conclusion

[Multiple Choice Questions](#)

[Answers](#)

[Further Reading](#)

[5. REST API for User Module](#)

[Introduction](#)

[Structure](#)

[Base Controller](#)

[Base Service](#)

[Role Management](#)

[Role Service](#)

[Input Validation](#)

[Add Role](#)

[GetAll Roles](#)

[GetOne Role](#)

[Update Role](#)

[Delete Role](#)

[Add Default Role from System](#)

[User Management](#)

[User Service](#)

[Input Validation](#)

[User Onboarding](#)

[Add Default User from System](#)

[User Sign-In](#)

[Authentication](#)

[Authorization](#)

[GetAll Users](#)

[GetOne User](#)

[Update User](#)

[Delete User](#)

[Password Management](#)

[Change Own Password](#)

[Recover Password](#)

[Reset Password](#)

[Conclusion](#)

[6. REST API for Project and Task Modules](#)

[Introduction](#)
[Structure](#)
[Project Management](#)

[Project Service](#)
[Input Validation](#)
[Add Project](#)
[GetAll Project](#)
[Search Project by Name](#)
[GetOne Project](#)
[Update Project](#)
[Delete Project](#)
[Project Util](#)

[Task Management](#)
[Task Service](#)
[Input Validation](#)
[Add Task](#)
[GetAll Task](#)
[Search Task](#)
[GetOne Task](#)
[Update Task](#)
[Delete Task](#)
[Upload Supported Files](#)

[Conclusion](#)
[Further Reading](#)

7. API Caching

[Introduction](#)
[Structure](#)
[Understanding Caching](#)
[Introduction to Redis](#)
[Setting Up Redis Server](#)

[Installing Redis Server on Mac OS](#)
[Installing Redis Server on Ubuntu / Linux](#)
[Installing Redis Server on Rocky \(RHEL-based\)](#)

[Pros and Cons of Caching](#)

[Pros of Caching](#)
[Cons of Caching](#)

[Using Redis for Caching](#)
[Updating Project Dependencies](#)
[Cache Utility](#)
[Caching Entities](#)
[Building Cache at Startup](#)
[Consideration when Using Redis](#)
[Conclusion](#)
[Multiple Choice Questions](#)
[Answers](#)
[Further Readings](#)

8. Notification Module

[Introduction](#)
[Structure](#)
[Understanding Notification Module](#)
[Implementing Queue](#)
[Using Redis for Queue](#)
[Handling Failures](#)
[Notifying About the New Task](#)
[Considerations while Implementing Queues](#)
[Conclusion](#)
[Multiple Choice Questions](#)
[Answers](#)
[Further Readings](#)

9. Testing API

[Introduction](#)
[Structure](#)
[Overview of Unit Testing](#)
[Mocha Framework](#)
[Installing Mocha and Chai](#)
[Defining a Test Case](#)
[Configuring the Application](#)
[Hooks](#)
[Verifying APIs through Test cases](#)
[Login Test](#)
[List of User Test](#)

[Add User Test](#)
[Delete User Test](#)
[Mocking Database Connection](#)
[Conclusion](#)

10. Building and Deploying Application

[Introduction](#)
[Structure](#)
[Code Obfuscation](#)
[Common Techniques](#)
[Installing Required Dependencies](#)
[Creating an Obfuscation Script](#)
[Downside of Code Obfuscation](#)
[Building the Application](#)
[Deploying the Application](#)
[AWS Server Setup](#)
[Signing in to the AWS Management Console](#)
[Navigating to EC2](#)
[Choosing an Amazon Machine Image \(AMI\)](#)
[Key Pair Generation](#)
[Network Settings](#)
[Configuring Storage](#)
[Launching an Instance](#)
[Connecting the Server](#)
[Deploying Code on Server](#)
[Other Methods For Deployment](#)
[Conclusion](#)
[Multiple Choice Questions](#)
[Answers](#)
[Further Reading](#)

11. The Journey Ahead

[Introduction](#)
[Structure](#)
[The Story So far](#)
[Next Steps](#)
[FrontEnd](#)

[Reporting](#)

[Applying Machine Learning](#)

[Server Monitoring](#)

[Security Features](#)

[SSL/TLS Encryption](#)

[Social Media Login](#)

[Two-Factor Authentication](#)

[LDAP Integration](#)

[Container-Based Deployments](#)

[Swagger UI](#)

[Staying Ahead](#)

[Further Reading](#)

[Advanced Node.js Development](#)

[Full Stack Development](#)

[Test-Driven Development](#)

[Performance Tuning](#)

[Other Topics](#)

[Final Thoughts](#)

[Index](#)

[OceanofPDF.com](#)

CHAPTER 1

Introduction to Node.js

Introduction

The popularity of Node.js is booming day by day in the IT market worldwide. Through this book, any JavaScript developer can easily learn about Node.js from basic to advanced levels. This chapter talks about the Node.js basics and architecture. We will also learn how to write a simple Node.js program.

Structure

In this chapter, we will be covering the following topics:

- Defining Node.Js and Where It is Used
- Pros and Cons of Node.Js
- Installing Node.Js on Various Platforms
- Understanding Event-Driven Programming
- Node.js Architectures
- Writing HTTP and HTTPS Server
- Using the Cluster Module

Defining Node.Js

When Ryan Dahl demonstrated his remarkable work, *Node.js at JSConf 2009*, it was the beginning of a new era. He stated that the concurrency was achieved by threads in most of the top languages and that using threads has certain problems as context switching between threads is costly. Using the event loop, he showed that Node.js achieves way higher concurrency than any of the existing languages. People at the conference welcomed the idea and applauded Dahl. Thus began the much-awaited shift in the programming world.

Node.js is an open source cross-platform JavaScript runtime environment. It states that anyone can use it free of cost on any operating system such as Windows, Linux, Unix, Mac, and more. JavaScript is the foundation of Node.js. The code of any node.js application is written in JavaScript. This code runs on

Google Chrome's V8 JavaScript engine which converts source code to machine code directly without interpreting and then it gets executed without the need for a web browser. Node.js provides the necessary environment for the code to run.

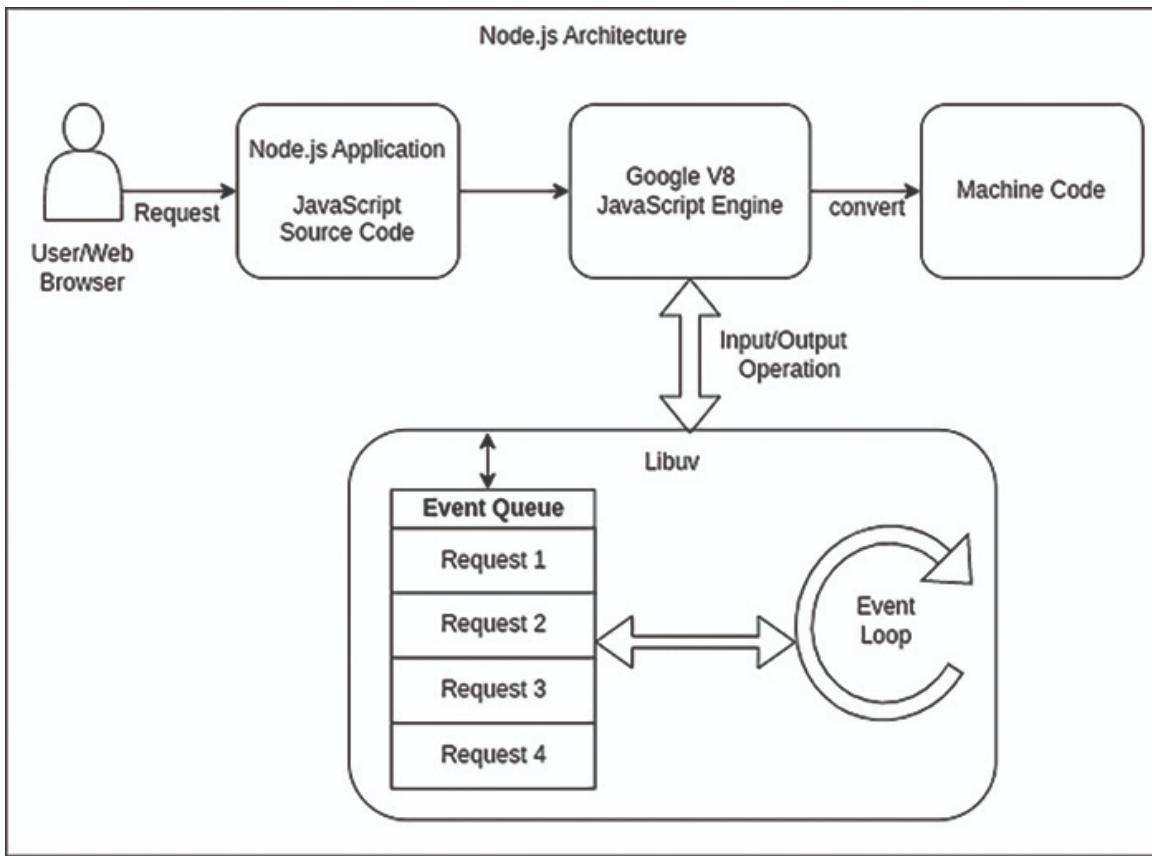


Figure 1.1: Node.js Architecture

[Figure 1.1](#) shows a high level architecture of Node.js. Even though Node.js is single threaded, it still handles lots of concurrent requests at a time through a mechanism of asynchronous non-blocking I/O operations which provides hidden threads from the **libuv** library that executes itself as multi-threaded.

Due to non-blocking I/O operation, Node.js is fast compared to other languages because the request does not wait for its response and parallelly executes another request. All requests are first sent to the event queue, processed in the event loop, and then sent back to the V8 engine through the queue as displayed in Node.js architecture. More about the event loop is elaborated in the section, “*Event-driven Mechanism*” later in this chapter.

Node.js is not only written for backend-side server programming applications but also developed as node modules and used on the client side, which is beneficial for developers as the same language is used on both sides.

Applications of Node.js

The usage of Node.js has been growing at a fast pace in IT Industries due to its features and different types of application. Here are a few examples of different types of applications related to Node.js.

Single-Page Applications

Currently, there are a lot of organizations and enterprises that provide complex and real time solutions to their clients by developing Node.js applications as server side applications through single-page applications. For example, Gmail, Twitter, Facebook, Trello, and many more applications are developed as SPA (Single-Page Applications). A single-page application communicates with the user's actions by rewriting data on a single web page instead of reloading the whole web page.

Real-time Applications

Node.js is an ideal model for real-time applications because it gives responses to numerous requests at the same time. If there are a large number of users that need real-time response, Node.js is a better choice. You can use Node.js with WebSockets for continuous connection and to provide a faster response time. Applications such as audio, video, chat, multiplayer games, and stock trading are developed in this manner.

IoT Devices Applications

Due to its faster response time and ability to handle large numbers of requests concurrently, Node.js is a good choice for Internet of Things (IoT) apps where devices or sensors are connected to the internet and send huge amounts of data continuously. IoT use cases such as fire detection, noise pollution measure, fitness tracker, health monitoring are many such applications where Node.js is playing a big role.

Data Streaming Application

Node.js allows working with an abstract interface as a stream for data streaming. Large media files are divided into small chunks and sent as buffers. These buffers are transformed into meaningful data. Netflix-like streaming services use Node.js where data is transferred in chunks instead of whole large streams that reduce loading and delay while streaming happens.

The uses of Node.js are not limited to the preceding types of applications but also many more types of applications developed in Node.js such as making proxy or signaling servers, monitoring data-based applications, and more.

Pros of Node.js

Node.js is a very powerful runtime environment for JavaScript. It allows developers to build high performance and scalable applications. Some of the key advantages that Node.js offers are as follows:

- **Cross platform:** Node.js comes up with cross-platform functionality so the application can be easily developed on any OS and deployed on any platform. Key platforms supported by Node.js are Windows, Mac(Intel), Mac(ARM), and Linux (Intel/ARM). Probably every major platform is supported.
- **High performance:** Node.js offers high performance due to asynchronous non-blocking I/O operations which execute requests in parallel without waiting for the response of any other request.
- **Easy to scale:** Node.js is itself single threaded but in heavy traffic, it handles a lot of requests at the same time to scale up with the “*cluster*” module which creates child processes and reduces the load on the application.
- **Caching:** Node.js allows storing data in temporary memory that is not updated frequently, which reduces loading time and saves database transactions. This is called caching.
- **Huge community:** Ever since Node.js appeared, its community size has been increasing day by day. The language used for programming is JavaScript which has been the backbone of the internet and almost every front-end developer was already familiar with it. This made learning easy and made the community grow rapidly. There are more than 1.3 million open-source libraries available for use.

There are many other advantages as well such as cost-effectiveness, ease of learning, and adaptability. Node.js is a technique that has really made a difference.

Cons of Node.js

There are some disadvantages of Node.js too. However many of these can be overcome using best practices.

- **Single threaded:** Node.js is single threaded, which is an advantage as well as a pitfall because it is unable to process heavy CPU oriented computation quickly. When requests which need more CPU for processing come in the event loop, they keep piling up because until it finishes one request, it will not pick other requests from the event queue. However, this happens only when there are only CPU centric tasks. If a request needs some IO to happen, another request will be picked while a request waits for IO to complete. CPU centric tasks make the performance low and delay the response. For example, for searching algorithms and mathematical calculations where complexity is high at that time, Node.js is not recommended due to poor performance.
- **Callback hell:** Asynchronous programming in Node.js can be challenging for some developers, especially when using callbacks. Callback hell is a situation when callback functions are nested. This can make code difficult to read and maintain. However to avoid this, developers can use promises, async-await, or libraries such as Async.js.
- **Library compatibility:** Although there are more than a million libraries available yet those being open sourced by individual developers might not be up-to-date to the latest versions. This sometimes makes it difficult to use those libs in projects.

Installing Node.js

Now when we have a high level understanding of what Node.js is and what it offers, let us jump to the setup. There are different ways to download and install Node.js in your system but here, we give the easiest and best way. Download the LTS (Long Term Support) Version of Node.js from its official site (<https://nodejs.org/en/download>) based on your operating system. On the site, there will be LTS and Current Version, choose the LTS version because it is stable and recommended for complex projects.

At the time of writing, Node.js version 20 is ACTIVE.

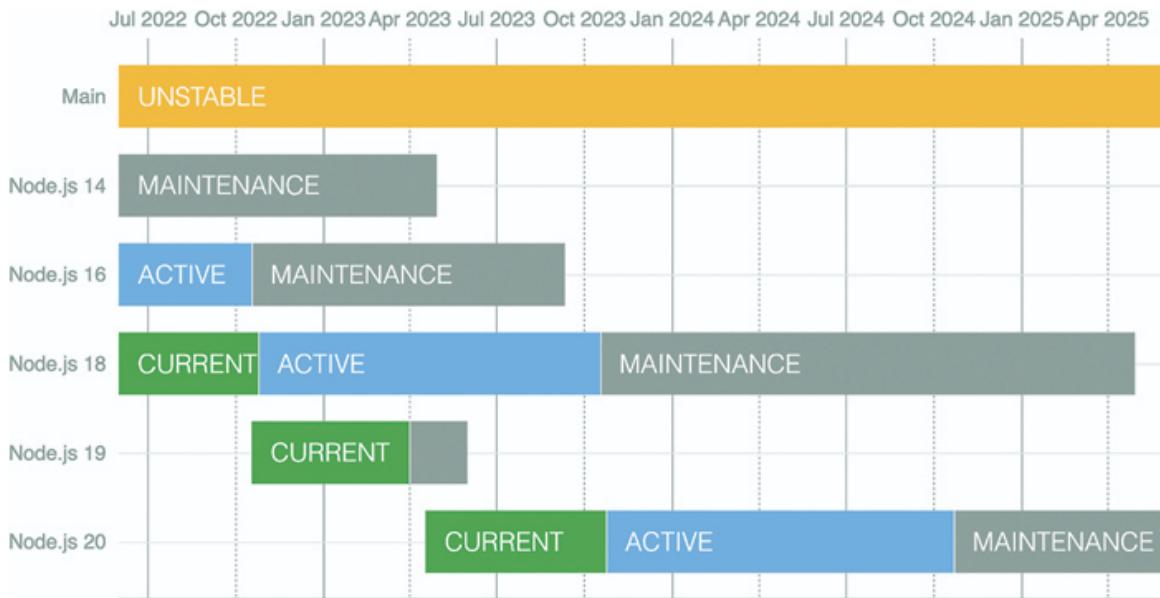


Figure 1.2: Node.js Versions

The preceding up-to-date release schedule can be seen on Node.js GitHub page—<https://github.com/nodejs/release#release-schedule>.

Installing Node For Linux/Ubuntu

NPM (Node Package Manager) is the default package manager for node.js and also a library of JavaScript software packages. It is open source so that developers can install other modules in their project via npm free of cost.

Node Version Manager (NVM) is a shell script that manages multiple node versions and uses it on different projects.

We can highlight the importance of installing Node.js through **NVM** with real use cases.

Using NVM, you can easily manage multiple Node.js versions on the same machine. Here is how it helps:

- **Version Management:** NVM allows you to install multiple versions of Node.js on your system. This means you can switch between different versions seamlessly based on the requirements of your projects.
- **Isolated Environments:** Each Node.js version installed through NVM is isolated from others. This ensures that changes made to one version won't affect the others. It is particularly useful when you are working on projects with conflicting dependencies or when you need to maintain compatibility with older versions.

- **Flexibility:** With NVM, you have the flexibility to switch between Node.js versions effortlessly. This allows you to test your applications across different versions, ensuring compatibility and stability.
- **Project-specific Versioning:** NVM allows you to specify the Node.js version required for a particular project. This ensures that each project uses the correct version of Node.js without interfering with others.
- **Easy Updates:** NVM simplifies the process of updating Node.js to the latest version. You can easily upgrade or downgrade Node.js versions with a single command, ensuring that your development environment stays up-to-date.

NVM for managing Node.js versions provides a streamlined and efficient workflow, enhancing productivity and reducing potential conflicts between projects. It is an essential tool for developers working on multiple Node.js projects simultaneously.

Based on the advantages of NVM, we will install Node.js through NVM on different platforms.

Let us first install NVM and then install Node.js by NVM. Open terminal/console or cmd and follow three given steps:

1. Update your system with the latest versions of packages.
`$ sudo apt-get update`
2. Download and install NVM using this command:
`$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash`
3. Before running the preceding command, make sure curl is installed on the system. If it is not installed then run the following commands to install and verify:
`$ sudo apt install curl $ curl -version`
4. Verify NVM version:
`$ nvm -version`
5. Install Node.js as follows:
`$ nvm install node`
This command will install the latest stable version of node.
6. To install Node.js with LTS version, use this command:
`$ nvm install -lts`

7. If anyone wants specific version of node, then add specific version at end of NVM and install as follows:

```
$ nvm install 18.15.0
```

Or

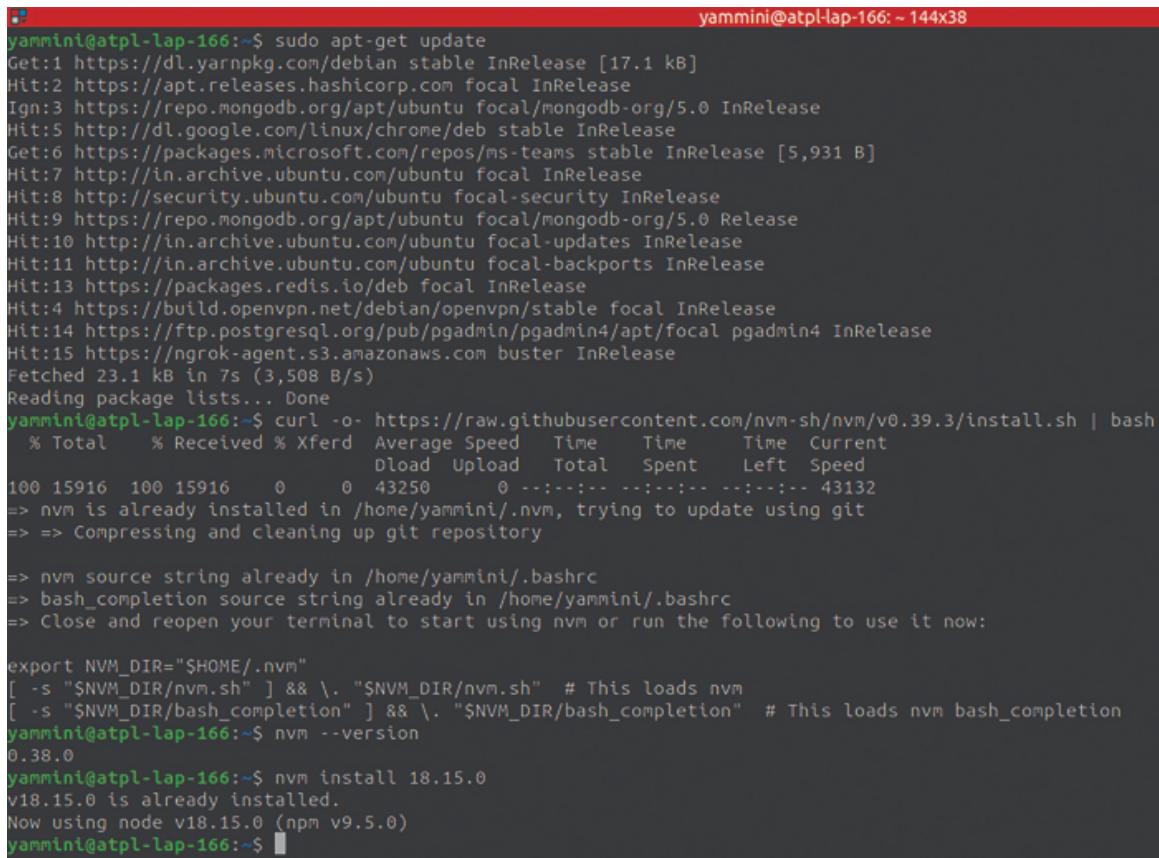
```
$ nvm install 18.x
```

In the preceding commands, 18.15.0 is a specific version of node.js and 18.x means it will consider the highest version of 18 above and below 19.

8. Verify Node.js version as follows:

```
$ node -version
```

After successfully executing the above steps, you can expect the following output to be displayed in the command prompt for your reference.



The screenshot shows a terminal window with a red header bar. The terminal prompt is `yammini@atpl-lap-166:~`. The window title is `yammini@atpl-lap-166: ~ 144x38`. The terminal content shows the following steps:

```
yammini@atpl-lap-166:~$ sudo apt-get update
Get:1 https://dl.yarnpkg.com/debian stable InRelease [17.1 kB]
Hit:2 https://apt.releases.hashicorp.com focal InRelease
Ign:3 https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0 InRelease
Hit:5 http://dl.google.com/linux/chrome/deb stable InRelease
Get:6 https://packages.microsoft.com/repos/ms-teams stable InRelease [5,931 B]
Hit:7 http://in.archive.ubuntu.com/ubuntu focal InRelease
Hit:8 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:9 https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0 Release
Hit:10 http://in.archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:11 http://in.archive.ubuntu.com/ubuntu focal-backports InRelease
Hit:13 https://packages.redis.io/deb focal InRelease
Hit:4 https://build.openvpn.net/debian/openvpn/stable focal InRelease
Hit:14 https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/focal pgadmin4 InRelease
Hit:15 https://ngrok-agent.s3.amazonaws.com buster InRelease
Fetched 23.1 kB in 7s (3,508 B/s)
Reading package lists... Done
yammini@atpl-lap-166:~$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
  % Total    % Received % Xferd  Average Speed   Time   Time     Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 15916  100 15916    0     0  43250      0 --:--:-- --:--:-- 43132
=> nvm is already installed in /home/yammini/.nvm, trying to update using git
=> => Compressing and cleaning up git repository

=> nvm source string already in /home/yammini/.bashrc
=> bash_completion source string already in /home/yammini/.bashrc
=> Close and reopen your terminal to start using nvm or run the following to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # This loads nvm bash_completion
yammini@atpl-lap-166:~$ nvm --version
0.38.0
yammini@atpl-lap-166:~$ nvm install 18.15.0
v18.15.0 is already installed.
Now using node v18.15.0 (npm v9.5.0)
yammini@atpl-lap-166:~$
```

Figure 1.3: Linux Node.js Installation

Once Node.js installed, by default NPM is also installed with Node.js installation package, which can be verified with `$ npm -version`

Other NVM commands which can be helpful for developers to play with node versions on different projects are as follows:

- `$ nvm ls` - Checks list of node version in the system
- `$ nvm use 18.x` – For specific use of node version on the project
- `$ nvm alias default 18.x` – It is to set default for all projects in the system
- `$ nvm uninstall 18.x` - It will uninstall that 18.x version from system

Installing Node.js for Windows

While we have covered Linux installation, now let us proceed with the installation process on Windows. You can follow the steps outlined below for Windows installation.

Installing Node.js through NVM on Windows via the command prompt (**cmd**) requires the usage of a specialized tool called "**nvm-windows**". Here are the steps to install Node.js on Windows using **nvm-windows** via the command prompt:

1. Download NVM for Windows:

Go to the GitHub repository of **nvm-windows**: `nvm-windows`. You can explore more **nvm** for windows on <https://github.com/coreybutler/nvm-windows>.

Download the latest installer (.zip file) from the Releases section from following link: <https://github.com/coreybutler/nvm-windows/releases> Here, we will download **nvm-setup.zip** file

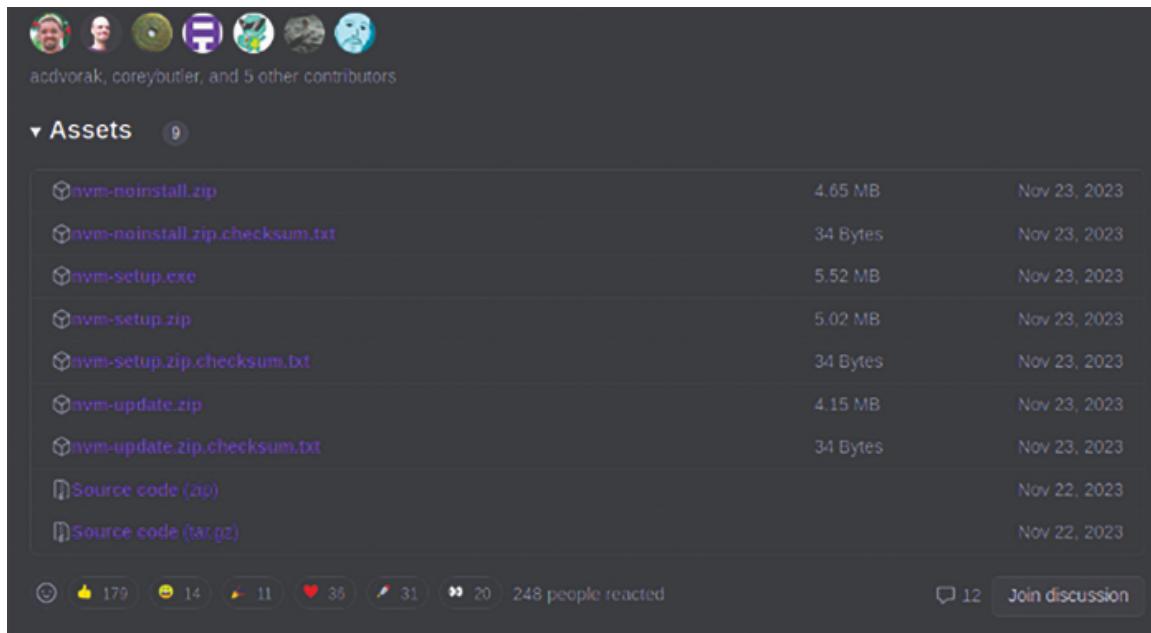


Figure 1.4: Windows Node.js Download Zip File

2. Extract the Zip File:

Extract the downloaded .zip file to a directory on your system.

3. Install **NVM** for Windows:

Open the Command Prompt as an administrator (right-click and select "**Run as administrator**"). Navigate to the directory where you extracted the **nvm-windows** files.

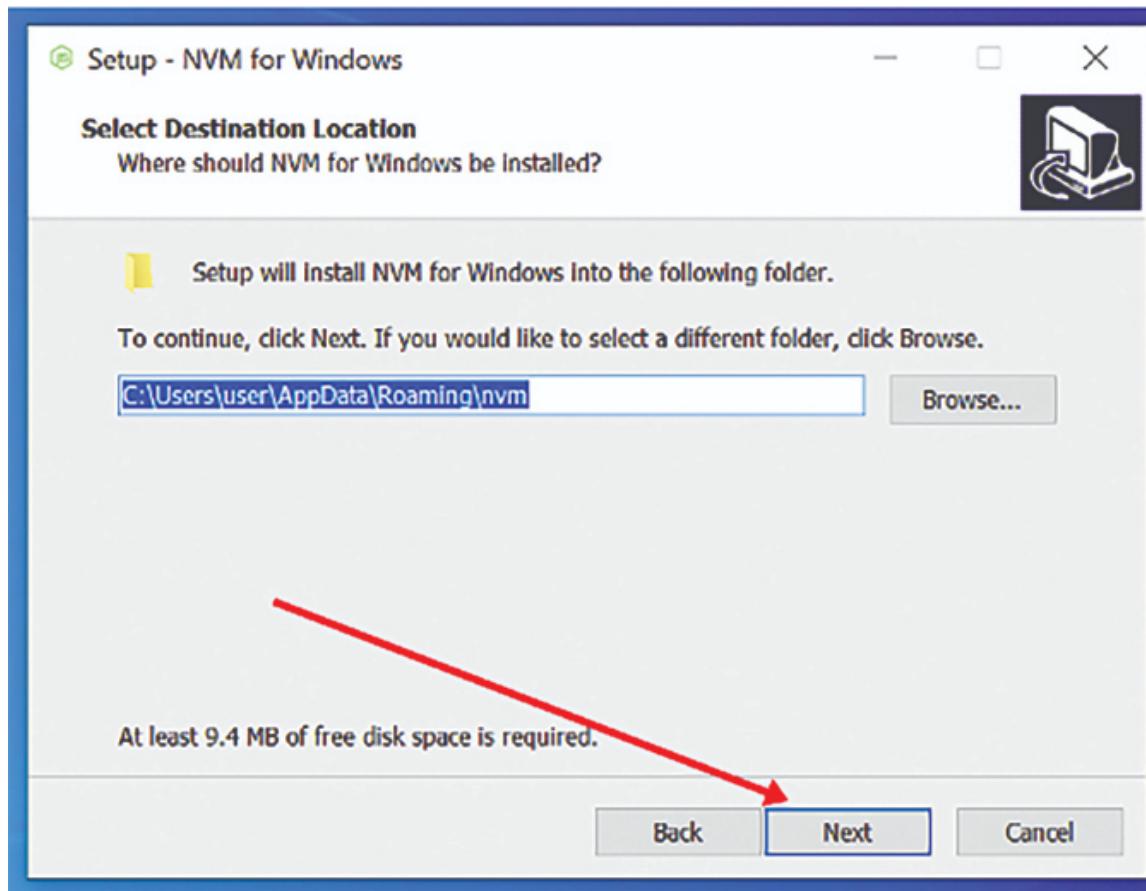


Figure 1.5: NVM Install Select Location

4. Click **Finish** to complete the process:

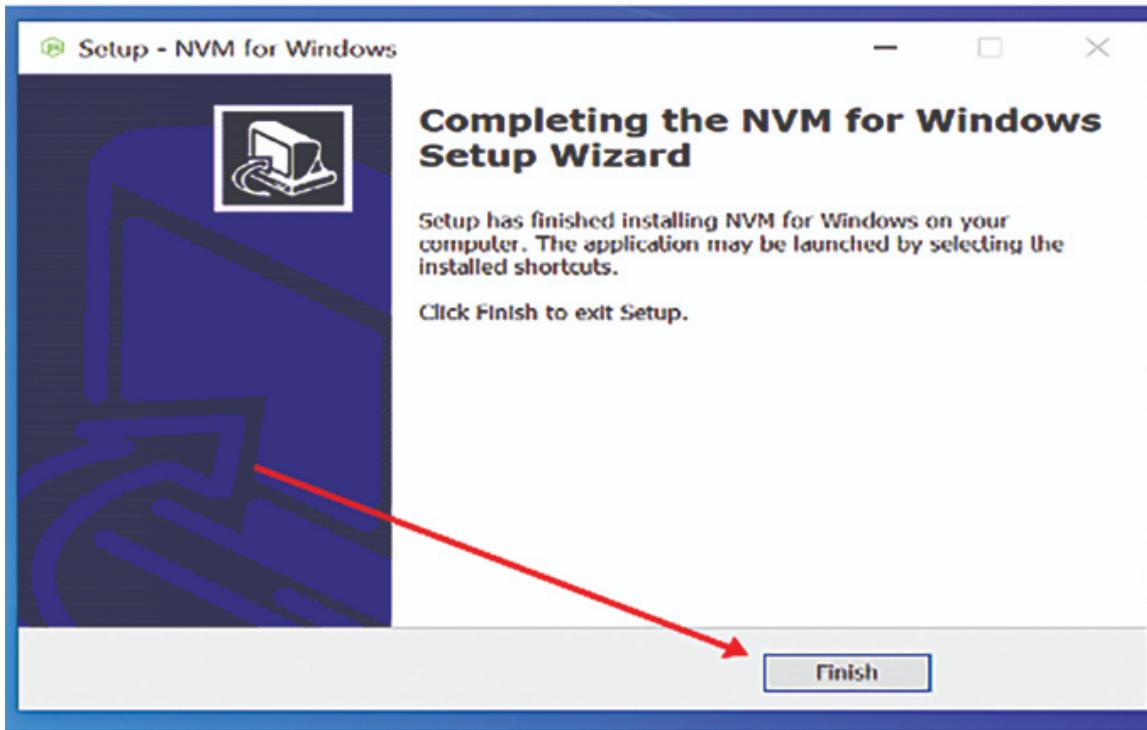


Figure 1.6: NVM Finish Install Process

Run the **nvm-setup.zip** executable to start the installation process. Follow the on-screen instructions to complete the installation.

5. Verify NVM Installation:

Close and reopen the Command prompt as administrator. Run the command `NVM` version to ensure that NVM is installed correctly.

6. Install Node.js:

Once NVM is installed, you can use it to install Node.js. To install a specific version of Node.js, use the command `nvm install <version>` (for example, `nvm install 18.0.0`).

After the installation is complete, you can switch between Node.js versions using the `nvm use <version>` command.

7. Verify Node.js Installation:

Run the command `node -v` to verify that Node.js is installed and the correct version is active.

Installing Node.js for mac

Installation of Node.js on Mac OS is similar to that of Linux. Follow the given steps:

1. Install NVM: To install NVM, we just need to run the following command:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
```

Please make sure that **curl** is available.

2. Install Node.js using NVM. In case, we just want to install latest Node.js, run the following command:

```
nvm install node
```

This command will automatically download and install the latest Node.js version. In case you want to install the LTS (Long term support) version, run this command:

```
nvm install --lts
```

(Please note that there are two '-' hyphens before lts).

3. Verify the installation by opening the console and run the following command:

```
node --version
```

This will output the version installed, for example:

```
v20.0.0
```

4. We can also check the npm version using this command:

```
npm -- version
```



```
9.6.4
```

Event-Driven Mechanism

Node.js is an asynchronous non-blocking event-driven programming. Any action that happens is called an event and it is either done by the user or the system itself. Node.js provides an inbuilt module **Event**, which is an instance of **EventEmitter**. Event is an I/O request that is first sent to the Event Queue. If there are multiple concurrent requests coming to the queue, then the queue passes it to the event loop.

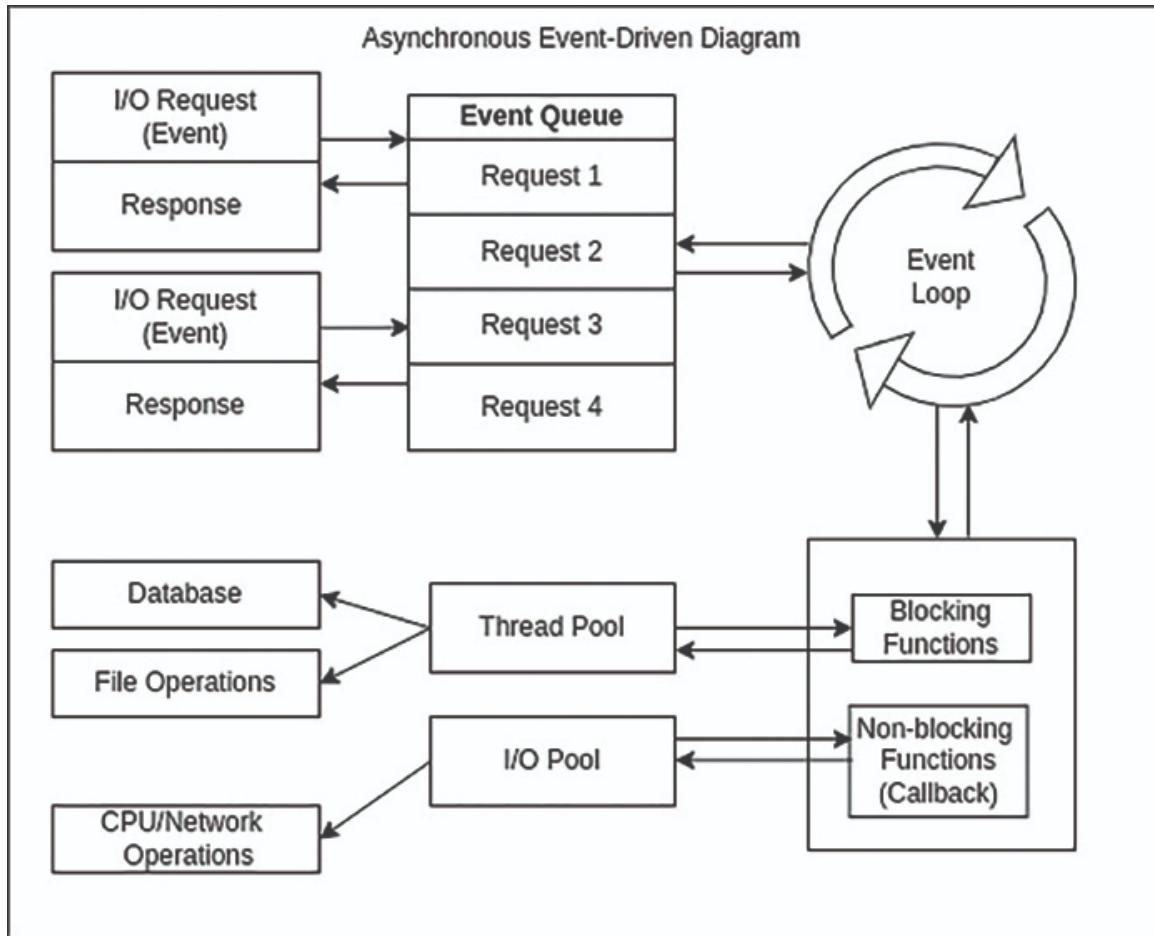


Figure 1.7: Event-Driven Diagram

Event loop monitors the event queue, collects the events from it, then processes it and executes based on blocking and non-blocking functions. Blocking functions are executed sequentially, one after the other, and the second function is not called until the first one responds. Sometimes it depends on external resources and waits for its response which takes longer time, whereas non-blocking functions do not need to wait for any response. It executes asynchronously, which means multiple functions run parallel at a time so that they are not depending on one another. Blocking and non-blocking functions send thread and I/O Pool to its pools, respectively. Once the actual operation is done, the response of that request is sent back to the event queue via event loop. In nutshell, events are emitted and then registered or unregistered through a queue, which is monitored by the event loop, binding the appropriate handlers accordingly.

Node.js follows a non-blocking asynchronous model even though it has a single thread that handles multiple requests at a time, without blocking its call respective handlers.

Example of Event Programming

Create a file save with **event_index.js** and paste the following code:

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

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

// Binds event handler for send message
eventEmitter.on('send_message', function () {
  console.log('Hi, This is my first message');
});

// Handler associated with the connection event
const connectHandler = function connected() {
  console.log('Connection is created');
  // Trigger the corresponding event
  eventEmitter.emit('send_message');
};

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

// Trigger the connection event
eventEmitter.emit('connection');

console.log("Finish");
```

Run the file with **\$ node event_index.js** and the following output will be shown:

```
Connection is created
Hi, This is my first message
Finish
```

Example of Synchronous Code

Create one file named **hello.txt** and paste the following text in it:

```
Hello, I am Developer
```

Create another file named **index.js** within the same folder, paste the following code and save it:

```
const fs = require('fs');
console.log('Start');
const data = fs.readFileSync('hello.txt');
```

```
console.log(data.toString());
console.log('End');
```

Run the code as follows:

```
$ node sync_index.js
```

You will get an output as follows:

```
Start
Hello, I am Developer
End
```

Here, fs is a file system module importing it and `fs.readFileSync()` is a synchronous function which waits until file read is complete and assigns that response to the data variable. It prints line-by-line and executes synchronously.

Example of Asynchronous code

Create one file named `hello.txt` and paste the following text:

```
Hello, I am Developer
```

Create another file named `index.js` within the same folder, paste the following code and save it:

```
const fs = require('fs');
console.log('Start');
fs.readFile('hello.txt', function (err, data) {
  if (err) {
    return console.error(err);
  }
  console.log(data.toString());
});
console.log('End');
```

Run the code as follows:

```
$ node async_index.js
```

You will get an output as follows:

```
Start
End
Hello, I am Developer
```

Here, fs is a file system module. We import it, and `fs.readFile()` is an asynchronous function. This function does not wait for the file read to complete. Instead, it has a callback function. Once the file read operation is finished, the

`callback` function executes and prints the data. Therefore, the line after the callback is executed asynchronously.

Types of Node.js Architectures

When we start developing the applications using Node.js, it is important to decide how your application should be structured. There are many ways to structure your Node.js application with different kinds of architecture. Let us discuss those briefly here.

Monolithic Architecture

In this architecture, all components or modules of business logic are blended together in a single unit. Almost all web servers or server-side frameworks are built using monolithic architecture (see [Figure 1.8](#)), which is the easiest way for developers:

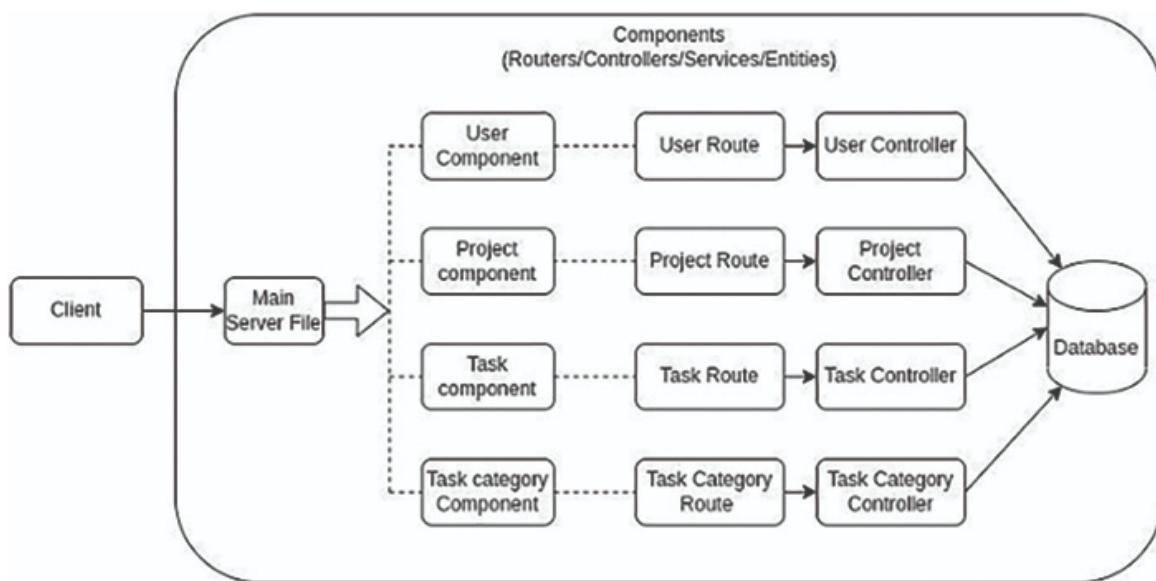


Figure 1.8: Monolithic Architecture

For smaller applications that do not require extensive scalability, this architecture can be suitable. However, it may not be well-suited for larger and more complex applications. As the traffic load on your server-side application grows, you will need to scale it to handle the increased demand. In this architecture, you have a single main Node.js server file that routes all API requests to controllers and services, managing database transactions.

You can scale a monolithic architecture using clusters to reduce the load. However, there are instances when a single server is unable to handle the

incoming traffic. In such cases, you can deploy the same code on multiple servers, run application servers, and employ a load balancer like Nginx. The load balancer, using a round-robin approach, becomes a reliable solution, especially for very large and heavily used applications. We will delve into this aspect in more detail in the deployment section. The biggest drawback of this structure is that if there is a small change needed on any component, then it needs to be done in all servers and rebuilt and redeployed again.

Microservice Architecture

A microservices architecture is a type of architecture that is developed as a collection of services. The framework provided here allows us to develop and deploy the microservices along with maintaining them independently. Microservices sort out the challenges of monolithic systems by fragmenting the application from a whole into several smaller parts. It is reliable and suitable for large and complex applications such as e-commerce platforms, social sites where multiple features are provided to millions of users at same time. Hence, during maintenance or while adding new features, it does not interrupt other existing features and deploy only updated services. Nowadays, it is trending more and more for its flexibility where multiple developers work individually and are only responsible for their own small code instead of whole system code.

In this architecture, all components or modules of business logic are individual. Many large enterprises use this kind of microservice architecture (see [Figure 1.9](#)):

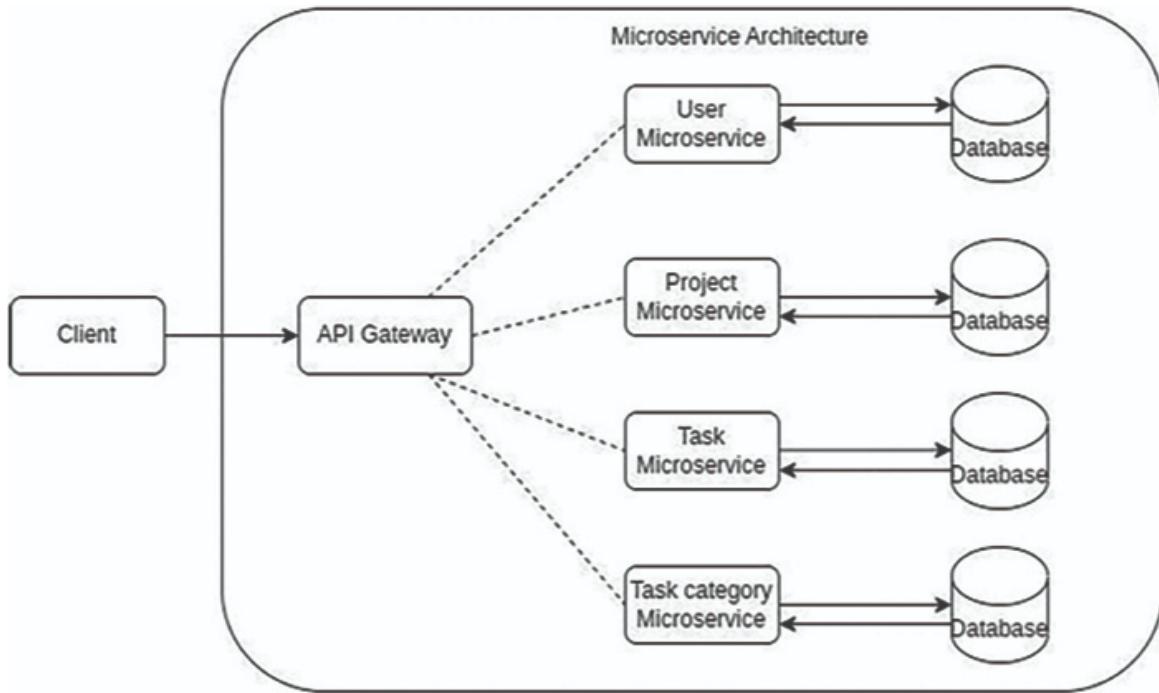


Figure 1.9: Microservice Architecture

As per the preceding diagram, client as user or UI sends a request, which is collected by API Gateway and passed to the respective microservice, which has its own function (Lambda Function). This function connects to the database and gives back a response accordingly. Each microservice can be easily changed and deployed without affecting each other. In addition, these microservices also call each other through API HTTP service or gRPC (Google Remote Procedure Call) which is a generic flow of microservice architecture. However, while it is a cost-effective and time-saving architecture for development, it may not be suitable for smaller applications. This is because it relies on cloud-based solutions, which can become expensive even with minimal setup requirements. This cost issue can often be mitigated by adopting more budget-friendly solutions offered by monolithic architectures. In fact, microservices represent a way to leverage serverless architecture within the realm of cloud computing.

Serverless Architecture

Serverless architecture is the approach for developing and building applications without managing infrastructure. Basically any app is developed and deployed on a specific server. However, managing the server hosting process can be a cumbersome task for developers. This is where serverless architecture becomes a boon for those who want to avoid server management and only pay for what they

use. In serverless architecture, everything is handled by third-party services provided by cloud computing providers like AWS, Azure, Google, etc. These providers offer respective functions like AWS Lambda functions, Microsoft Azure functions, and Google Cloud functions, which is why it is also referred to as “*Function as a Service*” (FaaS). However, it is important to note that this approach has its drawbacks, as it involves entrusting everything to third parties, which can raise security concerns. Even though it has some limitations, it is still becoming more popular because organizations focus on actual products and services, not infrastructure, so it will be cost effective for those who spend a lot of effort on infrastructure.

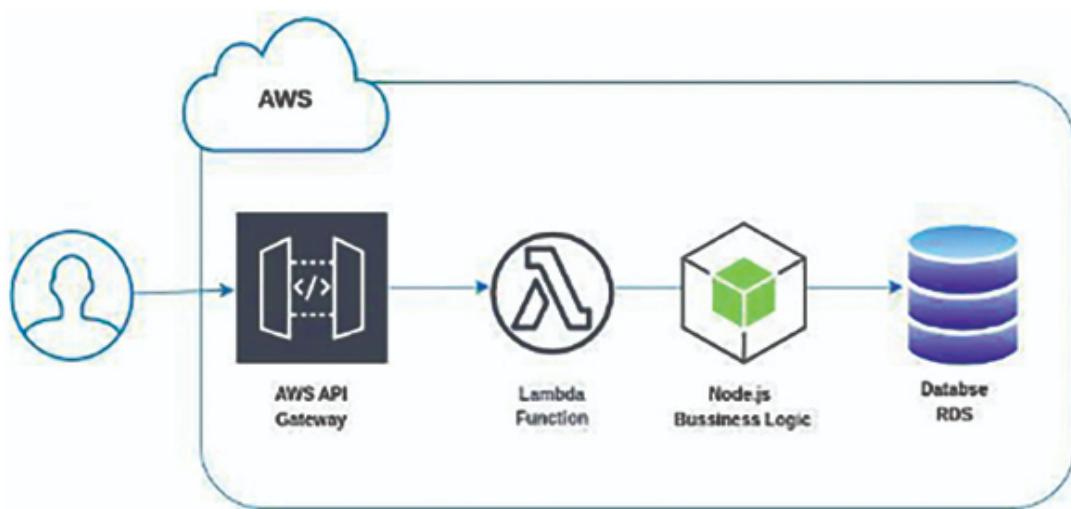


Figure 1.10: Serverless Architecture

In this serverless architecture, a transformative approach for application development and deployment that eliminates the need for traditional server management. Instead, it enables developers to focus solely on writing code while cloud providers handle the underlying infrastructure. Below is a visual representation of this serverless concept (see [Figure 1.10](#)).

It is an example of AWS serverless architecture in which an AWS API gateway is used to route REST API calls and is based on the route Lambda function called with an attached gateway. Lambda functions can be written in different languages but we consider them to be written in Node.js code that have the actual business logic to perform action to the database. AWS Cloud provides various database instances such as DynamoDB, MySQL, PostgreSQL, and others. Furthermore, it can effortlessly scale to accommodate increasing workloads. AWS offers auto-scaling capabilities, which means it can automatically add more EC2 instances when there is a surge in load and reduce instances when the load decreases.

Aspect	Monolithic Architecture	Microservices Architecture	Serverless Architecture
Development	Easier setup and development process	Complex setup, but independent services promote scalability	Focuses on writing functions without managing infrastructure
Scalability	Limited scalability due to the entire application being scaled	Scalable, as each service can be independently scaled	Automatically scales based on demand
Maintenance	Single codebase makes maintenance easier	Requires management of multiple services and communication	Less management overhead; managed by cloud provider
Technology Stack	Limited flexibility; all components use the same technology stack	Flexibility to use different technologies for each service	Limited control over underlying infrastructure and runtime
Deployment	Simple deployment process; deploy as a single unit	Deployment complexity due to multiple services	Simplified deployment process
Resource Utilization	Resource utilization may be inefficient	Optimized resource utilization with services scaled as needed	Efficient resource utilization with on-demand execution
Cost	Lower upfront costs; higher operational costs in the long term	Higher initial setup costs; potential cost savings with scale	Pay-per-use model can be cost-effective for low traffic apps
Fault Isolation	A bug in one part can affect the entire application	Faults are isolated to specific services; others remain unaffected	Managed by cloud provider, potential vendor lock-in
Flexibility	Limited flexibility due to the monolithic structure	Flexibility to use different technologies and languages	Limited control over underlying infrastructure and runtime

Table 1.1: Comparisons of Architecture

Table 1.1 provides a comparison of various aspects of monolithic, microservices, and serverless architectures in Node.js, outlining their respective advantages and disadvantages. Depending on specific project requirements, one architecture may be more suitable than the others.

Let us create a basic http and https server with programming.

Writing an HTTP Server

Now when Node.js is properly set up and running in your system, let us do the famous "**Hello World**" in Node.js way by serving this through an HTTP server.

Let us create a file with name index.js, and copy the following code into it:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

In the provided code, "**http**" is a default module provided by Node.js, so there is no need to install it separately. However, for modules not included in Node.js's built-in modules, you can install them from the npm library.

The npm library contains millions of registered packages, and you can install them using the following code:

```
$ npm install package-name
ex. npm install express
```

package-name can be as "**body-parser**", "**express**", "**moment**", and so on.

The http module creates a http server that runs on specific port 3000. Ideally, Node.js runs on port 3000 but developers can assign different ports such as 3001, 4000 or any port. Just need to make sure that the port number does not conflict with other applications on the system.

To run the program, open console with source code directory and paste the following command:

```
$ node index.js
```

Once it runs, open the browser and go to the URL as <http://localhost:3000>; it will print "**Hello World**".

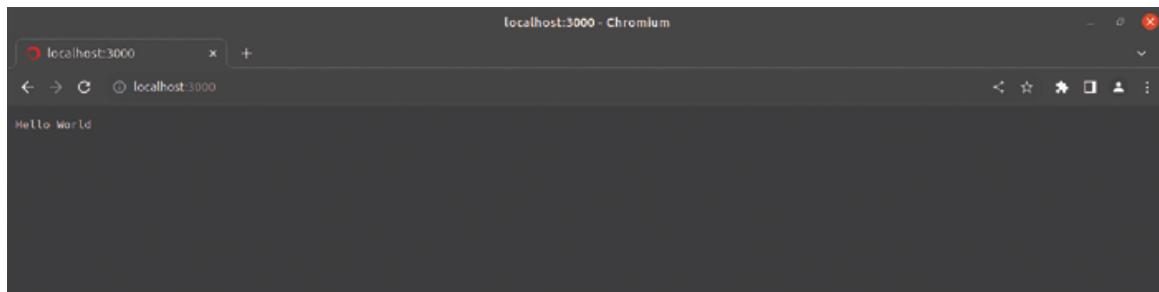


Figure 1.11: HTTP Server Program Output

Making it HTTPS

The server we just created does not provide a secure way of serving APIs. Most often, we need to serve the APIs using HTTPS rather than HTTP.

HTTP does not encrypt the data, thus it is not secure where the information is leaked during transmission. On the other hand, HTTPS encrypts data during transmission on request from client to server that makes it secure.

Let us rewrite the same code with the https server. For HTTPS, we need SSL certificates.

Let us first create a self-signed SSL certificate:

1. Open console and install **openssl** if not installed in your system.
2. For Debian Linux such as ubuntu, installation can be done using the **apt** command:

```
$ sudo apt install openssl
```

3. For Centos, Fedora, and Rocky Linux, yum can be used to install **openssl**:

```
$ sudo yum install openssl
```

4. Make **openssl** directory as follows:

```
$mkdir openssl  
$cd openssl
```

5. Request to generate **ssl** certificate with the following command:

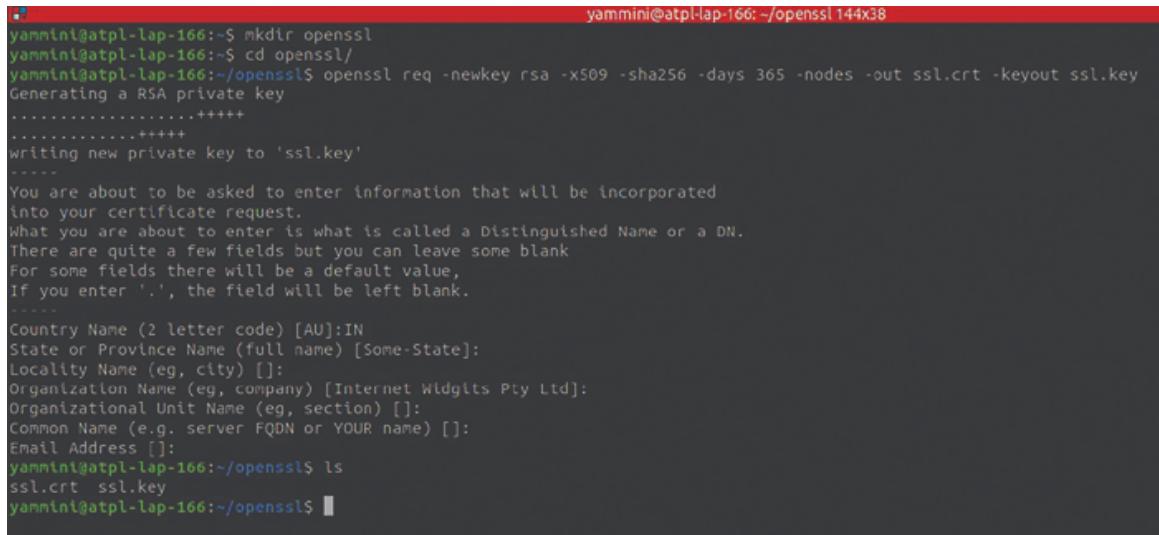
```
$ openssl req -newkey rsa -x509 -sha256 -days 365 -nodes -out  
ssl.crt -keyout ssl.key
```

Let us understand the preceding command:

- **-newkey rsa** : create new key with **rsa** algorithm default 2048 bit
- **-x509** : creates a X.509 Certificate
- **-sha256** : use 256-bit SHA (Secure Hash Algorithm)
- **-days 365** : The number of days to certify the certificate for 365. You can use any positive integer
- **-nodes** : creates a key without a passphrase.
- **-out ssl.crt** : Specifies the filename to write the newly created certificate to. You can specify any file name.

- **-keyout ssl.key** : Specifies the filename to write the newly created private key to. You can specify any file name.

Once you enter this command, it will prompt the following questions (as shown in [Figure 1.12](#)). Press enter until done and check that folder which has **ssl.crt** and **ssl.key** files:



```

yammini@atpl-lap-166:~$ mkdir openssl
yammini@atpl-lap-166:~$ cd openssl/
yammini@atpl-lap-166:~/openssl$ openssl req -newkey rsa -x509 -sha256 -days 365 -nodes -out ssl.crt -keyout ssl.key
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'ssl.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Wldglts Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
yammini@atpl-lap-166:~/openssl$ ls
ssl.crt  ssl.key
yammini@atpl-lap-166:~/openssl$ 
```

Figure 1.12: Openssl Certificate Generation

Now create **main.js** and write the following code to create **https server** with **port 3000**:

```

// import https module
const https = require(`https`);
//import fs module to read files
const fs = require(`fs`);
const options = {
  key: fs.readFileSync(`./openssl/ssl.key`),
  cert: fs.readFileSync(`./openssl/ssl.crt`)
};
// create https server on port 3000
https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end(`hello world from https server \n`);
}).listen(3000); 
```

Run the code with node **main.js** and open the browser with **https://localhost:3000** which displays "**hello world from https server**". This way a secure https server is built with very basic examples.

Using Cluster Module

Node.js can easily make applications highly scalable through the cluster module. Cluster is creating a child process so that it raises another process which splits single thread into multi thread. Due to that heavy traffic load is reduced and shared on different instances of thread with the same port. It is a built-in module of Node.js. As Node.js supports async single thread and sometimes when blocking functions are more, the application performance gets down. Cluster is most important and useful to improve it.

Node.js server initiates multiple incoming requests. First, it points to the main process that is called primary process or master, which is a single one. Afterwards it splits into different child processes from its parent process. Child process is called the worker process, which can be multiple and has its own event loop that processes it simultaneously. Clustering has two ways of distribution process. The first one is the default round robin method in which the master listens to server requests and sends them to the worker in an equally circular order, and the other one is socket based where the master listens and assigns only interested workers who want to do the process.

Cluster architecture harnesses the capabilities of multiple interconnected servers, elevating the performance, reliability, and scalability of contemporary applications. [Figure 1.13](#) provides a visual representation of how cluster nodes collaborate seamlessly to efficiently manage incoming requests.

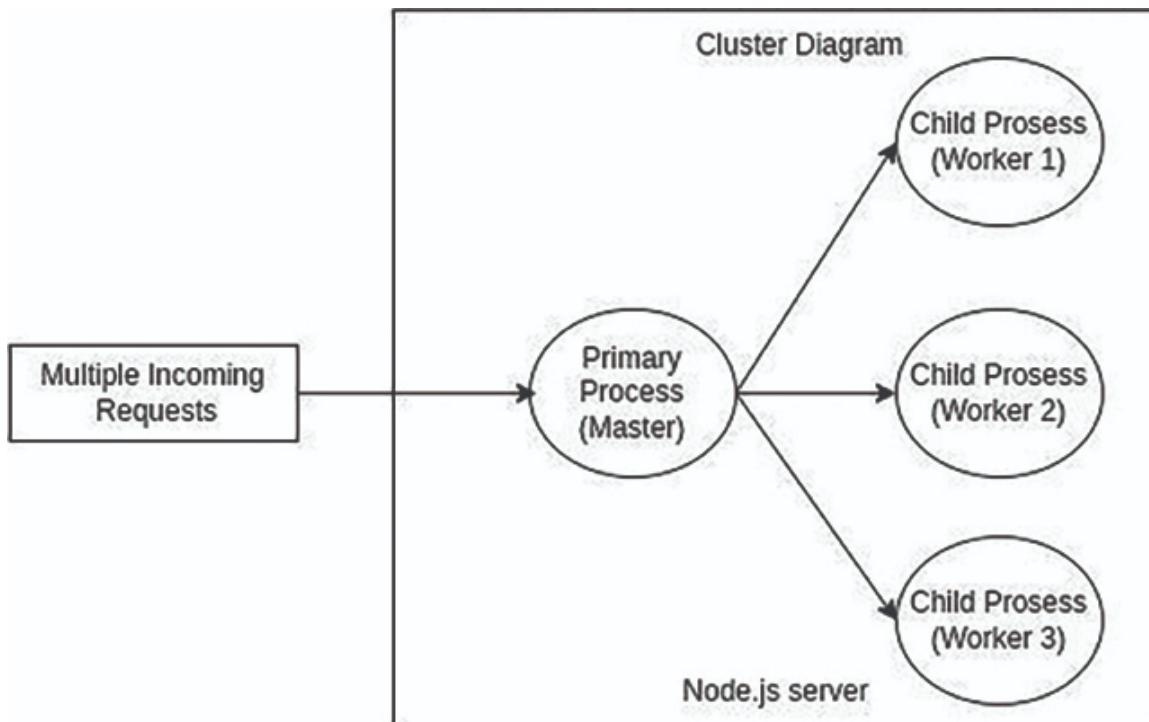


Figure 1.13: Cluster Diagram

Programming Without Cluster Module Example

Create a file named as `without_cluster.js` and save the following code:

```
//Import http module for create server
const http = require('http');
// create server and api for test
http.createServer(function (req, res) {
  if (req.url === "/api/test" && req.method === "GET") {
    console.time('API_without_cluster');
    let result = 0;
    for (let i = 0; i < 5000000; i++) {
      result += i;
    }
    console.timeEnd('API_without_cluster');
    console.log(`Result = ${result} - on process ${process.pid}`);
    res.end(`Result = ${result}`);
  }
}).listen(3001);
```

Run the code with this command:

```
$ node without_cluster.js
```

Now we can test this in the browser with URL `http://localhost:3001/api/test` and call it multiple times continuously by hitting the refresh button multiple times.

The following output will be displayed in the console:

```
yammini@atpl-lap-166:~/Desktop$ node without_cluster.js
API_without_cluster: 14.077ms
Result = 12499997500000 - on process 76523
API_without_cluster: 14.445ms
Result = 12499997500000 - on process 76523
API_without_cluster: 19.866ms
Result = 12499997500000 - on process 76523
API_without_cluster: 22.245ms
Result = 12499997500000 - on process 76523
API_without_cluster: 22.777ms
Result = 12499997500000 - on process 76523
API_without_cluster: 19.854ms
Result = 12499997500000 - on process 76523
API_without_cluster: 22.299ms
Result = 12499997500000 - on process 76523
API_without_cluster: 20.373ms
Result = 12499997500000 - on process 76523
```

Figure 1.14: Output Without Cluster

Programming With Cluster Module Example

Now create another file named **cluster.js** and save the following code:

```
//Import cluster module
const cluster = require('cluster');
//Import http module for create server
const http = require('http');
//check if it is master process then create child process through
fork() method
if (cluster.isMaster) {
  const numWorkers = require('os').cpus().length;
  console.log(`Master ${process.pid} started`);
  console.log(`Number of workers => ${numWorkers}`);
  for (var i = 0; i < numWorkers; i++) {
    cluster.fork();
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
    console.log("Let's fork another worker!");
    cluster.fork();
  });
}
```

```
    } else {
        // it is worker process so run multiple process with same 3000
        // port
        console.log(`Worker ${process.pid} started`);
        http.createServer(function (req, res) {
            if (req.url === "/api/test" && req.method === "GET") {
                console.time('API_with_cluster')
                let result = 0;
                for (let i = 0; i < 50000000; i++) {
                    result += i;
                }
                console.timeEnd('API_with_cluster');
                console.log(`Result = ${result} - on process ${process.pid}`);
                res.end(`Result = ${result}`);
            }
        }).listen(3000);
    }
}
```

Now run the code using this command:

```
$ node cluster.js
```

Open the browser with URL <http://localhost:3000/api/test> and call it multiple times. It will give the following output in the console:

```
yammini@atpl-lap-166:~/Desktop$ node cluster.js
Master 79030 started
Number of workers => 8
Worker 79037 started
Worker 79038 started
Worker 79044 started
Worker 79050 started
Worker 79056 started
Worker 79062 started
Worker 79068 started
Worker 79070 started
API_with_cluster: 10.762ms
Result = 12499997500000 - on process 79037
API_with_cluster: 12.031ms
Result = 12499997500000 - on process 79038
API_with_cluster: 13.834ms
Result = 12499997500000 - on process 79044
API_with_cluster: 12.075ms
Result = 12499997500000 - on process 79050
API_with_cluster: 13.294ms
Result = 12499997500000 - on process 79056
API_with_cluster: 12.642ms
Result = 12499997500000 - on process 79062
API_with_cluster: 16.445ms
Result = 12499997500000 - on process 79068
API_with_cluster: 12.777ms
Result = 12499997500000 - on process 79070
API_with_cluster: 15.666ms
Result = 12499997500000 - on process 79037
API_with_cluster: 13.546ms
Result = 12499997500000 - on process 79038
```

Figure 1.15: Output With Cluster

As we can see that when we use the cluster module, the responses take less time between 12 and 16 ms, but without the cluster module, the time goes higher – 14 to 22 ms. The difference here is not much since the code we are using has almost no logic, database operations, or any other IO. The time may be changed with implementation so cluster is useful when computation is heavy, but if there are not

too many computations it might not be beneficial. Basically, cluster allows us to run multiple workers which can utilize more than one CPU.

The cluster module can also be used to set up a master-worker setup where master monitors the workers and in case a worker stops or crashes, master can start another worker. This way, we can handle errors safely in the application and prevent applications from completely crashing. In [Chapter 4, Planning the Application](#), we will see it in action.

Conclusion

In this chapter, we got an introduction to Node.js and what it offers along with its pros and cons. We learned how to install Node.js and created a simple server. We also got familiar with how Node.js makes use of event loop and different types of architecture. Later we created a web server with HTTP and HTTPS. Finally, we saw how the cluster module can be used.

In this chapter, we used JavaScript as a programming language, which is not maintainable when the project size becomes big. A better approach is to use Typescript instead of JavaScript. In the next chapter, we will learn the basics of Typescript.

Multiple Choice Questions

1. What is Node.js and which of the following statements about it is true?
 - a. Node.js is a closed-source JavaScript runtime environment
 - b. Node.js can only be used on Windows operating systems
 - c. Node.js is primarily based on Python code
 - d. Node.js is an open-source JavaScript runtime environment that can be used on various operating systems
2. For which types of applications is Node.js commonly used?
 - a. Node.js is mainly used for desktop applications and gaming
 - b. Node.js is primarily utilized for mobile app development
 - c. Node.js is commonly employed for single-page applications (SPAs), real-time applications, Internet of Things (IoT) devices applications, and data streaming applications
 - d. Node.js is exclusively used for web-based email services like Gmail

3. What is one of the key advantages of using Node.js for real-time applications?
 - a. Node.js is the only option for building real-time applications
 - b. Node.js provides a graphical user interface for real-time applications
 - c. Node.js offers a continuous connection through WebSockets, enabling faster response times
 - d. Node.js can only be used for audio and video streaming applications
4. How can you check the version of Node.js installed on your system?
 - a. Run the command **node version** in the terminal
 - b. Run the command **node info** in the terminal
 - c. Run the command **node --v** in the terminal
 - d. Run the command **node -v** in the terminal
5. What is the key characteristic of the event loop in Node.js?
 - a. It executes blocking functions in parallel to improve performance
 - b. It waits for all functions to complete before moving to the next
 - c. It handles rendering and user interface tasks in Node.js applications
 - d. It manages asynchronous operations, ensuring non-blocking execution
6. How does microservices architecture differ from monolithic architecture?
 - a. Microservices use a single codebase for all components
 - b. Microservices are tightly coupled and run as a single application
 - c. Microservices are loosely coupled and consist of independently deployable services
 - d. Microservices communicate only via RESTful APIs
7. When is serverless architecture a suitable choice for application development?
 - a. When you want to focus on writing code and not worry about server provisioning
 - b. When the application has a monolithic codebase
 - c. When you want to minimize development costs
 - d. When you need full control over server management

8. Which method in the HTTP module is used to create an HTTP server in Node.js?

- a. `http.createServer()`
- b. `http.request()`
- c. `http.get()`
- d. `http.post()`

9. Which method is used to create a cluster of Node.js processes using the Cluster module?

- a. `cluster.start()`
- b. `cluster.fork()`
- c. `cluster.create()`
- d. `cluster.spawn()`

10. How does the Cluster module enhance the performance of a Node.js application?

- a. By creating multiple instances of the Node.js application
- b. By managing database connections more efficiently
- c. By reducing the number of available CPU cores
- d. By slowing down the application's response time

Answers

- 1. d
- 2. c
- 3. c
- 4. d
- 5. d
- 6. c
- 7. d
- 8. a
- 9. b
- 10. a

Further Reading

<https://nodejs.org/en>

OceanofPDF.com

CHAPTER 2

Introduction to TypeScript

Introduction

Millions of applications are developed in JavaScript, but developers face many issues while developing it. JavaScript runs directly without compiling the code first. Due to dynamic typing and lack of compile-time checks, issues like runtime errors, code complexity, and maintenance of large codebase are common. TypeScript was introduced as a superset of JavaScript to address all such issues.

In this chapter, we will discuss the basic concepts of TypeScript and how to use it instead of JavaScript in your application while developing.

Structure

In this chapter, we will cover the following topics:

- Overview of TypeScript
- Advantages of TypeScript
- Pitfalls of TypeScript
- Installing TypeScript
- Building a Basic Application with TypeScript
- OOP (Object-Oriented Programming) Concepts
- Features of ECMAScript
- EsLint

Overview of TypeScript

TypeScript is a superset of JavaScript, meaning all JavaScript features are present and other extra features are bundled in it. JavaScript is an object-based language but does not provide all the concepts of object-oriented programming, so the missing part is achieved through TypeScript. TypeScript is an open-source programming language that is developed and maintained by Microsoft. It was released in 2012 and is often abbreviated as TS.

Displayed in the image (see [Figure 2.1](#)) is the representation of TypeScript, which functions as a superset of JavaScript, compiled by the TypeScript compiler (TSC) and subsequently transformed into JavaScript code.

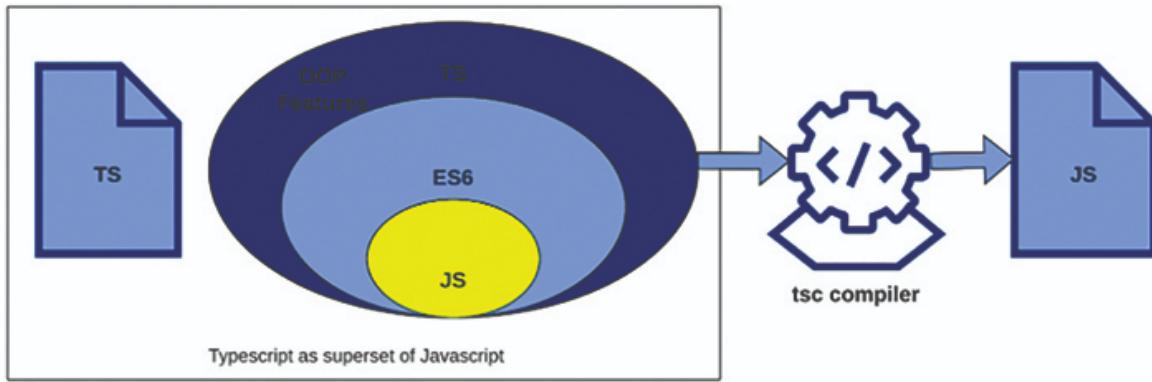


Figure 2.1: TypeScript as a Superset of JavaScript

JavaScript is an interpreted language which runs directly without the code being compiled. So if any error occurs, it will be detected at run time. TypeScript can easily deal with that problem, because it is strict in checking the type of data. If it is followed properly, there are fewer chances of error at run time. TypeScript first compiles the code and then transpiles the code. Basically here transpiles means converting TypeScript code to JavaScript code. The default compiler for it is known as **tsc**. TypeScript supports all ES6 (ECMAScript6) features and oops concepts. TypeScript is **static checking** which states that it detects errors without executing code.

Let us consider an example:

JS code

```
let name = "Jack";  
name = 12;
```

No error

TS Code

```
let name: string = "Jack";  
name = 12;  
  
Error: Type 'number' is not assignable to type 'string'
```

This is a very basic example of both JS and TS codes. The difference is that JS automatically considers the string type and once a number value is assigned, it converts number to string, and hence, it will not give any error. However, this can lead to runtime issues when used in subsequent code sections involving number-

related operations. In contrast, TypeScript identifies such issues during the build process, generating errors at that stage.

Advantages of TypeScript

There are many benefits of using TypeScript. Some key advantages are as follows:

- **Open source:** TypeScript is free and open source, strongly type checking programming language. It can be easily installed and used free of cost.
- **Cross platform:** TypeScript supports cross platform and has cross browser compatibility. After all, it is run as JavaScript code once the compilation is finished.
- **Supports OOP and ES6:** TypeScript has the capability to use the most powerful object oriented programming concept, the same as other high level oops languages such as JAVA which is not enabled for Js. It supports classes, interfaces, access modifiers, abstraction, inheritance, and so on. It also has the ability to use ES6 features for better coding standards and readability.
- **Bug detector:** TypeScript provides a bug detection facility while developing applications that give errors with description before running it; so developers can assuredly save time for debugging and develop it with robust code.
- **Optional type checking:** TypeScript is basically for strong type checking but it can be optional because it allows static as well as dynamic checking. For dynamic checking, it needs to be optional.
- **Flexible and maintainable:** TypeScript has huge flexibility to be used with different JavaScript frameworks such as React.js, Angular, Nest.js, Express.js, loopback, and so on. It can be used as JavaScript on both client as well as server sides. Moreover, code can be easily maintained even though different developers collaborate and work together by following standard coding practices.
- **Better IDE support:** TypeScript provides much better IDE support than JavaScript. All popular IDEs provide features such as code completion, parameter hints, type checking, and so on, which makes development faster and more efficient.

Pitfalls of TypeScript

While TypeScript has several advantages, there are some disadvantages as well:

- **Unsuitable for small applications:** TypeScript is not fit for all applications, mostly for small-scale applications because it becomes complicated to implement minor features where type checking is not needed whereas in JS it can be easily done.
- **Compilation time:** TypeScript code is first compiled, so it takes time in the overall process of transpiling which must need a compiler for compilation whereas JavaScript does not require it.
- **Learning curve:** Being a bit more complex than JavaScript, developers need to invest some time to learn and adapt the syntax. However, it is usually only for the first time. Once a developer is familiar and comfortable with TypeScript, because of TypeScript's advantages, a lot of time is saved.

Installing TypeScript

TypeScript can be installed and set up in your system in two ways. First one is installing it globally and another one is installing it locally as a dev dependency. We can install it through the npm module which must be installed before in your system.

Global Installation

Global installation approach proves beneficial when handling multiple projects, eliminating the need for project-specific installations. However, it assumes that the same version is employed across various projects.

Open the terminal and paste the following command. It will install the latest version in your system.

```
$ npm install typescript -g
```

Once installation is done, if you want to verify the version of installed TypeScript, paste the following code:

```
$ tsc -v
```

Output of the preceding command will give like "**Version 5.3.3**"

```
yammini@atpl-lap-166:~$ npm install typescript -g
changed 1 package in 3s
yammini@atpl-lap-166:~$ tsc -v
Version 5.3.3
yammini@atpl-lap-166:~$ █
```

Figure 2.2: Typescript Installation

Project-wise Installation

In many of the cases there is a need to use different versions of TypeScript with different projects, that time locally installation is necessary.

Create one folder for the project and install it locally as follows:

```
$ mkdir basic-typescript-project
$ cd basic-typescript-project
$ npm install typescript --save-dev
```

This will install TypeScript on that project locally. We can now jump into developing basic applications with TS.

Building a Basic Application with TypeScript

Once installation is done, go to the root directory of the created project and follow the given steps:

1. **Initialize the project:** Enter the following command that will create a **package.json** file for initializing the node project, and it will prompt some questions for information about the project such as project name, version, description, entry point (main file), and so on. So enter those details as shown in [Figure 2.2](#).

```
$ npm init
```

```

yammini@atpl-lap-166:~/projects/basic-typescript-project
yammini@atpl-lap-166:~/projects/basic-typescript-project 185x42
yammini@atpl-lap-166:~/projects/basic-typescript-project$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See 'npm help init' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg>' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (basic-typescript-project)
version: (1.0.0)
description: Basic startup project for typescript with node.js
entry point: (index.js) main.js
test command:
git repository:
keywords:
author: Yamini Panchal
license: (ISC)
About to write to /home/yammini/projects/basic-typescript-project/package.json:

{
  "name": "basic-typescript-project",
  "version": "1.0.0",
  "description": "Basic startup project for typescript with node.js",
  "main": "main.js",
  "dependencies": {
    "typescript": "^5.0.4"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Yamini Panchal",
  "license": "ISC"
}

Is this OK? (yes) yes
yammini@atpl-lap-166:~/projects/basic-typescript-project$ 

```

Figure 2.3: Project Initialization Step

2. **Install TypeScript as dev dependency:** Now run the following command which will add TypeScript as dev dependency (development dependency) purpose:

```
$ npm install typescript --save-dev
```

3. **Install node as dev dependency with TypeScript:** While working with Node.js projects, it is better to install it with types. This will allow all default modules of Node.js to be used with TypeScript such as http, https, fs, and so on.

```
$ npm install @types/node --save-dev
```

Any extra package can be installed with `@types/pkg_name --save-dev` while developing so that the type definitions can be made available in the IDE.

4. **Create a `tsconfig.json` file and define compiler options:** Create a `tsconfig.json` file in the root directory and then configure the compiler options as follows:

```
{
  "compilerOptions": {
    "target": "es2019",
    "module": "commonjs",
    "moduleResolution": "node",
    "pretty": true,
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./lib",
    "noImplicitAny": false,
    "esModuleInterop": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "experimentalDecorators": true,
    "alwaysStrict": true,
    "forceConsistentCasingInFileNames": true,
    "emitDecoratorMetadata": true,
    "resolveJsonModule": true,
    "skipLibCheck": true
  },
  "include": [
    "lib/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ],
  "files": [
    "node_modules/@types/node/index.d.ts"
  ]
}
```

Let us understand these compiler options one by one and then exclude and include options in the **tsconfig.json** file:

- **target:** Specifies the ECMAScript Target Version. By default it is '**ES3**' and can be set to '**ES5**', '**ES6**', '**ES2015**', '**ES2016**', '**ES2017**', '**ES2018**', '**ES2019**', or '**ESNEXT**'. It sets the JavaScript language version for emitted JavaScript and include compatible library declarations.
- **module:** Specifies what module code is generated. If the target attribute is **ES5** or **ES3**, then the default value is '**commonjs**' else '**ES6**', '**AMD**',

'System', 'ES2015', or 'ESNEXT'.

- **moduleResolution**: Specifies module resolution strategy as '**node**' (Node.js) or '**classic**' (TypeScript). It is always set as a node for modern JS, otherwise classic. It specifies how TypeScript looks up a file from a given module specifier.
- **pretty**: Enables color and formatting in output to make compiler errors easier to read. This can be set as either true or false.
- **sourceMap**: This enables generating source map files for emitted JavaScript or not, it helps in debugging and reporting errors.
- **outDir**: Specify an output folder for storing all transpiled files.
- **baseUrl**: Specify the base directory to resolve non-relative module names where all TypeScript files are located.
- **noImplicitAny**: Enable error reporting for expressions and declarations with an implied any type.
- **esModuleInterop**: Emit additional JavaScript to ease support for importing CommonJS modules. This enables **allowSyntheticDefaultImports** for type compatibility.
- **removeComments**: Disable emitting comments, which means do not emit comments to output.
- **preserveConstEnums**: Disable erasing const **enum** declarations in generated code. When set as true, the **enum** exists at runtime and provides a way to reduce the overall memory footprint of your application at runtime by emitting the enum value instead of a reference.
- **experimentalDecorators**: This option enables using decorators in TS projects. ES has not yet introduced decorators, so they are disabled by default.
- **alwaysStrict**: Ensures that your files are parsed in the **ECMAScript** strict mode, and emit "**use strict**" for each source file. Default value is false.
- **forceConsistentCasingInFileNames**: Ensure that casing is correct in imports which disallow inconsistently cased references to the same file.
- **emitDecoratorMetadata**: Enables experimental support for emitting type metadata for decorators.
- **resolveJsonModule**: Enable importing **.json** files.

- **skipLibCheck**: Skip type checking all `.d.ts` files.
- **include**: Specifies a list of glob patterns that match files to be included in compilation. If no ‘files’ or ‘include’ property is present in `tsconfig.json`, the compiler defaults to including all files in the containing directory and subdirectories except those specified by ‘exclude’.
- **exclude**: Specifies a list of files to be excluded from compilation. The ‘exclude’ property only affects the files included via the ‘include’ property and not the ‘files’ property.
- **files**: If no ‘files’ or ‘include’ property is present in `tsconfig.json`, the compiler defaults to including all files in the containing directory and subdirectories except those specified by ‘exclude’. When a ‘files’ property is specified, only those files and those specified by ‘include’ are included.

5. **Create a basic application with TypeScript**: Create a **lib** folder and create the `main.ts` file in that as defined in `tsconfig.json` include **lib** folder which compiles all `.ts` files.

```
$ mkdir lib
$ cd lib
$ touch main.ts
```

Now write the following code in the `main.ts` file and save it:

```
import * as http from 'http';
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running with TypeScript project at
  http://${hostname}:${port}/`);
});
```

6. **Run TypeScript Code**: Install the `ts-node` package for compilation and `nodemon` so that every time the changes in files can be auto reload or rerun. This way we do not need to run the server again after some changes. Go to the root directory and enter the following commands:

```
$ npm install ts-node nodemon --save-dev
$ tsc
```

As soon as the **tsc** command runs, a new directory is created named **dist** (we specified it in **tsconfig.json** as `outDir`). All TS files are compiled as JS and placed in this directory. Now we can run the application on the specified port, which is usually 3000 by default. If the port is 3000, then the application would be available at <http://127.0.0.1:3000>.

nodemon can detect new changes in the **dist** directory and restart the server as soon as there is a change. We just need to run the server using **nodemon** instead of **node**:

```
$ nodemon dist/main.js
```

```
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ mkdir lib
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ cd lib/
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project/lib$ touch main.ts
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project/lib$ cd ../
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ npm install ts-node nodemon --save-dev
added 51 packages, and audited 54 packages in 8s

3 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ tsc
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ nodemon dist/main.js
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/main.js'
Server running with TypeScript project at http://127.0.0.1:3000/
```

Figure 2.4: Compile Typescript Application

nodemon can detect the changes from the **dist** directory and run **main.js** again, but since we are coding in TS, the files need to be recompiled after each change. Manual recompilation using **tsc** is also a tedious task. To solve this, we can use the `watch` option so that every time a TS file is changed, it will be compiled automatically. To run the **tsc** in watch mode, we can use `-w` flag as follows:

```
$ tsc -w
```

This way we can build a basic TypeScript application. In case, if you want to explore more about **tsconfig.js** compiler options, you can visit <https://www.TypeScriptlang.org/tsconfig> for reference.

We have seen building basic TypeScript application, now we will look into OOP concepts which are not supported in JavaScript.

OOP Concepts in TS

TypeScript supports object-oriented programming, where we will learn about class, interface, abstraction, inheritance, encapsulation, and polymorphism. However, before that we need to know about data types in TypeScript. So let us take a look at the supported data types.

Data Types

TypeScript has primitive and non-primitive data types. Primitive data types means value is assigned directly, which are also known as basic data types, whereas non primitive can be reference or subtype of primitive type.

Primitive Types

For Primitive data types, the values are assigned directly. These are also known as basic data types:

- **String:** It is used for textual data that is enclosed with “(double quote) or ‘ (single quote). Example:

```
let name:string = "Tom";  
name = "Jerry";
```

- **Number:** It is used for numeric data. It also supports decimal, binary, hexadecimal, and octal data as well. Example:

```
let total :number = 100;  
let binaryNumber :number = 0b0101;  
let hexaNumber :number = 0xf00d;  
let octalNumber :number = 0o555;
```

- **Bigint:** It is used for Numeric data. It also supports big integers. Example:

```
let bigNumber :bigint = 200n;  
let anotherBigNumber :bigint = BigInt(200);
```

- **Boolean:** It is used when you need either a **true** or **false** case. Example:

```
let mute :boolean = true;  
mute = false;
```

- **Null and Undefined:** It is the same as JavaScript: **null** represents an object having nothing, while **undefined** means type is not initialized. Both are subtypes of other types.

```
let name:undefined = undefined;  
let num:null = null;
```

- **Symbol**: It is a new data type introduced in **ES6**. This creates a unique symbol from the **Symbol** function.

```
let symbolTest = Symbol();
```

Non-Primitive Types

- **Object**: It represents a key-value pair in which key as property and value can be any primitive type. For example:

```
let student :Object = {id: 1, name :"Jack"}; //declare and assign
let employee :{emp_id:number, name:string}; //declare
employee = {emp_id: 2, name :"Jack"}; //assign value
```

- **Array**: It is similar to JavaScript arrays where you can store multiple elements in a single variable. It can be written in two ways:

```
let colors: string[ ] = ["Black", "white", "Red"];
let colors: Array<string> = ["Black", "white", "Red"];
```

First way to define is using the type of elements followed by [] (Square bracket). Second way is to use Array followed by **<typeof elements in Array>**.

- **Enums**: Enumeration is used when possible values are from a set of few fixed constant values. It can be number **enums**, string **enums**, heterogeneous **enums**, or any primitive type **enums**. An example of the **enums** can be for status of a task:

```
enum Status {
  "pending",
  "in_progress",
  "completed"
}
```

- **Tuple**: It represents an array in fixed size with known data types of each element in the array. For example,

```
//correct value need to match
let tupleExample : [string,number] = ["Jack",1]
//gives error
let tupleExampleIncorrect : [string,number] =[1,"Jack"];
```

In this example, the first element should be a string and the second would be a number. Providing them in incorrect order would throw an error.

- **Any:** It is a special type when the value type is not known. In simple words, after specifying any data type, checks are not done and it does not give any error. It is not recommended to use it except in the cases when there is no choice left.

```
let data: any = {};
data.geometry={
  "type": "Polygon",
  "coordinates": [
    [
      [123.61, -22.14], [122.38, -21.73], [121.06, -21.69]
    ]
  ]
}
data.latitude = 90;
data.longitude = 90;
```

- **Custom:** It allows you to define your own custom type using primitive and non-primitive types. For example a type project is created as follows:

```
type project = {
  name: string;
  description: string;
  start_time: number;
  owner_name: string[];
};

const projectData: project = {
  name: 'Project Management',
  description: 'It is about to manage any project from on board to deployment process',
  start_time: 1682516674343,
  owner_name: ["Jack", "John"]
};
```

We have seen most used data types, but there are some other data types such as union, unknown, void, never, functions, and so on, which are used rarely. If you want to explore more about it, then click on this official reference **TypeScript Types** (<https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>).

Now when we know about the data types offered, let us jump to Object Oriented Programming with TypeScript.

Class

Class is a collection of objects which has some common properties that works as components which can be reusable. In simple terms, it is a blueprint of objects. JavaScript supports class after **ES2015**, so TypeScript is also allowed to use it. Class basically contains objects, constructors, and methods.

Let us take an example of a class as follows:

```
export class Book {  
    private title: string;  
    private author: string;  
    private price: number;  
  
    constructor(title: string, author: string, price: number) {  
        this.title = title;  
        this.author = author;  
        this.price = price;  
    }  
  
    // Getter method for title  
    getTitle(): string {  
        return this.title;  
    }  
  
    // Setter method for title  
    setTitle(value: string) {  
        this.title = value;  
    }  
  
    // Getter method for author  
    getAuthor(): string {  
        return this.author;  
    }  
  
    // Setter method for author  
    setAuthor(value: string) {  
        this.author = value;  
    }  
  
    // Getter method for price  
    getPrice(): number {  
        return this.price;  
    }  
  
    // Setter method for price
```

```

setPrice(value: number) {
  this.price = value;
}

displayInfo() {
  console.log(`Title: ${this.title}`);
  console.log(`Author: ${this.author}`);
  console.log(`Price: ${this.price}`);
}

// Create an instance of the Book class
const myBook = new Book('The Great Gatsby', 'F. Scott Fitzgerald',
15.99);

// Use getter and setter methods to update book information
myBook.title = 'To Kill a Mockingbird';
myBook.author = 'Harper Lee';
myBook.price = 1200.99;

// Display updated book information
myBook.displayInfo();

```

Here, we have defined a class named `Book`. Keeping the first letter of class name as capital is a standard practice. We exported that class using keyword **export** so that it can be imported in other files. The class consists of a property called **title**, **author**, **price**, a **constructor**, and methods **get** and **set** respective properties.

A class object is created with a **new** keyword and followed by the class name as function and pass argument for required constructor defined. For example, **new Book('The Great Gatsby', 'F. Scott Fitzgerald', 15.99);**.

In Class, **constructor** is called first automatically. In this case, **constructor** is setting the property **name** with the given values. We can access class properties using **this** keyword anywhere inside the class. In this example, each property (**title**, **author**, and **price**) has a corresponding getter and setter method. These methods provide controlled access to the private properties of the `Book` class.

The output of the preceding code would be:

Output :

```

Title: To Kill a Mockingbird
Author: Harper Lee
Price: $1200.99

```

Inheritance

TypeScript supports inheritance in which one class can be extended by another class in which all its properties and methods are inherited from base class to derived class.

Consider the following example:

```
import { Book } from './class';

export class EBook extends Book {
    private format: string;

    constructor(title: string, author: string, price: number, format: string) {
        super(title, author, price);
        this._format = format;
    }

    // Override displayInfo method to include format
    displayInfo() {
        super.displayInfo(); // Call base class method
        console.log(`Format: ${this._format}`);
    }
}

// Create an instance of the EBook class
const myEBook = new EBook('The Great Gatsby', 'F. Scott Fitzgerald',
15.99, 'PDF');

// Display EBook information
myEBook.displayInfo();
```

The output would be as follows:

```
Title: To Kill a Mockingbird
Author: Harper Lee
Price: $1200.99
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Price: $15.99
Format: PDF
```

As per the preceding example, **EBook** is a child class of **Book** that extends all its properties and methods to derived (**EBook**) class. This means that **EBook** inherits all properties and methods from **Book**. The constructor of **EBook** takes additional parameters (**title**, **author**, **price**, and **format**) compared to the base class. It

calls the constructor of the base class (`super`) with the title, author, and price, and initializes the `_format` property specific to `EBook`. The `displayInfo` method is overridden in `EBook` to include information about the e-book's format. The `super.displayInfo()` call invokes the `displayInfo` method of the base class to display the book's title, author, and price. Then, it logs the format of the e-book. This code demonstrates how to extend a base class (`Book`) to create a subclass (`EBook`) with additional properties and behavior, while also leveraging inheritance and method overriding to reuse and customize functionality from the base class.

Access Modifiers

TypeScript has four different access modifiers for property(field) and method such as `public`, `private`, `protected`, and `static`:

- **Public:** It is the default access modifier which can be used anywhere; as the name `public`, it can be accessed in the same class, child class, or to any other class.

```
class Car {  
  public brand: string;  
  
  constructor(brand: string) {  
    this.brand = brand;  
  }  
  
  public startEngine() {  
    console.log(`Starting the engine of ${this.brand} car. `);  
  }  
}  
  
const myCar = new Car('Toyota');  
console.log(myCar.brand); // Accessing public property  
myCar.startEngine(); // Accessing public method  
  
Output:  
Toyota  
Starting the engine of Toyota car.
```

- **Private:** It is accessible only within the same class.

```
class BankAccount {  
  private balance: number;  
  
  constructor(initialBalance: number) {  
    this.balance = initialBalance;  
  }  
}
```

```

public deposit(amount: number) {
  this.balance += amount;
}

public getBalance() {
  return this.balance; // Accessing private property
}
}

const account = new BankAccount(1000);
console.log(account.getBalance()); // Accessing public method
account.deposit(500);           // Accessing public method
console.log(account.balance);   // Error: Property 'balance'
                                is private

```

- **Protected:** It can be accessible in the same class and all of the child classes only. In the example, the **price** is defined as protected, so it can be called from within the same and child class.

```

class MyBook {
  // Public properties
  public title: string;
  public author: string;
  protected price: number; // Protected property

  constructor(title: string, author: string, price: number) {
    this.title = title;
    this.author = author;
    this.price = price;
  }
}

class EBook extends MyBook {
  private format: string;

  constructor(title: string, author: string, price: number,
  format: string) {
    super(title, author, price);
    this.format = format;
  }

  public displayInfo() {
    console.log(`Price: ${this.price}`);
  }
}

```

```

// Create an instance of the Book class
const myBook = new MyBook('The Great Gatsby',
'F. Scott Fitzgerald', 15.99);

// console.log(myBook.price); // Error: Property 'price' is
protected and only accessible within the class and its
subclasses

// Create an instance of the EBook class
const myEBook = new EBook('The Great Gatsby',
'F. Scott Fitzgerald', 15.99, 'PDF');

// Accessing public properties and method of EBook
myEBook.displayInfo(); // Accessible

Output:
Price: 15.99

```

- **Static:** It can be used directly without creating an object of class. Properties and methods created using **static** belong to the class itself rather than to instances of the class.

```

class User {
  static user_name: string = 'Jack';
  public static calculateWorkingHoursPerMonth(hrsPerDay:
  number) {
    return hrsPerDay * 30;
  }
  public static setUserName(name: string) {
    User.user_name = name;
  }
}
const user = User.user_name;
console.log(`Username = ${user}`);
const totalHrs = User.calculateWorkingHoursPerMonth(8);
console.log(`Total hrs per month = ${totalHrs}`);
User.setUserName("Panchal");
console.log(`Modified Username = ${User.user_name}`);
Output:
Username = Jack
Total hrs per month = 240
Modified Username = Panchal

```

Interface

Interface is an **abstract** and **static** type that represents the behavior of a class which just describes the class and not the actual implementation. It is defined by the keyword **interface**. For example, an interface **IUser** is defined as follows:

```
interface IUser {  
  first_name: string;  
  last_name: string;  
  email_id: string;  
  assigned_project_code: number;  
  assigned_project_name?: string;  
  working_hrs_per_day: number;  
}  
  
const userData: IUser = {  
  first_name: 'Jack',  
  last_name: 'Panchal',  
  email_id: 'yami@gmail.com',  
  assigned_project_code: 1,  
  working_hrs_per_day: 8  
};
```

Note: Using **I** to start an interface name is a standard practice.

An optional property can be declared with a question mark (?). In the preceding example, **assigned_project_name** is marked as optional. **userData** is assigned a value to the interface in which **assigned_project_name** is not specified and it is not giving error but if any other property, for example, **email_id** is missed, TS would throw an error as follows:

```
Property 'email_id' is missing in type '{ first_name: string;  
last_name: string; assigned_project_code: number;  
working_hrs_per_day: number; }' but required in type 'IUser'.
```

We just saw a way to use interfaces to define an object directly. Another way is to create a class and implement the interface as follows:

```
class User implements IUser {  
  first_name: string;  
  last_name: string;  
  email_id: string;  
  assigned_project_code: number;  
  assigned_project_name?: string;  
  working_hrs_per_day: number;  
  
  constructor(first_name: string, last_name: string) {
```

```

        this.first_name = first_name;
        this.last_name = last_name;
    }
}
const newUser = new User("Jack", "Panchal");
newUser.email_id = 'yami@gmail.com';
console.log(`User = ${JSON.stringify(newUser)}`);
Output:
User =
{"first_name": "Jack", "last_name": "Panchal", "email_id": "yami@gmail.co
m"};

```

In the preceding example, we are creating a class `User` by implementing the `IUser` interface. After defining the class, we are creating a `user` variable to create an object of the class `user`. This can be rewritten as:

```
const user:IUser = new User("Jack", "Panchal");
```

Here, we are using interface `IUser` to let typescript know that the variable `user` is of type `IUser` and the `user` can contain only those properties which are declared by the interface `IUser`.

Abstraction

Abstraction is a fundamental concept which allows us to define abstract classes and methods. The methods defined this way do not specify any implementation.

Consider the following example where a `BaseClass` is defined as abstract using the keyword `abstract`. A `setName()` method is also defined as abstract. We can see that there is no implementation provided for this method.

```

abstract class Book {
    protected title: string;
    protected author: string;
    protected price: number;

    constructor(title: string, author: string, price: number) {
        this.title = title;
        this.author = author;
        this.price = price;
    }

    // Abstract method to display book information
    abstract displayInfo(): void;

```

```
}
```

Implementation of the **displayInfo** method is the responsibility of the classes which will extend the **Book** class. Now, let us create another class which extends the **Book** class we just created:

```
// Concrete class representing a PrintedBook, extending Book
class PrintedBook extends Book {
    private format: string;

    constructor(title: string, author: string, price: number, format: string) {
        super(title, author, price);
        this.format = format;
    }

    // Implementation of abstract method to display book information
    displayInfo(): void {
        console.log(`Title: ${this.title}`);
        console.log(`Author: ${this.author}`);
        console.log(`Price: ${this.price}`);
        console.log(`Format: ${this.format}`);
    }
}

// Create instances of books
const printedBook = new PrintedBook('The Great Gatsby', 'F. Scott Fitzgerald', 15.99, 'Hardcover');

// Display book information
printedBook.displayInfo();
```

Output:

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Price: $15.99
Format: Hardcover
```

The example of the class **PrintedBook** did its own implementation of the **displayInfo()** method. Any other class which extends **Book** class can choose their own implementation for the method.

Encapsulation

Encapsulation is hiding internal data and making it available only through some other way such as creating a getter function. In simple terms, it restricts visibility of all actual details of code and displays only the outer layer of code through access modifiers.

Example:

```
class BankAccountClass {  
  
    private accountNumber: string;  
    private balance: number;  
  
    constructor(accountNumber: string, initialBalance: number) {  
        this.accountNumber = accountNumber;  
        this.balance = initialBalance;  
    }  
  
    deposit(amount: number): void {  
        this.balance += amount;  
        console.log(`Deposited ${amount} into account  
        ${this.accountNumber}. New balance: ${this.balance}`);  
    }  
  
    withdraw(amount: number): void {  
        if (amount > this.balance) {  
            console.log("Insufficient funds.");  
        } else {  
            this.balance -= amount;  
            console.log(`Withdrawn ${amount} from account  
            ${this.accountNumber}. New balance: ${this.balance}`);  
        }  
    }  
  
    getAccountInfo(): void {  
        console.log(`Account Number: ${this.accountNumber}, Balance:  
        ${this.balance}`);  
    }  
}  
  
// Create an instance of the BankAccount class  
const myAccount = new BankAccountClass('123456789', 1000);  
  
// Accessing properties and methods using encapsulation  
myAccount.deposit(500); // Deposited 500 into account 123456789. New  
balance: 1500
```

```
myAccount.withdraw(200); // Withdrawn 200 from account 123456789.  
New balance: 1300  
myAccount.getAccountInfo(); // Account Number: 123456789, Balance:  
1300  
// Attempting to access private members directly (will result in  
TypeScript compilation error)  
// console.log(myAccount.accountNumber); // Error: Property  
'accountNumber' is private and only accessible within class  
'BankAccount'.  
// console.log(myAccount.balance); // Error: Property 'balance' is  
private and only accessible within class 'BankAccount'.
```

Output:

```
Original customer details:  
Name: John Doe, Email: john@example.com, Phone Number: 123-456-7890  
Updated customer details:  
Name: Jane Smith, Email: jane@example.com, Phone Number: 987-654-  
3210
```

In the example, the **accountNumber** and **balance** properties are private, and they can only be accessed or modified within the **BankAccount** class itself. Encapsulation ensures that the internal state of the **BankAccount** object is protected, and external code cannot directly manipulate it. Instead, interactions with the object's state are performed through well-defined methods like **deposit**, **withdraw**, and **getAccountInfo**, which encapsulate the underlying data and behavior.

In addition, **accountNumber** and **balance** are private members of class so that is not accessible outside of the class. If anyone tries to set value directly, then it gives an error of not being accessible; so sometimes it provides advantage of security using private members.

```
// Attempting to access private members directly (will result in  
TypeScript compilation error)  
console.log(myAccount.accountNumber); // Error: Property  
'accountNumber' is private and only accessible within class  
'BankAccount'.  
console.log(myAccount.balance); // Error: Property 'balance' is  
private and only accessible within class 'BankAccount'.
```

Polymorphism

In *Polymorphism*, *Poly* means many and *morphism* means form. It is a concept that refers to the ability of objects to take on multiple forms depending on the context.

We can use polymorphism to create classes which implement the same interface or base class. Let us consider a simple example:

```
// Define an interface for a printable item
interface IPrintable {
  print(): void;
}
```

Now let's create two classes implementing the interface:

```
// Implement the Printable interface for a Book class
class Books implements IPrintable {
  constructor(private title: string, private author: string) { }

  // Implement the print method from the Printable interface
  print(): void {
    console.log(`Title: ${this.title}`);
    console.log(`Author: ${this.author}`);
  }
}

// Implement the IPrintable interface for a Document class
class Documents implements IPrintable {
  constructor(private name: string) { }

  // Implement the print method from the Printable interface
  print(): void {
    console.log(`Document Name: ${this.name}`);
    console.log("This is a Printable document.");
  }
}

// Create instances of Book and Document
const book = new Books("The Great Gatsby", "F. Scott Fitzgerald");
const doc = new Documents("Sample Document");

// Call the printItem function with different Printable items
book.print();
doc.print();
```

Output:

```
Title: The Great Gatsby
```

Author: F. Scott Fitzgerald
Document Name: Sample Document
This is a printable document.

In the preceding example, both the **Books** and **Documents** classes implement the **IPrintable** interface, which defines a **print()** method. By implementing the same method with different behavior in each class, we create objects of different types that can be used interchangeably wherever an **IPrintable** is expected.

The **print()** method in each class is specific to the type of **book**, and the actual behavior is determined at runtime based on the specific object being used. This is an example of runtime polymorphism, where the behavior of the method is determined at runtime based on the actual type of the object, rather than at compile-time.

So, when we call **book.print()**, the **print()** method in the **Books** class is executed, and when we call **Document.print()**, the **print()** method in the **Documents** class is executed. This demonstrates the ability of objects to take on multiple forms depending on their specific implementation, which is the essence of polymorphism.

ECMAScript Features

ECMAScript is a standard of JavaScript that was first introduced in 2015; then onwards every year in the month of June, a new version is released. Currently, ES2022 is the latest version for ECMAScript. It is the 13th edition. The name ECMAScript comes from the fact that it was developed by Ecma International, a non-profit organization that was formerly known as the European Computer Manufacturers Association (ECMA).

Let us quickly jump into the features of ECMAScript.

Arrow Functions

Arrow Function is an anonymous function with short syntax compared to vanilla JavaScript. It does not have self-binding so cannot be used as a method or constructor. In normal JavaScript function, "**this**" keyword refers to the function where it is called, whereas in arrow function, "**this**" keyword refers to a global object or its parent object. It is also not a binding of arguments.

Syntax of arrow function is as following:

```
let arrowFunction = (arg1, arg2, ...argN) => {  
  statement(s)  
}
```

Let us take an example of a simple function that just adds two numbers. The normal JavaScript function would be:

```
function sum (num1, num2) {  
    return num1 + num2;  
}
```

The same function in form of arrow functions can be written as:

```
((num1, num2) => {  
    return num1 + num2;  
})
```

Using this Keyword

Consider the following example:

```
let projectName = "PMS";  
let project = {  
    projectName: "Project Management System",  
    normalFunction() {  
        console.log(this.projectName, this);  
    },  
    arrowFunction: () => {  
        console.log(this.projectName, this);  
    }  
};
```

In the preceding example, we have two functions available for the variable `project`. See that we have `projectName` as variable on global scope and `projectName` as a property inside the `project` object.

If we make a call to normal function, the output would be:

```
project.normalFunction();  
// output in browser  
Project Management System, {projectName: 'Project Management  
System', normalFunction: f, arrowFunction: f}
```

The '`this`' keyword here refers to the object `project` and prints the `project` object.

Now let us call the arrow function:

```
project.arrowFunction();  
// Output in browser  
undefined, Window {window: Window, self: Window, document: document,  
name: '', location: Location, ...}
```

Here, in the arrow function, the binding does not happen and the '**this**' keyword refers to the Window object of the browser. In the window object, there is no **projectName** variable, hence we got undefined as the first value in the log.

The behavior would be different for the arrow function if we run the same function in the terminal. Since there is no window object in the terminal, it outputs a blank object.

```
//In Terminal
undefined, {}
```

Using the new Keyword

In JavaScript, the new keyword is used to create a new instance of an object. In the case of arrow function, this behavior is different. If we try to use a new keyword to create an object of an arrow function, we would get a `TypeError`. The arrow functions are designed to be used as expressions rather than constructors.

```
// arrow function with new keyword
const Task = () => {};
const obj = new Task(); // TypeError: Task is not a constructor
Output:
VM37:2 Uncaught TypeError: Task is not a constructor at
<anonymous>:2:13
(anonymous) @ VM37:2
```

Blocking Scopes

In JavaScript, variables can be declared with the "**var**" keyword which can be redeclared and reassigned that has functional scope, whereas in Es6, **let** and **const** are introduced with different usage of scope.

The let Keyword

The variables declared using the let keyword have a blocking scope and these are accessible only in the block where they are declared. These variables can be reassigned new values but not redeclared.

Let us consider the following example:

```
let name = "Project";
let nameIsDeclared = true;
if(nameIsDeclared) {
```

```

let name = "PMS";
console.log(name);
}
console.log(name);
// Output
PMS
Project

```

As per the preceding example, the name variable is declared and assigned with "Project" value which has global scope. After that, the same name variable is declared and assigned with PMS in the block function. In this case, it is redeclared and does not give an error due to changing its scope from global to block.

However, if it is redeclared in the same global scope, then it gives an error as **caught SyntaxError: Identifier 'name' has already been declared.**

The const Keyword

The **const** keyword has also a blocking scope, same as **let**, but it cannot be reassigned to a new value. Its value remains constant as the name suggests.

In the following example, we are trying to re-assign a value to the **const name** and that is not allowed, hence we get a compile time error:

```

const name = "PMS";
name = "Project Management system";
Output:
VM1717:2 Uncaught TypeError: Assignment to constant variable.
at <anonymous>:2:6

```

In case of variables holding objects, the behavior is a bit different. We know that Object is one kind of reference. If a const is created to hold the object, its properties can be changed, but the whole object cannot be replaced with another.

Let us consider the following example:

```

const Task = {
  name: "Insert Data to database",
  efforts_hrs: 10
};
Task.name = " Fetch Data from database";
console.log(Task.name);

```

Output:

Fetch Data from database

In this case, the name property of Task could be changed.

Template Literals

ECMAScript supports template string which has syntax backticks (` `) enclosed with string. Inside the string we can use the dollar (\$) sign to put a variable or an expression, which will be evaluated and its value will be printed.

```
let name = "PMS";
let task = "Insert data to database";
console.log("Task => " + task + " and name => " + name); // without
template
console.log(`Task => ${task} and name => ${name}`); // with
template
```

This avoids the use of + (plus) sign to concatenate multiple strings or variables.

Classes

ES6 allows the use of object-oriented concepts and classes are one of them. A class consists of objects, methods, and constructor. Classes can be declared with the class keyword.

Syntax :

```
class class_name { }
```

Example:

```
class Project {
  constructor() { }
}
```

We already have seen details of class in OOPs concept earlier in this chapter.

Promises

Before the introduction of **ES6**, callbacks were used to perform or handle asynchronous tasks. **Callback** is a type of function that gets called after the execution of the actual function is done. When there are multiple nested callbacks, it creates a situation called callback hell. To solve this problem, **Promises** were introduced.

The syntax of a promise is:

```
new Promise((resolve, reject) => {...});
```

A promise has following three states:

- **pending**: It is the initial state that is before the reject or resolve state.
- **reject**: It is the state when failure happens.
- **resolve**: It is the state when execution completes successfully.

Let us consider the following example:

```
const example = new Promise((resolve, reject) => {
  try {
    const options = {
      body: JSON.stringify({test:"test"}),
      headers: { 'Content-Type': 'application/json' },
      method: 'POST',
    };
    const url = `https://www.google.com`;
    fetch(url, options)
      .then(async (response) => {
        console.log(`Got response - ${await response.text()}`);
        resolve(response);
      })
      .catch((err) => {
        console.log('Catch => ', err);
        reject(err);
      })
      .finally(() => {
        console.log("Finally");
      });
  } catch (error) {
    console.log('Catch 2 => ', error);
    reject(error);
  }
});
```

In the preceding example, when the result of fetch is returned, based on the good or bad result, we can call either **resolve()** or **reject()**. As we can see that on successful results, we are making a call to **resolve()**, and in case there is an error caught, we are calling **reject()**.

Using the syntax of promise is also now outdated. Instead of promises, **async await** is used for handling asynchronous requests. The same function can be rewritten using **async await** as:

```

async function example() { // line 1
  try {
    const options = {
      body: JSON.stringify({test: "test"}),
      headers: {'Content-Type': 'application/json'},
      method: 'POST'
    };
    const url = 'https://www.google.com';
    const response = await fetch(url, options);
    console.log(`Got response - ${await response.text()}`);
    return response;
  } catch (error) {
    console.log('Catch => ', error);
    throw error;
  } finally {
    console.log('Finally');
  }
}

```

To make a function use `async-await`, we need to declare the function as `async` similar to what we have done in line 1. Now, we can use `await` for the calls for which we want to wait. In the preceding example, we have used `await` before `fetch()`. The execution is going to wait for the fetch to complete and return with some result.

Destructuring

Destructuring allows us to make properties of objects and values of arrays into identical variables.

Object Destructuring

Let us consider the following example:

```

const obj = { a: 1, b: 2 };
const { a, b } = obj;
console.log(a);
console.log(b);

```

Output:

```

1
2

```

In this example, we could declare two variables **a** and **b** directly from the object's properties.

Array Destructuring

Let us consider the following example to understand how it works:

```
const numbers = ["1", "2", "3"];
const [red, yellow, green] = numbers;
console.log(red); // "1"
console.log(yellow); // "2"
console.log(green); // "3"
```

Output:

```
1
2
3
```

In the case of an array, the values can be assigned to the variables in the same order as the values appear in the array.

Default Parameters

In ES6, it is possible to set the default value to any variable through the parameter of function.

The syntax for setting the default value is following:

```
function fnName(param1 = defaultValue1, /* ..., */ paramN =
defaultValueN) {
  // ...
}
```

Let us define a function to add two values:

```
function sum(num1, num2=5) {
  let sum = num1 + num2;
  return sum;
}
```

The preceding function can be called as:

```
sum(1, 3) // returns 4
sum(3) // returns 8
```

The function can be called without specifying a value to the default parameters.

Modules

ES6 provides export and import facilities. In this any written JavaScript code can be transferred from one file to another file as a module for reuse as export from one file to import in another file.

Let us create a file named project.js as:

```
export class Project {  
  //objects  
  //constructor  
  //methods  
}
```

The class Project is exported using the export keyword. Let us create another file main.js in which we can import the class Project as:

```
import {Person} from './project.js'
```

Now, all properties and functions can be used in the main.js file. This way export-import is possible in JavaScript after ES6 was introduced.

Enhanced Object Literals

In ES6, it is possible to make dynamic properties for objects effectively. It also enables shorthand syntax for initializing objects.

Let us consider the example of a function **getTask()** in regular JavaScript:

```
// regular JavaScript  
function getTask(name, hrs) {  
  return {  
    name: name,  
    hrs: hrs  
  }  
}  
getTask("Task 1", 10);  
Output :  
{name: 'Task 1', hrs: 10}
```

We can rewrite the same function in **ES6** as follows:

```
//ES6  
function getTask(name, hrs) {  
  return { name, hrs };  
}
```

```
getTask("Task 1", 10);
```

Output:

```
{name: 'Task 1', hrs: 10}
```

Now let us add some dynamic properties:

```
let name = "t";
let i = 0;
const task = {
  [name + ++i]: "Add",
  [name + ++i]: "Update",
  [name + ++i]: "Delete"
};
```

We added three more properties to the task object as **t1**, **t2**, and **t3**:

```
console.log(task.t1);
console.log(task.t2);
console.log(task.t3);
```

Output:

```
Add
Update
Delete
```

These were the most used features of **ES6**. There are some other features which can be explored and be used with the help of various examples through reference from the official website of **ECMAScript**: <https://262.ecma-international.org>.

EsLint

EsLint is one of the best tools that checks JavaScript and ECMAScript code based on a set of rules defined and highlights what is not done correctly as per the rules. All major IDEs support **eslint**, and if that rule is violated, the developer is notified immediately. This helps to save a lot of development efforts.

Installing Eslint

To install **EsLint** in your project, simply install the following packages:

```
npm install eslint @typescript-eslint/parser
@typescript-eslint/eslint-plugin --save-dev
```

Configuring Eslint

Let us create a file named `.eslintrc` in the root directory of the project. Now save the following code in the `.eslintrc` file with defined rules and parser options:

```
{  
  "env": {  
    "browser": true,  
    "es2021": true,  
    "node": true  
  },  
  "parser": "@typescript-eslint/parser",  
  "parserOptions": {  
    "ecmaVersion": 12,  
    "sourceType": "module"  
  },  
  "plugins": [  
    "@typescript-eslint"  
  ],  
  "extends": [  
    "eslint:recommended",  
    "plugin:@typescript-eslint/recommended"  
  ],  
  "rules": {  
    "@typescript-eslint/explicit-module-boundary-types": "off",  
    "@typescript-eslint/no-unused-vars": "error",  
    "no-console": "warn",  
    "indent": ["error", 2],  
    "quotes": ["error", "single"],  
    "semi": ["error", "always"]  
  }  
}
```

Running Eslint

Although all IDEs will immediately start evaluating the code based on the provided `.eslintrc` file, we can run `eslint` manually in the terminal as well.

Open the `package.json` file of the project and save the following lint script:

```
"scripts": {  
  ...  
  "lint": "eslint . --ext .ts"
```

```
}
```

Now we can run **lint** script:

```
$ npm run lint
```

```
yarminni@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ npm run lint

> basic-typescript-project@1.0.0 lint
> eslint . --ext .ts

/home/yarminni/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/main.ts
  1:14  error  Require statement not part of import statement  @typescript-eslint/no-var-requires
  6:1  error  Expected indentation of 2 spaces but found 4  indent
  7:1  error  Expected indentation of 2 spaces but found 4  indent
  8:1  error  Expected indentation of 2 spaces but found 4  indent
 12:1  error  Expected indentation of 2 spaces but found 4  indent
 12:5  warning  Unexpected console statement  no-console

/home/yarminni/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/oops/abstraction.ts
  3:1  error  Expected indentation of 2 spaces but found 4  indent
  4:1  error  Expected indentation of 2 spaces but found 4  indent
  5:1  error  Expected indentation of 2 spaces but found 4  indent
  7:1  error  Expected indentation of 2 spaces but found 4  indent
  8:1  error  Expected indentation of 4 spaces but found 8  indent
  9:1  error  Expected indentation of 4 spaces but found 8  indent
 10:1  error  Expected indentation of 4 spaces but found 8  indent
 11:1  error  Expected indentation of 2 spaces but found 4  indent
 19:1  error  Expected indentation of 2 spaces but found 4  indent
 21:1  error  Expected indentation of 2 spaces but found 4  indent
 22:1  error  Expected indentation of 4 spaces but found 8  indent
 23:1  error  Expected indentation of 4 spaces but found 8  indent
 24:1  error  Expected indentation of 2 spaces but found 4  indent
 26:1  error  Expected indentation of 2 spaces but found 4  indent
 27:1  error  Expected indentation of 2 spaces but found 4  indent
 28:1  error  Expected indentation of 4 spaces but found 8  indent
 28:9  warning  Unexpected console statement  no-console
 29:1  error  Expected indentation of 4 spaces but found 8  indent
 29:9  warning  Unexpected console statement  no-console
 30:1  error  Expected indentation of 4 spaces but found 8  indent
 30:9  warning  Unexpected console statement  no-console
 31:1  error  Expected indentation of 4 spaces but found 8  indent
```

Figure 2.5: Run Eslint

Some of the **Eslint** issues are auto-fixable and we can fix those by using **-- fix** option in script:

```
"scripts": {
  ...
  "lint": "eslint . --ext .ts --fix"
}
```

```
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter2/basic-typescript-project$ npm run lint
> basic-typescript-project@1.0.0 lint
> eslint . --ext .ts --fix

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/main.ts
 12:3  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/abstraction.ts
 28:5  warning  Unexpected console statement  no-console
 29:5  warning  Unexpected console statement  no-console
 30:5  warning  Unexpected console statement  no-console
 31:5  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/class.ts
 43:5  warning  Unexpected console statement  no-console
 44:5  warning  Unexpected console statement  no-console
 45:5  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/encapsulation.ts
 13:5  warning  Unexpected console statement  no-console
 18:7  warning  Unexpected console statement  no-console
 21:7  warning  Unexpected console statement  no-console
 26:5  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/inheritance.ts
 14:5  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/interface.ts
 10:7  error    'userData' is assigned a value but never used  @typescript-eslint/no-unused-vars
 34:1  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/modifiers/private.ts
 18:1  warning  Unexpected console statement  no-console

/home/yammini/projects/book/kriti/docs/chapter2/basic-typescript-project/lib/cops/modifiers/protected.ts
```

Figure 2.6: Fix Eslint Issue

This is helpful when there are small errors like missing semicolon, extra whitespace, and so on.

As shown above, some fixes need to be done manually here. For demonstration purpose, we did not remove the console statements, which need to be manually removed or an eslint rule should be added.

That is overview of how **Eslint** is used in project for better development; if you need to explore more about it, follow the reference EsLint Documentation (<https://eslint.org/docs/latest/>)

Earlier, there was a typescript linting tool named **tslint**, which is now deprecated. The plan is to use **Eslint** for typescript projects as well. The roadmap can be tracked at <https://github.com/palantir/tslint/issues/4534>.

Conclusion

In this chapter, we learned about basic TypeScript with examples, its installation, and configuration. We learned about the OOP concepts along with what is offered by ECMAScript.

In the next chapter, we will learn Express.js, the most used framework of Node.js. Express.js will help us to build the APIs we need in the chapters ahead in the book.

Multiple Choice Questions

1. What is TypeScript?
 - a. A JavaScript library for creating user interfaces
 - b. A statically typed superset of JavaScript
 - c. A database management system
 - d. A programming language for server-side development
2. Which of the following is a benefit of using TypeScript?
 - a. It runs in a web browser without any compilation
 - b. It enforces strict typing rules to catch errors early
 - c. It provides built-in support for database connections
 - d. It can be used for mobile app development
3. How is TypeScript code compiled into JavaScript?
 - a. It is interpreted by the browser
 - b. It is manually transcribed by developers
 - c. It is compiled using the TypeScript compiler (tsc)
 - d. It is converted by a built-in JavaScript converter
4. Which command is used to compile a TypeScript file (for example, `app.ts`) into JavaScript?
 - a. `ts compile app.ts`
 - b. `tsc app.ts`
 - c. `typescript app.ts`
 - d. `transpile app.ts`
5. Which TypeScript feature allows you to define reusable types for complex data structures?
 - a. Type annotations
 - b. Type inference
 - c. Interfaces and custom types
 - d. Polymorphism
6. Which OOP concept promotes the idea of bundling data and methods that operate on that data into a single unit called an object?

- a. Inheritance
 - b. Encapsulation
 - c. Polymorphism
 - d. Abstraction
7. Which OOP principle states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program?
- a. Encapsulation
 - b. Abstraction
 - c. Inheritance
 - d. Polymorphism
8. Which ES6 feature allows you to unpack elements from arrays or properties from objects into distinct variables?
- a. Destructuring Assignment
 - b. Object Literals
 - c. Spread Operator
 - d. Template Literals
9. Which ES6 feature allows you to create multi-line strings with embedded expressions?
- a. Template Strings
 - b. String Templates
 - c. Multi-line Strings
 - d. String Literals
10. Given an array of numbers, how do you create a new array with all the elements of numbers followed by 5 using the spread operator?
- a. [...numbers + 5]
 - b. [...numbers, 5]
 - c. [numbers + ...5]
 - d. [5 ...numbers]

Answers

1. b
2. b
3. c
4. b
5. c
6. b
7. d
8. a
9. a
10. b

Further Reading

<https://www.typescriptlang.org>

OceanofPDF.com

CHAPTER 3

Overview of Express.js

Introduction

Express.js is a highly recognized and frequently utilized open-source framework of Node.js that facilitates the creation of web applications and REST APIs. It is a robust and adaptable framework that is favored by developers seeking to construct efficient and expandable web applications.

It is a very popular framework due to its simplicity, flexibility, and scalability. Its popularity can be observed by looking at the average downloads per week. According to the npm registry, the Express.js package has been downloaded on average over 27 million times per week.

Structure

In this chapter, we will discuss the following topics:

- Defining Express.js
- Advantages and Limitations of Express.js
- Express.js Installation and Creating a Basic Application
- Core Features of Express.js
- Security and Performance Best Practices

Defining Express.js

In software development, a **framework** is a pre-written code that provides a set of generic functionality, tools, and guidelines for building applications. It is essentially a structured and standardized way of organizing and developing software, which helps to reduce development time and effort by providing pre-existing components and patterns that can be reused across multiple projects.

Express.js is one of the most popular open source, fast, and flexible frameworks of Node.js. It follows the “*unopinionated*” approach, which means that it does not enforce any specific architecture or patterns and allows developers to build their applications using their preferred tools and techniques. It provides a set of tools

and features for building web applications and APIs using Node.js, including handling HTTP requests and responses, routing, middleware, templating engines, static file serving, and more. It has a large and active community of developers contributing to its development and maintenance.

Express.js is also known for its performance and scalability, with a lightweight and efficient core that allows it to handle high-traffic loads. Overall, Express.js is a powerful framework of Node.js that has all the required features to develop secure and scalable web applications.

Advantages of Express.js

Express.js has several advantages over other web application frameworks, including:

- **Minimalist**

Express.js has a straightforward design that helps to reduce the learning curve for developers who are new to the framework, enabling them to get up and running with their projects more quickly and easily.

- **Flexible**

Express.js is a highly customizable and flexible framework. One such example of its flexibility is the middleware system, which allows developers to add custom logic to incoming requests or outgoing responses.

- **Scalable**

Express.js provides built-in support for asynchronous programming using JavaScript promises and `async/await` syntax, which enables developers to write scalable code that can handle a large number of concurrent requests. It is well-suited for building large, complex applications that can handle high levels of traffic and data.

- **Compatibility with Node.js**

Express.js is built specifically to work with Node.js, which means that it is highly compatible with Node.js and its related libraries. Express.js is designed to leverage the features and capabilities of Node.js, such as its event-driven architecture and non-blocking I/O model. It is also able to seamlessly integrate with other Node.js libraries and tools.

- **Collaborative community**

It has a very large and active community which is a key advantage of it. It ensures that the framework is constantly evolving and improving. Moreover,

it provides valuable support for developers, making it easier for them to troubleshoot issues and learn from the experience of others.

Express.js not only offers the aforementioned benefits, but also provides numerous additional advantages.

Limitations of Express.js

Like every coin has two sides, Express.js also has its limitations as follows:

- **Incompatible with client-side application**

Express.js is primarily focused on server-side web development, so it is not used for building complex client-side applications. However, it can be used in conjunction with other tools and frameworks to build full-stack applications.

- **Lack of built-in features**

Express.js is a minimalist framework, which means that it does not come with all the built-in features and tools that some other frameworks provide. Developers may need to install and configure additional modules or libraries to add certain functionalities to their applications.

- **Inconsistency**

Since Express.js is a minimalist framework, it does not enforce any standard way to structure an application or organize its code. As a result, developers are free to design their structures, which can pose a challenge for new developers trying to contribute to any existing project.

Express.js has fewer limitations compared to other frameworks and offers more benefits, making it an easily approachable framework for developers.

Express.js Installation and Creating a Basic Application

To install Express.js, prior make sure you have Node.js installed on your machine, then you can perform the following steps:

1. Create a new project directory and navigate into it using the command prompt or terminal.

```
$ mkdir my-express-app  
$ cd my-express-app
```

Initialize a new Node.js project. This will create a **package.json** file in your project directory.

```
$ npm init
```

2. Install **Express.js** and typescript dependency with run the following command:

```
$ npm install express typescript --save
```

The **--save** option will automatically update your **package.json** file with the installed package and its version.

3. Install development dependency as dev dependency for typescript with express.js

```
$ npm install @types/express @types/node --save-dev
```

4. Create a new file called **app.ts** in the root directory of your project, and add the following code:

```
import * as express from 'express';
import { Request, Response } from 'express';
const app: express.Application = express();

app.get('/', (req: Request, res: Response) => {
  res.send('Hello World!');
});

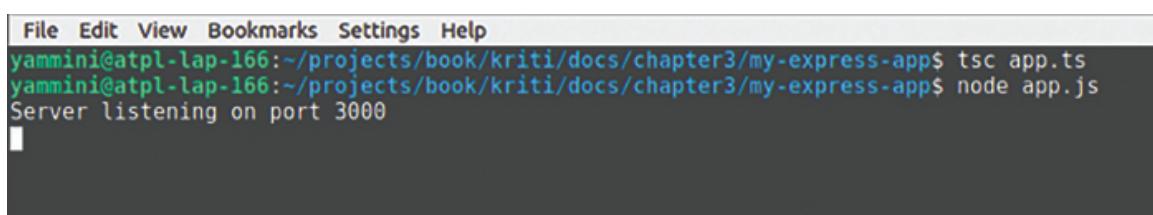
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

5. Compile the TypeScript code into JavaScript using the **tsc** command:

```
$ tsc app.ts
```

6. Run the server with the **node** command:

```
$ node app.js
```



```
File Edit View Bookmarks Settings Help
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ tsc app.ts
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ node app.js
Server listening on port 3000
```

Figure 3.1: Compile and Run Express Application

Now you should be able to visit **http://localhost:3000** in your web browser and see the message "**Hello World!**" displayed on the page:

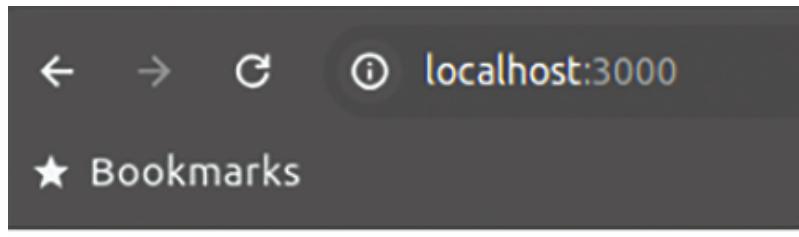


Figure 3.2: Launch Express Application

Core Features of Express.js

Express.js is equipped with several fundamental features, some of which are as follows:

REST APIs

A RESTful API is a web-based API that follows the principles of Representational State Transfer (REST) architectural style. It is a way of designing web services that are lightweight, maintainable, and scalable. RESTful APIs use HTTP methods to interact with resources that are identified by URIs.

There are certain principles which should be followed while building REST APIs. These principles are a set of guidelines for designing an application.

REST Principles

There are a total of six guiding principles. In general, not all of these are mandatory to be followed to build applications; however, using these principles ensures better performance, efficiency, and scalability. These principles are:

- **Client-Server Architecture**

The system should be divided into client and server. The client should communicate to the server over the network. Server-side and Client-side responsibilities must be independent and to be implemented by respective sides. This allows the evolution of both client and server independently.

- **Stateless Design**

Whenever a client needs any data from the server it sends a request to the backend server. This principle says that each request from client to server

must contain all of the information that server would need to understand the request. Server must not store the state of the session about the client.

- **Cacheable**

Sometimes, there are responses which do not change too frequently. Such responses can be cached to improve performance. Responses should be defined as cacheable so that client can also know if it can reuse the same data or should request again.

- **Uniform Interface**

The uniform interface simplifies the architecture by making it more modular, and allows easier development and deployment. There are four constraints which define a uniform interface :

- **Resource Identification:** Resources can be identified through request URIs. For example, /projects clearly says that we are requesting the list of projects. A URI /projects/23 says that we are requesting a project using unique id 23.
- **Resource Manipulation through Representations:** A resource is a conceptual entity identified by URI and the representation is the form of the resource when it is transferred over the network. The representation can be as JSON, XML, HTML, and so on. By sending or receiving these representations, clients manipulate resources.

A project can be represented (in JSON format) as

```
{  
  "name": "Mobile App",  
  "description": "This project is to manage development of  
  Mobile App"  
}
```

- **Self-descriptive Messages:** The messages being transferred between server and client should include enough information to describe how to process the message. This helps in decoupling of the client and server. Let us consider a request:

```
GET /projects/23 HTTP/1.1  
Host: example.com  
Accept: application/json
```

The above can be a request from client to server. This clearly shows that we are making a GET request over HTTP for host example.com and it desires the response in JSON format using the Accept header. Let us see a sample response to this:

```

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 122
{
  "id": 23,
  "name": "Mobile App",
  "description": "This project is to manage development of
  Mobile App"
}

```

The preceding response shows that the status of the request was **200 OK**, the returned data is in JSON format and the body contains the requested project.

- **Hypermedia as the Engine of Application State (HATEOAS):** The principle says that the client should interact with a RESTful application entirely through the hypermedia provided dynamically by the application servers. The client would have only the initial URI of the application. The hyperlinks needed further should be inside the response. This allows the dynamic discovery of actions and helps to decouple the client from the server. Each communication here would have self-descriptive messages.

- **Layered Architecture**

This principle insists that the application architecture should be divided into hierarchical layers. Each layer performs specific tasks. Let us consider a simple web application. The layers in it can be:

- **Client Layer:** User interacts with this layer, for example, web app or mobile app.
- **API Gateway Layer:** Entry point, every request goes through this layer.
- **Application Layer:** Handles the business logic of processing the user requests.
- **Service Layer:** Contains helper services such as notification service, and so on.
- **Data Access Layer:** Contains logic needed to fetch, store or update data.
- **Database Layer:** The layer which communicates with the database to fetch or store data.

These layers should give an idea about the principle. This is an example and the layers can vary depending on the system being developed.

- **Code on Demand**

This is an optional principle. This allows the client to extend the client functionality by providing the code in response. In this case, the client makes a request and server responses with a code which is usually a script which can be run at the client side. This principle allows flexibility and on-the-fly customization of the client application. However, we must be careful and should consider the security aspects.

While it is true that REST principles enhance the scalability, performance, and maintainability of the APIs, not each principle is mandatory in all contexts. The last principle — Code on Demand is optional. However, to achieve the full benefits of the REST architecture, it is recommended that these principles should be followed as closely as possible.

Building REST API

Express.js is well-suited for building RESTful APIs, with support for HTTP. It allows developers to easily handle HTTP requests and responses. There are different types of HTTP methods such as **GET, POST, PUT, DELETE, PATCH, HEAD**.

Let us build a rest api to get a list of users using the GET Method of HTTP.

We have already created a basic-typescript-project in the previous chapter, so let us take that as a starting point and install **express** in that project.

```
$ npm install express --save
```

In a REST API **body-parser** is a very useful npm package that is used as Node.js parsing middleware. It extracts the body portion of the incoming request and parses it based on the **Content-Type** request header. The parsed body data is then made available through the **req.body** property. Let us install it using the npm package manager:

```
$ npm install body-parser --save
```

Additionally, let us install the TypeScript definitions for Express and **body-parser** as dev dependencies:

```
$ npm install -D @types/express @types/body-parser
```

Now, let us update the **main.ts** file in the root directory of the project with the following code:

```
import express from 'express';
```

```

import * as bodyParser from 'body-parser';
import { users } from "./users/user";
import { Application } from 'express';
const app: Application = express();
app.use(bodyParser.json());
app.get('/api/users', (req, res) => {
  res.json(users);
});
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});

```

Next, create a **user.ts** file in the user directory inside the **lib** directory and put the following code:

```

interface User {
  id: number;
  name: string;
  email: string;
}

export const users: User[] = [
  { id: 1, name: 'John', email: 'john@example.com' },
  { id: 2, name: 'Jane', email: 'jane@example.com' },
];

```

To run the server, we need to compile the TypeScript code with **tsc** and start the server with node:

```

$ tsc
$ node dist/main.js

```

Now we can test the API using a tool like Postman/curl or directly open the browser with url `http://localhost:3000/api/users`, it gives the following JSON output:

```
[{"id":1,"name":"John","email":"john@example.com"}, {"id":2,"name":"Jane","email":"jane@example.com"}]
```

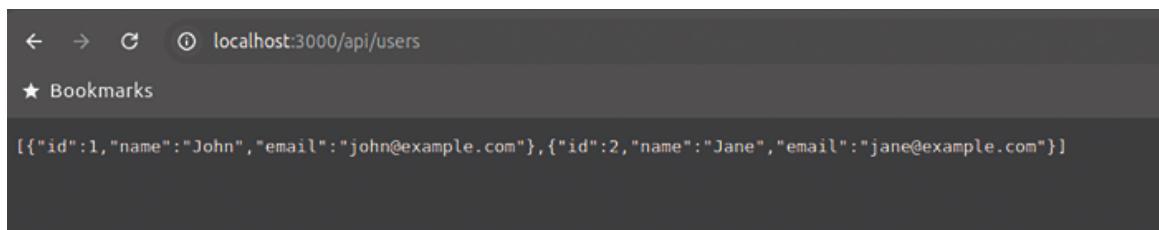


Figure 3.3: Get API Users

In the preceding example, we created an API endpoint `/api/users` which returned us the users list. We also used `body-parser` middleware. We will learn about middlewares later in this chapter.

Routing

Express.js provides a simple and flexible routing system that allows developers to define URL routes for handling incoming HTTP requests. Routing in Express.js refers to the mechanism of defining and handling endpoints (URL paths) for web applications and APIs.

It is a crucial aspect of any web framework, as it helps to determine how the application responds to client requests. In Express.js, routing is accomplished using the `express.Router()` class, which creates modular, mountable route handlers. The router consists of route method, route path, and callback handler.

`app.METHOD(PATH, HANDLER)`

app : It is an instance of express.

METHOD : It is an HTTP request method, in lowercase.

PATH : It is a path on the server.

HANDLER : It is the function executed when the route is matched.

Route Methods

There are most commonly used methods are as follows:

GET : It is used for retrieving data.

POST : It is used for creating or adding new data.

PUT : It is used to update an existing data.

DELETE : It is used to delete data.

PATCH : It is used to partially update an existing data.

OPTIONS : It is used to retrieve information about available options for data.

HEAD : It is similar to the `GET` method but only retrieves the response headers without the response body.

You can also use `app.all()` to handle all HTTP methods.

```
app.all('/', (req, res, next) => {
  console.log('all method...')
  next() // pass control to the next handler
```

```
})
```

Route Paths

A path can be string, string pattern, or a regular expression.

This route path will match requests to **/users** specific string.

```
app.get('/users', (req, res) => {
  res.send('users')
})
```

This route path will be matched with string patterns such as **abcd**, **abbcd**, **abbcd**, and so on.

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

This route path will match blueberries and strawberries, raspberries, but not **blueberriesfruit**, **strawberriesfruit**, and so on.

```
app.get('.*berries$', (req, res) => {
  res.send('/.*berries$')
})
```

These are different ways of defining route paths.

Route Parameters

Express.js treats certain characters differently in string-based paths compared to their regular expression counterparts.

For example, ?, +, *, and () are all subsets of their regular expression counterparts. On the other hand, the hyphen (-) and the dot (.) are interpreted literally when used in string-based paths.

Consider the following example:

```
app.get('/projects/:projectCode', function(req, res) {
  var projectCode = req.params.projectCode;
  // Do something with the project code
  res.send('project code: ' + projectCode);
});

app.get('/users/:user-email', function(req, res) {
  var userEmail = req.params['user-email'];
  // Do something with the user email
  res.send('User email: ' + userEmail);
```

```
});

app.get('/files/:file_name.pdf', function(req, res) {
  var fileName = req.params['file_name'];
  // Do something with the file name
  res.send('File name: ' + fileName);
});
```

In this example, there are three routes with hyphens and dots in their parameters. The first route `/projects/:projectCode` accepts a project code parameter, which can contain hyphens. The second route `/users/:user-email` accepts a user email parameter, which can contain hyphens. The third route `/files/:file_name.pdf` accepts a file name parameter, which can contain hyphens and ends with `.pdf`.

Route Handlers

When a route is matched in Express.js, it can have one or more handler functions associated with it, which are executed. Route handlers are responsible for processing requests, accessing data, and returning responses to the client.

Let us take the following example:

```
app.get('/api/users', (req, res) => {
  res.json(users);
});
```

In this example, the route handler function is `(req, res) => {...}`, which is executed when a **GET** request is received with a URL path that matches the pattern `/api/users`. The `req` parameter contains information about the incoming request, such as the request headers and parameters, while the `res` parameter is used to send a response back to the client.

Middleware

Express.js supports middleware functions that can modify incoming requests or outgoing responses as per the required custom logic. Middleware in Express.js refers to a series of functions that are executed in a specific order when a client sends a request to a server. These functions have access to the request and response objects and can modify them as needed. Middleware functions can be used for a variety of purposes such as logging, authentication, error handling, and more.

In Express.js, middleware functions can be added to the application or specific routes using the `use()` method.

Previously we already used body-parser as middleware while building rest api. This middleware is for each route as passed in `app.use()`, for example.

```
app.use(bodyParser.json());
```

There is another **example of middleware in which you do not wish to use it** for each route such as user validation on a specific route. They are executed in the order in which they are added and can be chained together using the `next()` function to pass control to the next middleware function in the stack.

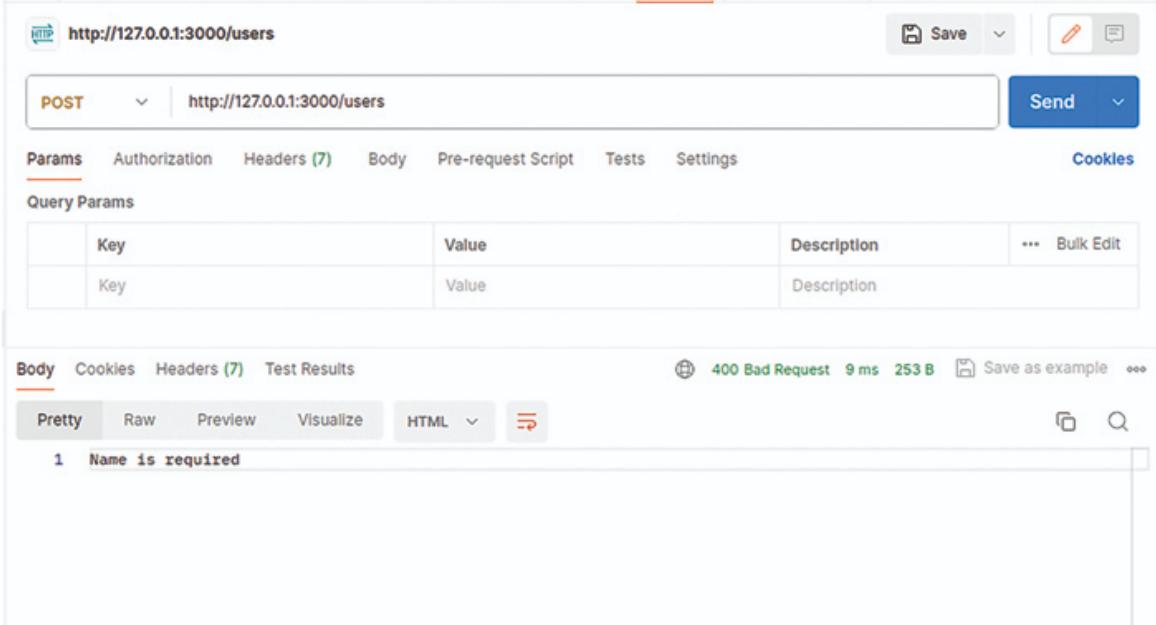
```
// Validation middleware
const validate = (req, res, next) => {
  const { name } = req.body;

  if (!name) {
    return res.status(400).send('Name is required');
  }
  next();
};

// Route
app.post('/users', validate, (req, res) => {
  const { name } = req.body;
  res.send(`Hello, ${name}!`);
});
```

In this example, the validate middleware function is defined to check if the name parameter is present in the body string. If it is not present, the middleware sends a 400 Bad Request response with an error message. If the name parameter is present, the middleware calls the `next()` function to pass control to the next middleware or route handler.

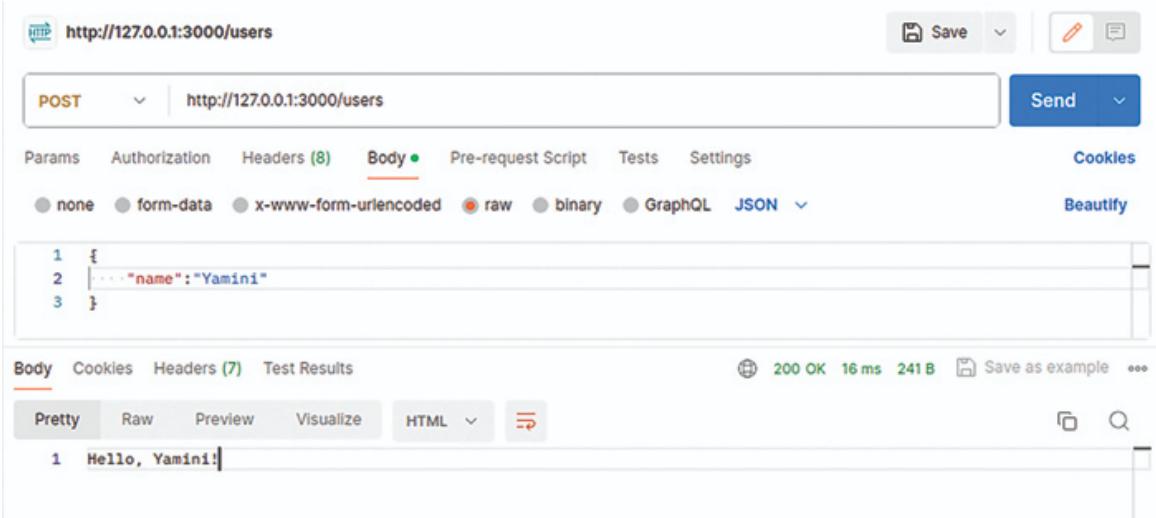
The validate middleware is then used in the `'/user'` route handler as the second argument to ensure that the name parameter is present before generating a user's response.



The screenshot shows the Postman interface for a POST request to `http://127.0.0.1:3000/users`. The 'Body' tab is selected, showing a JSON payload with a single field: `1 { 2 "name": "Yamini" 3 }`. The 'Headers' tab shows `Content-Type: application/json`. The 'Test Results' tab displays an error message: `1 Name is required`. The status bar at the bottom indicates a `400 Bad Request` response with `9 ms` and `253 B` size.

Figure 3.4: Post API User Bad Request

[**Figure 3.5**](#) shows the result "Hello, Yamini!":



The screenshot shows the Postman interface for a POST request to `http://127.0.0.1:3000/users`. The 'Body' tab is selected, showing a JSON payload with a single field: `1 { 2 "name": "Yamini" 3 }`. The 'Headers' tab shows `Content-Type: application/json`. The 'Test Results' tab displays the response message: `1 Hello, Yamini!`. The status bar at the bottom indicates a `200 OK` response with `16 ms` and `241 B` size.

Figure 3.5: Post API User Valid Input Name

Error Handling

Error handling is an important aspect of building robust applications. The framework provides a few ways to handle errors:

Built-in Error Handling

Express.js provides a built-in error-handling middleware function that can be used to handle errors in the application. This middleware function can be used to catch any unhandled errors that occur during the execution of the application.

In Express.js, another way to handle errors is by utilizing middleware functions with error-first callbacks or functions. Here is an example demonstrating how it can be implemented:

```
import express, { Request, Response, NextFunction } from 'express';
const app = express();
app.get('/', (req: Request, res: Response) => {
  throw new Error('Oops! Something went wrong.');
});
app.use((err: Error, req: Request, res: Response, next: NextFunction) => {
  res.status(500).send('Something went wrong!');
});
app.listen(3000, () => {
  console.log('Server listening on port 3000!');
});
```

In this example, we have defined an Express app with a single route that throws an error. We then use the built-in error handling middleware function `app.use` to catch the error and send a **500** status code with a message to the client.

The error-handling middleware function takes four arguments: `err`, `req`, `res`, and `next`. The first argument (`err`) is the error that was thrown, the second argument (`req`) is the request object, the third argument (`res`) is the response object, and the fourth argument (`next`) is a function that is used to pass control to the next middleware function in the stack.

If an error is thrown within any middleware or route handler function, Express.js will automatically call the error handling middleware function with the thrown error object as the first argument.

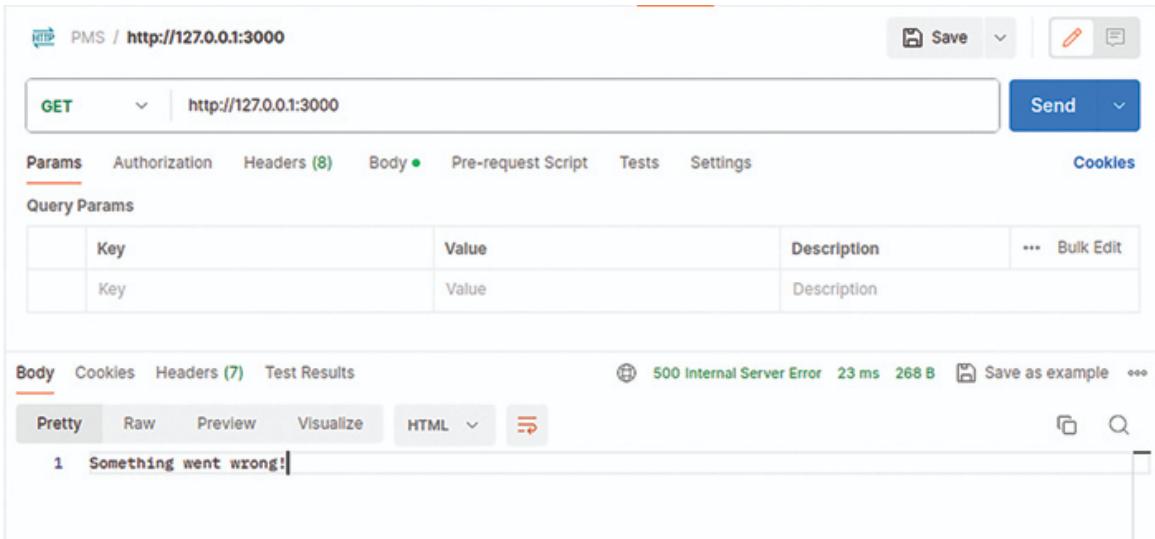


Figure 3.6: Error Handling

Custom Error Handling

Developers can also create their own custom error-handling middleware to handle specific types of errors. This middleware function can be added to the middleware stack and used to catch errors that are specific to the application.

Let us update the `main.ts` file with the following code:

```
import express, { Application, Request, Response, NextFunction } from 'express';
import { HttpException, NotFoundException } from './utils/errorHandler';
import * as bodyParser from 'body-parser';
import { users, Users } from "./users/user";
const app: Application = express();
app.use(bodyParser.json());
app.get('/api/users', (req, res) => {
  res.json(users);
});
app.get('/users/:id', (req, res, next) => {
  const userId = req.params.id;
  const user = new Users();
  const isUserExist = user.getUserById(userId);
  if (!isUserExist) {
```

```

        return next(new NotFoundException(`User with ID ${userId} not
        found`));
    }

    res.status(200).json(user);
});
app.use((err: HttpException, req: Request, res: Response, next:
NextFunction) => {
    const status = err.status || 500;
    const message = err.message || 'Internal server error';

    res.status(status).json({ error: message });
});

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

```

Create `utils` folder and create `errorHandler.ts` file into that directory the paste the following code:

```

export class HttpException extends Error {
    status: number;
    message: string;

    constructor(status: number, message: string) {
        super(message);
        this.status = status;
        this.message = message;
    }
}

export class NotFoundException extends HttpException {
    constructor(message: string = 'Not Found') {
        super(404, message);
    }
}

```

Now update the `user.ts` file in `users` directory with the following code:

```

interface User {
    id: number;
    name: string;
    email: string;
}

```

```

export const users: User[] = [
  { id: 1, name: 'John', email: 'john@example.com' },
  { id: 2, name: 'Jane', email: 'jane@example.com' },
];
export class Users {
  public getUserId(userId) {
    if (users.find(i => i.id == userId)) {
      return true;
    } else {
      return false;
    }
  }
}

```

In this example, there is a custom **HttpException** class that extends the Error class and adds a status property. There is also a **NotFoundException** class that extends the **HttpException** class and sets the status to 404 by default.

In the route handler for `/users/:id`, if the requested user is not found, a **NotFoundException** is thrown and passed to the next function, which triggers the custom error handling middleware.

The screenshot shows a Postman request for `http://127.0.0.1:3000/users/1234`. The response status is 404 Not Found, with a message: "User with ID 1234 not found".

Key	Value	Description
		Description

Body	Cookies	Headers (7)	Test Results	404 Not Found	6 ms	281 B	Save as example
<pre> 1 2 "error": "User with ID 1234 not found" 3 </pre>							

Figure 3.7: User Not Found Exception

The custom error handling middleware checks if the error is an instance of **HttpException** and uses the status and message properties to send a JSON response with the appropriate HTTP status code. If the error is not an instance of **HttpException**, it sends a generic 500 error response.



```
File Edit Selection View Go Run Terminal Help

EXPLORER
MY-EXPRESS-APP
  > dist
  > lib
  > users
  > utils
  TS main.ts
  > node_modules
  JS app.js
  TS app.ts
  package-lock.json
  package.json
  tsconfig.json

TS main.ts
lib > TS main.ts > ...
1  import express, { Application, Request, Response, NextFunction } from 'express';
2  import { HttpException, NotFoundException } from './utils/errorHandler';
3  import * as bodyParser from 'body-parser';
4  import { users, Users } from './users/user';
5
6  const app: Application = express();
7
8  app.use(bodyParser.json());
9
10 app.get('/api/users', (req, res) => {
11   res.json(users);
12 });
13
14 app.get('/users/:id', (req, res, next) => {
15   const userId = req.params.id;
16   const user = new Users();
17   const isUserExist = user.getUserById(userId);
18
19   if (!isUserExist) {
20     return next(new NotFoundException(`User with ID ${userId} not found`));
21   }
22
23   res.status(200).json(user);
24 });

PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT
yannmin@atl-p186:~/projects/book/kriti/docs/chapter3/my-express-app$ tsc
yannmin@atl-p186:~/projects/book/kriti/docs/chapter3/my-express-app$ node dist/main.js
Server listening on port 3000!
```

Figure 3.8: Sample code in VsCode Editor

Async Error Handling

In Express.js, asynchronous errors can be handled using try-catch blocks or by returning a rejected Promise.

Add the following code in `main.ts` file:

```
// Async function that throws an error
async function asyncFunction(): Promise<void> {
  throw new Error('Async error');
}

// Async route handler that calls the async function
app.get('/async-error', async (req: Request, res: Response, next: NextFunction) => {
  try {
    await asyncFunction();
    res.send('Success');
  } catch (error) {
    next(error);
  }
});

// Error handling middleware
app.use((err: Error, req: Request, res: Response, next: NextFunction) => {
  console.error(err.message);
})
```

```

    res.status(500).send('Something broke!');
  });

```

Now compile and run the code with `$ tsc` and then `$ node dist/main.js`. After that open browser with `http://localhost:3000/async-error` it displays as Something broke!

In this example, we have an async function called **asyncFunction** that throws an error. We have a route handler that calls this function and catches any errors that occur using a try-catch block. If an error occurs, the next function is called with the error parameter to pass the error to the error-handling middleware.

The error-handling middleware function takes four parameters: `err`, `req`, `res`, and `next`. If an error occurs in any middleware or route handler before this function, it will be passed to this middleware function. The middleware function logs the error to the console and sends a **500 Internal Server Error** response to the client.

Note that the **async** keyword is used before the route handler function to indicate that it is an asynchronous function. Also, the **await** keyword is used before the call to the **asyncFunction** to wait for the function to complete before proceeding to the next line of code.

The screenshot shows the Postman interface with the following details:

- URL:** http://127.0.0.1:3000
- Method:** GET
- Request URL:** http://localhost:3000/async-error
- Body:** (Pretty) 1 Something broke!
- Headers:** (7)
- Response:** 500 Internal Server Error 17 ms 263 B
- Message:** Something broke!

Figure 3.9: Async Error API

Figure 3.10 shows the terminal output:

```

● yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ tsc
○ yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ node dist/main.js
Server listening on port 3000!
Async error

```

Figure 3.10: Async Error Terminal Output

It is important to handle errors properly in an Express.js application to ensure that the application is robust and reliable.

Static File Serving

In Express.js, we can serve static files, such as images, CSS, JavaScript files, and more, using the **express.static()** middleware function, for example.

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

In the preceding example, we are serving static files from the public directory. The **express.static()** middleware function takes one argument, which is the name of the directory that contains the static files.

Once the middleware is set up, you can access your static files by specifying their URL relative to the public directory. For example, if you have a file called **profilePic.png** in the **public/images** directory, you can access it at **http://localhost:3000/images/profilePic.png**

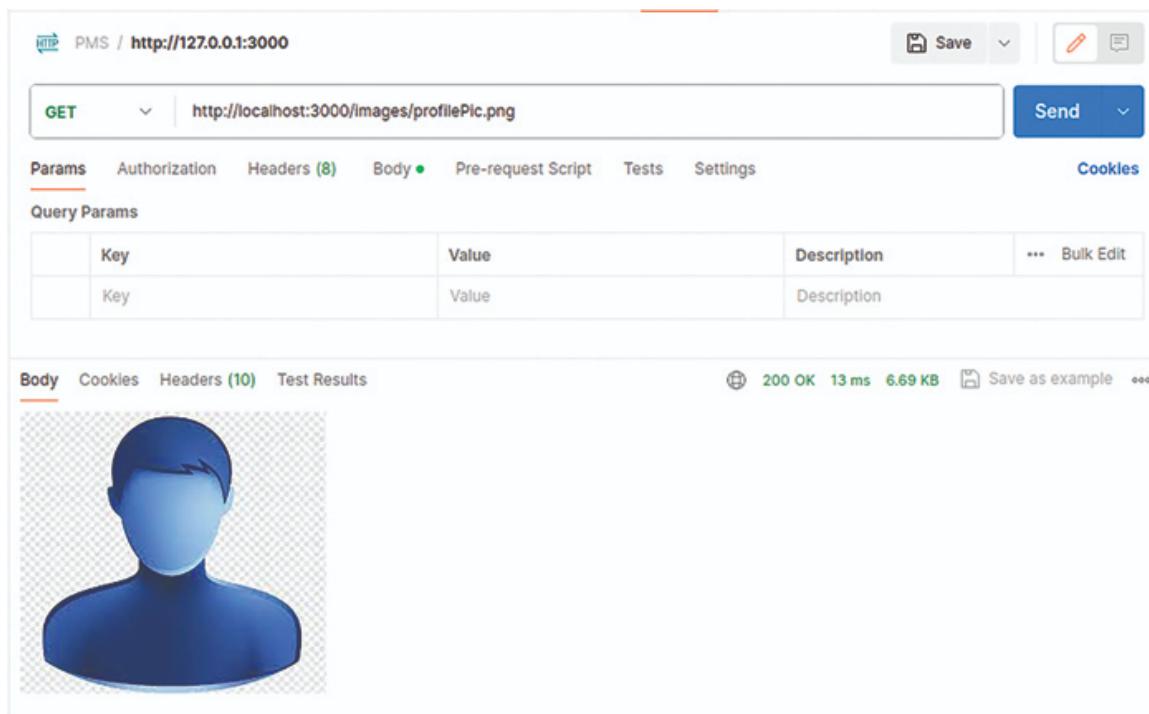


Figure 3.11: Static Serving Image File

Templating Engines

In Express.js, templating engines are used to generate HTML markup and dynamically render views. Templating engines allow you to create templates with

placeholders for dynamic data that can be replaced with real data when the template is rendered.

Some of the popular templating engines supported by Express.js include:

- EJS (Embedded JavaScript)
- Pug (formerly Jade)
- Handlebars
- Mustache

To use a templating engine in an Express.js application, you need to install the engine using npm and set it as the default view engine in the app configuration. Then you can create views using the syntax and features of the chosen templating engine.

Open the terminal and install dependency for **ejs** with the following command:

```
$ npm install ejs --save
```

Once **ejs** is installed, we can try the following example code:

```
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));
app.get('/ejs', (req, res) => {
  const data = {
    title: 'My App',
    message: 'Hello, I am from EJS !!!'
  };
  res.render('index', data);
});
```

Create **index.ejs** file and paste html code.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= message %></h1>
  </body>
</html>
```

In this example, we use the `<%= %>` syntax to output the "**title**" and "**message**" variables that were passed to the view in the route handler. When the view is rendered, these variables will be replaced with their respective values.

The TypeScript compiler handles the task of generating JavaScript files and transferring them to the `dist` folder. However, it does not handle the copying of other necessary project files like EJS view templates. To address this, you can create a build script responsible for copying all additional files to the `dist` folder.

To automatically copy files from the `views` folder to the `dist` folder after compiling your TypeScript code, you can use a build tool like `copyfiles` or `copy`.

First, install the `copyfiles` package as a dev dependency with the following command executed in the terminal:

```
$npm install --save-dev copyfiles
```

Now update the scripts in `package.json` with the following code:

```
"scripts": {  
  "build": "tsc && npm run copy-views",  
  "copy-views": "cpy 'views/*' dist/views/ --recursive"  
}
```

Then execute script from the terminal with the following code:

```
$ npm run build
```

```
• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ npm run build  
> my-express-app@1.0.0 build  
> tsc && npm run copy-views  
  
> my-express-app@1.0.0 copy-views  
> cpy 'views/*' dist/views/ --recursive  
○ yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter3/my-express-app$ node dist/main.js  
  Server listening on port 3000!  
  □
```

Figure 3.12: Build Application

After running the application, you can open your web browser and navigate to `http://localhost:3000/ejs`. This will display the HTML output in the browser, as shown in the following image.

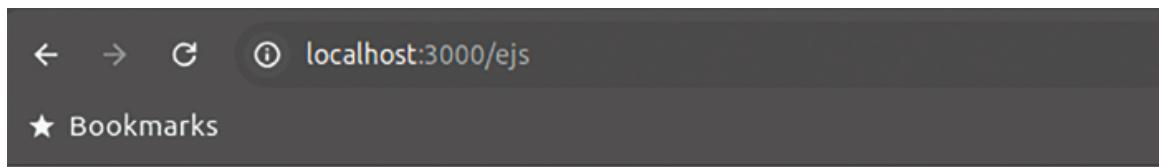


Figure 3.13: Browser EJS Template

Security and Performance Best Practices

There are several security best practices to follow when developing applications with Express.js:

- **Use secure HTTP protocols:** Always use HTTPS instead of HTTP to ensure secure communication between client and server.
- **Use the latest version:** Keep your Express.js version up to date, and apply security patches as soon as they become available.
- **Avoid using deprecated or vulnerable packages:** Use only up-to-date, well-maintained packages and avoid deprecated or vulnerable ones.
- **Validate user input:** Always validate user input to prevent injection attacks, cross-site scripting (XSS) attacks, and other malicious activities.
- **Use a Content Security Policy (CSP):** Implement a Content Security Policy (CSP) to protect against XSS attacks by limiting the resources that a page can load.

Example:

```
app.use((req, res, next) => {
  res.setHeader(
    'Content-Security-Policy',
    "default-src 'self'; script-src 'self' 'unsafe-inline'; img-
    src 'self' data:; font-src 'self' data:;"
  );
  next();
});
```

The CSP header is then set in a middleware function that is added to the Express app. The CSP policy in this example allows scripts to be loaded from the same domain ('`self`') and also allows inline scripts ('`unsafe-inline`'). Images and fonts are allowed from the same domain as well as from the data: protocol.

- **Implement rate limiting:** Implement rate limiting to prevent brute force attacks and other types of attacks that involve repeated requests.

Example:

First, install the `express-rate-limit` package using npm:

```
$ npm install express-rate-limit
```

Then, require the package and create a new rate limiter object with the desired options:

```
const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests per windowMs
});
```

In this example, we are creating a rate limiter that limits each IP address to 100 requests every 15 minutes.

Finally, apply the rate limiter middleware to the desired routes:

```
app.use(limiter);

app.get("/", (req, res) => {
  res.send("Hello World!");
});
```

Now, each incoming request to the root route ("") will be checked against the rate limiter. If the IP address has exceeded the maximum number of requests within the specified time window, the middleware will return a 429 "Too Many Requests" error.

- **Use helmet:** Use the helmet middleware to add additional security headers to HTTP responses, such as the X-XSS-Protection, X-Content-Type-Options, and X-Frame-Options headers.

Example:

```
import helmet from "helmet";

// Use Helmet middleware
app.use(helmet());
```

```
// Add routes to the app
app.get("/", (req, res) => {
  res.send("Hello, world!");
});
```

In this example, the helmet middleware is imported from the helmet package and used in the application using `app.use(helmet())`. This will automatically add security headers to HTTP responses, such as setting the X-Content-Type-Options header to `nosniff` to prevent browsers from interpreting response content as a different MIME type.

Note that this is just a basic example, and additional configuration may be necessary depending on the specific security needs of your application.

- **Use secure cookies:** When using cookies, set the secure and `httpOnly` flags to prevent cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

Example:

```
import cookieParser from 'cookie-parser';
app.use(cookieParser('secret'));
app.get('/set-cookie', (req, res) => {
  res.cookie('myCookie', 'someValue', {
    httpOnly: true,
    sameSite: 'strict',
    secure: true
  });
  res.send('Cookie set successfully!');
});
```

In this example, the cookie-parser middleware is used to parse cookies in incoming requests. The secret argument is used to sign and encrypt the cookies.

The `/set-cookie` route sets a new cookie with the `res.cookie()` method. The `httpOnly` option prevents the cookie from being accessed by JavaScript code, making it more difficult for an attacker to steal the cookie using a cross-site scripting (XSS) attack. The `sameSite` option limits the scope of the cookie to the same site that set it, reducing the risk of cross-site request forgery (CSRF) attacks. The `secure` option ensures that the cookie is only sent over HTTPS, protecting it from interception by network attackers.

By following these best practices for secure cookie handling, you can significantly improve the security of your Express.js application.

- **Implement a secure deployment process:** Implement a secure deployment process that includes secure configurations, code review, and testing to prevent security vulnerabilities from being introduced into production.
- **Use a linter and security scanner:** Use a linter and security scanner to detect and fix potential security issues in your code. The best example is **EsLint**.

By following these security best practices, you can help ensure that your Express.js applications are secure and protected against common security threats.

There are some performance best practices to follow when developing applications with Express.js:

- **Avoid the use of synchronous functions:** It is recommended to use asynchronous code because synchronous code in production slows down the application, so try to avoid unnecessary synchronous functions and use **async/await** with promises.
- **Exception handling:** Use always try catch to handle the exception at the code level so it does not break the application on run time.
- **Reduce middleware usage:** Use only the required middleware and avoid excessive usage. Middleware can be resource-intensive and may slow down the application.
- **Caching:** Implement caching for frequently requested data such as static files or API responses. By caching this data, you can reduce the number of requests that the server has to process and significantly improve the response times of your application.
- **Use cluster:** Use Node.js Cluster mode to utilize all available CPU cores and distribute the load evenly across all cores which improves the performance of the application.
- **Auto restart application:** Make sure that if anytime application crashes then it automatically restarts so use a process manager or packages such as PM2 or Forever for that.

By following these recommended practices, you can enhance the performance of your Express.js application, resulting in a faster and more efficient user experience.

Conclusion

In this chapter, we learned about Express.js, a popular and powerful web application framework for Node.js. We learned about its offerings, upsides, and limitations. While it has a few limitations, the benefits of Express.js, such as its active community, extensive documentation, and support for REST API development, make it an excellent choice for web development projects.

In the next chapter, we will start building a project management system application. We will take this as a big exercise to learn the concepts of TypeScript and Express.js.

Multiple Choice Questions

1. What is Express.js primarily used for?
 - a. Database management
 - b. Front-end development
 - c. Building web applications and APIs
 - d. Machine learning
2. What advantages and drawbacks are associated with the Express.js framework?
 - a. Advantages encompass its lightweight and minimalist design, while drawbacks involve the absence of built-in features for complex applications.
 - b. Advantages comprise a wide range of built-in features for complex applications, while drawbacks pertain to suboptimal performance.
 - c. Advantages include automated scaling for high-traffic applications, while drawbacks involve a challenging learning curve.
 - d. Advantages entail effortless integration with databases, while drawbacks concern the absence of routing support.
3. What is the purpose of the Express.js Router object?
 - a. To define routes for multiple applications
 - b. To create middleware functions
 - c. To handle errors in the application
 - d. To define routes for a specific part of the application
4. What role does middleware play in an Express.js application?
 - a. To host static files

- b. To establish routes
 - c. To manage incoming requests and responses
 - d. To facilitate authentication
5. How can you handle routing parameters in Express.js?
- a. Using the `req.routeParams` object
 - b. By defining separate route handlers for each parameter
 - c. Accessing them directly from the URL
 - d. Using the `req.params` object
6. What does the `next()` function do in Express.js middleware?
- a. Ends the request-response cycle
 - b. Passes control to the next middleware function
 - c. Sends a response to the client
 - d. Logs information to the console
7. How can you handle errors in an Express.js application using middleware?
- a. Use the `catchError` middleware function
 - b. Wrap the code in `try-catch` blocks
 - c. Use the `error` event on the `app` object
 - d. Define an error-handling middleware with four parameters
8. Which of the following Express.js middleware is commonly used for parsing JSON requests?
- a. `express-static`
 - b. `body-parser`
 - c. `cookie-parser`
 - d. `express-session`
9. What is the primary purpose of the "cookie-parser" middleware in an Express.js application?
- a. To generate random cookies for user sessions.
 - b. To parse and handle incoming HTTP requests.
 - c. To parse cookies attached to incoming HTTP requests.
 - d. To set secure HTTP headers for cookie handling.

10. Which middleware is used for handling Cross-Origin Resource Sharing (CORS) in Express.js applications?

- a. express-cors
- b. cors-express
- c. cross-origin
- d. cors

Answers

- 1. c
- 2. a
- 3. d
- 4. c
- 5. d
- 6. b
- 7. d
- 8. b
- 9. c
- 10. d

Further Readings

<https://expressjs.com>

<https://blog.dreamfactory.com/rest-apis-an-overview-of-basic-principles/>

<https://restfulapi.net/>

OceanofPDF.com

CHAPTER 4

Planning the App

Introduction

Node.Js has become the preferred choice when it comes to writing an application. There are more than 30 million websites powered by Node.js. These applications or websites or projects in general have a fair share of complexity. Netflix, LinkedIn, Uber, Paypal, and even NASA-like organizations use Node.js for their applications.

For developing an application, planning is the crucial part. It helps to establish a clear goal and a path to follow to achieve the defined goal efficiently.

For the rest of the book, we are going to follow one common example to learn the important aspects of an application. We will see how to write, optimize, and test the app with standard techniques. We will be writing a project management software.

Structure

In this chapter, we will discuss the following topics:

- Overview of Project Management Application
- Database Design
- Setting up the Project Structure
 - Installation of Project Dependency
 - Project Directory Structure
- Database Models (entities)
- Routing

Overview of the Application

Any project management software has a simple goal defined. Let the individuals and teams plan, organize, and execute the tasks to effectively take the project to its completion. A PMS should allow the respective users to create and manage projects. For each project, there can be many tasks to perform. Users should be able to communicate with each other. We will soon find out about these tasks and much more.

Our objective or scope of the Project Management System (PMS) project is to develop a small web application that serves as a basic project management tool. The application will have various features including secure user login, the ability to create projects and tasks, task assignment based on user roles, and the implementation of email notifications when tasks are assigned or completed.

Additionally, the project will explore the implementation of a caching mechanism for APIs to enhance response times for requests. Moreover, it covers information for developers to do unit testing, along with how to build the application for the production environment and finally the deployment.

In order to get familiar with what we are going to build, we need to define the purpose, tasks we are going to perform, architecture, project structure, database design, and much more.

Roadmap

A typical roadmap for a project involves several key stages and activities. Here is a high-level overview of what the roadmap covers.

- Planning the application
- Defining the modules
- Database design
- Setting up the project structure
- Init the project
- Connecting the database
- API development
- API Caching
- Unit Testing
- Deployment

In this chapter, we will cover the first six of the preceding list. The last four topics have their dedicated chapters.

Scope

The purpose aforementioned has already given a good idea that we are going to develop a project management system. At times, we will refer to this as PMS also. Our first goal is to identify what our PMS should be capable of.

We should be able to:

- Manage users and login into the application.
- Create and manage projects.
- Create and manage tasks for the created projects.
- Should be able to access and update the task details.
- Should be able to communicate with other members on a task.
- Some basic project reports on a Dashboard.

Defining the Modules

Based on the tasks we should be able to perform, we can see that the PMS should have these must-have modules in the system.

User Module

This module will allow us to manage users. We should be able to perform the following:

- Add new users.
- Edit users.
- Delete users.
- User Login.
- Get a user profile.
- Reset user password.

Project Module

The project module will allow us to manage the projects. The following key tasks should be performed with the help of this module:

- Create new projects.
- Update project details.
- Delete projects.
- Get project detail.
- Get project statistics.
- Manage project memberships.

Task Module

This module will allow us to manage tasks within a project. This is a key module which will enable us to:

- Create new tasks.
- Update task details.
- Delete tasks.
- Get task details.
- Attach supporting files.

Comment Module

A task will always need a way to enable communication among team members. A comment module along with the tasks will facilitate this feature. The following tasks we should be able to perform:

- Add comments on tasks.
- Allow users to update their own comments.
- Allow users and admins to delete the comments.

Database Design

Designing the database for big projects is a tricky, on-going activity that is improvised with project timelines. Although it must be defined at the earliest. It involves defining the schema, tables, relationships, constraints, and data types to meet the specific requirements of the application.

Normalization helps optimize storage that includes data integrity by applying normalization rules to eliminate data redundancy. The goal of normalization is to avoid data duplication and maintain data integrity. Each piece of data is stored in only one place, reducing the risk of inconsistencies. The relationship between tables is maintained using primary and foreign keys.

Normalization has its benefits in terms of storage, non-redundancy, and cleanliness. However, as the database grows with the number of tables along with complex features added to the application, the number of joins and complexity in the queries increase. This impacts readability, and results in slower performance if there are heavy read operations.

In this section, let us design the tables needed for our entities. We will try to keep the relationships to the minimum while maintaining a balance between redundancy and normalization wherever possible.

Note: *For this book, we are going to use the PostgreSQL database. Hence the data type in the following tables is specific to PostgreSQL. For each table, we will also specify if the column is a primary key or a unique key and if it can contain null values. If we are going to reference a column from another table, then we will mention the reference table and column.*

For all of the modules defined above, we need the following tables for our database:

- **User:** Represents all of the users who can access the system
- **Role:** Each user will have a role assigned, for example, Project Manager, Admin, and others.
- **Project:** Represents the project entity.
- **Task:** Tasks associated with each project.
- **Comment:** Comments added to each task.

[Figure 4.1](#) shows a basic entity relationship diagram. This figure uses only primary keys and foreign keys used for each table.

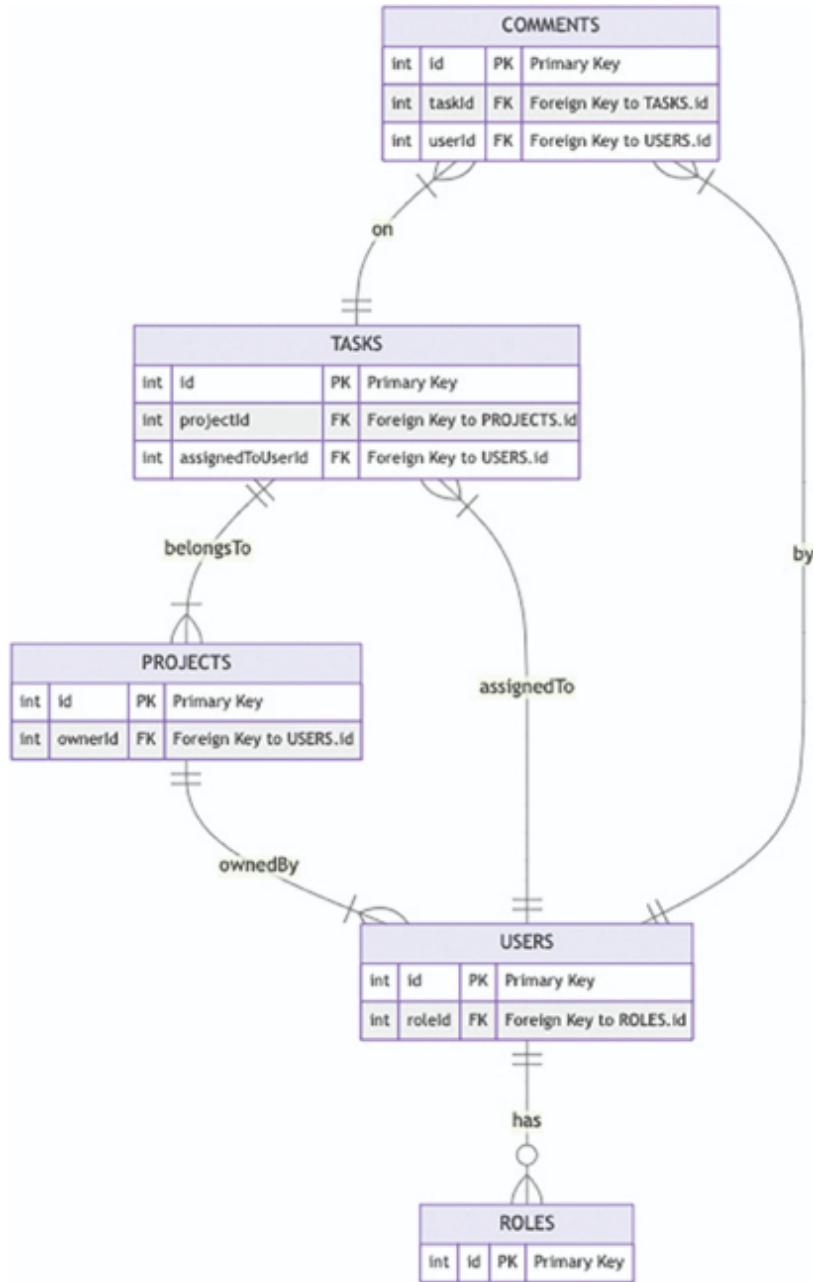


Figure 4.1: High-Level Entity Relationship Diagram

The figure shows that each user has a role assigned. Each project is owned by a user. Each project has tasks assigned and each task can have comments. This diagram is a high-level ER Diagram. Now when we understand the relationship among the tables, we can explore each entity in detail followed by a detailed ER Diagram.

User Schema Table

Let us begin with the User table which is vital for user authentication, authorization, and maintaining user-related data. It plays a crucial role in ensuring that each user's information is accurately captured and secured while enabling the PMS to identify and manage individual users effectively.

For the sake of simplicity, we are adding only the user id, name, password, and some relevant details needed.

Column Name	Type	Data Type	Description	Primary?	Is Null	Unique	Reference
user_id	uuid	uuid	User's unique identifier id	true	false	true	-
email	email	varchar(60)	User's Email-id	false	false	true	-
full_name	string	varchar(30)	User's Full Name	false	true	false	-
username	string	varchar(30)	User's Unique Name	false	false	true	-
password	string	varchar(100)	User's hashed Password	false	false	false	-
role_id	uuid	uuid	User's role id	false	false	false	Role Table (role_id)
created_at	date	timestamp	User's Creation Time	false	false	false	-
updated_at	date	timestamp	User's Data Updation Time	false	true	false	-

Table 4.1: User Table

Role Schema Table

User access to the application is determined by the assigned role and the permissions granted to that role. Users are allowed to perform specific actions within the application based on the rights associated with their assigned role. We have defined the role schema table for applications with constraints as follows.

Column Name	Type	Data Type	Description	Primary?	Is Null	Unique
-------------	------	-----------	-------------	----------	---------	--------

role_id	uuid	uuid	Role's unique identifier id	true	false	true
name	email	varchar(60)	Role's name	false	false	true
description	string	varchar(30)	Role's description	false	true	false
rights	string	text	Rights as permission of diff modules	false	false	false
created_at	date	timestamp	Role Creation Time	false	false	false
updated_at	date	timestamp	Role's Data Updation Time	false	true	false

Table 4.2: Role Table

Project Schema Table

In a Project Management System (PMS), the project table serves as a central repository for storing information about different projects managed within the system. Each row in the project table represents a specific project, while each column stores various attributes or details related to those projects. We defined the schema for that as follows:

Column Name	Type	Data Type	Description	Primary?	Is Null	Unique	Reference
project_id	uuid	integer	Project's unique identifier id	true	false	true	-
name	string	varchar(60)	Project's Name	false	false	true	-
description	string	varchar(200)	Project's Description	false	true	false	-
user_ids	string	text	Project's assigned users_ids	false	true	false	User Table (user_id)
start_time	date	timestamp	Project's Start Date	false	true	false	-
end_time	date	timestamp	Project's End Date	false	true	false	-
status	enum	varchar(30)	Current Project Status	false	false	false	-

created_at	date	timestamp	User Creation Time	false	false	false	-
updated_at	date	timestamp	User Data Updation Time	false	true	false	

Table 4.3: Project Table

Task Schema Table

The task table is designed to store information about various tasks or activities associated with a specific project. Each row in the task table represents a specific task, while each column holds different attributes or details related to those tasks as follows:

Column Name	Type	Data Type	Description	Primar y?	Is Nu ll	Uniq ue	Reference
task_id	uuid	uuid	Task's unique identifier id	true	false	true	-
name	string	varchar(60)	Task's Name	false	false	true	-
description	string	varchar(30 0)	Task's Description	false	true	false	-
project_id	uuid	varchar(35)	Associated Project Id	false	true	false	Project Table (project_id)
user_id	uuid	uuid	Assigned User Id	false	true	false	User Table (user_id)
estimated_start_time	date	timestamp	Task's Estimated Start Date	false	true	false	-
estimated_end_time	date	timestamp	Task's Estimated End Date	false	true	false	-
actual_start_time	date	timestamp	Task's Actual Start Date	false	true	false	-
actual_end_time	date	timestamp	Task's Actual End Date	false	true	false	-
priority	enum	varchar(20)	Task's priority e.g., High, Low,	false	true	false	-

			Medium					
status	enum	varchar(30)	Current Status	Task	false	false	false	-
supported_files	string	text	Supported Files url	url	false	true	false	-
created_at	date	timestamp	Task's Creation Time	Time	false	false	false	-
updated_at	date	timestamp	Task's Data Updation Time	Time	false	true	false	-

Table 4.4: Task Table

Comment Schema Table

The comment table is designed to store comments or notes associated with various tasks. Each row in the comment table represents a specific comment related to a particular task, while each column holds different attributes or details related to those comments as follows.

Column Name	Type	Data Type	Description	Primary?	Is Null	Unique	Reference
comment_id	uuid	uuid	Comment's unique identifier id	true	false	true	-
comment	string	varchar(300)	Comment	false	false	false	-
task_id	uuid	uuid	Task's Id of given comment	false	false	false	Task Table (task_id)
user_id	uuid	uuid	User's Id on given comment	false	false	false	User Table (user_id)
supported_files	string	text	attached file's url	false	true	false	-
created_at	date	timestamp	Comment's Creation Time	false	false	false	-
updated_at	date	timestamp	Comment's Data Updation Time	false	true	false	-

Table 4.5: Comment Table

This is the overview of PMS database design with raw schema tables. Now that we have the schema ready, we can head to start writing our project. We will first start by defining the project file structure.

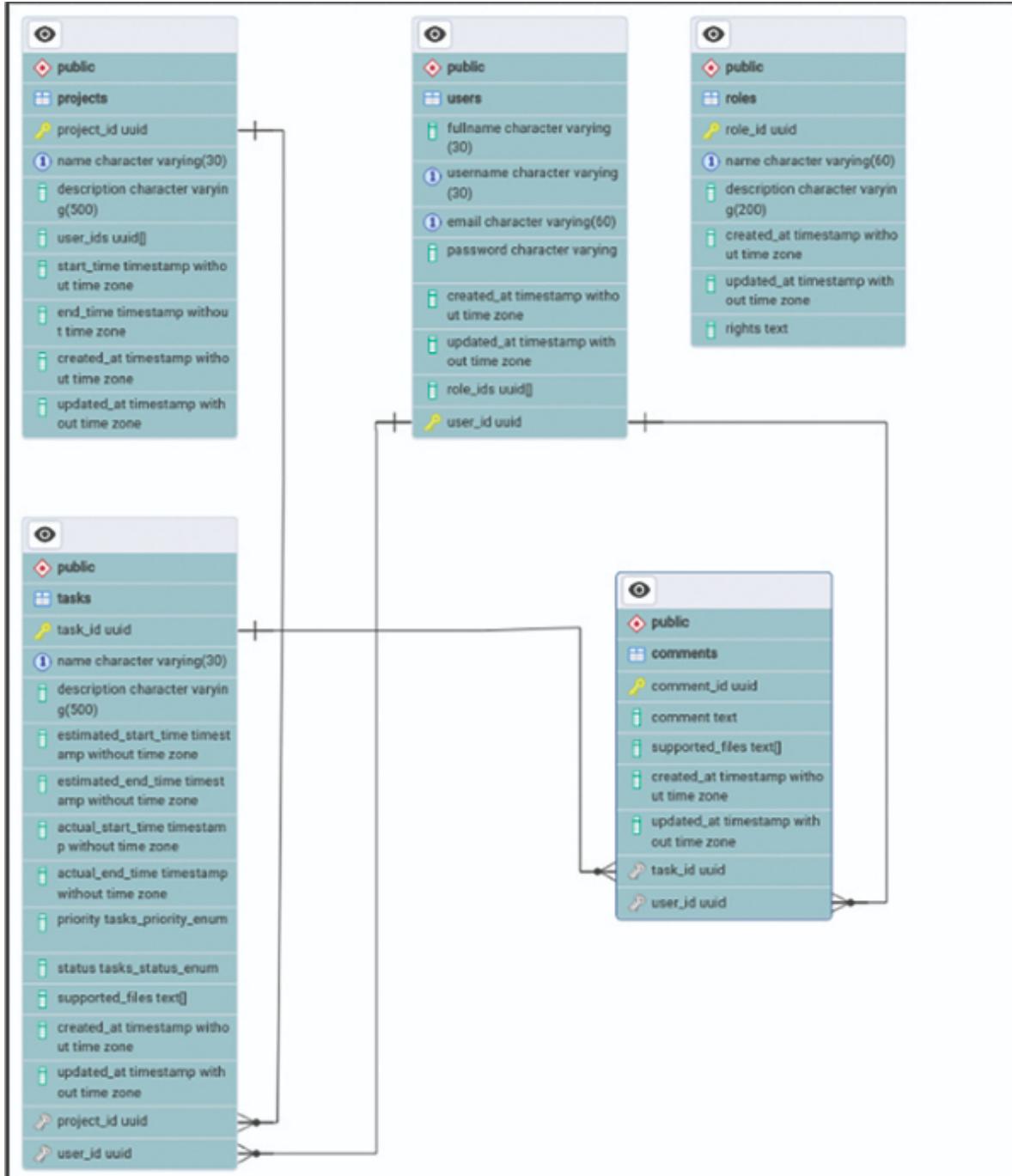


Figure 4.2: Entity Relationship Diagram

Setting Up the Project Structure

The project structure is flexible and can be customized to suit the specific needs of your Node.js application, and we aim to minimize developers' efforts by adhering to established standards throughout the development process.

Let us start developing a PMS application in Node.js through the initialization project.

Init the project

Before you begin, ensure that you have installed the LTS (Long Term Support) version of Node.js and PostgreSQL on your system.

PostgreSQL can be downloaded from <https://www.postgresql.org/download/>. There are installers available for Linux, MacOs, Windows, BSD and Solaris. Since there are many flavors of Linux, the official website mentioned above also has separate instructions for each flavor.

The directory name for the project can be kept as `pms-be`. `pms` stands for Project Management System and `-be` can be appended because we are creating an API backend of the app. Let us create a new directory `'pms-be'` and navigate to it using the command prompt (`cmd`). Inside the directory, initiate a new Node.js project by running the command "`npm init`".

Once "`npm init`" is executed, the setup process will prompt you to provide various details step by step. You can add the appropriate information and press "**Enter**" for each prompt:

- Package Name: (Default is "pms-be")
- Version: (Default is "1.0.0")
- Description: "Project Management System"
- Entry Point: (Default is "index.js") You can change it to "main.js" if needed.
- Test Command: If you don't have a specific test command, you can leave this field blank.
- Git Repository: If your project is not using Git, leave this field blank.
- Keywords: Add relevant keywords separated by commas.

- Author: Add the name of the project's author.
- License: (Default is "ISC") You can choose a different license or keep it as "ISC."
- Press "Enter" once more to complete the initialization process.

With this, your project's initialization process will be successfully completed and the **package.json** file will be created.

```
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter4/pms-be$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (pms-be)
version: (1.0.0)
description: Project Management System
entry point: (index.js) main.js
test command:
git repository:
keywords:
author: Yamini Panchal & Ravi Gupta
license: (ISC)
About to write to /home/yammini/projects/book/kriti/docs/chapter4/pms-be/package.json:

{
  "name": "pms-be",
  "version": "1.0.0",
  "description": "Project Management System",
  "main": "main.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Yamini Panchal & Ravi Gupta",
  "license": "ISC"
}

Is this OK? (yes) yes
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter4/pms-be$ █
```

Figure 4.3: Init Project

Installation of Project Dependency

Once the project is initialized, the next step involves installing the necessary packages and tools as dependencies. We highly recommend using Visual Studio Code as the development tool because it offers various extensions, such as intellisense for programming languages, code prettier, and git

packages, which significantly streamline the development process. These extensions help save a considerable amount of effort for developers, making the development experience more efficient and enjoyable.

Now let us start to install the most required packages for project management system development such as **express.js**, **postgresql(pg)**, **typescript**, **typeorm**, **ts-node**, **ts-lint**, and so on, and later on we will install other packages as required while development.

```
npm install express --save
npm install typescript -D
npm install pg --save
npm install typeorm --save
npm install @types/express -D
npm install @types/node -D
npm install @types/pg -D
npm install reflect-metadata --save
npm install uuid --save
npm install @types/uuid -D
npm install ts-node prettier eslint eslint-config-prettier
eslint-plugin
-prettier -D
npm install @typescript-eslint/eslint-plugin @typescript-
eslint/parser -D
```

Here, we share the **package.json** file for reference to install all packages and it will also change day by day through development.

```
// package.json
{
  "name": "pms-be",
  "version": "1.0.0",
  "description": "Project Management System",
  "main": "main.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1",
    "start": "nodemon dist/src/main.js"
  },
  "author": "Ravi Kumar Gupta, Yamini Panchal",
  "license": "Proprietary",
```

```

"devDependencies": {
  "@types/express": "4.17.17",
  "@types/node": "20.4.2",
  "@types/pg": "8.10.2",
  "@types/uuid": "9.0.2",
  "@typescript-eslint/eslint-plugin": "5.59.8",
  "@typescript-eslint/parser": "5.59.8",
  "eslint": "8.41.0",
  "eslint-config-prettier": "8.8.0",
  "eslint-plugin-prettier": "4.2.1",
  "nodemon": "2.0.22",
  "prettier": "2.8.8",
  "ts-node": "10.9.1",
  "typescript": "5.0.4"
},
"dependencies": {
  "express": "4.18.2",
  "pg": "8.11.1",
  "reflect-metadata": "0.1.13",
  "typeorm": "0.3.17",
  "uuid": "9.0.0"
}
}

```

Next, create a **tsconfig.json** file in the main directory of the project with the following code to ensure TypeScript compiles the code:

```

// tsconfig.json
{
  "compilerOptions": {
    "module": "commonjs",
    "moduleResolution": "node",
    "pretty": true,
    "sourceMap": false,
    "target": "ESNext",
    "outDir": "./dist",
    "baseUrl": "./src",
    "noImplicitAny": false,
    "esModuleInterop": true,
  }
}

```

```

"removeComments": true,
"preserveConstEnums": true,
"experimentalDecorators": true,
"alwaysStrict": true,
"forceConsistentCasingInFileNames": true,
"emitDecoratorMetadata": true,
"resolveJsonModule": true,
"skipLibCheck": true
},
"include": ["src/**/*.ts", "src/**/*.json"],
"exclude": ["node_modules"],
"files": ["node_modules/@types/node/index.d.ts"]
}

```

Thereafter, create `server_config.json` which will be used for all configurations, such as, `port`, `database config`, and so on.

```

// server_config.json
{
  "port": 3000,
  "db_config": {
    "db": "postgres",
    "username": "root",
    "password": "123456",
    "host": "localhost",
    "port": 5432,
    "dbname": "pms"
  }
}

```

Note: Please always keep stronger passwords for any account.

As part of the configuration process, we will integrate **ESLint** to enhance development practices. We will include a `.eslintrc.json` file in the project's root directory, as demonstrated in the code snippet as follows:

```

.eslintrc.json
{
  "env": {
    "browser": true,
    "es2021": true
  }
}

```

```
},
"extends": ["eslint:recommended", "plugin:@typescript-eslint/recommended"],
"overrides": [],
"parser": "@typescript-eslint/parser",
"parserOptions": {
  "ecmaVersion": "latest",
  "sourceType": "module"
},
"plugins": ["@typescript-eslint"],
"rules": {
  "linebreak-style": ["error", "unix"],
  "quotes": ["error", "single"],
  "semi": ["error", "always"]
}
}
```

Nodemon is beneficial for Node.js developers as it enhances development workflow, improves productivity, and provides real-time feedback during the development process.

With Nodemon, developers receive immediate feedback on code changes as the server restarts instantly. This allows for quick validation of code modifications and helps catch errors early in the development process. This eliminates the need to manually stop and restart the server after every code modification, saving time and streamlining the development process.

Let us install **nodemon** open root directory of project on terminal and paste following command:

```
$ npm install nodemon -D
```

Project Directory Structure

When structuring a **Node.js** project with **Express** and **TypeScript**, a common directory structure can be as follows:

```
pms-be/
| -- src/
|   | -- components/
|   |   | -- users/
```

```
|   |   |   |-- users_controller.ts
|   |   |   |-- users_entity.ts
|   |   |   |-- users_routes.ts
|   |   |   |-- users_service.ts
|   |   |-- projects/
|   |   |   |-- projects_controller.ts
|   |   |   |-- projects_entity.ts
|   |   |   |-- projects_routes.ts
|   |   |   |-- projects_service.ts
|   |   |-- tasks/
|   |   |   |-- tasks_controller.ts
|   |   |   |-- tasks_entity.ts
|   |   |   |-- tasks_routes.ts
|   |   |   |-- tasks_service.ts
|   |-- utils/
|   |   |-- db_utils.ts
|   |   |-- common.ts
|   |-- routes/
|   |   |-- index.ts
|-- express_server.ts
|-- main.ts
|-- tests/
|   |-- users_spec.tjs
|-- package-lock.json
|-- package.json
|-- node_modules
|-- tsconfig.json
|-- server_config.json
|-- .gitignore
|-- README.md
```

As shown in the project directory structure, we have already set up **server_config.json** and **tsconfig.json** files. As we proceed with the development, we will create various folders and files to organize the code and manage different aspects of the project.

[Create Express Server](#)

Before creating an express server, add one **README.md** file that includes how this project will run and the required configuration so anyone can easily work on that.

README.md

```
# Project Management System API

## Prerequisites
- Node.js: Ensure that Node.js is installed on your machine.
You can download it from [here](https://nodejs.org).

## Install Project Dependency Packages
```
npm install
```

## Configuration
create server_config.json file in the root directory and save it. Inside that file, add the desired configuration values in the format KEY:VALUE.

- Configuration Options
| Configuration Key | Description |
| ----- | ----- |
| PORT | The port number on which project run |

## Compile Project
Once you are in the project's root directory, you can use the tsc command with the --watch flag to enable automatic recompilation when changes are detected. Run the following command
```
tsc --watch
```

## Run Project
- on local machine run during development
```
nodemon dist/src/main.js
```

```

```
- on server after development
```

```
```
```

```
node src/main.js
```

```
```
```

Let us create a new directory called "src" and inside it, create two files named "main.ts" and "express_server.ts" to set up an Express HTTP server with the following code. As our project is currently small-scale, we will not be using **cluster** and **forks** at the OS level, but we will implement reconnection to the server in case of any exceptions. Depending on your project's specific requirements, you can choose to use the "main.ts" file with or without **cluster** functionality.

```
// main.ts
import cluster from 'cluster';
import { ExpressServer } from './express_server';
// connect the express server
const server = new ExpressServer();

process.on('uncaughtException', (error: Error) => {
  console.error(`Uncaught exception in worker process
${process.pid}:`, error);

// Close any open connections or resources
server.closeServer();

setTimeout(() => {
  cluster.fork();
  cluster.worker?.disconnect();
}, 1000);
});

// Gracefully handle termination signals
process.on('SIGINT', () => {
  console.log('Received SIGINT signal');
  // Close any open connections or resources
  server.closeServer();
});

process.on('SIGTERM', () => {
  console.log('Received SIGTERM signal');
  // Close any open connections or resources
```

```
server.closeServer();
});
```

With Cluster for a Large Project

With the presence of the cluster module, if the server crashes due to any reason, we can catch the event when the process exits and spawn another child. This way, we ensure that the server does not crash and it remains available for serving upcoming requests.

```
// main.ts
import cluster from 'cluster';
import os from 'os';
const numCPUs = os.cpus().length;
import { ExpressServer } from './express_server';
if (cluster.isPrimary) {
  console.log(`Master process PID: ${process.pid}`);

  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker process ${worker.process.pid} exited with
    code ${code} and signal ${signal}`);
    setTimeout(() => {
      cluster.fork();
    }, 1000);
  });
}

} else {
  // connect the express server
  const server = new ExpressServer();
  process.on('uncaughtException', (error: Error) => {
    console.error(`Uncaught exception in worker process
    ${process.pid}:`, error);
    // Close any open connections or resources
    server.closeServer();

    setTimeout(() => {
      cluster.fork();
    }, 1000);
  });
}
```

```

        cluster.worker?.disconnect();
    }, 1000);
});

// Gracefully handle termination signals
process.on('SIGINT', () => {
    console.log('Received SIGINT signal');
    // Close any open connections or resources
    server.closeServer();
});

process.on('SIGTERM', () => {
    console.log('Received SIGTERM signal');
    // Close any open connections or resources
    server.closeServer();
});
}

```

As per the preceding code, we need to create **express_server.ts** as it is imported. So let us make it.

```

// express_server.ts
import express, { Application } from 'express';
import { IServerConfig } from './utils/config';
import * as config from '../server_config.json';
export class ExpressServer {

    private static server = null;
    public server_config: IServerConfig = config;
    constructor() {
        const port = this.server_config.port ?? 3000;

        // initialize express app
        const app = express();
        app.get('/ping', (req, res) => {
            res.send('pong');
        });

        ExpressServer.server = app.listen(port, () => {
            console.log(`Server is running on port ${port} with pid = ${process.pid}`);
        });
    }
}

```

```

    });
}

//close the express server for safe on uncaughtException
public closeServer(): void {
    ExpressServer.server.close(() => {
        console.log('Server closed');
        process.exit(0);
    });
}
}
}

```

Afterwards, create a directory with **utils** and inside add **config.ts** file for define config interface as follows:

```

// config.ts
export interface IServerConfig {
    port: number;
    db_config: {
        'db': string;
        'username': string;
        'password': string;
        'host': string;
        'port': number;
        'dbname': string;
    };
}

```

Now to run the app, first compile the code with **tsc --watch**. After that, a **dist** folder will be created, and then run with **nodemon dist/src/main.js** while developing the app, the following output will be displayed.

Output :



```
GINLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 2
yammini@atpl-lap-166:~/projects/book/kriti/pms-be$ nodemon dist/src/main.js
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/src/main.js`
Server is running on port 3000 with pid = 16847
```

Figure 4.4: Output Run Application

Note: If you are using Visual Studio Code, you can configure and run the TypeScript watch task easily. To do this, press "Ctrl + P", then select "Tasks: Run Task" and configure a task with **TypeScript**. Choose "**tsc:watch -tsconfig.json**" to enable watch mode for TypeScript compilation based on your **tsconfig.json** file.

Connecting the Database

TypeORM is widely used in modern TypeScript and JavaScript applications for its ease of use, flexibility, and abstraction of database operations. It simplifies the development process and improves maintainability when working with databases in Node.js. We already installed packages for **postgreSQL** and **typeorm**, respectively, using the following commands: **npm i pg --save**, **npm i @types/pg -D** and **npm i typeorm -- save**.

Now creates **db.ts** file in **utils** directory with the following code:

```
// db.ts

import { DataSource } from 'typeorm';
import { IServerConfig } from './config';
import * as config from '../../server_config.json';
export class DatabaseUtil {
  public server_config: IServerConfig = config;

  constructor() {
    this.connectDatabase();
  }
}
```

```

private connectDatabase() {
  try {
    const db_config = this.server_config.db_config;
    const AppDataSource = new DataSource({
      type: 'postgres',
      host: db_config.host,
      port: db_config.port,
      username: db_config.username,
      password: db_config.password,
      database: db_config dbname,
      entities: [],
      synchronize: true,
      logging: false,
    });
    AppDataSource.initialize()
      .then(() => {
        console.log('Connected to the database');
      })
      .catch((error) => console.log(error));
  } catch (error) {
    console.error('Error connecting to the database:', error);
  }
}
}

```

For now, we have entities as blank arrays since we do not have any entities defined. Then import the database connection in `main.ts` with the updated following code:

```

// main.ts
import cluster from 'cluster';
import { ExpressServer } from './express_server';
import { DatabaseUtil } from './utils/db';
// connect the express server
const server = new ExpressServer();
// connect the database
new DatabaseUtil();
process.on('uncaughtException', (error: Error) => {

```

```

        console.error(`Uncaught exception in worker process
${process.pid}:`, error);

        // Close any open connections or resources
        server.closeServer();

        setTimeout(() => {
            cluster.fork();
            cluster.worker?.disconnect();
        }, 1000);

    });

    // Gracefully handle termination signals
    process.on('SIGINT', () => {
        console.log('Received SIGINT signal');
        // Close any open connections or resources
        server.closeServer();
    });

    process.on('SIGTERM', () => {
        console.log('Received SIGTERM signal');
        // Close any open connections or resources
        server.closeServer();
    });
}

```

After running, the following output will be displayed on a successfully connected database.

Output:



The screenshot shows a terminal window with the following output:

```

GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT
yammini@atpl-lap-166:~/projects/book/kriti/pms-be$ nodemon dist/src/main.js
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/src/main.js`
Server is running on port 3000 with pid = 12272
Connected to the database

```

Figure 4.5: Database Connected Output

This way using typeorm postgreSQL database connected based on the database config defined in **server_config.json**.

Database Models (Entities)

We have already seen database schema tables . To integrate the database schema into the code, we will create a components directory and subdirectories for roles, users, projects, tasks, and comments. In each of these directories, we will create a corresponding entity file.

```
components
  └── roles
    └── roles_entity.ts
  └── users
    └── users_entity.ts
  └── projects
    └── projects_entity.ts
  └── tasks
    └── tasks_entity.ts
  └── comments
    └── comments_entity.ts
```

We will start by defining the Roles entity:

Role Entity

Create file **role_entity.ts** in role directory with the following code.

```
// role_entity.ts

import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn,
UpdateDateColumn } from 'typeorm';

@Entity()
export class Roles {
  @PrimaryGeneratedColumn('uuid')
  role_id: string;

  @Column({ length: 60, nullable: false, unique: true })
  name: string;
```

```

@Column({ length: 200 })
description: string;

@Column({ type: 'text' })
rights: string;

@CreateDateColumn()
created_at: Date;

@UpdateDateColumn()
updated_at: Date;
}

```

The code imports necessary decorators from TypeORM. These decorators are used to define the entity and its properties.

The **@Entity()** decorator marks the class `Roles` as a TypeORM entity, indicating that it represents a database table. It has the `role_id` as the primary key decorated with **PrimaryGeneratedColumn** with an auto-generated UUID, and the name field is defined as unique and not nullable.

The `description` field has a maximum length of 200 characters. The `rights` field uses the `text` data type to store larger textual data. The **@CreateDateColumn()** and **@UpdateDateColumn()** decorators handle the automatic insertion of data for respective insertion and updation time.

User Entity

The `Users` entity represents the user information in the application's database. Create `users_entity.ts` with the following code:

```

// user_entity.ts
import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn,
UpdateDateColumn, OneToOne, JoinColumn } from 'typeorm';
import { Roles } from '../roles/roles_entity';

@Entity()
export class Users {
  @PrimaryGeneratedColumn('uuid')
  user_id: string;

  @Column({ length: 50, nullable: true })

```

```

  fullname: string;

  @Column({ length: 30, nullable: false, unique: true })
  username: string;

  @Column({ length: 60, nullable: false, unique: true })
  email: string;

  @Column({ nullable: false })
  password: string;

  @Column({ nullable: false })
  @ManyToOne(() => Roles)
  @JoinColumn({ name: 'role_id' })
  role_id: Roles['role_id'];

  @CreateDateColumn()
  created_at: Date;

  @UpdateDateColumn()
  updated_at: Date;
}

```

As per user-defined entity, **user_id** serves as the primary key with auto-generated **uuid**, **username** and **email** will be unique with not-null constraints, **fullname** allows maximum 50 characters, and **password** also has a not-null constraints.

In this context, each user is linked with one role, establishing a many-to-one relationship between them, so there is many-to-one join established between two tables users and roles.

@ManyToOne(() => Roles): This decorator specifies a many-to-one relationship between entities. It indicates that the current entity (likely representing a user or some other entity) has a many-to-one relationship with the Roles entity. The Roles entity is specified using a function that returns the target entity class.

@JoinColumn({ name: 'role_id' }): This decorator specifies the column in the current entity that serves as the foreign key to establish the relationship. In this case, the **role_id** column is used as the foreign key.

role_id: Roles['role_id']: This line defines a TypeScript property **role_id** of type **Roles['role_id']**. It's likely that Roles is an entity class

representing roles in your application, and **role_id** is a property of that class representing the primary key or unique identifier of a role.

This is a basic overview of a many-to-one relationship.

Project Entity

Now create a file **projects_entity.ts** with the following code:

```
// projects_entity.ts

import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn,
UpdateDateColumn } from 'typeorm';

@Entity()
export class Projects {
  @PrimaryGeneratedColumn('uuid')
  project_id: string;

  @Column({ length: 30, nullable: false, unique: true })
  name: string;

  @Column({ length: 500 })
  description: string;

  @Column('uuid', { array: true, default: [] })
  user_ids: string[];

  @Column()
  start_time: Date;

  @Column()
  end_time: Date;

  @CreateDateColumn()
  created_at: Date;

  @UpdateDateColumn()
  updated_at: Date;
}
```

The **@PrimaryGeneratedColumn('uuid')** decorator specifies that the **project_id** property is the primary key of the **projects** table and will be automatically generated as a **UUID** when a new project is inserted.

The `@Column({ length: 30, nullable: false, unique: true })` decorator is applied to the `name` property, indicating that it is a string with a maximum length of 30 characters. It is also marked as `nullable: false` (required) and `unique: true` (must be unique among projects).

The `@Column({ length: 500 })` decorator is used for the `description` property, representing a string with a maximum length of 500 characters.

The `@Column('uuid', { array: true, default: [] })` decorator is applied to the `user_ids` property, indicating that it is an array of strings (`string[]`) that will store user IDs associated with the project. The default value for the array is set to an empty array (`[]`).

The `@Column()` decorator is used for both the `start_time` and `end_time` properties, which represent `Date` objects.

In the context of the application, every project comprises multiple tasks, and each task is linked to a single project. Consequently, within the `tasks` entity, there exists a one-to-one relationship with the `project` entity. Similarly `user` and `role` joins here `project` and `task` join will be established with join columns as `project_id` for both tables in the following `task` entity that we will explore in detail.

Task Entity

Create a new file named `tasks_entity.ts` and include the following code in it:

```
// tasks_entity.ts

import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn,
UpdateDateColumn, ManyToOne, JoinColumn } from 'typeorm';
import { Users } from '../users/users_entity';
import { Projects } from '../projects/projects_entity';

export enum Status {
  NotStarted = 'Not-Started',
  InProgress = 'In-Progress',
  Completed = 'Completed',
}

export enum Priority {
```

```
Low = 'Low',
Medium = 'Medium',
High = 'High',
}

@Entity()
export class Tasks {
    @PrimaryGeneratedColumn('uuid')
    task_id: string;

    @Column({ length: 30, nullable: false, unique: true })
    name: string;

    @Column({ length: 500 })
    description: string;

    @Column()
    @ManyToOne(() => Projects, (projectData) =>
    projectData.project_id)
    @JoinColumn({ name: 'project_id' })
    project_id: string;

    @Column()
    @ManyToOne (() => Users, (userData) => userData.user_id)
    @JoinColumn({ name: 'user_id' })
    user_id: string;

    @Column()
    estimated_start_time: Date;

    @Column()
    estimated_end_time: Date;

    @Column()
    actual_start_time: Date;

    @Column()
    actual_end_time: Date;

    @Column({
        type: 'enum',
        enum: Priority, // Use the enum type here
    })
}
```

```

    default: Priority.Low, // Set a default value as Low
  })
priority: Priority;

@Column({
  type: 'enum',
  enum: Status, // Use the enum type here
  default: Status.NotStarted, // Set a default value as Not-
  Started
})
status: Status;

@Column('text', { array: true, default: [] })
supported_files: string[];

CreateDateColumn()
created_at: Date;

UpdateDateColumn()
updated_at: Date;
}

```

The code imports necessary decorators from TypeORM, including `Entity`, `PrimaryGeneratedColumn`, `Column`, `CreateDateColumn`, and `UpdateDateColumn`. It also imports the `Users` and `Projects` entities, as they will be used to establish one-to-one relationships with the `Tasks` entity.

`Priority` and `Status` `enums` define possible values for the status and priority columns of the `Tasks` entity. By using `enums`, we ensure that only specific predefined values can be assigned to these columns.

`@ManyToOne` decorators establish many-to-one relationships with the `Projects` and `Users` entities. The first argument of `@ManyToOne` is a function that returns the related `entity` class. This signifies that many tasks can be associated with one project or user. This second argument denotes the inverse side of the relationship, indicating the properties `project_id` and `user_id` in the `Projects` and `Users` entities, respectively, which reference the `tasks` entity. The `@JoinColumn` decorator specifies the name of the foreign key column in the `tasks` table that links to the related `Projects` and `Users` entities.

Comment Entity

Let us define the comment entity in **comments_entity.ts** with the following code:

```
// comments_entity.ts

import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn,
UpdateDateColumn, JoinColumn, OneToOne } from 'typeorm';
import { Users } from '../users/users_entity';
import { Tasks } from '../tasks/tasks_entity';

@Entity()
export class Comments {
  @PrimaryGeneratedColumn('uuid')
  comment_id: string;

  @Column({ type: 'text' })
  comment: string;

  @OneToOne(() => Users, (userData) => userData.user_id)
  @JoinColumn({ name: 'user_id' })
  user_id: string;

  @OneToOne(() => Tasks, (taskData) => taskData.task_id)
  @JoinColumn({ name: 'task_id' })
  task_id: string;

  @Column('text', { array: true, default: [] })
  supported_files: string[];

  @CreateDateColumn()
  created_at: Date;

  @UpdateDateColumn()
  updated_at: Date;
}
```

The provided code includes essential TypeORM decorators like **Entity**, **PrimaryGeneratedColumn**, **Column**, **CreateDateColumn**, and **UpdateDateColumn**. Additionally, it imports the **Users** and **Tasks** entities,

which play a crucial role in setting up one-to-one relationships with the **Comments** entity.

The `@PrimaryGeneratedColumn('uuid')` decorator specifies that the **comment_id** property is the primary key of the comments table and will be automatically generated as a UUID when a new comment is inserted. The **comment** property is of type text, allowing it to store larger textual data.

`@OneToOne` decorators establish one-to-one relationships with the **Users** and **Tasks** entities. The first argument of `@OneToOne` is a function that returns the related **entity** class. Meanwhile, the second argument defines the inverse side of the relationship, referring to the properties **user_id** and **task_id** in the **Users** and **Tasks** entities, respectively. The `@JoinColumn` decorator then specifies the name of the foreign key column in the **comments** table, which references the related **Users** and **Tasks** entities.

The **supported_files** property is an array of strings (`string[]`) which will store the urls of different files in the array.

After defining all the entities, their purpose remains incomplete without synchronization to the database. So, let us synchronize them with the database and automatically create the corresponding tables. To achieve this, we can update the **db.ts** file with the following code:

```
// db.ts

import { DataSource } from 'typeorm';
import { IServerConfig } from './config';
import * as config from '../../server_config.json';
import { Roles } from '../components/roles/roles_entity';
import { Users } from '../components/users/users_entity';
import { Projects } from
  '../components/projects/projects_entity';
import { Tasks } from '../components/tasks/tasks_entity';
import { Comments } from
  '../components/comments/comments_entity';

export class DatabaseUtil {
  public server_config: IServerConfig = config;

  constructor() {
    this.connectDatabase();
  }
}
```

```

private connectDatabase() {
  try {
    const db_config = this.server_config.db_config;
    const AppDataSource = new DataSource({
      type: 'postgres',
      host: db_config.host,
      port: db_config.port,
      username: db_config.username,
      password: db_config.password,
      database: db_config dbname,
      entities: [Roles, Users, Projects, Tasks, Comments],
      synchronize: true,
      logging: false,
    });
    AppDataSource.initialize()
      .then(() => {
        console.log('Connected to the database');
      })
      .catch((error) => console.log(error));
  } catch (error) {
    console.error('Error connecting to the database:', error);
  }
}
}

```

The entities array should include all the entity classes you have defined (**Users**, **Projects**, **Tasks**, and **Comments**) to be synchronized with the database.

Setting `synchronize: true` ensures that the database tables are automatically created or updated to match the defined entities.

Once you run this code, the defined entities will be synchronized with the database, and the corresponding tables will be created or updated accordingly.

Routes

Routing plays a crucial role in modern application development. Routing provides a structured way to navigate through an application. By establishing routes, users can seamlessly interact with distinct sections of the application through designated URLs.

Routing is basically a way to respond to any request coming to a specific endpoint based on method and parameters.

Route Definition follows this pattern:

`app.METHOD(PATH, HANDLER)`

"**app**" refers to an instance of Express.

"**METHOD**" pertains to an HTTP request method, written in lowercase.

"**PATH**" designates a specific server path.

"**HANDLER**" represents the function executed upon matching the route.

Routes consist of basically four parts:

- **URL Paths:**

Each route definition includes a URL path or pattern that users can enter in the browser's address bar or click as links.

For example, `"/api/login"`

- **HTTP Method:**

Routes specify the HTTP methods (**GET**, **POST**, **PUT**, **DELETE**, and so on) that are allowed for each path. Different HTTP methods trigger different actions in the application. For example, a **GET** request might retrieve data, while a **POST** request might submit data. You can explore it in detail through this link <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

- **Route Handlers:**

For each URL path, there is a corresponding handler function or controller that defines the actual logic for functionality to be executed. This can include rendering a specific view, fetching data from a database, processing user input, and more.

- **Route Parameters:** Some paths might include dynamics that act as placeholders for specific values. These are often indicated with a colon, like "`:id`".

For example, `"/users/:id"` could represent a user's profile page, where `:id` is replaced with the actual user's ID.

Let us construct routes following the outlined structure:

```
├── routes
|   └── index.ts
└── components
    ├── roles
    |   └── roles_entity.ts
    |   └── roles_routes.ts
    |   └── roles_controller.ts
    ├── users
    |   └── users_entity.ts
    |   └── users_routes.ts
    |   └── users_controller.ts
    ├── projects
    |   └── projects_entity.ts
    |   └── projects_routes.ts
    |   └── projects_controller.ts
    ├── tasks
    |   └── tasks_entity.ts
    |   └── tasks_routes.ts
    |   └── tasks_controller.ts
    └── comments
        └── comments_entity.ts
        └── comments_routes.ts
        └── comments_controller.ts
```

By the preceding structure, we will establish a `"routes"` directory, containing an `"index.ts"` file, within the `"src"` directory. This `"index.ts"` file will function as the root for the individual routes of each module. We will also create all individual routes and controller files for each component and update them as we develop APIs step by step.

Each module encompasses functionalities for adding, updating, retrieving, and deleting, along with obtaining specific details about a particular data. Each route can cover **CRUD (Create, Read, Update, Delete)** operations having a basepoint as `"/api/componentname"` where `componentname` can be `roles, users, projects, task` and `comments`:

- **Create:** This route allows the addition of new data. It is typically associated with an HTTP **POST** request to a path ex. `"/api/componentname"`.
- **Update:** The update route lets you modify existing data. It is triggered by an HTTP **PUT** request to a path like `"/api/componentname/:id"`, where `:id` represents the specific entries identifier.
- **Retrieve All:** This route enables the retrieval of a list of all roles. It is often invoked via an HTTP **GET** request to the `"/api/componentname"` path.
- **Retrieve Specific:** This route is used to fetch details about a specific role. It is initiated by an HTTP **GET** request to a path like `"/api/componentname/:id"`.
- **Delete:** The delete route allows the removal of roles. It is typically triggered by an HTTP **DELETE** request to a path like `"/api/componentname/:id"`.

Role Routes

Now, update `role_controller.ts` file with empty skeleton functions.

```
// role_controller.ts

export class RoleController {

  public addHandler() {
    // addHandler
  }
  public getAllHandler() {
    // getAllHandler
  }
  public getDetailsHandler() {
    // getDetailsHandler
  }
  public async updateHandler() {
    // updateHandler
  }
  public async deleteHandler() {
    // deleteHandler
  }
}
```

```
}
```

Afterwards, modified **role_routes.ts** with the following code:

```
// role_routes.ts

import { Express } from 'express';
import { RoleController } from './roles_controller';
export class RoleRoutes {

    private baseEndPoint = '/api/roles';

    constructor(app: Express) {

        const controller = new RoleController();

        app.route(this.baseEndPoint)
            .get(controller.getAllHandler)
            .post(controller.addHandler);

        app.route(this.baseEndPoint + '/:id')
            .get(controller.getDetailsHandler)
            .put(controller.updateHandler)
            .delete(controller.deleteHandler);
    }
}
```

Similarly, we will update all controller files by changing class names, such as **UserController**, **ProjectController**, **TaskController**, and **CommentController**, and import them in respective route files with modifying basepoint.

User Routes

Let us modify **users_controller.ts** and **users_routes.ts** files, respectively, as follows:

```
// users_controller.ts

export class UserController {

    public addHandler() {
        // addHandler
    }
    public getAllHandler() {
```

```

    // getAllHandler
}
public getDetailsHandler() {
    // getDetailsHandler
}
public async updateHandler() {
    // updateHandler
}
public async deleteHandler() {
    // deleteHandler
}
}

// users_routes.ts
import { Express } from 'express';
import { UserController } from './users_controller';
export class UserRoutes {

    private baseEndPoint = '/api/users';

    constructor(app: Express) {

        const controller = new UserController();

        app.route(this.baseEndPoint)
            .get(controller.getAllHandler)
            .post(controller.addHandler);

        app.route(this.baseEndPoint + '/:id')
            .get(controller.getDetailsHandler)
            .put(controller.updateHandler)
            .delete(controller.deleteHandler);
    }
}

```

Project Routes

Let us modify `projects_controller.ts` and `projects_routes.ts` files, respectively, as follows:

```
// project_controller.ts
```

```
export class ProjectController {  
  public addHandler() {  
    // addHandler  
  }  
  public getAllHandler() {  
    // getAllHandler  
  }  
  public getDetailsHandler() {  
    // getDetailsHandler  
  }  
  public async updateHandler() {  
    // updateHandler  
  }  
  public async deleteHandler() {  
    // deleteHandler  
  }  
}  
  
// projects_routes.ts  
import { Express } from 'express';  
import { ProjectController } from './projects_controller';  
export class ProjectRoutes {  
  private baseEndPoint = '/api/projects';  
  constructor(app: Express) {  
    const controller = new ProjectController();  
    app.route(this.baseEndPoint)  
      .get(controller.getAllHandler)  
      .post(controller.addHandler);  
    app.route(this.baseEndPoint + '/:id')  
      .get(controller.getDetailsHandler)  
      .put(controller.updateHandler)  
      .delete(controller.deleteHandler);  
  }  
}
```

Task Routes

Let us modify `tasks_controller.ts` and `tasks_routes.ts` files, respectively, as follows:

```
// tasks_controller.ts

export class TaskController {

    public addHandler() {
        // addHandler
    }

    public getAllHandler() {
        // getAllHandler
    }

    public getDetailsHandler() {
        // getDetailsHandler
    }

    public async updateHandler() {
        // updateHandler
    }

    public async deleteHandler() {
        // deleteHandler
    }
}

// tasks_routes.ts

import { Express } from 'express';
import { TaskController } from './tasks_controller';
export class TaskRoutes {

    private baseEndPoint = '/api/tasks';

    constructor(app: Express) {

        const controller = new TaskController();

        app.route(this.baseEndPoint)
            .get(controller.getAllHandler)
            .post(controller.addHandler);

        app.route(this.baseEndPoint + '/:id')
```

```

        .get(controller.getDetailsHandler)
        .put(controller.updateHandler)
        .delete(controller.deleteHandler);
    }
}

```

Comment Routes

Let us modify `comments_controller.ts` and `comments_routes.ts` files, respectively, as follows:

```

// comments_controller.ts

export class CommentController {

    public addHandler() {
        // addHandler
    }

    public getAllHandler() {
        // getAllHandler
    }

    public getDetailsHandler() {
        // getDetailsHandler
    }

    public async updateHandler() {
        // updateHandler
    }

    public async deleteHandler() {
        // deleteHandler
    }
}

// comments_routes.ts

import { Express } from 'express';
import { CommentController } from './comments_controller';
export class CommentRoutes {

    private baseEndPoint = '/api/comments';

    constructor(app: Express) {

```

```

const controller = new CommentController();

app.route(this.baseEndPoint)
  .get(controller.getAllHandler)
  .post(controller.addHandler);

app.route(this.baseEndPoint + '/:id')
  .get(controller.getDetailsHandler)
  .put(controller.updateHandler)
  .delete(controller.deleteHandler);
}

}

```

We define all routes based on components individually. However, these routes need to be initialized from the Express server, so import all routes in one file and call it from the Express server.

Create **routes** directory and **index.ts** with the following code:

```

// index.ts
import { Express, Router } from 'express';
import { RoleRoutes } from '../components/roles/roles_routes';
import { UserRoutes } from '../components/users/users_routes';
import { ProjectRoutes } from
  '../components/projects/projects_routes';
import { TaskRoutes } from '../components/tasks/tasks_routes';
import { CommentRoutes } from
  '../components/comments/comments_routes';

export class Routes {
  public router: Router;

  constructor(app: Express) {

    const routeClasses = [
      RoleRoutes,
      UserRoutes,
      ProjectRoutes,
      TaskRoutes,
      CommentRoutes
    ];

    for (const routeClass of routeClasses) {

```

```

        try {
            new routeClass(app);
            console.log(`Router : ${routeClass.name} - Connected`);
        } catch (error) {
            console.log(`Router : ${routeClass.name} - Failed`);
        }
    }
}
}

```

Afterward, modify **express_server.ts** with import **routes** in the app:

```

// express_server.ts
import express from 'express';
import { IServerConfig } from './utils/config';
import * as config from '../server_config.json';
import { Routes } from './routes';

export class ExpressServer {

    private static server = null;
    public server_config: IServerConfig = config;
    constructor() {
        const port = this.server_config.port ?? 3000;

        // initialize express app
        const app = express();

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

        app.get('/ping', (req, res) => {
            res.send('pong');
        });

        const routes = new Routes(app);
        if (routes) {
            console.log('Server Routes started for server');
        }
    }

    ExpressServer.server = app.listen(port, () => {
        console.log(`Server is running on port ${port} with pid =`);
    });
}

```

```

        `${process.pid}`);
    });
}

//close the express server for safe on uncaughtException
public closeServer(): void {
    ExpressServer.server.close(() => {
        console.log('Server closed');
        process.exit(0);
    });
}
}

```

While running the application, the following output will display in the terminal:

Output:

```

Router : RoleRoutes - Connected
Router : UserRoutes - Connected
Router : ProjectRoutes - Connected
Router : TaskRoutes - Connected
Router : CommentRoutes - Connected
Server Routes started for server
Server is running on port 5000 with pid = 251784
Connected to the database

```

Conclusion

At this moment, now our server is running with a database utility, all entities along with their routes.

We learned about the use of the cluster module, using TypeORM for our database connection and query needs. We have our entities with all required columns.

As of now, we have empty and un-implemented route functions. In the next chapters, we will take each entity and implement the API.

Multiple Choice Questions

1. What is the primary goal of any project management software, including the PMS (Project Management System) discussed here?
 - a. To develop web applications
 - b. To allow individuals and teams to plan and execute tasks
 - c. To organize data effectively
 - d. To provide secure user login
2. Which modules are considered must-have in the PMS system based on the defined tasks?
 - a. User Module, Project Module, Task Module, and Comment Module
 - b. Planning Module, API Development Module, and Unit Testing Module
 - c. Communication Module, Deployment Module, and Database Design Module
 - d. API Caching Module, Project Reports Module, and User Profile Module
3. What is the primary objective of normalization in database design?
 - a. To increase data redundancy
 - b. To maintain data integrity and avoid data duplication
 - c. To slow down query performance
 - d. To increase the complexity of queries
4. What role does the Role Table play in the PMS database?
 - a. It defines the rights and permissions of users
 - b. It stores project-related data
 - c. It manages user profiles
 - d. It records project creation and modification times
5. Which file is typically generated as a result of running "npm init" for a Node.js project?
 - a. package-lock.json
 - b. tsconfig.json

- c. server_config.json
 - d. package.json
6. What does the "tsconfig.json" file specify in a TypeScript project?
- a. The project's description
 - b. The project's dependencies
 - c. TypeScript compiler options and settings
 - d. The project's test commands
7. Which of the following decorators from TypeORM is used to mark a class as a TypeORM entity representing a database table?
- a. @Entity()
 - b. @Table()
 - c. @Database()
 - d. @Model()
8. Which TypeScript feature is used to define and enforce specific predefined values for the "status" and "priority" columns in the "Tasks" entity?
- a. Type assertions
 - b. Type inference
 - c. Enums
 - d. Generics
9. What is the purpose of the @JoinColumn decorator in the "Tasks" entity class?
- a. To specify the name of the entity class
 - b. To define a foreign key constraint
 - c. To specify the name of the foreign key column
 - d. To create a new table in the database
10. How would you trigger the update route for modifying existing data in Express.js?
- a. Send an HTTP POST request to a path like "/api/componentname"

- b. Send an HTTP PUT request to a path like "/api/componentname/:id"
- c. Send an HTTP GET request to a path like "/api/componentname"
- d. Send an HTTP DELETE request to a path like "/api/componentname/:id"

Answers

- 1. b
- 2. a
- 3. b
- 4. a
- 5. d
- 6. c
- 7. a
- 8. c
- 9. c
- 10. b

Further Reading

<https://www.postgresqltutorial.com>

<https://typeorm.io>

<https://www.npmjs.com/package/pg>

OceanofPDF.com

CHAPTER 5

REST API for User Module

Introduction

At the heart of any application lies the User module, a foundational component that orchestrates the management of user-centric features. This module allows users to administer the user accounts, enables authentication and authorization, and various user-specific operations, such as adding or registering users, updating user profiles, deleting users, password management, role-based permission, logging, and many more. The User module enhances the user experience and promotes the seamless operation of the application.

Structure

In this chapter, we will discuss the following topics:

- Base Controller and Base Service for REST API development
- Role Management with Input Validation
- User Management with Input Validation
 - User Onboarding
 - User Sign-In
 - Authentication and Authorization Mechanism
 - Update User Data
 - Delete User Account
 - Password Management and Recovery with Email Notification

Base Controller

We have seen in the previous chapter how we constructed individual controllers for each module. For each controller, there were some common methods, such as handlers for add, get all, get one, update, and so on. Since

there is one controller for each entity, this set of common functions must be implemented by each controller. We could write an abstract class for the controllers to extend and force them to provide an implementation. Therefore, adhering to established norms, we are now introducing a foundational concept known as **Base Controller**. It is an abstract class that serves as a blueprint for other classes to inherit its predefined methods that facilitate operations like creating, updating, retrieving all, retrieving one, and deleting data.

Let's create the `base_controller.ts` in the `utils` directory using the following code:

```
// base_controller.ts
import { Request, Response } from 'express';

export abstract class BaseController {
  public abstract addHandler(req: Request, res: Response): void;
  public abstract getAllHandler(req: Request, res: Response): void;
  public abstract getOneHandler(req: Request, res: Response): void;
  public abstract updateHandler(req: Request, res: Response): void;
  public abstract deleteHandler(req: Request, res: Response): void;
}
```

The purpose of this abstract class is to furnish a uniform framework that other classes (controllers) can adhere to while implementing these methods. When a class extends the **BaseController**, it becomes obligatory to furnish implementations for these abstract methods. This practice guarantees that controllers across various routes maintain consistent method names and parameters, even though the specific execution details might vary.

Please note that an individual controller can still write their own additional methods.

The code for this chapter can be downloaded from <https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/tree/main/ch-05-code-files>

Base Service

Base Service is a foundational structure that provides common functionalities for managing data operations in an application. It serves as a blueprint that other services can inherit to avoid repetitive code and ensure consistent patterns for data manipulation. The primary purpose of a base service is to encapsulate commonly used data operations, such as create, read, update, and delete, and make them available to other services. This reduces code duplication and enforces consistent practices across different modules of an application. Other services requiring data operations can inherit from the base service. By extending the base service, these child services gain access to the common methods defined in the base service.

Let's create the **base_service.ts** using the following code:

https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/base_service.ts

If you had downloaded the source code for this chapter, then you can find the **base_service.ts** inside. We have added comments for each method to explain what it does and how it works. This base service class provides common CRUD operations, **findByIds**, and a custom query runner along with handling API responses and error cases. This service class can be inherited by other services.

It is important to manage database connections efficiently. If we make separate database connections for each operation, the application would likely crash during high loads. Using a database pool is the best practice to limit and reuse the connections from a pool.

Let's change the **db.ts** file using the following code for adding a database pool and to use a single connection in the whole app:

<https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/db.ts>

The **connectDatabase** function is responsible for establishing a database connection or returning an existing connection if available. It first checks if a valid connection already exists, and if not, it initializes a new connection and stores it for future use.

The **getInstance** function retrieves a database instance, ensuring it is connected before providing access. Unlike the **connectDatabase** function, **getInstance** waits until the connection is established before returning, ensuring that it can only be used once the connection is ready.

The **getRepository** function is designed to retrieve a repository instance for a given entity. It checks if a valid database connection exists and creates the repository instance if it doesn't already exist. If there's no valid connection, it returns null.

Role Management

Role management is a critical aspect of application security that involves controlling and defining the access and permissions of users within a system. It ensures that users have the appropriate rights to perform specific actions based on their roles or responsibilities. Role management is essential in preventing unauthorized access and data breaches and maintaining the overall integrity of an application.

Different users have different levels of access and privileges. For example, an application might have different user roles such as **Super Admin**, **Project Manager**, and **Guest**, each with distinct sets of permissions. Admins typically have access to all features and functionalities, project managers have limited access, and guests might have very restricted access.

Role Service

Role Service will be used to perform role-based operations in a database that extends from the base service. So, let's create the **roles_service.ts** file in the roles component using the following code:

```
// role_service.ts
import { Repository } from 'typeorm';
import { BaseService } from '../../../../../utils/base_service';
import { DatabaseUtil } from '../../../../../utils/db';
import { Roles } from './roles_entity';

export class RolesService extends BaseService<Roles> {

  constructor() {
    // Create an instance of DatabaseUtil
```

```

const databaseUtil = new DatabaseUtil();

// Get the repository for the Roles entity
const roleRepository: Repository<Roles> =
databaseUtil.getRepository(Roles);
// Call the constructor of the BaseService class with the
// repository as a parameter
super(roleRepository);
}
}

```

The **RolesService** class is designed to extend the functionality provided by the **BaseService** class. It uses the **DatabaseUtil** class to get the repository for the **Roles** entity and then passes that repository to the constructor of the **BaseService** class. This allows the **RolesService** class to inherit and use the CRUD methods defined in the **BaseService** class for working with the **Roles** entity.

We will develop the REST API for roles as follows:

- Add Roles
- Get All Roles
- GetOne Role
- Update Role
- Delete Role

Before developing the actual **Add Role** API, we need to define input validation for roles while adding them to the database. So, let's use an Express Validator to validate input requests.

Input Validation

First, install the Express Validator module so paste the following command in **cmd**:

```
npm i express-validator --save
```

Now create the **validator.ts** file in the **utils** directory using the following code:

<https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms->

[be/src/utils/validator.ts](#)

The provided code exports a function named **validate** that generates **middleware** for validating request data using the express-validator package. This middleware function runs the provided validation functions, checks for validation errors using **validationResult**, and sends a response with a 400 status and error messages if validation fails. The structure of the error messages aligns with the **IValidationResult** interface. This approach is commonly used to handle request validation in Express applications.

Next, create the **validRoleInput** in the **roles_routers.ts** file using the following code:

```
// roles_routers.ts
import { Express } from 'express';
import { RoleController, RolesUtil } from './roles_controller';
import { validate } from '../../utils/validator';
import { body } from 'express-validator';

const validRoleInput = [
  body('name').trim().notEmpty().withMessage('It should be required'),
  body('description').isLength({ max: 200 }).withMessage('It has maximum limit of 200 characters'),
];

export class RoleRoutes {
  private baseEndPoint = '/api/roles';

  constructor(app: Express) {
    const controller = new RoleController();

    app.route(this.baseEndPoint)
      .get(controller.getAllHandler)
      .post(validate(validRoleInput), controller.addHandler);

    app.route(this.baseEndPoint + '/:id')
      .get(controller.getOneHandler)
      .put(validate(validRoleInput), controller.updateHandler)
      .delete(controller.deleteHandler);
  }
}
```

In the preceding class, the `baseEndPoint` variable is used. This is part of the API endpoints, which is going to be the same for all of the role APIs.

Notice the `validRoleInput` variable, which is an array. This array contains a series of validation checks for each input field expected in a role. Each element of this array is a validator function that checks a specific aspect of the data.

The validator for the `name` field in request body would be processed as follows:

```
body('name').trim().notEmpty().withMessage('It should be required')
```

This validator is applied to the `name` field in the request body and performs the following functions:

- `trim()`: Removes any leading and trailing whitespace from the input.
- `notEmpty()`: Checks that the input is not empty.
- `withMessage('It should be required')`: If the validation fails, this message will be included in the error response.

Similarly, the validator for the `description` field would be as follows:

```
body('description').isLength({ max: 200 }).withMessage('It has a maximum limit of 200 characters')
```

This validator is applied to the `description` field in the request body. It checks that the length of the input does not exceed `200` characters.

Overall, this code defines a set of validation rules for different fields of role data, including checking for the presence, length, and validity of access rights. If any of the validations fail, the corresponding error message will be included in the error response with status code `400: bad request`.

Each role consists of a set of rights. These rights are nothing but string keys, which can help us to understand whether a logged-in user has a particular right assigned to let them do a corresponding task. A right for adding a task could be as simple as `add_task`, which could be added to the role assigned to the user.

Let's define all of the necessary application rights in `common.ts` file as follows:

<https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/common.ts>

These rights are application rights, which are used to check permission during user login based on the assigned Role. When a role is created, it has a rights value that is saved as comma-separated from these application rights.

Let's create a **RolesUtil** class with **getAllPermissionsFromRights** function in the **role_controller.ts** file using the following code:

```
// role_controller.ts
export class RolesUtil {

    /**
     * Retrieves all possible permissions from the defined rights
     * in the Rights object.
     * @returns {string[]} An array of permissions
     */
    public static getAllPermissionsFromRights(): string[] {

        // Initialize an empty array to collect values;
        let permissions = [];

        // Iterate through each section of the Rights object
        for (const module in Rights) {

            // Check if rights for ALL are defined for the current
            // module
            if (Rights[module]['ALL']) {
                let sectionValues = Rights[module]['ALL'];
                sectionValues = sectionValues.split(',');
                permissions = [...permissions, ...sectionValues];
            }
        }
        // Return the collected permissions
        return permissions;
    }
}
```

This function effectively compiles all the available permissions from the defined Rights object, which can then be used for validation and other

purposes in the application.

Now add validation for the **rights** field in **validRoleInput** using the following code:

```
const validRoleInput = [
  body('name').trim().notEmpty().withMessage('It should be
  required'),
  body('description').isLength({ max: 200 }).withMessage('It has
  maximum limit of 200 characters'),
  body('rights').custom((value: string) => {
    const accessRights = value?.split(',');
    if (accessRights?.length > 0) {
      const validRights =
        RolesUtil.getAllPermissionsFromRights();
      const areAllRightsValid = accessRights.every(right =>
        validRights.includes(right));
      if (!areAllRightsValid) {
        throw new Error('Invalid permission');
      }
    }
    return true; // Validation passed
  })
];
```

Based on the provided code, the custom validation function ensures the validity of rights during the process of adding or updating a role. It evaluates the rights received in the request and verifies whether they are valid or not. If any of the rights are found to be invalid, an error is generated.

Add Role

When using the REST API to add a role, you typically provide the necessary data in the request body, such as the role's name, description, and the rights it should have. The API endpoint responsible for this action is designed to receive this data, validate it according to predefined rules, and create a new role based on the provided information.

We have already created **roles_controller.ts** as a skeleton class. Now, let's change it with an extended Base Controller and use the base service to perform database operations using the following code:

```

// roles_controller.ts
import { Response, Request } from 'express';
import { RolesService } from './roles_service';
import { BaseController } from '../../utils/base_controller';
import { Rights } from '../../utils/common';
export class RoleController extends BaseController {

  public async addHandler(req: Request, res: Response): Promise<void> {
    const role = req.body;
    const service = new RolesService();
    const result = await service.create(role);
    res.status(result.statusCode).json(result);
  }

  public async getAllHandler(req: Request, res: Response) {}
  public async getOneHandler(req: Request, res: Response) {}
  public async updateHandler(req: Request, res: Response) {}
  public async deleteHandler(req: Request, res: Response) {}
}

```

In this context, the `addHandler` method captures the data provided within the incoming request body. Subsequently, it forwards this data to the Base service by invoking the create method, which is responsible for adding the information to the database. The Base service then returns a response to this handler, signifying either a successful or unsuccessful outcome. This response is effectively transmitted as the ultimate outcome for the associated REST API operation.

Now call **addHandler** in role routes with change in **roles_router.ts** file as follows:

```

//roles_router.ts
export class RoleRoutes {

  private baseEndPoint = '/api/roles';

  constructor(app: Express) {

    const controller = new RoleController();
    app.route(this.baseEndPoint)

```

```
        .post(validate(validRoleInput), controller.addHandler);  
    }  
}
```

We have established routes for adding roles and incorporated middleware to validate requests before inserting data into the database.

You can test the REST APIs in Postman, cURL, or .http file with the following request and get their responses. The easiest method is to simply use VSCode, install REST Client extension, and create a new file with the .http extension and put the request as shown in [Figure 5.1](#):

```
### Create a new Role  
Send Request  
POST http://127.0.0.1:3000/api/roles  
Content-Type: application/json  
{  
    "name": "Super Admin",  
    "description": "Having All Rights",  
    "rights": "add_role,edit_role,get_all_roles,get_details_role,delete_role",  
}
```

Figure 5.1: Making a request using VSCode REST Client Extension

For testing the API for adding a role, we need to make a **POST** call to **/api/roles**. The preceding figure shows the way it can be called with the request body.

REST API Add Role

Request

```
URL : http://127.0.0.1:3000/api/roles  
Method: POST  
body :  
{  
    "name": "Super Admin",  
    "description": "Having All Rights",  
    "rights": "add_role,edit_role,get_all_roles,get_details_role,  
    delete_role,  
    add_user,edit_user,get_all_users,get_details_user,delete_user,  
    add_project,edit_project,get_all_projects,get_details_project,  
    delete_project,add_task,edit_task,get_all_tasks,get_details_task,  
    delete_task,add_comment,edit_comment,get_all_comments,
```

```
get_details_comment,delete_comment"
}
```

Response

```
{
  "statusCode": 201,
  "status": "success",
  "data": {
    "role_id": "b88cc70d-ab0a-4464-9562-f6320df519f6",
    "name": "Super Admin",
    "description": "Having All Rights",
    "rights": "add_role,edit_role,get_all_roles,get_details_role,
delete_role,add_user,edit_user,get_all_users,get_details_user,
delete_user,add_project,edit_project,get_all_projects,
get_details_project,delete_project,add_task,edit_task,get_all_t
asks,
get_details_task,delete_task,add_comment,edit_comment,
get_all_comments,get_details_comment,delete_comment",
    "created_at": "2023-08-16T16:39:14.047Z",
    "updated_at": "2023-08-16T16:39:14.047Z"
  }
}
```

In case of a unique role name, trying again with the same request gives an error as 409 conflict code:

```
{
  "statusCode": 409,
  "status": "error",
  "message": "Key (name)=(Super Admin) already exists."
}
```

In another case, if you change the rights as "rights":"no_rights", it gives an error for Bad Request with 400 status as follows:

```
{
  "statusCode": 400,
  "status": "error",
  "errors": [
    {
      "rights": "Invalid permission"
    }
  ]
}
```

```
        }
    ]
}
```

GetAll Roles

Once a role has been successfully added to the database, we can proceed to retrieve the newly inserted roles from the database. So, let's change the `getAllHandler` method in the `roles_controller.ts` file using the following code:

```
// roles_controller.ts
public async getAllHandler(req: Request, res: Response): Promise<void> {
    const service = new RolesService();
    const result = await service.findAll(req.query);
    res.status(result.statusCode).json(result);
}
```

The `getAllHandler` method uses the `RolesService` class to retrieve all roles from the database based on the query parameters in the request. The resulting data is then sent back to the client with an appropriate HTTP status code and formatted as JSON.

This `controller` method call in routes with change in `roles_routes.ts` as follows:

```
// roles_routes.ts
app.route(this.baseEndPoint)
    .post(validate(validRoleInput), controller.addHandler)
    .get(controller.getAllHandler);
```

By employing this approach, we establish a `GET` route that fetches all roles stored in the database, effectively functioning as a REST API endpoint for retrieving role data.

REST API GetAll Roles

Request

URL : `http://127.0.0.1:3000/api/roles`
Method: GET
Query Params: {}

Response

```
{  
  "statusCode": 200,  
  "status": "success",  
  "data": [  
    {  
      "role_id": "b88cc70d-ab0a-4464-9562-f6320df519f6",  
      "name": "Super Admin",  
      "description": "Having All Rights",  
      "rights":  
        "add_role,edit_role,get_all_roles,get_details_role,delete_r  
        ole,add_user,edit_user,get_all_users,get_details_user,  
      delete_user,add_project,edit_project,get_all_projects,  
      get_details_project,delete_project,add_task,edit_task,  
      get_all_tasks,get_details_task,delete_task,add_comment,  
      edit_comment,get_all_comments,get_details_comment,delete_commen  
      t",  
      "created_at": "2023-08-16T16:39:14.047Z",  
      "updated_at": "2023-08-16T16:39:14.047Z"  
    }, {  
      "role_id": "5f11a06b-e9a7-438d-9f49-757e8239e238",  
      "name": "visitor",  
      "description": null,  
      "rights": null,  
      "created_at": "2023-08-15T13:04:50.314Z",  
      "updated_at": "2023-08-15T13:04:50.314Z"  
    }  
  ]  
}
```

If you want to filter or search by exact name, you can change query params as follows:

```
URL : http://127.0.0.1:5000/api/roles?name=visitor  
It gives only matched data as a response, as follows  
{  
  "statusCode": 200,  
  "status": "success",  
  "data": [
```

```

{
  "role_id": "5f11a06b-e9a7-438d-9f49-757e8239e238",
  "name": "visitor",
  "description": null,
  "rights": null,
  "created_at": "2023-08-15T13:04:50.314Z",
  "updated_at": "2023-08-15T13:04:50.314Z"
}
]
}

```

Note: In base service, query params will now be applicable for only exact matches. We will explore search functionality later on.

GetOne Role

GetOne role endpoint is a fundamental part of role management systems, allowing users to view specific role information without having to retrieve the entire list of roles. It's essential for providing targeted insights into each role's attributes and permissions.

To implement the GetOne Role API, make the following changes in the **getOneHandler** code in the role controller:

```

// roles_controller.ts
public async getOneHandler(req: Request, res: Response): Promise<void> {
  const service = new RolesService();
  const result = await service.findOne(req.params.id);
  res.status(result.statusCode).json(result);
}

```

The **getOneHandler** function serves as the bridge between the incoming client request, the service layer that interacts with the database, and the outgoing HTTP response. It retrieves a single role's details from the database based on the provided role ID and sends the role information back to the client.

This method called from the routes file with making a new route for it as follows:

```
// roles_routes.ts
```

```
app.route(this.baseEndPoint + '/:id')
    .get(controller.getOneHandler);
```

Here, `/:id` will be a request parameter meant to capture the ID of the role that the user wants to retrieve.

REST API GetOne Role

Request

URL: `http://127.0.0.1:3000/api/roles/5f11a06b-e9a7-438d-9f49-757e8239e238`
Method: GET

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "role_id": "5f11a06b-e9a7-438d-9f49-757e8239e238",
    "name": "visitor",
    "description": null,
    "rights": null,
    "created_at": "2023-08-15T13:04:50.314Z",
    "updated_at": "2023-08-15T13:04:50.314Z"
  }
}
```

Providing a valid role ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was not found.

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

Update Role

Updating a role involves modifying the existing data of a specific role stored in the database. This process allows you to adjust the attributes of a role,

such as its name, description, and associated rights. By performing an update, you can ensure that the role's information remains accurate and up-to-date. This operation is particularly useful when there are changes in a role's permissions, and you need to reflect those changes in the database.

To implement the Update Role API, make the following changes in the **updateHandler** code in the role controller:

```
// roles_controller.ts
public async updateHandler(req: Request, res: Response) : Promise<void> {
  const role = req.body;
  const service = new RolesService();
  const result = await service.update(req.params.id, role);
  res.status(result.statusCode).json(result);
}
```

The **updateHandler** function is intended to handle the updating of a role in the database.

It operates by retrieving data from the incoming HTTP request body, which is then utilized to update the corresponding role data in the database based on the role's unique identifier (**role_id**) as request parameter ID. The function subsequently generates a response indicating whether the update operation was successful or unsuccessful, providing details about the updated data or an appropriate error message if needed.

This method called from the routes file with making a new route for it as follows:

```
// roles_routes.ts
app.route(this.baseEndPoint + '/:id')
  .get(controller.getOneHandler)
  .put(validate(validRoleInput), controller.updateHandler);
```

Here, **/:id** will be a request parameter meant to capture the ID of the role that the user wants to retrieve, and it also validates data before updating in the database.

REST API Update Role

Request

URL: <http://127.0.0.1:3000/api/roles/5f11a06b-e9a7-438d-9f49-757e8239e238>

```
Method: PUT
body :
{
  "name": "visitor",
  "Description": "Allow read projects",
  "rights": "get_all_projects, get_details_project"
}
```

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "role_id": "5f11a06b-e9a7-438d-9f49-757e8239e238",
    "name": "visitor",
    "description": null,
    "created_at": "2023-08-15T13:04:50.314Z",
    "updated_at": "2023-08-18T07:35:06.238Z",
    "rights": "get_all_projects, get_details_project"
  }
}
```

Providing a valid role ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was not found.

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

Delete Role

The **delete** functionality for roles in a REST API involves the removal of a specific role from the database. This process is managed through an endpoint dedicated to role deletion. When a request is made to this endpoint, it triggers a function that handles the deletion process. The incoming request typically contains the unique identifier (**role_id**) of the role that needs to be deleted.

To implement **Delete** Role API, make the following changes in the **deleteHandler** code in the role controller:

```
// roles_controller.ts
public async deleteHandler(req: Request, res: Response) : Promise<void> {
  const service = new RolesService();
  const result = await service.delete(req.params.id);
  res.status(result.statusCode).json(result);
}
```

The **deleteHandler** processes the request by utilizing a service that interacts with the database. This service is responsible for executing the deletion operation. If the requested role exists in the database and the deletion is successful, the function responds with a success message and an appropriate status code, such as **200 OK**. If the role does not exist, the function returns an error response with a status code of 404 Not Found, indicating that the role was not located in the database.

This method called from the routes file with making a new route for it as follows:

```
// roles_routes.ts
app.route(this.baseEndPoint + '/:id')
  .get(controller.getOneHandler)
  .put(validate(validRoleInput), controller.updateHandler)
  .delete(controller.deleteHandler);
```

Here, **/:id** will be a request parameter meant to capture the ID of the role that the user wants to delete.

REST API Delete Role

Request

URL: <http://127.0.0.1:3000/api/roles/5f11a06b-e9a7-438d-9f49-757e8239e238>

Method: DELETE

Response

```
{
  "statusCode": 200,
  "status": "success"
}
```

In case of already deleted or not exist in database:

```
{  
  "statusCode": 404,  
  "status": "error",  
  "message": "Not Found"  
}
```

Add Default Role from System

After implementing APIs related to roles, let's add a feature to create a default role for **SuperAdmin** who has all rights in the system during project initialization. This additional functionality can be utilized later for authorization purposes.

Create one file in utils directory with name **ddl_util.ts** with the following code:

https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/ddl_util.ts

This DDLUtil class contains a static method **addDefaultRole()** responsible for adding a default role to the system.

Now do following changes in main.ts file:

<https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/main.ts>

As per code, it checks if command-line arguments are provided and if the first argument is **--init**. If these conditions are met, an asynchronous function is immediately invoked using an **async** function. It awaits the execution of **DDLUtil.addDefaultRole()**, which presumably initializes a default role in the system. Once **addDefaultRole()** completes, **process.exit()** is called to terminate the Node.js process. In summary, this code block is responsible for initializing a default role in the system when the program is run with the **--init** flag, and then exiting the process afterward.

```

• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter5/pms-be$ node dist/src/main.js --init
Master process PID: 69660
Connected to the database
Connected to the database
Add Default Role Result {
  statusCode: 201,
  status: 'success',
  data: Roles {
    role_id: 'e1cf6aae-a52a-4697-8607-c263e6372b66',
    name: 'SuperAdmin',
    description: 'Admin with having all permission',
    rights: 'add_role,edit_role,get_all_roles,get_details_role,delete_role,add_user,edit_user,get_all_users,ge
t_details_user,delete_user,add_project,edit_project,get_all_projects,get_details_project,delete_project,add_ta
sk,edit_task,get_all_tasks,get_details_task,delete_task,add_comment,edit_comment,get_all_comments,get_details_
comment,delete_comment',
    created_at: 2024-03-16T08:39:33.619Z,
    updated_at: 2024-03-16T08:39:33.619Z
  }
}
• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter5/pms-be$ █

```

Figure 5.2: Invoke Add Default Role Script

Output:



The screenshot shows a PostgreSQL query tool interface. The 'Query' tab is active, displaying the SQL command:

```

1 SELECT * FROM public.roles
2 ORDER BY role_id ASC

```

The 'Data Output' tab is active, showing the results of the query:

role_id	name	description	rights	created_at	updated_at
e1cf6aae-a52a-4697-8607-c263e6372b66	SuperAdmin	Admin with having all permission	add_role,edit_role,get_all_roles,get_details_role,delete_role,add_user,edit_user,get_all_users,ge t_details_user,delete_user,add_project,edit_project,get_all_projects,get_details_project,delete_project,add_ta sk,edit_task,get_all_tasks,get_details_task,delete_task,add_comment,edit_comment,get_all_comments,get_details_ comment,delete_comment	2024-03-16 14:09:33.619	2024-03-16 14:09:33.619

Figure 5.3: Postgres Added Role Output

In summary, role management plays a critical role in ensuring the security and controlled access of users within a system. It allows organizations to define and assign specific permissions and rights to different user roles, ensuring that each user can only perform actions that are relevant to their role and responsibilities.

User Management

User management refers to the process of handling user-related functionalities in an application. This includes creating, updating, retrieving, and deleting user accounts, as well as managing user roles, permissions, and authentication. User management is a crucial aspect of many applications, especially those that require user registration, authentication, and authorization.

User Service

Let's create the first user service to perform user table operations. Create **users_service.ts** using the following code:

```
// users_service.ts
import { Repository } from 'typeorm';
import { BaseService } from '../../utils/base_service';
import { DatabaseUtil } from '../../utils/db';
import { Users } from './users_entity';

export class UsersService extends BaseService<Users> {
  constructor() {
    let userRepository: Repository<Users> | null = null;
    userRepository = new DatabaseUtil().getRepository(Users);
    super(userRepository);
  }
}
```

The **UsersService** class is designed to extend the functionality provided by the **BaseService** class. It uses the **DatabaseUtil** class to get the repository for the User entity and then passes that repository to the constructor of the **BaseService** class. This allows the **UsersService** class to inherit and use the CRUD methods defined in the **BaseService** class for working with the Users entity.

Input Validation

User input validation is a crucial aspect of developing web applications to ensure data integrity, security, and a smooth user experience. It involves checking and sanitizing the data submitted by users through various input fields, such as **email**, **username**, **password**, **role**, and so on.

Now add **validUserInput** in the **users_routes.ts** file using the following code:

```
// user_routes.ts
import { body } from 'express-validator';
import { validate } from '../../utils/validator';
const validUserInput = [
```

```

body('username').trim().notEmpty().withMessage('It should be
required'),
body('email').isEmail().withMessage('It should be valid
emailId'),
body('password')
  .isLength({ min: 6, max: 12 }).withMessage('It must be
  between 6 and 12 characters in length')
  .isStrongPassword({ minLowercase: 1, minUppercase: 1,
  minSymbols: 1, minNumbers: 1 })
  .withMessage('It should include at least one uppercase
  letter, one lowercase letter, one special symbol, and one
  numerical digit.'),
body('role_ids').isArray().withMessage('It must be an array of
  uuids of roles')
  .custom((value: string[]) => {
    if (value?.length > 0 && value instanceof Array) {
      const uuidPattern = /^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-
      fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$/;
      const isValid = value?.every(uuid =>
        uuidPattern.test(uuid.trim()));
      if (!isValid) {
        throw new Error('It has invalid uuids for role');
      }
    }
    return true; // Validation passed
  })
];

```

- **validUserInput** array: This array contains a series of validation checks for each input field expected in a user. Each element of this array is a validator function that checks a specific aspect of the data.
- **body('username').trim().notEmpty().withMessage('It should be required')**: This validator is applied to the username field in the request body. It removes any leading and trailing whitespace from the input and checks that the input is not empty. If the validation fails, this message will be included in the error response.

- `body('email').isEmail().withMessage('It should be valid emailId')`: This validator is applied to the email field in the request body. It checks if a valid email ID is provided or not. If the validation fails, it displays a message as given in `withMessage`.
- `body('password').isLength({ min: 6, max: 12 }).withMessage('It must be between 6 and 12 characters in length').isStrongPassword({ minLowercase: 1, minUppercase: 1, minSymbols: 1, minNumbers: 1 }).withMessage('It should include at least one uppercase letter, one lowercase letter, one special symbol, and one numerical digit.')`: This validator is applied to the password field. It checks that the password should be between 6 and 12 characters long and have strong passwords as specified in the given option. If the validation fails, it gives a defined error message.
- `body('role_ids')...` : It validates the given `role_ids` array of UUIDs or not. If any of them are not matched in the db, it gives an error.

User Onboarding

In PMS, a user onboarding facility is provided by a super admin or someone who has permission to add users to the system. First, we create one master or super admin user who has all rights, meaning we assign a role that has each and every right. We have already created a super admin role, so we will assign that `role_id` to the user and create a super admin user.

We will store the user's password in the database in an encrypted form instead of plain text for security reasons. So, we will install an npm package and encryption function, and then compare encrypted password validation during login.

To begin, open the terminal with the root directory of the project and paste the following command:

```
npm install bcrypt --save
```

Next, open the `common.ts` file and add the following functions:

```
// common.ts
/**
 * Encrypts a string using bcrypt hashing.
 */
```

```

    * @param {string} s - The string to be encrypted.
    * @returns {Promise<string>} - The encrypted string.
    */
export const encryptString = async (s: string) => {
  const encryptedString = await bcrypt.hash(s, 8);
  return encryptedString;
};

/**
 * Compares a plain string with a bcrypt hash to determine if
they match.
*
* @param {string} s - The plain string to be compared.
* @param {string} hash - The bcrypt hash to compare against.
* @returns {Promise<boolean>} - A promise that resolves to
true if the comparison is successful, otherwise false.
*/
export const bcryptCompare = async (s, hash) => {
  return await bcrypt.compare(s, hash);
};

```

The **encryptString** function takes a string as input and uses the **bcrypt.hash** function to perform a one-way hashing with a cost factor of 8. The result is an encrypted string that can be stored securely. The **async** keyword indicates that the function is asynchronous, meaning it returns a promise that resolves to the encrypted string once the hashing is complete.

The **bcryptCompare** function takes a plain string and a **bcrypt** hash as input. It uses the **bcrypt.compare** function to compare the plain string with the provided hash. The function returns a promise that resolves to true if the comparison is successful (that is, the plain string matches the hash), and false otherwise. This comparison is used for verifying passwords during authentication processes. The **async** keyword indicates that the function is asynchronous, meaning it returns a promise that holds the comparison result.

In **validUserInput**, we have seen that we just check if the **role_ids** field has a value in array form. However, it is necessary to check whether these role IDs exist in the db or not before user insertion in the db. So, we will create the **checkValidRoleIds** function in **RolesUtil** class.

```
public static async checkValidRoleIds(role_ids: string[]) {
```

```

const roleService = new RolesService();

// Query the database to check if all role_ids are valid
const roles = await roleService.findByIds(role_ids);

// Check if all role_ids are found in the database
return roles.data.length === role_ids.length;
}

```

It queries the database using the **findByIds** method from the **roleService** instance to retrieve roles based on the provided role IDs, and then it checks whether the number of roles retrieved from the database matches the number of input role IDs. If they match, it indicates that all the provided role IDs are valid; otherwise, it implies that it is not valid **role_ids**.

Let's update the **users_controller.ts** file using the following code:

https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/components/users/users_controller.ts

This function appears to handle the addition of a new user by processing the user data, encrypting the password, and checking the validity of role IDs before creating the user using a service class. It also handles errors gracefully by providing appropriate error responses.

Now call **addHandler** in user routes with change in the **users_router.ts** file as follows:

```

// users_routes.ts
export class UserRoutes {

  private baseEndPoint = '/api/users';

  constructor(app: Express) {

    const controller = new UserController();

    app.route(this.baseEndPoint)
      .post(validate(validUserInput), controller.addHandler);
  }
}

```

This way we created an add user router with input validation for user onboarding through middleware of validation. If validation happens successfully, then the user is inserted in the database.

REST API Add User

Request

```
URL: http://127.0.0.1:  
Method: POST  
body :  
{  
  "fullname": "Super Admin",  
  "username": "pms-admin",  
  "email": "admin@pms.com",  
  "password": "Admin@pms1",  
  "role_ids": ["b88cc70d-ab0a-4464-9562-f6320df519f6"]  
}
```

Response

```
{  
  "statusCode": 201,  
  "status": "success",  
  "data": {  
    "user_id": "f249e681-57d8-4f91-addd-a8c615b43a37",  
    "fullname": "Super Admin",  
    "username": "pms-admin",  
    "email": "admin@pms.com",  
    "password":  
      "$2b$08$pSsEBELbLrXjDfqBJY/7EuyygVuqDzLyBA0J08pPnWDYJYp5.015G",  
    "role_ids": [  
      "b88cc70d-ab0a-4464-9562-f6320df519f6"  
    ],  
    "created_at": "2023-08-19T18:12:55.315Z",  
    "updated_at": "2023-08-19T18:12:55.315Z"  
  }  
}
```

In case, if the **role_id** has the wrong UUID or does not exist in the role table, then it gives an error with status **400** as follows:

```
{  
  "statusCode": 400,  
  "status": "error",
```

```
        "message": "Invalid role_ids"
    }
```

Here, we onboarded super admin. This admin similarly adds other users and shares their credentials manually with them. Afterward, users can change their own password, which we will cover later on.

Add Default User from System

After establishing the default role, let's proceed to create a default user with super admin privileges, granting access to all APIs associated with the default role.

Create one **addDefaultUser** method in **dd1_util.ts** file with the following code:

```
public static async addDefaultUser(): Promise<boolean> {
    try {
        await DatabaseUtil.getInstance();
        const service = new UsersService();

        const user: Users = {
            user_id: v4(),
            fullname: 'Super Admin',
            username: 'superadmin',
            email: config.default_user.email,
            password: await
                encryptString(config.default_user.password),
            role_id: this.superAdminRoleId,
            created_at: new Date(),
            updated_at: new Date()
        };
        const result = await service.create(user);
        console.log('Add Default User Result', result);
        if (result.statusCode === 201) {
            return true;
        }
        return false;
    } catch (error) {
        console.error(`Error while addDefaultRole() =>

```

```

        ${error.message}`);
        return false;
    }
}

```

Here, we added the default user's email and password in **server_config.json** file and used it from there, so you can add it in a similar manner as per your convenience.

This function is invoked from the **main.ts** file, after creating the default role, as follows:

```

if (args.length > 0 && args[0] === '--init') {
  (async () => {
    await DatabaseUtil.getInstance();
    await DDLUtil.addDefaultRole();
    await DDLUtil.addDefaultUser();
    process.exit();
  })();
}

```

Now run the script and see the following output:

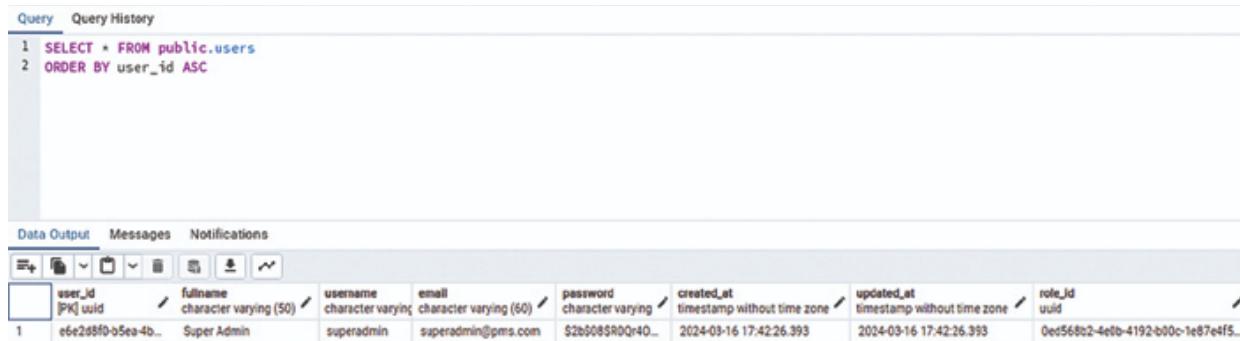
```

• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter5/pms-be$ node dist/src/main.js --init
Master process PID: 123518
Connected to the database
Connected to the database
Add Default Role Result {
  statusCode: 409,
  status: 'error',
  message: 'Key (name)=(SuperAdmin) already exists.'
}
Add Default User Result {
  statusCode: 201,
  status: 'success',
  data: Users {
    user_id: 'e6e2d8f0-b5ea-4b2b-b116-c3ffc6a007a5',
    fullname: 'Super Admin',
    username: 'superadmin',
    email: 'superadmin@pms.com',
    password: '$2b$08$R00r40e.KDk4njkeF.MGQ0gxcM7G7sXiDhnccKycImA65unS3HIRa',
    role_id: '0ed568b2-4e0b-4192-b00c-1e87e4f54cbb',
    created_at: 2024-03-16T12:12:26.393Z,
    updated_at: 2024-03-16T12:12:26.393Z
  }
}
• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter5/pms-be$ []

```

Figure 5.4: Add Default User from Script

Output:



The screenshot shows a PostgreSQL database interface. The top bar has tabs for 'Query' (which is selected) and 'Query History'. Below the bar is a code editor with the following SQL query:

```

1 SELECT * FROM public.users
2 ORDER BY user_id ASC

```

Below the code editor is a 'Data Output' tab, which is selected. The data is presented in a table with the following columns: user_id, fullname, username, email, password, created_at, updated_at, and role_id. There is one row of data:

user_id	fullname	username	email	password	created_at	updated_at	role_id
e6e2d8f0-b5ea-4b...	Super Admin	superadmin	superadmin@pms.com	\$2b\$08\$R0Q40...	2024-03-16 17:42:26.393	2024-03-16 17:42:26.393	0ed568b2-4e0b-4192-b00c-1e87e4f5...

Figure 5.5: Postgres Database Added User Output

User Sign-In

Once the user registration process is completed, users should have the capability to log into the application. As a component of the user sign-in procedure, an API will be established to enable users to request the server using their email and password. The server's role is to verify whether the user exists and if the provided password is valid. If the validation fails, an error response is generated. However, if the validation is successful, the server responds with both an access token and a refresh token. This entire process is known as **authentication**.

Upon successful login, each subsequent API request must incorporate the access token in the request header. The server's task at this point is to verify the validity of the token. If the token is invalid, an error response is generated, preventing further actions. Conversely, if the token is valid, the server grants permission for the requested actions to be executed. This aspect of validating the access token and permitting or denying actions is referred to as **authorization**.

To create **AccessToken**, we will use the **jsonwebtoken**, also known as JWT token. First, you will need to install the npm package. For more information, please visit <https://www.npmjs.com/package/jsonwebtoken>

Open the terminal with the root directory and paste the following command:

```

npm install jsonwebtoken -- save
npm install @types/jsonwebtoken -D

```

Here is a general overview of how JWT works:

- **Token Structure:** A JWT consists of three parts: header, payload, and signature. These parts are base64-encoded and combined with periods.

- **Header:** Contains info about the signing algorithm and token type.
- **Payload:** Contains claims (user info, expiration time, and so on).
- **Signature:** Used for verification and to ensure data integrity.
- **Token Creation:** When a user logs in, the server constructs the header and payload, including user details and other info. The server signs the JWT using a secret key or private key.
- **Token Issuance:** The server sends the JWT to the client after successful login. The client stores it, often in cookies or local storage.
- **Token Usage:** In subsequent requests, the client sends the JWT in the Authorization header or as a parameter. This helps the server verify the request's authenticity and identify the user by decoding the payload.
- **Token Verification:** The server verifies the JWT by recalculating the signature using the same key. If the recalculated signature matches the JWT's signature, the token is valid.
- **Token Expiration:** JWTs usually have an expiration time (exp claim) to prevent indefinite validity. Servers can reject expired tokens by checking the expiration time in the payload.
- **Token Revocation (Optional):** JWTs are stateless, meaning the server doesn't keep track of them after issuing. If you need to revoke a JWT before it expires, extra measures like maintaining a list of revoked tokens are required.

Remember, the security of JWTs relies on protecting the secret key (or private key) and ensuring proper token verification on the server side.

Let's define constants for JWT secret and expiration times in the **common.ts** file as follows:

```
// common.ts
export const SERVER_CONST = {
  JWTSECRET: 'SecretKeyOfPMs-SECRET',
  ACCESS_TOKEN_EXPIRY_TIME_SECONDS: 1 * 8 * 60 * 60, // 8 hours
  REFRESH_TOKEN_EXPIRY_TIME_SECONDS: 5 * 7 * 24 * 60 * 60, // one week
};
```

The mentioned parameters are flexible and adaptable according to your needs.

Authentication

In the standard login process, two tokens are usually created: an access token and a refresh token, each having distinct expiration times. The access token has a shorter lifespan, while the refresh token has a longer one. If the access token expires, the refresh token can be used to generate a new access token, facilitating seamless reauthentication without requiring manual login. This approach ensures continuous access without interruption.

This process is called **Authentication** in terms of generation of tokens.

Let's create login function in the **users_controller.ts** file as follows:

```
// user_controller.ts
import * as jwt from 'jsonwebtoken';

/**
 * Handles user login by checking credentials, generating
 * tokens, and responding with tokens.
 *
 * @param {Request} req - The request object.
 * @param {Response} res - The response object.
 */
public async login(req: Request, res: Response): Promise<void>
{
  const { email, password } = req.body;
  const service = new UsersService();

  // Find user by email
  const result = await service.findAll({ email: email });
  if (result.data.length < 1) {
    res.status(404).json({ statusCode: 404, status: 'error',
    message: 'Email
    not found' });
    return;
  } else {
    const user = result.data[0];
```

```

// Compare provided password with stored hashed password
const comparePasswords = await bcryptCompare(password,
user.password);
if (!comparePasswords) {
  res.status(400).json({ statusCode: 400, status: 'error',
  message: 'Password is not valid' });
  return;
}

// Generate access and refresh tokens
const accessToken: string = jwt.sign({
  email: user.email,
  username: user.username
}, SERVER_CONST.JWTSECRET, { expiresIn:
SERVER_CONST.ACCESS_TOKEN_EXPIRY_TIME_SECONDS });

const refreshToken: string = jwt.sign({
  email: user.email,
  username: user.username
}, SERVER_CONST.JWTSECRET, { expiresIn:
SERVER_CONST.REFRESH_TOKEN_EXPIRY_TIME_SECONDS });

// Respond with tokens
res.status(200).json({ statusCode: 200, status: 'success',
data: {
  accessToken, refreshToken } });
return;
}
}

```

In this code, the login function handles user login functionality using the following steps:

- It receives the email and password from the request body.
- It creates an instance of the **UsersService** class to interact with user data.
- It queries the database to find a user based on the provided email.
- If no user is found with the provided email, it sends a 404 error response.

- If a user is found, it compares the provided password with the hashed password stored in the user's data. If the passwords don't match, it sends a 400 error response.
- If the passwords match, it generates an access token and a refresh token using the `jwt.sign` function. The access token contains user information and has a short expiration time, while the refresh token has a longer expiration time.
- It sends a success response (status **200**) containing the generated access and refresh tokens.

Now add another function `getAccessTokenFromRefreshToken` in the same file using the following code:

```
/**
 * Generates a new access token using a valid refresh token.
 *
 * @param {Request} req - The request object.
 * @param {Response} res - The response object.
 */
public async getAccessTokenFromRefreshToken(req: Request, res: Response): Promise<void> {
  // Get the refresh token from the request body
  const refreshToken = req.body.refreshToken;

  // Verify the refresh token
  jwt.verify(refreshToken, SERVER_CONST.JWTSECRET, (err, user) => {
    if (err) {
      // If refresh token is invalid, send a 403 error response
      res.status(403).json({ statusCode: 403, status: 'error', message: 'Invalid Refresh Token' });
      return;
    }
    // Generate a new access token using user information from
    // the refresh token
    const accessToken = jwt.sign(user, SERVER_CONST.JWTSECRET, { expiresIn: SERVER_CONST.ACCESS_TOKEN_EXPIRY_TIME_SECONDS });
  });
}
```

```

        // Respond with the new access token
        res.status(200).json({ statusCode: 200, status: 'success',
        data: { accessToken } });
        return;
    });
}

```

The **getAccessTokenFromRefreshToken** function handles the generation of a new access token using a valid refresh token. It retrieves the refresh token from the request body. It uses the **jwt.verify** function to verify the refresh token. If the refresh token is invalid or has expired, an error will be caught in the **err** parameter. If the refresh token is invalid, it sends a 403 error response indicating an invalid refresh token. If the refresh token is valid, it uses the user information decoded from the refresh token to generate a new access token using the same **jwt.sign** function. The new access token is signed with the same secret key and has a shorter expiration time. It sends a success response (status **200**) containing the newly generated access token.

These two functions are called from routes for HTTP requests, so let's add two routes in the **users_router.ts** file as follows:

```

// users_router.ts
app.route('/api/login')
    .post(controller.login);

app.route('/api/refresh_token')
    .post(controller.getAccessTokenFromRefreshToken);

```

Sign In API

REST API Login

Request

URL: <http://127.0.0.1:3001/api/login>
 Method: POST
 body :
 {
 "email": "admin@pms.com",
 "password": "Admin@pms1"
 }

Response

{

```

"statusCode": 200,
"status": "success",
"data": {
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InlhbwIwYW5
  jaGFsMTk5M0BnbWFpbC5jb20iLCJ1c2VybmcFtZSI6InlhbwIuaSIsInR5cCI6
  IkJlYXJlcIIsImlhhdCI6MTY5MjY0MjgwNywiZXhwIjoxNjkyNjcxNjA3fQ.Lz
  Yu6ZzT501MvRbuiZGNCv-kMD9UdWMG_iNYCuI3ta4",
  "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InlhbwIwYW5
  jaGFsMTk5M0BnbWFpbC5jb20iLCJ1c2VybmcFtZSI6InlhbwIuaSIsInR5cCI6
  I1JlZnJlc2giLCJpYXQiOjE2OTI2NDI4MDcsImV4cCI6MTY5NTY2NjgwN30.bbVs_7AUTpwqDx0hyx66A59uV-CirkPvEdJsDWD-OM"
}
}

```

In case of a wrong email ID passed in a request, the server will respond with the following error message:

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Email not found"
}
```

Similarly, if a wrong password is sent in the request, then it will result in a **400 error** code, as follows:

```
{
  "statusCode": 400,
  "status": "error",
  "message": "Password is not valid"
}
```

In this manner, users can effortlessly and securely log into the application.

Authorization

Authorization is the process of determining whether a user or entity has the right permissions to access certain resources or perform specific actions within an application. It is a critical component of security that ensures that

users can only access the data and functionality they are allowed to while protecting sensitive information and preventing unauthorized actions.

We are going to create a middleware function responsible for authorizing whether the JWT token is valid or not. If it is valid, then only pass it to the next function or the actual API; otherwise, we will restrict and give an error as 401 unauthorized.

Create **custom.d.ts** file in the **src** directory with the following code:

```
// custom.d.ts
declare namespace Express {
  interface Request {
    user?: {
      username?: string;
      email?: string;
      rights?: string[];
      user_id?: string;
    };
    // Add any other custom properties you need
  }
}
```

In this part of the code, a **TypeScript** namespace declaration is used to extend the Request interface provided by the **Express.js** framework. It adds a custom property called `user` to the Request object. This `user` property is an optional object that can contain properties such as `username`, `email`, and `rights`. The comment suggests that you can add any other custom properties you might need here.

Now, let's make **UsersUtils** in the **users_controller.ts** file with the following code:

```
// users_controller.ts
export class UsersUtil {
  public static async getUserFromUsername(username: string) {
    try {
      if (username) {
        const service = new UsersService();
        const users = await service.customQuery(`username =
          '${username}'`);
        if (users && users.length > 0) {
```

```

        return users[0];
    }
}
} catch (error) {
    console.error(`Error while getUserFromToken() =>
    ${error.message}`);
}
return null;
}
}

```

The `getUserFromUsername` function accepts a `username` as its input parameter and retrieves the corresponding user from the database. If the provided `username` does not have a match in the database, the function returns a null response.

After that, add one method in `RolesUtil` to get rights from roles:

```

public static async getAllRightsFromRoles(role_ids: string[]): Promise<string[]> {

    // Create an instance of RolesService to interact with the
    // roles
    const roleService = new RolesService();

    // Initialize an array to store the collected rights
    let rights: string[] = [];

    // Query the database to validate the provided role_ids
    const queryData = await roleService.findIds(role_ids);
    const roles: Roles[] = queryData.data ? queryData.data : [];

    // Extract rights from each role and add them to the rights
    // array
    roles.forEach((role) => {
        const rightFromRole: string[] = role.rights.split(',');
        rights = [...new Set(rights.concat(rightFromRole))];
    });

    // Return the accumulated rights
    return rights;
}

```

The `getAllRightsFromRoles` is designed to retrieve and consolidate rights associated with a collection of role IDs. The function queries the database using the provided role IDs to fetch corresponding role data. For each retrieved role, the associated rights are extracted by splitting the rights string. These rights are then added to the rights array while avoiding duplicates using a set-based approach. Finally, the function returns an array containing the accumulated and unique rights gathered from all the roles.

Let's create `auth_util.ts` in the utils directory using the following code:

https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/auth_util.ts

The `authorize` middleware function is responsible for verifying the authorization token, decoding it, and attaching user-related information to the `req.user` object. It first checks if the authorization token exists in the request headers. If not, it returns a 401 Unauthorized response indicating a missing token. The token is split to extract the actual token value after the “Bearer” prefix. The token is then verified using the provided JWT secret. If the token is successfully decoded, it extracts the username and email from the decoded token and assigns them to `req.user.username` and `req.user.email`, respectively. If a valid username exists, it fetches the user's information and assigned it to `req.user`. Finally, if everything is successful, the middleware proceeds to the next middleware using the `next()` function.

The `hasPermission` function takes an array of user rights and a desired right as parameters. It checks if the array of user rights includes the desired right. If the desired right is found in the user rights, the function returns true; otherwise, it returns false.

We created raw functions of authorization, and now change call the `authorize` function in the routes file for each API, as follows:

```
// users_routes.ts
export class UserRoutes {

  private baseEndPoint = '/api/users';

  constructor(app: Express) {

    const controller = new UserController();
    app.route(this.baseEndPoint)
```

```

    .all(authorize) // Apply authorization middleware to all
    routes under this endpoint
    .get(controller.getAllHandler)
    .post(validate(validUserInput), controller.addHandler);

  app.route(this.baseEndPoint + '/:id')
    .all(authorize) // Apply authorization middleware to all
    routes under this endpoint
    .get(controller.getOneHandler)
    .put(controller.updateHandler)
    .delete(controller.deleteHandler);

  app.route('/api/login')
    .post(controller.login);

  app.route('/api/refresh_token')
    .post(controller.getAccessTokenFromRefreshToken);
}
}

```

Authorization is not required for the login and refresh token API since it falls under the category of authentication APIs:

```

// roles_routes.ts
export class RoleRoutes {

  private baseEndPoint = '/api/roles';

  constructor(app: Express) {

    const controller = new RoleController();
    app.route(this.baseEndPoint)
      .all(authorize)
      .post(validate(validRoleInput), controller.addHandler)
      .get(controller.getAllHandler);

    app.route(this.baseEndPoint + '/:id')
      .all(authorize)
      .get(controller.getOneHandler)
      .put(validate(validRoleInput), controller.updateHandler)
      .delete(controller.deleteHandler);
  }
}

```

```
}
```

From now onwards, make sure, except for the login and refresh token APIs, you have to pass the access token with bearer type in the header as Authorization in the request; otherwise, it gives an error:

```
{
  "statusCode": 401,
  "status": "error",
  "message": "Missing Authorization Token"
}
```

Checking for valid rights of permission is also a part of Authorization. We have already created a **hasPermission** method that will be the first call in each API controller with respective rights.

In the User Controller, add the following checks with their respective methods:

```
// addHandler
  public async addHandler(req: Request, res: Response): Promise<void> {
    if (!hasPermission(req.user.rights, 'add_user')) {
      res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
      return;
    }
    ...
  }

// getAllHandler
  public async getAllHandler(req: Request, res: Response): Promise<void> {
    if (!hasPermission(req.user.rights, 'get_all_users')) {
      res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
      return;
    }
    ...
  }

// getOneHandler
```

```

public async getOneHandler(req: Request, res: Response):
Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  ...
}

// updateHandler
public async updateHandler(req: Request, res: Response):
Promise<void> {
  if (!hasPermission(req.user.rights, 'edit_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  ...
}

// deleteHandler
public async deleteHandler(req: Request, res: Response):
Promise<void> {
  if (!hasPermission(req.user.rights, 'delete_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  ...
}

```

Similarly, add in the Role controller to check permission for roles API:

```

// addHandler
public async addHandler(req: Request, res: Response):
Promise<void> {
  if (!hasPermission(req.user.rights, 'add_role')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
  }
}

```

```
    return;
}
...
}
// getAllHandler
public async getAllHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_all_roles')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
...
}
// getOneHandler
public async getOneHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_role')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
...
}
// updateHandler
public async updateHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'edit_role')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
...
}
// deleteHandler
```

```

public async deleteHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'delete_role')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  ...
}

```

This is how middleware functions for authorization and permissions work in an **Express.js** application. It extends the Request interface to include a user object with properties such as username, email, and rights. The **authorize** middleware verifies JSON Web Tokens (JWTs) from the request header, extracts user information, and checks permissions. The **hasPermission** function checks if a user's rights include the desired permission. Moving forward, except for the login and refresh token APIs, the access token should be passed in the header as a bearer type using the **Authorization** field.

GetAll Users

After successfully onboarding users, we can proceed to retrieve the newly inserted users from the database. So, update the **getAllHandler** method in the **users_controller.ts** file using the following code:

```

public async getAllHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_all_users')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  const service = new UsersService();
  const result = await service.findAll(req.query);
  if (result.statusCode === 200) {
    // Remove password field to send in response
    result.data.forEach(i => delete i.password);
  }
  res.status(result.statusCode).json(result);
}

```

```
    return;
}
```

The `getAllHandler` method uses the `UserService` class to retrieve all users from the database based on the query parameters in the request. The resulting data is then sent back to the client with an appropriate HTTP status code and formatted as JSON.

This controller method call in routes with a change in `roles_routes.ts` as follows:

```
app.route(this.baseEndPoint)
  .all(authorize) // Apply authorization middleware to all
  routes under this endpoint
  .get(controller.getAllHandler)
  .post(validate(validUserInput), controller.addHandler);
```

By employing this approach, we establish a GET route that fetches all roles stored in the database, effectively functioning as a REST API endpoint for retrieving role data.

REST API GetAll Roles

Request

URL: `http://127.0.0.1:3000/api/users`
Method: GET
Query Params: {}

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": [
    {
      "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd",
      "fullname": "Super Admin",
      "username": "pms-admin",
      "email": "admin@pms.com",
      "role_ids": [
        "dbda47e4-f843-4263-a4d6-69ef80156f81"
      ],
      "created_at": "2023-08-22T17:08:24.722Z",
    }
  ]
}
```

```

    "updated_at": "2023-08-22T17:08:24.722Z"
  },
  {
    "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
    "fullname": "Yami Panchal",
    "username": "yamini",
    "email": "yami@gmail.com",
    "role_ids": [
      "b88cc70d-ab0a-4464-9562-f6320df519f6"
    ],
    "created_at": "2023-08-18T17:57:38.744Z",
    "updated_at": "2023-08-24T16:57:19.110Z"
  }
]
}

```

If you want to filter or search by exact name, you can change query params as follows:

URL: <http://127.0.0.1:3000/api/users?username=pms-admin>

It gives only matched data as a response, as follows

```

{
  "statusCode": 200,
  "status": "success",
  "data": [
    {
      "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd",
      "fullname": "Super Admin",
      "username": "pms-admin",
      "email": "admin@pms.com",
      "role_ids": [
        "dbda47e4-f843-4263-a4d6-69ef80156f81"
      ],
      "created_at": "2023-08-22T17:08:24.722Z",
      "updated_at": "2023-08-22T17:08:24.722Z"
    }
  ]
}

```

GetOne User

This endpoint offers a valuable function by providing specific user information. It serves as a crucial tool for users to access their own details and also enables administrators to retrieve specific user information whenever necessary.

To implement the **GetOne** User API, make the following changes in the **getOneHandler** code in the user controller:

```
// users_controller.ts
public async getOneHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  const service = new UsersService();
  const result = await service.findOne(req.params.id);
  if (result.statusCode === 200) {
    delete result.data.password;
  }
  res.status(result.statusCode).json(result);
  return;
}
```

The **getOneHandler** function serves as the bridge between the incoming client request, the service layer that interacts with the database, and the outgoing HTTP response. It retrieves a single user's details from the database based on the provided user ID and sends the user information back to the client, except password data for security reasons.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize) // Apply authorization middleware to all
  routes under this endpoint
  .get(controller.getOneHandler)
```

Here, `/:id` will be a request parameter meant to capture the ID of the user that wants to retrieve.

REST API GetOne Role

Request

URL: `http://127.0.0.1:5000/api/users/b930d02c-43af-4875-b7e9-546c9f4c23dd`
Method: GET

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd",
    "fullname": "Super Admin",
    "username": "pms-admin",
    "email": "admin@pms.com",
    "role_ids": [
      "dbda47e4-f843-4263-a4d6-69ef80156f81"
    ],
    "created_at": "2023-08-22T17:08:24.722Z",
    "updated_at": "2023-08-22T17:08:24.722Z"
  }
}
```

Providing a valid role ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was **Not found**.

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

Update User

In updating user information, it's important to note that the `username` and `email` fields won't be modified. These fields are unique identifiers and will

retain their initial values as set during creation. Additionally, to change the password, a separate API will be implemented specifically for that purpose, known as the **change password API**.

Let's develop **updateHandler** in the **users_controller.ts** using the following code:

```
public async updateHandler(req: Request, res: Response): Promise<void> {

  if (!hasPermission(req.user.rights, 'edit_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }

  const service = new UsersService();
  const user = req.body;

  // we will not update email and username once inserted so
  // remove it from body
  delete user?.email;
  delete user?.username;

  // we will also not update password from here it will be from
  // changePassword function separate
  delete user?.password;
  const result = await service.update(req.params.id, user);
  if (result.statusCode === 200) {
    delete result.data.password;
  }
  res.status(result.statusCode).json(result);
  return;
}
```

The **updateHandler** function is intended to handle the updating of a user in the database.

It operates by retrieving data from the incoming HTTP request body, which is then utilized to update the corresponding role data in the database based on the user's unique identifier (**user_id**) as request parameter ID. The

function subsequently generates a response indicating whether the update operation was successful or unsuccessful, providing details about the updated data or an appropriate error message if needed. We will never send a password data response, so delete that field from the response.

To update a user's time, we just need to validate that the **role_ids** are proper. So, perform the validation as follows:

```
const updateValidUserInput = [
  body('role_ids').isArray().withMessage('It must be an array of
  uuids of roles')
  .custom((value: string[]) => {
    if (value?.length > 0 && value instanceof Array) {
      const uuidPattern = /^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-
      fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$/;
      const isValid = value?.every(uuid =>
        uuidPattern.test(uuid.trim())));
      if (!isValid) {
        throw new Error('It has invalid uuids for role');
      }
    }
    return true; // Validation passed
  })
];

```

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize) // Apply authorization middleware to all
  routes under this endpoint
  .get(controller.getOneHandler)
  .put(validate(updateValidUserInput),
  controller.updateHandler);
```

REST API Update User

Request

URL: <http://127.0.0.1:3000/api/users/d166945a-f85d-485c-bdac-0c8056b3188a>
Method: PUT
body :

```
{  
  "fullname": "Yami Panchal",  
  "role_ids": ["b88cc70d-ab0a-4464-9562-f6320df519f6"]  
}
```

Response

```
{  
  "statusCode": 200,  
  "status": "success",  
  "data": {  
    "fullname": "Yami Panchal",  
    "username": "yamini",  
    "email": "yamipanchal1993@gmail.com",  
    "created_at": "2023-08-18T17:57:38.744Z",  
    "updated_at": "2023-08-24T16:57:19.110Z",  
    "role_ids": [  
      "b88cc70d-ab0a-4464-9562-f6320df519f6"  
    ],  
    "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a"  
  }  
}
```

Providing a valid role ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was **Not found**.

```
{  
  "statusCode": 404,  
  "status": "error",  
  "message": "Not Found"  
}
```

Delete User

The **delete** functionality for users in a REST API involves the removal of a specific user from the database. This process is managed through an endpoint dedicated to user deletion. When a request is made to this endpoint, it triggers a function that handles the deletion process. The incoming request typically contains the unique identifier (**user_id**) of the user that needs to be deleted.

To implement the **Delete User** API, make the following changes in the **deleteHandler** code in the user controller:

```
public async deleteHandler(req: Request, res: Response):  
Promise<void> {  
  if (!hasPermission(req.user.rights, 'delete_user')) {  
    res.status(403).json({ statusCode: 403, status: 'error',  
    message: 'Unauthorised' });  
    return;  
  }  
  const service = new UsersService();  
  const result = await service.delete(req.params.id);  
  res.status(result.statusCode).json(result);  
  return;  
}
```

The **deleteHandler** processes the request by utilizing a service that interacts with the database. This service is responsible for executing the deletion operation. If the requested user exists in the database and the deletion is successful, the function responds with a success message and an appropriate status code, such as **200 OK**. If the role does not exist, the function returns an error response with a status code of **404 Not Found**, indicating that the user was not located in the database.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')  
  .all(authorize)  
  .get(controller.getOneHandler)  
  .put(validate(updateValidUserInput),  
    controller.updateHandler)  
  .delete(controller.deleteHandler);
```

Here, **:id** will be a request parameter meant to capture the ID of the role that the user wants to delete.

REST API Delete Role

Request

URL: <http://127.0.0.1:3000/api/users/5f11a06b-e9a7-438d-9f49-757e8239e238>

Method: DELETE

Response

```
{  
  "statusCode": 200,  
  "status": "success"  
}
```

In case of already deleted or not exist in database:

```
{  
  "statusCode": 404,  
  "status": "error",  
  "message": "Not Found"  
}
```

Password Management

Password Management involves implementing strategies and practices to secure and effectively manage user passwords within digital systems. This encompasses various methods to ensure the confidentiality and integrity of user passwords, as well as enabling users to securely create, update, and recover their passwords. Effective password management is vital for safeguarding sensitive data, preventing unauthorized access, and maintaining the overall security posture of an application or system.

Change Own Password

During the user information update process, it's been observed that for security reasons, password modifications are not carried out within the same API. Therefore, a distinct API will be developed to address this security concern.

Now, create one method in the `users_controller.ts` file for `changePassword` as follows:

```
// users_controller.ts  
public async changePassword(req: Request, res: Response):  
Promise<void> {  
  const { oldPassword, newPassword } = req.body;  
  const service = new UsersService();
```

```

const findUserResult = await service.findOne(req.params.id);
if (findUserResult.statusCode !== 200) {
  res.status(404).send({ statusCode: 404, status: 'error',
  message: 'User Not Found' });
  return;
}
const user = findUserResult.data;

// check requested user_id and session user_id is same
if (user?.username !== req.user.username) {
  res.status(400).send({ statusCode: 400, status: 'error',
  message: 'User can change only own password' });
  return;
}

// verify old password is valid
const comparePasswords = await bcryptCompare(oldPassword,
user.password);
if (!comparePasswords) {
  res.status(400).json({ statusCode: 400, status: 'error',
  message: 'oldPassword is not matched' });
  return;
}

// Encrypt the user's new password
user.password = await encryptString(newPassword);
const result = await service.update(req.params.id, user);
if (result.statusCode === 200) {
  res.status(200).json({ statusCode: 200, status: 'success',
  message: 'Password is updated successfully' });
  return;
} else {
  res.status(result.statusCode).json(result);
  return;
}
}

```

The **changePassword** function ensures that the user requesting the password change is authorized to do so, verifies the old password, securely encrypts the new password, and updates the user's password in the database.

This method is called from the `users_routes.ts` file, so add the following code to the route file:

```
// users_routes.ts
const validChangePassword = [
  body('oldPassword').trim().notEmpty().withMessage('It should
  be
  required'),
  body('newPassword')
    .isLength({ min: 6, max: 12 }).withMessage('It must be
    between 6 and 12 characters in length')
    .isStrongPassword({ minLowercase: 1, minUppercase: 1,
    minSymbols: 1, minNumbers: 1 })
    .withMessage('It should include at least one uppercase
    letter, one lowercase letter, one special symbol, and one
    numerical digit.'), body('role_ids'),
];
app.route(this.baseEndPoint + '/changePassword/:id')
  .all(authorize)
  .post(validate(validChangePassword), controller.
changePassword);
```

It will validate old and new Passwords with authorization.

REST API Change Own Password

Request

URL: `http://127.0.0.1:5000/api/users/changePassword/b930d02c-43af-4875-b7e9-546c9f4c23dd`
Method: POST
body :
{
 "oldPassword": "Admin@pms1",
 "newPassword": "Admin@pms123"
}

Response

```
{
  "statusCode": 200,
  "status": "success",
```

```
    "message": "Password is updated successfully"
}
```

In case of a wrong **oldPassword** sent in a request, the server will respond with the following error message:

```
{
  "statusCode": 400,
  "status": "error",
  "message": "oldPassword is not matched"
}
```

Similarly, if a wrong **newPassword** is sent in the request then it will result in a 400 error code, as follows:

```
{
  "statusCode": 400,
  "status": "error",
  "errors": [
    {
      "newPassword": "It should include at least one uppercase
letter, one lowercase letter, one special symbol, and one
numerical digit."
    }
  ]
}
```

In another case, if the **user_id** in the request URL does not match the access token passed in the Authorization, then it gives the following response:

```
{
  "statusCode": 400,
  "status": "error",
  "message": "User can change only own password"
}
```

In this manner, users can effortlessly change their password successfully.

Recover Password

The password recovery process provides a crucial safety net for users who have forgotten their account credentials or need to reset their passwords. By offering a reliable and secure way to regain access to their accounts, this

feature enhances user experience and maintains the integrity of your application's security.

The password recovery process will be divided into two distinct steps. The first step involves the **Forgot Password** functionality, which triggers the sending of a reset link to the user's email address. In the second step, users will be able to reset their password by validating the reset link's token, thereby granting them access to set a new password.

Forgot Password

For sending mail to the user on forgot password, we need to install the **nodemailer** package. So, open the terminal with the root directory and paste the following command:

```
npm install nodemailer --save
npm install @types/nodemailer -D
```

For sending mail, you need to set up a Gmail account. If you don't have one, then create an account and afterward follow these steps:

1. Sign in to your Google Account:

Start by signing in to the Gmail account from which you want to send emails using Nodemailer.

2. Access Security Settings:

Click your profile picture or initials in the top-right corner of the Gmail interface. From the drop-down menu, select **Manage your Google Account**.

3. Navigate to Security:

In the Google Account settings, navigate to the left sidebar and click **Security**.

4. App Passwords:

Scroll down to the **Signing in to Google** section and find the **App passwords** option. Click **App passwords**.

5. Generate App Password:

You may need to verify your identity using your Google Account password. Once verified, you'll be able to generate an app-specific password.

1. Choose the app for which you want to generate the password. Select **Mail** if it's not listed and choose **Other (Custom Name)**.
2. Enter a custom name for your app-specific password (for example, **Nodemailer App**). Click **Generate**.
3. Copy the App Password:
Google will generate a unique app-specific password for your application. Copy this password and keep it secure.

6. Use App Password in Your Code:

Replace the regular password in your code with the app-specific password you generated. This app-specific password will be used to authenticate your application with Gmail's SMTP server.

Keep in mind that app-specific passwords are only shown once when generated. If you lose the password or need to generate a new one, you'll need to repeat the process.

Now add **email_configs** and front URL in **config.ts**:

```
// config.ts
export interface IServerConfig {
  port: number;
  db_config: {
    'db': string;
    'username': string;
    'password': string;
    'host': string;
    'port': number;
    'dbname': string;
  };
  email_config: {
    'from': string;
    'user': string;
    'password': string;
  };
  front_app_url: string;
}
```

Also, provide actual values in **server_config.json** file:

For example:

```

"email_config": {
  "from": "pms-support@pms.com",
  "user": "pmsbook2023@gmail.com",
  "password": "*****"
},
"front_app_url": "http://127.0.0.1:5000"

```

After that, create `email_util.ts` in the `utils` directory using the following code:

https://github.com/ava-orange-education/Hands-on-Rest-APIs-with-Node.js-and-Express/blob/main/ch-05-code-files/pms-be/src/utils/email_util.ts

The `sendMail` function provides a way to send emails using Gmail's SMTP server through Nodemailer. Make sure to replace the Gmail email, password, and other configurations with your actual values and customize the email content as needed.

For the **Forgot Password** feature, users will initiate the process by submitting their email address. To facilitate this, we will enhance the `UsersUtil` with a function named `getUserByEmail`, responsible for retrieving user information based on their email. Here's the modified `UsersUtil` with the added function:

```

export class UsersUtil {
  // ... Existing methods and functionalities ...
  public static async getUserByEmail(email: string) {
    try {
      if (email) {
        const service = new UsersService();
        const users = await service.customQuery(`email =
          '${email}'`);
        if (users && users.length > 0) {
          return users[0];
        }
      }
    } catch (error) {
      console.error(`Error while getUserFromToken() =>
        ${error.message}`);
    }
  }
}

```

```

        return null;
    }
    // ... Other methods ...
}

```

Now create forgot password function in the **users_controller.ts** file as follows:

```

// users_controller.ts
public async forgotPassword(req: Request, res: Response): Promise<void> {
    const { email } = req.body;
    const emailPattern = /^[^@\s]+@[^\s@]+\.[^\s@]+$/;
    if (!emailPattern.test(email)) {
        res.status(400).send({ statusCode: 400, status: 'error',
            message: 'Invalid email' });
        return;
    }
    const user: Users = await UsersUtil.getUserByEmail(email);
    if (!user) {
        res.status(404).send({ statusCode: 404, status: 'error',
            message: 'User Not Found' });
        return;
    }
    // Generate a reset token for the user
    const resetToken: string = jwt.sign({ email: user.email },
        SERVER_CONST.JWTSECRET, {
            expiresIn: '1h',
        });
    // Generate the reset link
    const resetLink = `${config.front_app_url}/reset-password?
token=${resetToken}`;
    const mailOptions = {
        to: email,
        subject: 'Password Reset',
        html: ` Hello ${user.username},<p>We received a request to
        reset your password. If you didn't initiate this request,
        please ignore this email.</p>
        <p>To reset your password, please click the link below:</p>

```

```

<p><a href="${resetLink}" style="background-color: #007bff; color: #ffffff; text-decoration: none; padding: 10px 20px; border-radius: 5px; display: inline-block;">Reset Password</a></p>
<p>If the link doesn't work, you can copy and paste the following URL into your browser:</p>
<p>${resetLink}</p>
<p>This link will expire in 1 hour for security reasons.</p>
<p>If you didn't request a password reset, you can safely ignore this email.</p>
<p>Best regards,<br>PMS Team</p>`,
};

const emailStatus = await sendMail(mailOptions.to, mailOptions.subject, mailOptions.html);
if (emailStatus) {
  res.status(200).json({ statusCode: 200, status: 'success', message: 'Reset Link sent on your mailId', data: { 'resetToken': resetToken } });
} else {
  res.status(400).json({ statusCode: 400, status: 'error', message: 'something went wrong try again' });
}
return;
}

```

The **forgotPassword** function takes the user's email from the request body. It attempts to retrieve a user using the **getUserByEmail** function from **UsersUtil**. If no user is found, a **404** response is sent, indicating "**User Not Found.**" If a user is found, a reset token is generated using JWT with the user's email and a one-hour expiration. A reset link is composed using the front-end application URL and the generated reset token. An email template is constructed with the reset link and instructions for the user. The **sendMail** function is called to send the email, including recipient, subject, and HTML content. If the email is successfully sent, a **200** response is sent with a success message and the reset token. If sending the email fails, a **400** response is sent with an error message.

This function is called from `users_routes.ts`, let's create a route for it:

```
// users_routes.ts
export class UserRoutes {
//Other routes
...
app.route('/api/forgot_password')
  .post(controller.forgotPassword);
}
```

REST API Forgot Password

Request

URL: `http://127.0.0.1:3000/api/forgot_password`
Method: POST
body: {
 "email": "yamipanchal1993@gmail.com"
}

Response

```
{
  "statusCode": 200,
  "status": "success",
  "message": "Reset Link sent on your mailId",
  "data": {
    "resetToken":
      "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InlhbwlwYW5
      jaGFsMTk5M0BnbWFpbC5jb20iLCJpYXQiOjE20TMxNTU0MDIsImV4cCI6MTY5
      MzE10TAwMn0.dM10tE43CSt8nArHjgHxRElk3IRBmdd_NHqpL5f1viU"
  }
}
```

In case, if email is not passed or the requested email has no user, the server will respond with a **400** and **404** status code, respectively.

For example:

```
{
  "statusCode": 404,
  "status": "error",
  "message": "User Not Found"
}
```

When successfully sent, the mail will look like the following:

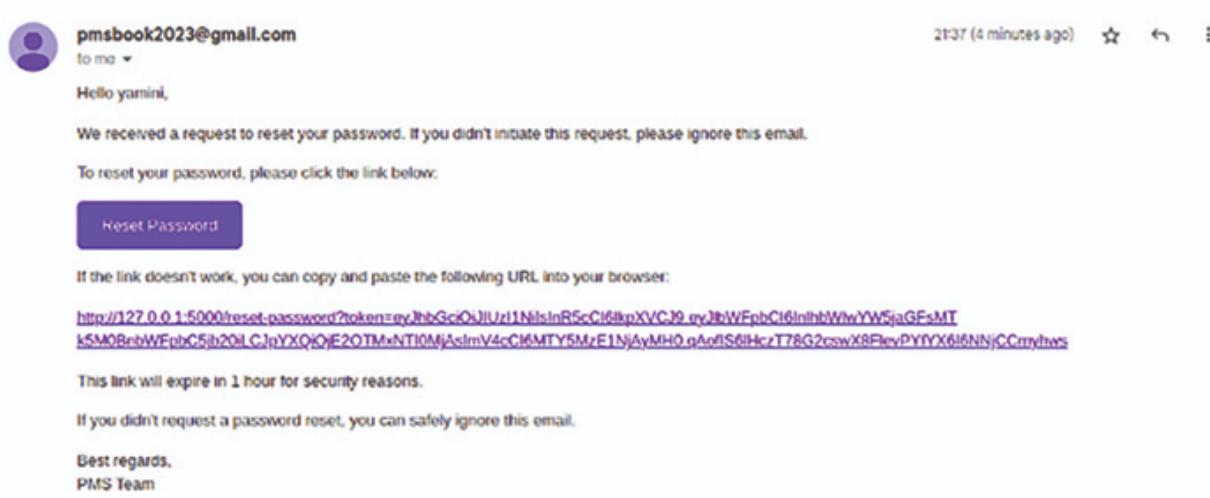


Figure 5.6: Forgot Mail Sent Output

Note: In the context of the **Forgot Password** functionality, please note that the inclusion of the **resetToken** in the API response is solely for explanatory purposes. In actual practice, it is not recommended to provide the **resetToken** in the API response for security reasons.

Reset Password

Reset password is based on the rest token given in the forgotten password. Upon clicking the reset link, users are directed to a secure web page where they can enter their new password. This page confirms the validity of the token and prompts them to create a strong password that adheres to the platform's security requirements. On the backend side, the API first validates the request token and password. If it is valid, then only the password will be reset.

Create method for **resetPassword** in the **users_controller.ts** file as follows:

```
// users_controller.ts
public async resetPassword(req: Request, res: Response): Promise<void> {
  const { newPassword, token } = req.body;
  const service = new UsersService();
  let email;
```

```

try {
  const decoded = jwt.verify(token, SERVER_CONST.JWTSECRET);
  if (!decoded) {
    throw new Error('Invalid Reset Token');
  }
  email = decoded['email'];
} catch (error) {
  res.status(400).json({ statusCode: 400, status: 'error',
  message: 'Reset Token is invalid or expired' }).end();
  return;
}
try {
  const user = await UsersUtil.getUserByEmail(email);
  if (!user) {
    res.status(404).json({ statusCode: 404, status: 'error',
    message: 'User not found' }).end();
    return;
  }
  // Encrypt the user's new password
  user.password = await encryptString(newPassword);
  const result = await service.update(user.user_id, user);
  if (result.statusCode === 200) {
    res.status(200).json({ statusCode: 200, status:
    'success', message: 'Password updated successfully' });
  } else {
    res.status(result.statusCode).json(result).end();
  }
} catch (error) {
  console.error(`Error while resetPassword =>
${error.message}`);
  res.status(500).json({ statusCode: 500, status: 'error',
  message: 'Internal Server error' }).end();
}
}

```

The `resetPassword` extracts the `newPassword` and `token` from the request body. It attempts to decode the token using JWT. If decoding the token fails, a **400** response is sent with an **Invalid Reset Token** error. If decoding the

token succeeds, the function attempts to retrieve the user using the decoded email from the token. If no user is found, a **404** response is sent with a “User not found” error. If a user is found, the new password is encrypted and assigned to the user's password property. The update method of **UsersService** is called to update the user's information in the database. If the update is successful (status code **200**), a **200** response is sent with a success message. If the update fails, the response status and result are based on the result of the update operation. In case of any error during the process, a **500** response is sent with an “Internal Server Error” message.

This function is initiated from **users_routes.ts** as follows:

```
const validResetPassword = [
  body('token').trim().notEmpty().withMessage('It should be
  required'),
  body('newPassword')
    .isLength({ min: 6, max: 12 }).withMessage('It must be
    between 6 and 12 characters in length')
    .isStrongPassword({ minLowercase: 1, minUppercase: 1,
    minSymbols: 1, minNumbers: 1 })
    .withMessage('It should include at least one uppercase
    letter, one lowercase letter, one special symbol, and one
    numerical digit.'), body('role_ids'),
];
export class UserRoutes {
  //Other routes
  ...
  app.route('/api/reset_password')
    .post(validate(validResetPassword),
    controller.resetPassword);
}
```

REST API Reset Password

Request

URL: http://127.0.0.1:3000/api/reset_password
Method: POST
body: {
 "newPassword": "Pmsbook@123",

```
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
        eyJlbWFpbCI6InlhbWlwYW5jaGFsMTk5M0BnbWFpbC5jb20iLCJpYXQiOjE2OT  
        MwNzc1MTYsImV4cCI6MTY5MzA4MTExNn0.tAJgxypxNm5-  
        xVgQtusLFIYomzax2-HQSGYPBNQUt4"  
    }
```

Response

```
{  
    statusCode: 200, status: 'success', message: 'Password  
    updated successfully'  
}
```

In case of token is not valid or expired, it gives 400 error as follows:

```
{  
    "statusCode": 400,  
    "status": "error",  
    "message": "Reset Token is invalid or expired"  
}
```

Effective password management is paramount for user security. Implementing strong encryption, token-based resets, and clear policies fortifies protection and enhances user trust. We follow that standard kind of flow for password management.

Conclusion

In this chapter, we covered a common base service for database operation and a base controller to perform REST API CRUD operations and other utility functions. We implemented the whole Role and User management with input validations for the PMS application. In addition, we also learned to send email functionality. In the next chapter, we will learn about the project and task core functionality of the PMS application.

CHAPTER 6

REST API for Project and Task Modules

Introduction

In project management systems (PMS), a “*project module*” typically refers to a specific component or core module. These modules are designed to streamline various aspects of project planning, execution, and monitoring, making it easier for project managers and teams to manage complex projects efficiently.

Each project module typically focuses on a specific area, such as task management, resource allocation, time tracking, or reporting, providing dedicated tools and features to support those functions. These modules are an essential part of PMS, enabling users to customize their project management approach based on the specific needs of their projects. The task module plays a crucial role in project management by providing a centralized and organized approach for managing and executing tasks within a project. It enhances collaboration, improves visibility into task progress, and helps ensure that projects are completed on time and within scope.

Structure

In this chapter, we will discuss the following topics:

- Project Management with Input Validation
 - Project Creation API with Assigned User
 - Project Update, List, Details of Project, and Delete APIs
- Task Management with Input Validation
 - Task Creation API with Assigned User to Task
 - Task Update, List, Details of Task, and Delete APIs

Project Management

The project module serves as a centralized platform that empowers project managers and teams to efficiently plan, execute, and complete projects while maintaining control, transparency, and collaboration.

Project Service

We have previously defined the entity for the project in [Chapter 4, Planning the App](#), so let's start by first creating the Project Service as `projects_service.ts` in the project directory with the following code:

```
import { Repository } from 'typeorm';
import { BaseService } from '../../utils/base_service';
import { DatabaseUtil } from '../../utils/db';
import { Projects } from './projects_entity';

export class ProjectsService extends BaseService<Projects> {

  constructor() {
    let projectRepository: Repository<Projects> | null = null;
    projectRepository = new
      DatabaseUtil().getRepository(Projects);
    super(projectRepository);
  }
}
```

The `ProjectsService` class extends a `BaseService` and uses TypeORM to work with the `Projects` entity in the database. The repository for this entity is obtained from the `DatabaseUtil` class. The service is structured to facilitate database operations on the `Projects` entity, such as creating, retrieving, updating, and deleting records.

We will develop the REST API for the project as follows:

- **Add Project**
- **GetAll Project**
- **GetOne Project**
- **Update Project**
- **Delete Project**

Input Validation

Input validation in a project module plays a critical role in ensuring the integrity and reliability of data used within a project management system. It involves examining and verifying user inputs or data received from various sources to ensure that it adheres to predefined criteria and meets the required standards.

Add the following function, **checkValidDate**, in the **common.ts** file in the **utils** directory to validate a date:

```
export const checkValidDate = function (value) {
  if (!moment(value, 'YYYY-MM-DD HH:mm:ss', true).isValid()) {
    return false;
  }
  return true;
};
```

Now, add **validProjectInput** in the **projects_routes.ts** file with the following code:

```
import { Express } from 'express';
import { ProjectController } from './projects_controller';
import { body } from 'express-validator';
import { validate } from '../../utils/validator';
import moment from 'moment';
import { authorize } from '../../utils/auth_util';
import { checkValidDate } from '../../utils/common'

const validProjectInput = [
  body('name').trim().notEmpty().withMessage('It should be required'),
  body('user_ids').isArray().withMessage('It should be ids of users array'),
  body('start_time').custom((value) => {
    if (!checkValidDate(value)) {
      throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(value);
    const currentTime = new Date();
    if (startTime <= currentTime) {
```

```

        throw new Error('Start time must be greater than the
        current time');
    }
    return true;
}),
body('end_time').custom((value, { req }) => {
    if (!checkValidDate(value)) {
        throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(req.body.start_time);
    const endTime = new Date(value);

    if (endTime <= startTime) {
        throw new Error('End time must be greater than the start
        time');
    }
    return true;
})
];

```

Here are the details of the key parts of the preceding code:

- **validProjectInput Array:** This array contains a series of validation checks for each input field expected in a project. Each element of this array is a validator function that checks a specific aspect of the data.
- **body('name').trim().notEmpty().withMessage('It should be required'):** This validates the `name` field in the request `body`. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty; `.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the '`name`' field is required.
- **body('user_ids').isArray().withMessage('It should be ids of users array'):** This validates the `user_ids` field in the request `body`. `.isArray()` checks that the field is an array; `.withMessage('It should be ids of users array')` provides a custom error message if the validation fails, specifying that the '`user_ids`' field should be an array of user IDs.

- **body('start_time').custom((value) => { /* ... */ })**: This validates the `start_time` field in the request body using a custom validation function. The custom validation function checks if the value (the `'start_time'` value) is in a valid date format (`YYYY-MM-DD HH:mm:ss`). It then compares the `'start_time'` with the current time, ensuring that the `'start_time'` is greater than the current time. If any of these checks fail, it throws an error with a custom message.
- **body('end_time').custom((value, { req }) => { /* ... */ })**: This validates the `end_time` field in the request body using another custom validation function. Similar to the previous custom function, it checks if value (the `'end_time'` value) is in a valid date format (`YYYY-MM-DD HH:mm:ss`). It also accesses `req.body.start_time` to compare the `'end_time'` with the `'start_time'` to ensure that `'end_time'` is greater than `'start_time'`. If any of these checks fail, it throws an error with a custom message

Add Project

When utilizing the REST API to add a project, you typically furnish the required data within the request body, encompassing details such as the project's name, description, and any associated attributes. The designated API endpoint for this task is purpose-built to accept and validate this data in accordance with predefined criteria, culminating in the creation of a new project rooted in the supplied information.

We have previously created `projects_controller.ts` as a skeleton class. Now, let's change it with an extended Base Controller and use base service to perform database operations with the following code:

```
import { Response, Request } from 'express';
import { hasPermission } from '../../utils/auth_util';
import { ProjectsService } from './projects_service';
import { UsersUtil } from '../users/users_controller';

export class ProjectController extends BaseController {
  /**
   * Handles the addition of a new user.
   * @param {object} req - The request object.
   * @param {object} res - The response object.
   */
}
```

```
public async addHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'add_project')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  try {
    // Create an instance of the ProjectService
    const service = new ProjectsService();

    // Extract project data from the request body
    const project = req.body;

    // Check if the provided user_ids are valid
    const isValidUsers = await UsersUtil.checkValidUserIds
    (project.user_ids);

    if (!isValidUsers) {
      // If user_ids are invalid, send an error response
      res.status(400).json({ statusCode: 400, status: 'error',
      message: 'Invalid user_ids' });
      return;
    }

    // If user_ids are valid, create the user
    const createdProject = await service.create(project);
    res.status(201).json(createdProject);

  } catch (error) {
    // Handle errors and send an appropriate response
    console.error(`Error while addUser => ${error.message}`);
    res.status(500).json({ statusCode: 500, status: 'error',
    message: 'Internal server error' });
  }
}

public async getAllHandler(req: Request, res: Response) {}
public async getOneHandler(req: Request, res: Response) {}
public async updateHandler(req: Request, res: Response) {}
```

```
    public async deleteHandler(req: Request, res: Response) {}  
}
```

Add **checkValidUserIds** function in **UserUtil** Class with following code that checks if given **userIds** are valid and exist in the database or not:

```
public static async checkValidUserIds(user_ids: string[]) {  
    const userService = new UsersService();  
  
    // Query the database to check if all user_ids are valid  
    const users = await userService.findByIds(user_ids);  
  
    // Check if all user_ids are found in the database  
    return users.data.length === user_ids.length;  
}
```

In this context, the **addHandler** method acquires the data supplied in the incoming request body. It then proceeds to transmit this data to the base service by invoking the **create** method, which is tasked with incorporating this information into the database. Furthermore, we are including the addition of users to the project, associating users with the project during the project creation process.

Now, let's call **addHandler** in project routes with change in the **projects_router.ts** file as follows:

```
export class ProjectRoutes {  
  
    private baseEndPoint = '/api/projects';  
  
    constructor(app: Express) {  
        const controller = new ProjectController();  
        app.route(this.baseEndPoint)  
            .all(authorize)  
            .post(validate(validProjectInput), controller.addHandler);  
    }  
}
```

We have established routes for adding project and incorporated middleware to validate requests before inserting data into the database.

Once the API is successfully triggered, the following data will be added in PostgreSQL Database:

project_id	name	description	user_ids	start_time	end_time	created_at	updated_at
8eb68c03-d0c6-4f93-ac26...	Car Service Management	This Project is about ...	{64404d4e-f367-4c45-9c4...	2023-10-25 00:00:00	2023-12-15 00:00:00	2024-03-18 12:24:52.89559	2024-03-18 12:24:52.89559

Figure 6.1: Postgres Added Project Output

REST API Add Project

Request

URL : <http://127.0.0.1:3000/api/projects>
 Method: POST
 body :
 {
 "name": "Project Management",
 "description": "This Project is about for Management",
 "User_ids": ["b930d02c-43af-4875-b7e9-546c9f4c23dd",
 "611b346e-be39-4a7e-96d1-e7421193bd5a", "d166945a-f85d-485c-
 bdac-0c8056b3188a"],
 "start_time": "2023-09-25 00:00:00",
 "end_time": "2023-12-15 00:00:00"
 }

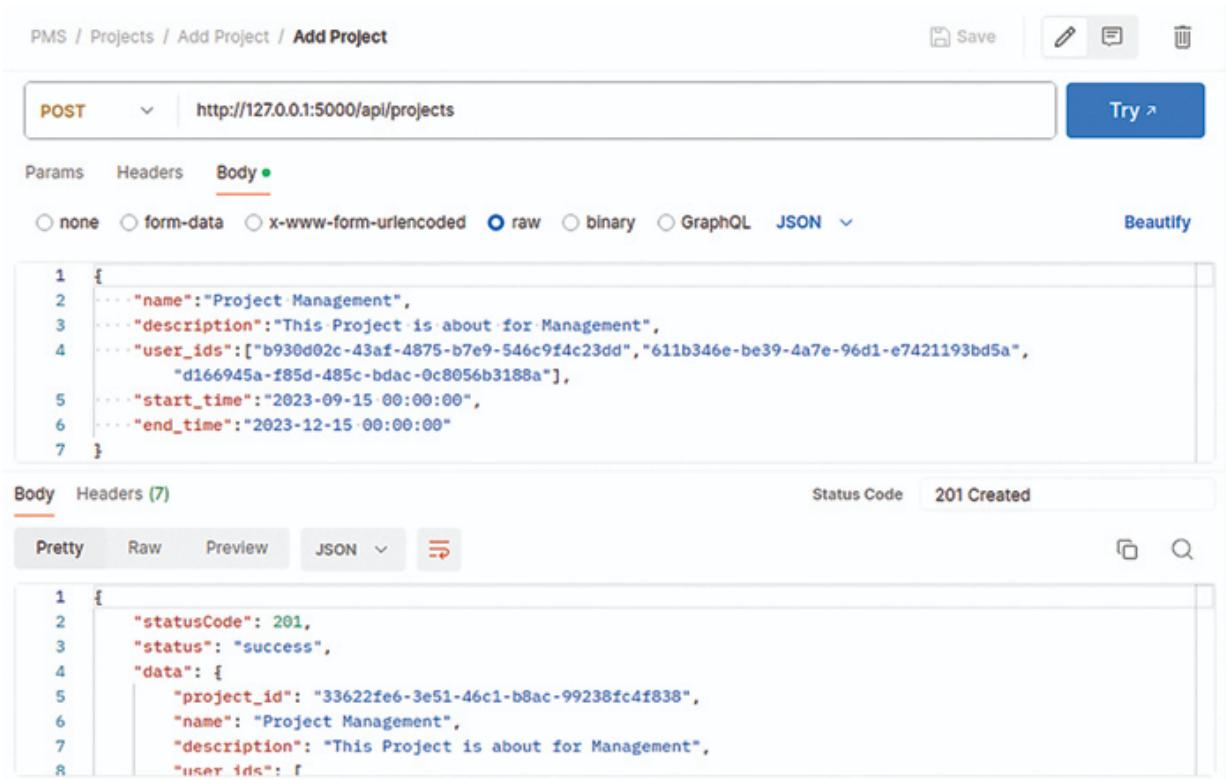
Response

{
 "statusCode": 201,
 "status": "success",
 "data": {
 "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
 "name": "Project Management",
 "description": "This Project is about for Management",
 "user_ids": [
 "b930d02c-43af-4875-b7e9-546c9f4c23dd",
 "611b346e-be39-4a7e-96d1-e7421193bd5a",
 "d166945a-f85d-485c-bdac-0c8056b3188a"
],
 },
 "message": "Project added successfully!"
}

```

    "start_time": "2023-09-25 00:00:00",
    "end_time": "2023-12-15 00:00:00",
    "created_at": "2023-09-23T17:23:36.061Z",
    "updated_at": "2023-09-23T17:23:36.061Z"
}
}

```



The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** <http://127.0.0.1:5000/api/projects>
- Body:** JSON (Pretty) - The request body is a JSON object representing a project:


```

1 {
2     "name": "Project Management",
3     "description": "This Project is about for Management",
4     "user_ids": ["b930d02c-43af-4875-b7e9-546c9f4c23dd", "611b346e-be39-4a7e-96d1-e7421193bd5a",
5                 "d166945a-f85d-485c-bdac-0c8056b3188a"],
6     "start_time": "2023-09-15 00:00:00",
7     "end_time": "2023-12-15 00:00:00"
}
      
```
- Response:** Status Code 201 Created
- Body (Pretty):** The response body is a JSON object:


```

1 {
2     "statusCode": 201,
3     "status": "success",
4     "data": {
5         "project_id": "33622fe6-3e51-46c1-b8ac-99238fc4f838",
6         "name": "Project Management",
7         "description": "This Project is about for Management",
8         "user_ids": [
      
```

Figure 6.2: Postman Response of Add Project Data 201

In the case of a unique project name, trying again with the same request gives an error as a **409** conflict code:

```

{
  "statusCode": 409,
  "status": "error",
  "message": "Key (name)=(Project Management) already exists."
}

```

The screenshot shows the Postman interface with a POST request to `http://127.0.0.1:3000/api/projects`. The request body is a JSON object representing a project with a name, description, user IDs, start time, and end time. The response is a 409 Conflict status code with a JSON body indicating that a project with the name 'Project Management' already exists.

```

1 {
2   ...
3   "name": "Project Management",
4   ...
5   "description": "This Project is about for Management",
6   ...
7   "user_ids": ["64404d4e-1367-4c45-9c4c-b45b3d8807d1"],
8   ...
9   "start_time": "2024-03-25 00:00:00",
10  ...
11  "end_time": "2024-06-15 00:00:00"
12 }

```

```

1 {
2   "statusCode": 409,
3   "status": "error",
4   "message": "Key (name)=(Project Management) already exists."
5 }

```

Figure 6.3: Postman Response of Already Exists Project 409

In another case, if you change in `start_time` less than current time, then it gives an error for a Bad Request with a **400** status code as follows:

```

{
  "statusCode": 400,
  "status": "error",
  "errors": [
    {
      "rights": "Start time must be greater than the current
      time"
    }
  ]
}

```

The screenshot shows a Postman request to `http://127.0.0.1:3000/api/projects` using the `POST` method. The request body is a JSON object with the following structure:

```

1  {
2    "name": "Project Management",
3    "description": "This Project is about for Management",
4    "user_ids": ["64404d4e-f367-4c45-9c4c-b45b3d8807d1"],
5    "start_time": "2023-03-25 00:00:00",
6    "end_time": "2024-06-15 00:00:00"
7  }

```

The response status is `400 Bad Request` with a `62 ms` response time and `357 B` of data. The response body is:

```

1  {
2    "statusCode": 400,
3    "status": "error",
4    "errors": [
5      {
6        "start_time": "Start time must be greater than the current time"
7      }
8    ]
9  }

```

Figure 6.4: Postman Response of Invalid Date Given to Project 400

GetAll Project

After successfully adding a project to the database, the next step involves retrieving the newly inserted project from the database. To achieve this, update the `getAllHandler` method in the `projects_controller.ts` file with the following code:

```

public async getAllHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_all_projects')) {
    res.status(403).json({ statusCode: 403, status: 'error', message: 'Unauthorised' });
    return;
  }
  const service = new ProjectsService();
  const result = await service.findAll(req.query);
  for (const project of result.data) {
    project['users'] = await UsersUtil.getUsernamesById(project.user_ids);
  }
}

```

```

        delete project.user_ids;
    }

    res.status(result.statusCode).json(result);
}

```

In the preceding code, we send the username along with the **user_id** of associated users in the project. So, we have created a method in the user utils that gives the username from **user_id** with the following code:

```

public static async getUsernamesById(user_ids: string[]) {
    const userService = new UsersService();

    // Query the database to check if all user_ids are valid
    const queryResult = await userService.findByIds(user_ids);
    if (queryResult.statusCode === 200) {
        const users = queryResult.data;
        const usernames = users.map((i) => {
            return {
                'username': i.username,
                'user_id': i.user_id
            };
        });
        return usernames;
    }
    return [];
}

```

The `**getAllHandler**` method uses the **ProjectsService** class to retrieve all projects from the database based on the query parameters in the request. The resulting data is then sent back to the client with an appropriate HTTP status code and formatted as JSON.

The routes class in **project_routes.ts** can be updated to make a call to the new handler for GET request as:

```

app.route(this.baseEndPoint)
    .all(authorize)
    .post(validate(validProjectInput), controller.addHandler)
    .get(controller.getAllHandler);

```

By employing this approach, we establish a **GET** route that fetches all projects stored in the database, effectively functioning as a REST API

endpoint for retrieving project data.

REST API GetAll Projects

Request

URL : `http://127.0.0.1:3000/api/projects`
Method: GET
Query Params: {}

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": [
    {
      "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
      "name": "Project Management",
      "description": "This Project is about for Management",
      "start_time": "2023-09-24T18:30:00.000Z",
      "end_time": "2023-12-14T18:30:00.000Z",
      "created_at": "2023-09-23T17:23:36.061Z",
      "updated_at": "2023-09-23T17:23:36.061Z",
      "users": [
        {
          "username": "pms-admin",
          "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd"
        },
        {
          "username": "pms-admin1",
          "user_id": "611b346e-be39-4a7e-96d1-e7421193bd5a"
        },
        {
          "username": "yamini",
          "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a"
        }
      ]
    }
  ]
}
```

Search Project by Name

To search for a project by name, you can utilize the same API endpoint as the one used for retrieving all projects. However, in this case, you will include query parameters to specify the search criteria.

Request

```
URL : http://127.0.0.1:3000/api/projects?name=Project  
Management  
Method: GET  
Query Params: {name: Project Management }
```

Response

```
{  
  "statusCode": 200,  
  "status": "success",  
  "data": [  
    {  
      "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",  
      "name": "Project Management",  
      "description": "This Project is about for Management",  
      "start_time": "2023-09-24T18:30:00.000Z",  
      "end_time": "2023-12-14T18:30:00.000Z",  
      "created_at": "2023-09-23T17:23:36.061Z",  
      "updated_at": "2023-09-23T17:23:36.061Z",  
      "users": [  
        {  
          "username": "pms-admin",  
          "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd"  
        },  
        {  
          "username": "pms-admin1",  
          "user_id": "611b346e-be39-4a7e-96d1-e7421193bd5a"  
        },  
        {  
          "username": "yamini",  
          "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a"  
        }  
      ]  
    }  
  ]
```

```
    }
]
}
```

GetOne Project

The “GetOne Project” endpoint plays a pivotal role in project management systems, allowing users to access in-depth information about a specific project without the requirement to fetch the complete project list. This functionality is indispensable for providing precise details regarding the attributes and permissions linked to each project.

To implement the “GetOne Project” API, modify the **getOneHandler** code in the project controller as follows:

```
public async getOneHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_project')) {
    res.status(403).json({ statusCode: 403, status: 'error',
      message: 'Unauthorised' });
  }
  const service = new ProjectsService();
  const result = await service.findOne(req.params.project_id);
  result.data['users'] = await
    UsersUtil.getUsernamesById(result.data.user_ids);
  delete result.data.user_ids;
  res.status(result.statusCode).json(result);
}
```

The `getOneHandler` function serves as the bridge between the incoming client request, the service layer that interacts with the database, and the outgoing HTTP response. It retrieves a single project’s details from the database based on the provided project ID and sends the project information back to the client.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:project_id')
  .all(authorize)
  .get(controller.getOneHandler);
```

Here, `/:project_id` will be a request parameter meant to capture the ID of the project that the user wants to retrieve.

Parameters passed to APIs can be read using `request.params`, as seen in the previous example while reading `project_id`:

```
const project_id = req.params.project_id;
```

REST API GetOne Project

Request

URL : `http://127.0.0.1:3000/api/projects/`

Method: GET

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
    "name": "Project Management",
    "description": "This Project is about for Management",
    "start_time": "2023-09-24T18:30:00.000Z",
    "end_time": "2023-12-24T18:30:00.000Z",
    "created_at": "2023-09-23T17:23:36.061Z",
    "updated_at": "2023-09-24T17:18:55.827Z",
    "users": [
      {
        "username": "pms-admin",
        "user_id": "b930d02c-43af-4875-b7e9-546c9f4c23dd"
      },
      {
        "username": "pms-admin1",
        "user_id": "611b346e-be39-4a7e-96d1-e7421193bd5a"
      },
      {
        "username": "yamini",
        "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a"
      }
    ]
  }
}
```

```
}
```

Providing a valid project ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was not found.

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

[Update Project](#)

The process of updating a project details involves making changes to the existing data of a particular project that is stored in the database. Through this process, you can modify attributes such as the project's name, description, and associated users. Updating a project is crucial for maintaining the accuracy and currency of project information, especially when there are alterations in project permissions that need to be reflected in the database.

To implement the "**Update Project**" API, make the following changes in the **updateHandler** code within the project controller:

```
public async updateHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'edit_project')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  const project = req.body;
  const service = new ProjectsService();
  const result = await service.update(req.params.id, project);
  res.status(result.statusCode).json(result);
}
```

The `updateHandler` function handles requests to update a task. It begins by checking if the user has the necessary permission, and if so, it extracts the updated project data, initializes the project service, performs the update

operation in the database, and responds to the client with the appropriate status code and result data.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize)
    .get(controller.getOneHandler)
    .put(validate(validProjectInput),
  controller.updateHandler);
```

Here, `/:id` will be a request parameter meant to capture the ID of the project that the user wants to retrieve, and it also validates data before updating in the database.

REST API Update Project

Request

```
URL : http://127.0.0.1:3000/api/projects/
Method: PUT
body :
{
  "name": "Project Management",
  "description": "This Project is about for Management",
  "user_ids": ["b930d02c-43af-4875-b7e9-546c9f4c23dd", "611b346e-be39-4a7e-96d1-e7421193bd5a", "d166945a-f85d-485c-bdac-0c8056b3188a"],
  "start_time": "2023-09-25 00:00:00",
  "end_time": "2023-12-25 00:00:00"
}
```

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
    "name": "Project Management",
    "description": "This Project is about for Management",
    "user_ids": [
```

```

        "b930d02c-43af-4875-b7e9-546c9f4c23dd",
        "611b346e-be39-4a7e-96d1-e7421193bd5a",
        "d166945a-f85d-485c-bdac-0c8056b3188a"
    ],
    "start_time": "2023-09-24T18:30:00.000Z",
    "end_time": "2023-12-24T18:30:00.000Z",
    "created_at": "2023-09-23T17:23:36.061Z",
    "updated_at": "2023-09-24T17:18:55.827Z"
}
}

```

Providing a valid project ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404 error**, signifying that the requested entity was not found.

```

{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}

```

Delete Project

The "**delete**" functionality for projects in a REST API involves the removal of a specific project from the database. This process is managed through an endpoint dedicated to project deletion. When a request is made to this endpoint, it triggers a function that handles the deletion process. The incoming request typically contains the unique identifier (**project_id**) of the project that needs to be deleted.

To implement the Delete Project API, make the following changes in the **deleteHandler** code in the project controller:

```

public async deleteHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'delete_project')) {
    res.status(403).json({ statusCode: 403, status: 'error', message: 'Unauthorised' });
    return;
  }
  const service = new ProjectsService();

```

```
    const result = await service.delete(req.params.id);
    res.status(result.statusCode).json(result);
}
```

The `deleteHandler` processes the request by utilizing a service that interacts with the database. This service is responsible for executing the deletion operation. If the requested project exists in the database and the deletion is successful, the function responds with a success message and an appropriate status code, such as **200 OK**. If the role does not exist, the function returns an error response with a status code of **404 Not Found**, indicating that the project was not located in the database.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize)
  .get(controller.getOneHandler)
  .put(validate(validProjectInput), controller.updateHandler)
  .delete(controller.deleteHandler);
```

Here, `/:id` will be a request parameter meant to capture the ID of the project that the user wants to delete.

REST API Delete Project

Request

URL : <http://127.0.0.1:3000/api/projects/>
Method: DELETE

Response

```
{
  "statusCode": 200,
  "status": "success"
}
```

In the case of an already deleted or not exist in the database:

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

Project Util

Following the development of the foundational base API for Project, several functions have been crafted within the Project Util class. These functions serve as valuable helpers and are intended for utilization in various other modules.

Now, let's proceed to establish the **ProjectUtil** class within the **projects_controller.ts** file. The provided code snippet outlines the structure of this class:

```
export class ProjectsUtil {  
  public static async checkValidProjectIds(project_ids: string[]) {  
    const projectService = new ProjectsService();  
    // Query the database to check if all project_ids are valid  
    const projects = await projectService.findByIds(project_ids);  
    // Check if all project_ids are found in the database  
    return projects.data.length === project_ids.length;  
  }  
}
```

In the preceding code, the `checkValidProjectIds` function checks whether the given **projects_ids** are valid or not in the database.

In summary, a well-designed RESTful API project module enables organizations to efficiently manage their projects, collaborate effectively, and integrate project data into their applications and systems while adhering to best practices in security, validation, and documentation.

Task Management

To manage tasks effectively, individuals and teams often use task management tools and software. This module provides features for creating, assigning, prioritizing, and tracking tasks. Tasks are the building blocks of project management. They help project managers and teams break down complex projects into manageable parts, allocate resources efficiently, and ensure that work progresses according to the project plan.

Let's start creating a task service first to manage tasks.

Task Service

Task Service is to facilitate the creation, updation, deletion, and retrieval of task details as required. Let's create a **tasks_services.ts** file in the tasks directory with the following code:

```
import { Repository } from 'typeorm';
import { BaseService } from '../../utils/base_service';
import { DatabaseUtil } from '../../utils/db';
import { Tasks } from './tasks_entity';

export class TasksService extends BaseService<Tasks> {

  constructor() {
    let taskRepository: Repository<Tasks> | null = null;
    taskRepository = new DatabaseUtil().getRepository(Tasks);
    super(taskRepository);
  }
}
```

The **TasksService** class inherits from a base service class, allowing it to interact with a database repository for managing tasks. The constructor initializes the database repository for tasks, facilitating database operations for tasks-related functionality in the application.

We will develop the REST API for the project as follows:

- **Add** Task
- **GetAll** Task
- **GetOne** Task
- **Update** Task
- **Delete** Task

Input Validation

Task input validation is the process of ensuring that data provided as input for tasks in a software application conforms to predefined criteria and constraints.

Now, let's add **validTaskInput** in the **tasks_routes.ts** file with the following code:

```
import { Express } from 'express';
import { TaskController } from './tasks_controller';
import { body } from 'express-validator';
import { checkValidDate } from '../../utils/common';
import { validate } from '../../utils/validator';
import { authorize } from '../../utils/auth_util';
const validTaskInput = [
  body('name').trim().notEmpty().withMessage('It should be required'),
  body('project_id').trim().notEmpty().withMessage('It should be required'),
  body('user_id').trim().notEmpty().withMessage('It should be required'),
  body('estimated_start_time').trim().notEmpty().withMessage('It should be required'),
  body('estimated_end_time').trim().notEmpty().withMessage('It should be required'),
  body('estimated_start_time').custom((value) => {
    if (!checkValidDate(value)) {
      throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(value);
    const currentTime = new Date();

    if (startTime <= currentTime) {
      throw new Error('Start time must be greater than the current time');
    }
    return true;
}),
  body('estimated_end_time').custom((value, { req }) => {
    if (!checkValidDate(value)) {
      throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(req.body.start_time);
    const endTime = new Date(value);

    if (endTime <= startTime) {
```

```
        throw new Error('End time must be greater than the start
        time');
    }
    return true;
})
];

```

Here are the details of the key parts of the preceding code:

- **body('name').trim().notEmpty().withMessage('It should be required')**: This validates the name field in the request body. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty; `.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the 'name' field is required.
- **body('project_id').trim().notEmpty().withMessage('It should be required')**: This validates the name field in the request `body`. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty; `.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the 'project_id' field is required.
- **body('user_id').trim().notEmpty().withMessage('It should be required')**: This validates the name field in the request `body`. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty; `.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the 'user_id' field is required.
- **body('estimated_start_time').trim().notEmpty().withMessage('It should be required')**: This validates the name field in the request `body`. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty; `.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the 'estimated_start_time' field is required.
- **body('estimated_end_time').trim().notEmpty().withMessage('It should be required')**: This validates the name field in the request `body`. `.trim()` removes any leading or trailing whitespace from the input; `.notEmpty()` checks that the field is not empty;

`.withMessage('It should be required')` provides a custom error message if the validation fails, indicating that the `"estimated_end_time"` field is required.

- `body('estimated_start_time').custom((value) => { /* ... */ })`: This validates the `start_time` field in the request body using a custom validation function. The custom validation function checks if the value (the `'start_time'` value) is in a valid date format (`YYYY-MM-DD HH:mm:ss`). It then compares the `'estimated_start_time'` with the current time, ensuring that the `'estimated_start_time'` is greater than the current time. If any of these checks fail, it throws an error with a custom message.
- `body('estimated_end_time').custom((value, { req }) => { /* ... */ })`: This validates the `estimated_start_time` field in the request body using another custom validation function. Similar to the previous custom function, it checks if the value (the `'estimated_end_time'` value) is in a valid date format (`YYYY-MM-DD HH:mm:ss`). It also accesses the `req.body` to compare the `'estimated_end_time'` with the `'estimated_start_time'` to ensure that `'estimated_end_time'` is greater than `'estimated_start_time'`. If any of these checks fail, it throws an error with a custom message.

In this manner, we can implement fundamental validation for tasks.

Add Task

When employing the REST API to initiate the addition of a task, the customary procedure involves providing essential information within the request body. This information typically encompasses details such as the task's name, description, and any related attributes. The dedicated API endpoint, tailored for this specific purpose, is designed to receive and meticulously validate this data, ensuring it aligns with predefined criteria. Subsequently, this validation process culminates in the creation of a fresh task, founded upon the information furnished in the request.

Previously, we established the initial structure of the `tasks_controller.ts` class as a skeletal outline. Now, let's proceed to enhance its functionality by extending it from the `BaseController` and leveraging the capabilities of the `BaseService` to carry out database operations.

Following is the code for this implementation:

```
import { Response, Request } from 'express';
import { hasPermission } from '../../utils/auth_util';
import { BaseController } from '../../utils/base_controller';
import { TasksService } from './tasks_service';
import { UsersUtil } from '../users/users_controller';
import { ProjectsUtil } from '../projects/projects_controller';
export class TaskController extends BaseController {

  /**
   * Handles the addition of a new user.
   * @param {object} req - The request object.
   * @param {object} res - The response object.
   */
  public async addHandler(req: Request, res: Response): Promise<void> {

    if (!hasPermission(req.user.rights, 'add_task')) {
      res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
      return;
    }
    try {
      // Create an instance of the ProjectService
      const service = new TasksService();

      // Extract task data from the request body
      const task = req.body;

      //check if the provided project_id is valid
      const isValidProject = await ProjectsUtil.
      checkValidProjectIds([task.project_id]);

      if (!isValidProject) {
        // If user_ids are invalid, send an error response
        res.status(400).json({ statusCode: 400, status: 'error',
          message: 'Invalid project_id' });
        return;
      }
    }
  }
}
```

```

// Check if the provided user_id is valid
const isValidUser = await
UsersUtil.checkValidUserIds([task.user_id]);

if (!isValidUser) {
  // If user_ids are invalid, send an error response
  res.status(400).json({ statusCode: 400, status: 'error',
  message: 'Invalid user_id' });
  return;
}

// If user_ids are valid, create the user
const createdTask = await service.create(task);
res.status(201).json(createdTask);

} catch (error) {
  // Handle errors and send an appropriate response
  console.error(`Error while addUser => ${error.message}`);
  res.status(500).json({ statusCode: 500, status: 'error',
  message: 'Internal server error' });
}

}

public async getAllHandler(req: Request, res: Response) {}
public async getOneHandler(req: Request, res: Response) {}
public async updateHandler(req: Request, res: Response) {}
public async deleteHandler(req: Request, res: Response) {}
}

```

The **addHandler** method retrieves the data provided within the incoming request body. Subsequently, it forwards this data to the Base service by calling the create method, responsible for integrating this information into the database. Additionally, we are extending this process to include the assignment of the user and project to the task, effectively associating the user and project with the task during its creation.

Now, let's call **addHandler** in the task routes with change in the **tasks_router.ts** file as follows:

```

export class TaskRoutes {

  private baseEndPoint = '/api/projects';

```

```

constructor(app: Express) {
  const controller = new ProjectController();

  app.route(this.baseEndPoint)
    .all(authorize)
    .post(validate(validTaskInput), controller.addHandler);
}

}

```

We have established routes for adding tasks and incorporated middleware to validate requests before inserting data into the database.

REST API Add Project

Request

URL : <http://127.0.0.1:3000/api/tasks>
 Method: POST
 body :
 {
 "name": "Setup Database",
 "description": "create one postgres database and setup database
 for project
 management project",
 "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
 "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
 "estimated_start_time": "2023-10-01 00:00:00",
 "estimated_end_time": "2023-10-02 00:00:00"
 }

Response

{
 "statusCode": 201,
 "status": "success",
 "data": {
 "task_id": "74f61799-7046-47d9-8f04-897f07b4e178",
 "name": "Setup Database",
 "description": "create one postgres database and setup
 database for project management project",
 "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
 "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
 }

```

    "estimated_start_time": "2023-10-01 00:00:00",
    "estimated_end_time": "2023-10-02 00:00:00",
    "actual_start_time": null,
    "actual_end_time": null,
    "priority": "Low",
    "status": "Not-Started",
    "supported_files": [],
    "created_at": "2023-09-30T12:17:51.138Z",
    "updated_at": "2023-09-30T12:17:51.138Z"
  }
}

```

In the case of a unique task name, trying again with the same request gives an error as **409** conflict code:

```
{
  "statusCode": 409,
  "status": "error",
  "message": "Key (name)=(Setup Database) already exists."
}
```

In another case, if you change in rights as “**rights**”:”**no_rights**”, it gives an error for a Bad Request with a **400** status code as follows:

```
{
  "statusCode": 400,
  "status": "error",
  "errors": [
    {
      "rights": "Invalid permission"
    }
  ]
}
```

In the database for the task entity, we added the default value for priority as “**Low**” and status as “**Not-Started**”, so that it takes automatically while creating the task.

[GetAll Task](#)

After successfully adding a task to the database, the next step involves retrieving the newly inserted task from the database. To achieve this, update the `getAllHandler` method in the `tasks_controller.ts` file with the following code:

```
public async getAllHandler(req: Request, res: Response):  
Promise<void> {  
  if (!hasPermission(req.user.rights, 'get_all_tasks')) {  
    res.status(403).json({ statusCode: 403, status: 'error',  
    message: 'Unauthorised' });  
    return;  
  }  
  const service = new TasksService();  
  const result = await service.findAll(req.query);  
  res.status(result.statusCode).json(result);  
}
```

A Task is associated with both a project and a user. In this API, we must also display the details of the project and user. Therefore, the standard `findAll` method from the base service does not provide the functionality we require. To achieve this, we need to override the `findAll` method in the `tasks_service.ts` file with the following code:

```
export class TasksService extends BaseService<Tasks> {  
  private taskRepository: Repository<Tasks> | null = null;  
  constructor() {  
    let taskRepository: Repository<Tasks> | null = null;  
    taskRepository = new DatabaseUtil().getRepository(Tasks);  
    super(taskRepository);  
    this.taskRepository = taskRepository;  
  }  
  
  // Override the method from the base service class  
  override async findAll(queryParams: object):  
  Promise<ApiResponse<Tasks[]>> {  
  
    const queryBuilder = await this.taskRepository  
      .createQueryBuilder('task')  
      .leftJoin('task.project_id', 'project')  
      .leftJoin('task.user_id', 'user')  
      .addSelect([
```

```
'task.*',
'task.project_id as project',
'project.project_id',
'project.name',
'user.user_id',
'user.username',
'user.email',
]);
// Build the WHERE clause conditionally based on the search
parameters
if (queryParams['username']) {
  queryBuilder.andWhere('user.username ILIKE :userName', {
    userName:
      `%%${queryParams['username']}%`});
}
if (queryParams['projectname']) {
  queryBuilder.andWhere('project.name ILIKE :projectName', {
    projectName:
      `%%${queryParams['projectname']}%`});
}
if (queryParams['project_id']) {
  queryBuilder.andWhere('task.project_id = :projectId', {
    projectId:
      queryParams['project_id']});
}
const data = await queryBuilder.getMany();
data.forEach((item) => {
  item['projectDetails'] = item.project_id;
  item['userDetails'] = item.user_id;
  delete item.project_id;
  delete item.user_id;
});
```

```
        return { statusCode: 200, status: 'success', data: data };
    }
}
```

We created a query builder using TypeORM to build an SQL query for retrieving tasks with specific details. We use joins to fetch related data from the project and user tables. We specify the columns we want to select in the query using the `addSelect` method. We conditionally add `WHERE` clauses to the query based on the values provided in the `queryParams` object. For example, if the `username` or `projectName` is provided in the `queryParams`, we add a condition to filter the results accordingly.

Overall, this custom `findAll` method enhances the base service's functionality to retrieve tasks with associated project and user details, while also allowing for conditional filtering based on specific criteria.

For a better understanding of which query is running, we can enable logging in the config of the database connection, in the `connectDatabase` function in `db.ts` file:

```
public async connectDatabase() {
    try {
        if (DatabaseUtil.connection) {
            return DatabaseUtil.connection;
        } else {
            const db_config = this.server_config.db_config;
            const AppSource = new DataSource({
                type: 'postgres',
                host: db_config.host,
                port: db_config.port,
                username: db_config.username,
                password: db_config.password,
                database: db_config.dbname,
                entities: [Roles, Users, Projects, Tasks, Comments],
                synchronize: true,
                logging: true,
                poolSize: 10
            });
            await AppSource.initialize();
            DatabaseUtil.connection = AppSource;
        }
    }
}
```

```

        console.log('Connected to the database');
        return DatabaseUtil.connection;
    }
} catch (error) {
    console.error('Error connecting to the database:', error);
}
}

```

The `getAllHandler` method uses the **TasksService** class to retrieve all projects from the database based on the query parameters in the request. The resulting data is then sent back to the client with an appropriate HTTP status code and formatted as JSON.

This controller method call in routes with a change in **tasks_routes.ts** as follows:

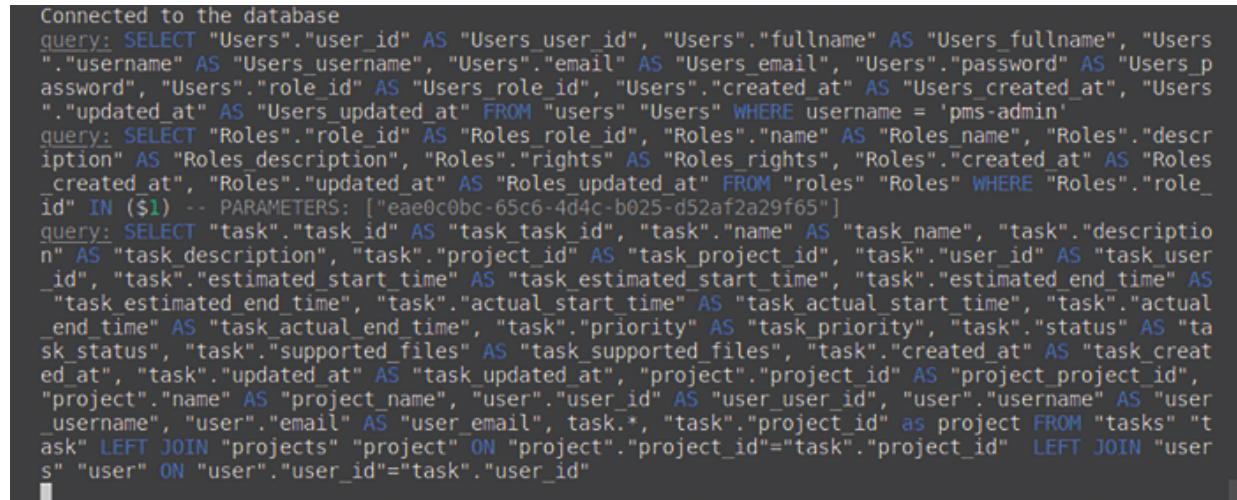
```

app.route(this.baseEndPoint)
    .all(authorize)
    .post(validate(validProjectInput), controller.addHandler)
    .get(controller.getAllHandler);

```

By employing this approach, we establish a **GET** route that fetches all tasks stored in the database, effectively functioning as a REST API endpoint for retrieving task data.

When above API is triggered in terminal, you can see query as follows :



Connected to the database

```

query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"
    ."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_p
    assword", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users
    ".updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'pms-admin'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."descr
    iption" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles
    _created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_
    id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "task"."task_id" AS "task_task_id", "task"."name" AS "task_name", "task"."descriptio
    n" AS "task_description", "task"."project_id" AS "task_project_id", "task"."user_id" AS "task_user
    _id", "task"."estimated_start_time" AS "task_estimated_start_time", "task"."estimated_end_time" AS
    "task_estimated_end_time", "task"."actual_start_time" AS "task_actual_start_time", "task"."actual
    _end_time" AS "task_actual_end_time", "task"."priority" AS "task_priority", "task"."status" AS "ta
    sk_status", "task"."supported_files" AS "task_supported_files", "task"."created_at" AS "task_creat
    ed_at", "task"."updated_at" AS "task_updated_at", "project"."project_id" AS "project_project_id",
    "project"."name" AS "project_name", "user"."user_id" AS "user_user_id", "user"."username" AS "user
    _username", "user"."email" AS "user_email", task.* , "task"."project_id" as project FROM "tasks" "t
    ask" LEFT JOIN "projects" "project" ON "project"."project_id"="task"."project_id" LEFT JOIN "user
    s" "user" ON "user"."user_id"="task"."user_id"

```

Figure 6.5: Enable Query Logging

Now, copy this query and run in **pgAdmin** or **DBeaver** as follows:

```

1 SELECT "task"."task_id" AS "task_task_id", "task"."name" AS "task_name", "task"."description" AS "task_description", "task"."project_id"
2 AS "task_project_id", "task"."user_id" AS "task_user_id", "task"."estimated_start_time" AS "task_estimated_start_time",
3 "task"."estimated_end_time" AS "task_estimated_end_time", "task"."actual_start_time" AS "task_actual_start_time", "task"."actual_end_time"
4 AS "task_actual_end_time", "task"."priority" AS "task_priority", "task"."status" AS "task_status", "task"."supported_files"
5 AS "task_supported_files", "task"."created_at" AS "task_created_at", "task"."updated_at" AS "task_updated_at", "project"."project_id"
6 AS "project_project_id", "project"."name" AS "project_name", "user"."user_id" AS "user_user_id", "user"."username" AS "user_username",
7 "user"."email" AS "user_email", task., "task"."project_id" as project FROM "tasks" "task" LEFT JOIN "projects" "project" ON
8 "project"."project_id"="task"."project_id"
9 LEFT JOIN "users" "user" ON "user"."user_id"="task"."user_id"

```

task_task_id	task_name	task_description	task_project_id	task_user_id	task_estimated_start_time	task_estimated_end_time	task_actual_start_time
2745b6ca-4004-4ad5-9fb4-2d638...	Setup Database	create one postgres database and...	8eb68c03-c0c6-4f93-a...	64404d4e-f367-4c45-9c4...	2024-04-01 00:00:00	2024-04-02 00:00:00	[null]

Figure 6.6: PgAdmin Run Query

REST API GetAll Tasks

Request

URL : <http://127.0.0.1:3000/api/tasks>
 Method: GET
 Query Params: {}

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": [
    {
      "task_id": "74f61799-7046-47d9-8f04-897f07b4e178",
      "name": "Setup Database",
      "description": "create one postgres database and setup database for project management project",
      "estimated_start_time": "2023-09-30T18:30:00.000Z",
      "estimated_end_time": "2023-10-01T18:30:00.000Z",
      "actual_start_time": null,
      "actual_end_time": null,
      "priority": "Low",
      "status": "Not-Started",
      "supported_files": [],
      "created_at": "2023-09-30T12:17:51.138Z",
      "updated_at": "2023-09-30T12:17:51.138Z",
      "projectDetails": {
        "project_id": "8eb68c03-c0c6-4f93-a000-000000000000",
        "name": "Project Management"
      }
    }
  ]
}
```

```

    "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
    "name": "Project Management"
  },
  "userDetails": {
    "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
    "username": "yamini",
    "email": "yamipanchal1993@gmail.com"
  }
}
]
}

```

Search Task

To search for a task by **project_id**, **projectname**, or **username**, you can utilize the same API endpoint as the one used for retrieving all tasks. However, in this case, you will include query parameters to specify the search criteria.

Request

URL : `http://127.0.0.1:3000/api/projects?projectname=project&username=yamini`
 Method: GET
 Query Params: {
 "projectname": "project",
 "username": "yamini"
 }

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": [
    {
      "task_id": "74f61799-7046-47d9-8f04-897f07b4e178",
      "name": "Setup Database",
      "description": "create one postgres database and setup database for project management project",
      "estimated_start_time": "2023-09-30T18:30:00.000Z",
    }
  ]
}
```

```

"estimated_end_time": "2023-10-01T18:30:00.000Z",
"actual_start_time": null,
"actual_end_time": null,
"priority": "Low",
"status": "Not-Started",
"supported_files": [],
"created_at": "2023-09-30T12:17:51.138Z",
"updated_at": "2023-09-30T12:17:51.138Z",
"projectDetails": {
  "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
  "name": "Project Management"
},
"userDetails": {
  "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
  "username": "yamini",
  "email": "yamipanchal1993@gmail.com"
}
}
]
}

```

GetOne Task

The “*Retrieve Individual Task*” endpoint holds a significant role within project management systems. It grants users access to comprehensive task details without necessitating the retrieval of the entire task catalog. This capability is crucial for delivering accurate insights into the specific attributes and permissions associated with each task.

To implement the “**GetOne Task**” API, modify the **getOneHandler** code in the task controller as follows:

```

public async getOneHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_task')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
  }
  const service = new TaskService();

```

```
    const result = await service.findOne(req.params.id);
    res.status(result.statusCode).json(result);
}
```

The `getOneHandler` function serves as the bridge between the incoming client request, the service layer that interacts with the database, and the outgoing HTTP response. It retrieves a single task's details from the database based on the provided task ID and sends the task information back to the client.

Similarly, override `findAll` method in service needs to override the `findOne` method in the `tasks_service.ts` file with the following code:

```
override async findOne(id: string): Promise<ApiResponse<Tasks>
|
undefined> {
    try {
        // Build the WHERE condition based on the primary key
        const where = {};
        const primaryKey: string =
            this.taskRepository.metadata.primaryColumns[0].databaseName;
        where[primaryKey] = id;

        // Use the repository to find the entity based on the
        // provided ID
        const data = await this.taskRepository
            .createQueryBuilder('task')
            .leftJoin('task.project_id', 'project')
            .leftJoin('task.user_id', 'user')
            .addSelect([
                'task.*',
                'task.project_id as project',
                'project.project_id',
                'project.name',
                'user.user_id',
                'user.username',
                'user.email',
            ])
            .where(where)
            .getOne();
```

```

if (data) {
  data['projectDetails'] = data.project_id;
  data['userDetails'] = data.user_id;
  delete data.project_id;
  delete data.user_id;
  return { statusCode: 200, status: 'success', data: data };
} else {
  return { statusCode: 404, status: 'error', message: 'Not
  Found' };
}
} catch (error) {
  return { statusCode: 500, status: 'error', message:
  error.message };
}
}

```

In this context, we have introduced a **"where"** condition to narrow down the data retrieval to a specific task. This condition is applied within the **getOne** method from the controller, which is invoked from the routes file by creating a new route as follows:

```

app.route(this.baseEndPoint + '/:id')
  .all(authorize)
  .get(controller.getOneHandler);

```

Here, `/:id` will be a request parameter meant to capture the ID of the task that the user wants to retrieve.

Once you trigger an API query, it will be logged in the terminal as shown below, and then you can copy it, change \$1 to actual parameters, and paste it into **pgAdmin** or **DBeaver** as follows:

```

query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."desc
ription" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles
_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_
id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "task"."task_id" AS "task_task_id", "task"."name" AS "task_name", "task"."descriptio
n" AS "task_description", "task"."project_id" AS "task_project_id", "task"."user_id" AS "task_user
_id", "task"."estimated_start_time" AS "task_estimated_start_time", "task"."estimated_end_time" AS
"task_estimated_end_time", "task"."actual_start_time" AS "task_actual_start_time", "task"."actual
_end_time" AS "task_actual_end_time", "task"."priority" AS "task_priority", "task"."status" AS "ta
sk_status", "task"."supported_files" AS "task_supported_files", "task"."created_at" AS "task_creat
ed_at", "task"."updated_at" AS "task_updated_at", "project"."project_id" AS "project_project_id",
"project"."name" AS "project_name", "user"."user_id" AS "user_user_id", "user"."username" AS "user
_username", "user"."email" AS "user_email", task.* , "task"."project_id" as project FROM "tasks" "t
ask" LEFT JOIN "projects" "project" ON "project"."project_id"="task"."project_id" LEFT JOIN "user
s" "user" ON "user"."user_id"="task"."user_id" WHERE "task"."task_id" = $1 -- PARAMETERS: ["2745b6
ca-4004-4ad6-9fb4-2d6387ba367a"]

```

Figure 6.7: Get One Task Query

```

1 SELECT "task"."task_id" AS "task_task_id", "task"."name" AS "task_name", "task"."description" AS "task_description", "task"."project_id" AS
2 "task_project_id", "task"."user_id" AS "task_user_id", "task"."estimated_start_time" AS "task_estimated_start_time", "task"."estimated_end_time" AS
3 "task_estimated_end_time", "task"."actual_start_time" AS "task_actual_start_time", "task"."actual_end_time" AS "task_actual_end_time",
4 "task"."priority" AS "task_priority", "task"."status" AS "task_status", "task"."supported_files" AS "task_supported_files", "task"."created_at" AS
5 "task_created_at", "task"."updated_at" AS "task_updated_at", "project"."project_id" AS "project_project_id", "project"."name" AS "project_name",
6 "user"."user_id" AS "user_user_id", "user"."username" AS "user_username", "user"."email" AS "user_email", task.* , "task"."project_id" as project
7 FROM "tasks" "task" LEFT JOIN "projects" "project" ON "project"."project_id"="task"."project_id"
8 LEFT JOIN "users" "user" ON "user"."user_id"="task"."user_id" WHERE "task"."task_id" ='2745b6ca-4004-4ad6-9fb4-2d6387ba367a'

```

Figure 6.8: PgAdmin GetOne Task Query Output

REST API GetOne Task

Request

URL : <http://127.0.0.1:3000/api/tasks/74f61799-7046-47d9-8f04-897f07b4e178>

Method: GET

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "task_id": "74f61799-7046-47d9-8f04-897f07b4e178",
    "name": "Setup Database",
    "description": "create one postgres database and setup database for project management project",
    "estimated_start_time": "2023-09-30T18:30:00.000Z",
    "estimated_end_time": "2023-10-01T18:30:00.000Z",
    "actual_start_time": null,
    "actual_end_time": null,
    "priority": "Low",
    "status": "Not-Started",
    "supported_files": [],
    "created_at": "2023-09-30T12:17:51.138Z",
    "updated_at": "2023-09-30T12:17:51.138Z",
    "projectDetails": {
      "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
      "name": "Project Management"
    }
  }
}
```

```

        },
        "userDetails": {
            "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
            "username": "yamini",
            "email": "yamipanchal1993@gmail.com"
        }
    }
}

```

Providing a valid task ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a 404 error, signifying that the requested entity was not found.

```

{
    "statusCode": 404,
    "status": "error",
    "message": "Not Found"
}

```

Update Task

The process of updating a task detail involves making changes to the existing data of a particular task that is stored in the database. Through this process, you can modify attributes such as the task's name, description, and associated users or projects. Updating a task is crucial for maintaining the accuracy and currency of task information, especially when there are alterations in task permissions that need to be reflected in the database.

To implement the "**Update Task**" API, make the following changes in the **updateHandler** code within the project controller:

```

public async updateHandler(req: Request, res: Response): Promise<void> {
    if (!hasPermission(req.user.rights, 'edit_task')) {
        res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
        return;
    }
    const task = req.body;
    const service = new TasksService();
    const result = await service.update(req.params.id, task);
}

```

```
    res.status(result.statusCode).json(result);
}
```

The `updateHandler` function handles requests to update a task. It begins by checking if the user has the necessary permission, and if so, it extracts the updated task data, initializes the task service, performs the update operation in the database, and responds to the client with the appropriate status code and result data. Here, we have added some validation in **updateTaskInput**.

This method is called from the routes file by creating a new route for it as follows:

```
const updateTaskInput = [
  body('estimated_start_time').custom((value) => {
    if (value && !checkValidDate(value)) {
      throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(value);
    const currentTime = new Date();

    if (startTime <= currentTime) {
      throw new Error('Start time must be greater than the
                     current time');
    }
    return true;
}),
  body('estimated_end_time').custom((value, { req }) => {
    if (value && !checkValidDate(value)) {
      throw new Error('Invalid date format YYYY-MM-DD HH:mm:ss');
    }
    const startTime = new Date(req.body.start_time);
    const endTime = new Date(value);

    if (endTime <= startTime) {
      throw new Error('End time must be greater than the start
                     time');
    }
    return true;
})
];
app.route(this.baseEndPoint + '/:id')
```

```
.all(authorize)
    .get(controller.getOneHandler)
    .put(validate(updateTaskInput), controller.updateHandler);
```

Here, `/:id` will be a request parameter meant to capture the ID of the task that the user wants to retrieve, and it also validates data before updating in the database.

REST API Update Task

Request

```
URL : http://127.0.0.1:3000/api/tasks/74f61799-7046-47d9-8f04-
897f07b4e178
Method: PUT
body :
{
  "name": "Create Microservices",
  "description": "Add microservice to store data on cloud",
  "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
  "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a",
  "status": "In-Progress"
}
```

Response

```
{
  "statusCode": 200,
  "status": "success",
  "data": {
    "task_id": "74f61799-7046-47d9-8f04-897f07b4e178",
    "name": "Create Microservices",
    "description": "Add microservice to store data on cloud",
    "estimated_start_time": "2023-09-30T18:30:00.000Z",
    "estimated_end_time": "2023-10-01T18:30:00.000Z",
    "actual_start_time": null,
    "actual_end_time": null,
    "priority": "Low",
    "status": "In-Progress",
    "supported_files": [],
    "created_at": "2023-09-30T12:17:51.138Z",
    "updated_at": "2023-10-07T15:57:21.912Z",
  }
}
```

```
        "project_id": "c2e9b17b-0af2-453b-b0c9-43ea2d304dca",
        "user_id": "d166945a-f85d-485c-bdac-0c8056b3188a"
    }
}
```

Providing a valid task ID will yield a successful response, while inputting an ID that doesn't correspond to an existing database entry will result in a **404** error, signifying that the requested entity was not found.

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

Delete Task

The "**delete**" functionality for tasks in a REST API involves the removal of a specific task from the database. This process is managed through an endpoint dedicated to task deletion. When a request is made to this endpoint, it triggers a function that handles the deletion process. The incoming request typically contains the unique identifier (**task_id**) of the task that needs to be deleted.

To implement the **Delete** Task API, make the following changes in the **deleteHandler** code in the task controller as follows:

```
public async deleteHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'delete_task')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
  const service = new TasksService();
  const result = await service.delete(req.params.id);
  res.status(result.statusCode).json(result);
}
```

The `**deleteHandler**` processes the request by utilizing a service that interacts with the database. This service is responsible for executing the

deletion operation. If the requested project exists in the database and the deletion is successful, the function responds with a success message and an appropriate status code, such as **200 OK**. If the role does not exist, the function returns an error response with a status code of **404 Not Found**, indicating that the project was not located in the database.

This method is called from the routes file by creating a new route for it as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize)
  .get(controller.getOneHandler)
  .put(validate(updateTaskInput), controller.updateHandler)
  .delete(controller.deleteHandler);
```

Here, `/:id` will be a request parameter meant to capture the ID of the task that the user wants to delete.

REST API Delete Task

Request

URL : <http://127.0.0.1:3000/api/tasks/74f61799-7046-47d9-8f04-897f07b4e178>
Method: DELETE

Response

```
{
  "statusCode": 200,
  "status": "success"
}
```

In the case of an already deleted or not exist in the database:

```
{
  "statusCode": 404,
  "status": "error",
  "message": "Not Found"
}
```

We've established various modules such as **Role**, **User**, **Project**, and **Task**. Similarly, you can create a **Comment** module and build APIs for handling comments. Within this module, comments are associated with a **task_id**. Users can add comments related to tasks, update their own comments, and

delete their own comments. This aspect of the project is meant for your personal learning and experimentation.

Upload Supported Files

In both tasks and comments, the ability to attach files is provided through the file module. You can create a file module that includes CRUD (Create, Read, Update, Delete) operations, following a structure similar to that of the project and task modules. The file module should define database fields such as `file_id`, `file_url`, `file_name`, `created_by`, `created_at`, `updated_at`, and more. Additionally, within the supported files array, you can add or remove `file_id` based on the corresponding action taken.

For reference, the `file_entity.ts` is as follows:

```
import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn, UpdateDateColumn, JoinColumn, ManyToOne }
from 'typeorm';
import { Users } from '../users/users_entity';
import { Tasks } from '../tasks/tasks_entity';

@Entity()
export class Files {
  @PrimaryGeneratedColumn('uuid')
  file_id: string;

  @Column({ length: 30, nullable: false, unique: true })
  file_name: string;

  @Column({ length: 30 })
  mime_type: string;

  @Column()
  @ManyToOne(() => Users, (userData) => userData.user_id)
  @JoinColumn({ name: 'user_id' })
  created_by: string;

  @Column()
  @ManyToOne(() => Tasks, (taskData) => taskData.task_id)
  @JoinColumn({ name: 'task_id' })
  task_id: string;
```

```
  @Column()
  @CreateDateColumn()
  created_at: Date;

  @Column()
  @UpdateDateColumn()
  updated_at: Date;
}
```

To send files from your website to your Node.js server, we will use **Multer**. It's widely used for its ability to easily handle file uploads. Once a file is uploaded, **Multer** lets you save it in a specific folder on your server. Not only does it help with storing files, but it also helps you keep track of their locations, making it easier to access and manage these files later.

Multer integrates with Node.js smoothly and offers a straightforward way to upload files. The following steps will help to set up **Multer** for our project:

- **Install Multer**

First, make sure that you have Multer installed in your **Node.js** project. If you haven't already, you can install it using npm:

```
$ npm install multer uuid --save
$ npm install @types/multer @types/uuid --save-dev
```

- Create one folder in root directory as **attachedFiles** where all files will be uploaded. In **server_config.json**, add one parameter for path of uploaded files absolute path and change in **IServerConfig** interface as follows:

```
export interface IServerConfig {
  port: number;
  db_config: {
    'db': string;
    'username': string;
    'password': string;
    'host': string;
    'port': number;
    'dbname': string;
  };
  email_config: {
    'from': string;
```

```

    'user': string;
    'password': string;
  };
  front_app_url: string;
  default_user?: {
    email: string;
    password: string;
  };
  attached_files_path?: string;
}

```

- **Develop Multer Middleware**

In the utils directory, create a file named **multer.ts** for a middleware that uploads files to a specific folder with the following code:

```

import multer from 'multer';
import { Request } from 'express';
import { IServerConfig } from './config';
import * as config from '../../server_config.json';

// Define the options for Multer
export const multerConfig = {
  storage: multer.diskStorage({
    destination: (req, file, cb) => {
      // Set the destination folder where files will be saved
      const server_config: IServerConfig = config;
      cb(null, server_config.attached_files_path); // Change
      'attachedFiles' to your desired folder name
    },
    filename: (req, file, cb) => {
      // Generate a unique filename for the uploaded file
      const uniqueFileName = `${Date.now()}-${file.
      originalname}`;
      cb(null, uniqueFileName);
    }
  }),
  fileFilter: (req, file, cb) => {
    // Define the allowed file types (MIME types) here.
  }
}

```

```

const allowedMimeTypes = ['image/jpeg', 'image/png',
'application/pdf'];

// Check if the uploaded file's MIME type is in the
allowed list.
if (allowedMimeTypes.includes(file.mimetype)) {
  cb(null, true); // Accept the file
} else {
  cb(new Error('Invalid file type. Only PDF, JPEG, and
PNG files are allowed.'), false); // Reject the file
}
};

const upload = multer(multerConfig);
// Export the Multer middleware
export const fileUploadMiddleware = upload.single('file');
export const uploadFile = (req: Request) => {

  if (!req.file) {
    throw new Error('No file provided');
  }

  // Here you can perform additional processing and file
  storage logic
  // For example, save the file to a storage service or
  local directory
  const fileData = req.file;

  return fileData;
};

```

Here in the code, **attachedFiles** is the directory name where uploaded files are stored. You can change the name as you want. We allow only PDF and image file types, which are also changeable.

- **Handle File Uploads in Your Route**

In your file module creation API, use the Multer middleware you created to handle file uploads. Here's an example of how to use it in an Express route:

```
import { Express } from 'express';
```

```

import { FileController } from './files_controller';
import { fileUploadMiddleware } from '../../utils/multer';
import { authorize } from '../../utils/auth_util';
export class FileRoutes {

  private baseEndPoint = '/api/files';

  constructor(app: Express) {

    const controller = new FileController();
    app.route(this.baseEndPoint)
      .all(authorize)
      .post(fileUploadMiddleware, controller.addHandler);
  }
}

```

In the controller, the following code is used to store **fileData** in the database, which is called from the router:

```

import { BaseController } from
'../../utils/base_controller';
import { uploadFile } from '../../utils/multer';
import { Request, Response } from 'express';
import { FilesService } from './files_service';
import { Files } from './files_entity';

export class FileController extends BaseController {

  /**
   * Handles the addition of a new file.
   * @param {object} req - The request object.
   * @param {object} res - The response object.
   */
  public async addHandler(req: Request, res: Response): Promise<void> {

    try {
      const fileDataFromMulter = uploadFile(req);
      // Create an instance of the ProjectService
      const service = new FilesService();
      const fileData = new Files();
      fileData.file_name = fileDataFromMulter.filename;
    }
  }
}

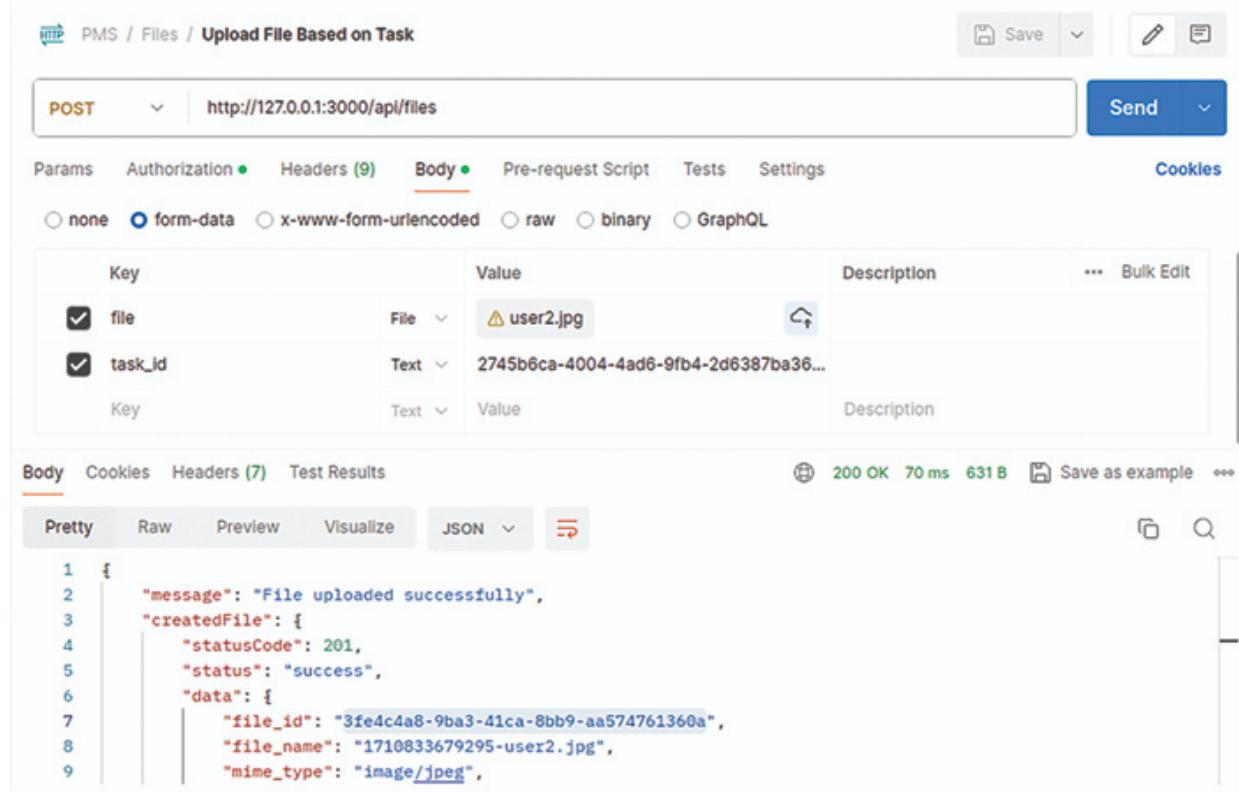
```

```
fileData.mime_type = fileDataFromMulter.mimetype;
fileData.created_by = req?.user?.user_id ?
  req?.user?.user_id :
  null;
const createdTask = await service.create(fileData);
res.status(201).json(createdTask);

res.status(200).json({ message: 'File uploaded
successfully', fileData });
} catch (error) {
  res.status(400).json({ error: error.message });
}
}

public async getAllHandler(req: Request, res: Response): Promise<void> { }
public async getOneHandler(req: Request, res: Response): Promise<void> { }
public async updateHandler(req: Request, res: Response): Promise<void> { }
public async deleteHandler(req: Request, res: Response): Promise<void> { }
}
```

From Postman, once you hit API, the following output will be displayed:



The screenshot shows the Postman interface for a POST request to `http://127.0.0.1:3000/api/files`. The 'Body' tab is selected, showing a 'form-data' structure with two fields: 'file' (selected) and 'task_id' (value: 2745b6ca-4004-4ad6-9fb4-2d6387ba36...). The response body is a JSON object:

```

1  {
2      "message": "File uploaded successfully",
3      "createdfile": {
4          "statusCode": 201,
5          "status": "success",
6          "data": {
7              "file_id": "3fe4c4a8-9ba3-41ca-8bb9-aa574761360a",
8              "file_name": "171083367925-user2.jpg",
9              "mime_type": "image/jpeg"
}

```

Figure 6.9: Upload File API

As per the code from the Multer middleware, the file is uploaded, and their data is stored in the database. Once the file data is stored, add that `file_id` in the task or comment module of the `supported_files` array.

- **Serve Uploaded Files**

To get uploaded file, we will create an API that will be `getOne` file, so update `getOneHandler` function in the controller with the following code:

```

public async getOneHandler(req: Request, res: Response): Promise<void> {
    try {
        const service = new FileService();
        const server_config: IServerConfig = config;
        const result = await service.findOne(req.params.id);
        const file_path =
` ${server_config.attached_files_path}/ ${result.data.fil
e_name}`;
        res.sendFile(file_path, (err) => {

```

```
if (err) {
  // Handle errors, such as file not found or
  permission issues
  console.error('Error sending file:', err);
  res.status(500).json({ error: err.message });
} else {
  res.status(200).end();
}
});
} catch (error) {
  res.status(400).json({ error: error.message });
}
}
```

This method called from the routes file as follows:

```
app.route(this.baseEndPoint + '/:id')
  .all(authorize)
  .get(controller.getOneHandler);
```

Once the API is triggered from Postman, the file associated with the directory will be downloaded directly, as follows:

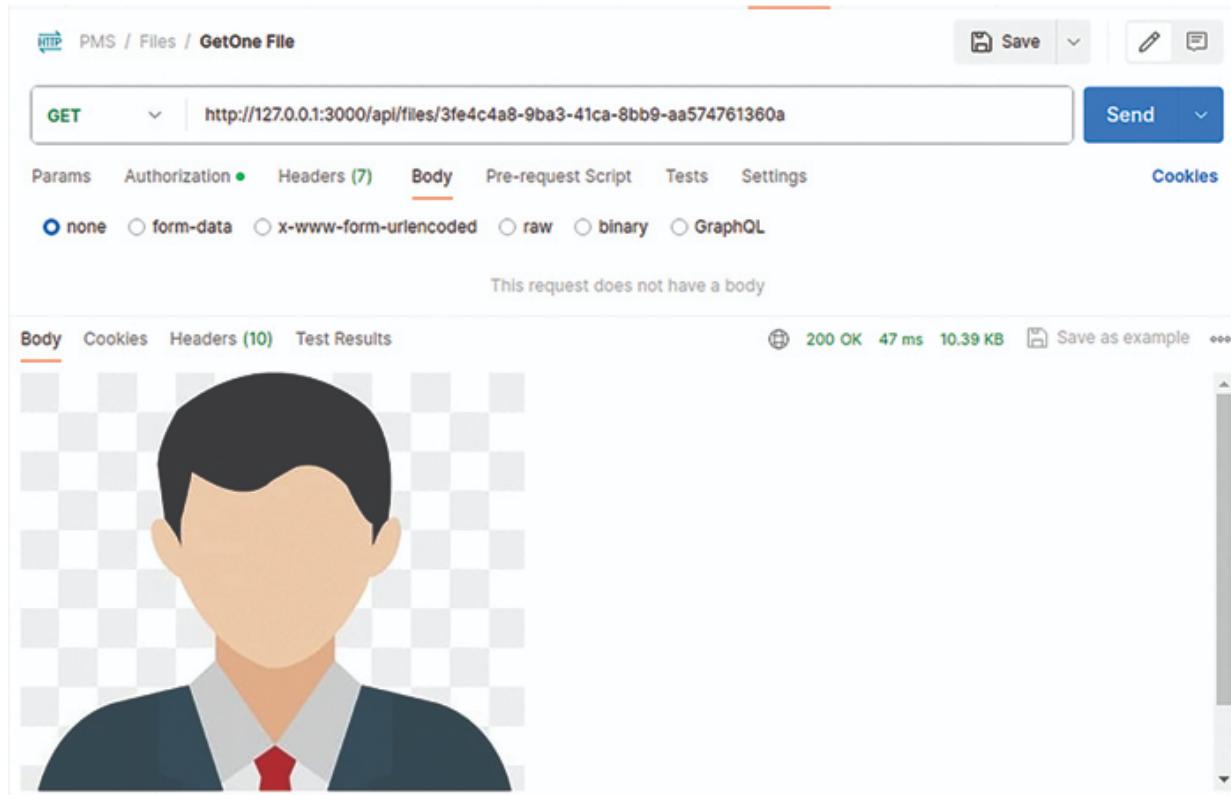


Figure 6.10: Get Uploaded File

By following these steps, you can use Multer to handle file uploads in your Node.js application, store files in a specific folder, and update file paths in your database.

Conclusion

In this chapter, we crafted REST APIs for both the project and task modules, incorporating robust input validation and facilitating comprehensive CRUD operations. We also delved into the realm of advanced query capabilities with TypeORM, skillfully selecting specific columns to optimize performance.

Moreover, we ventured into the territory of file uploads in Node.js, leveraging the powerful capabilities of the Multer package.

In the next chapter, we will learn about API Caching, which is helpful in improving your application performance.

Further Reading

<https://github.com/expressjs/multer>

OceanofPDF.com

CHAPTER 7

API Caching

Introduction

A user opens the browser and tries to access different parts of the application. This causes many API calls to the backend. Normally, with a low count of users, the response would be quick. However, when the data grows in the application and a significant number of users are accessing the data simultaneously, the response time increases. This may lead to poor user experience.

Usually, for a system, when the state of the data does not change frequently, for a given input, the output will mostly be the same unless something changes in data. Till the time, data does not change, the database query is not necessary to fetch the same data repetitively if you can keep a copy of the outcome somewhere. The process of storing this outcome and accessing it when needed is a fundamental concept of caching.

In this chapter, we will learn how to cache the data and use the cached data to serve APIs or save data of the queries so that we do not have to query the database for a certain time or till the time data is unchanged.

Structure

In this chapter, we will discuss the following topics:

- Understanding Caching
- Introduction to Redis
- Setting Up Redis server
- Pros and Cons of Redis/Caching
- Using Redis for Caching Data

Understanding Caching

Consider yourself trying to log in to the application. After entering a username and password, the application takes a couple of seconds to validate and then navigate to the homepage. At the homepage, there are many things: project list you are part of, team activities on the project tasks, your high-priority tasks list, and many more. Bringing data for each of them is going to take a good amount of time if the system is busy serving too many requests already.

All of these things on the homepage would require some data to be fetched from the database. The database queries are going to take time every single time you or other users open the application. These database query results can be *cached*. If cached, whenever the homepage is requested, the database queries would be avoided, and data will be read from the cache and sent in response. This process of caching-retrieving data would make responses faster and improve the user experience.

By definition, caching is a technique to store fetched data, or calculated results in *cache* so that any future request asking for that data can be served faster. When we need to access the data, the cache will be checked first to see if the desired data is cached or not. If yes, then serve the data from cache. Otherwise, fetch the data, process it if needed, and then store it in cache so that the next request can be served quicker.

Caching is a critical component when it comes to optimization of system performance. It helps to lower the response time or latency, and makes the system highly scalable. Cache can be considered a high-speed data storage system. There are many types of caching: memory cache, disk cache, browser cache, database cache, and so on. For our use-case, we will cache the data needed for API response, and database query results. For the caching, we will use a software called Redis.

Introduction to Redis

Quoting the content from Redis.io website:

“The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.”

In simple words, Redis can store various types of data. Most of them are simple Key-Value pairs. It supports various data structures such as strings,

hashes, lists, sets, sorted sets, bitmaps, and more. We can store these kinds of data in Redis and access via a key.

Being in-memory by nature makes Redis blazingly fast for read and write operations, thus, making it a popular choice for caching.

Apart from the core data structures which help us store various types of data, Redis also offers features including data replication, persistence, sharding, and transaction capabilities. Redis is adopted by developers for a variety of applications. Redis is used by GitHub, Twitter, snapchat, stackoverflow, and many more. Techstacks.io maintains a list of popular websites which utilize Redis for their use cases.

More on Redis can be learned at - <https://redis.io>.

Setting Up Redis Server

Let us start by setting up the Redis server. This section will cover the installation for MacOS, Ubuntu(debian), and Rocky Linux.

Installing Redis Server on Mac OS

The easiest way to install most of the softwares on a Mac is to use **homebrew**. If **homebrew** is not installed, it can be installed by running:

```
/bin/bash -c "$(curl -fSSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

This will make available a command named **brew** which we can use to install Redis server.

```
brew install redis
```

Once this completes, the server can be started with:

```
brew services start redis
```

To verify if Redis is installed or not, we can try running **redis-cli** which is a command line interface for accessing Redis.

```
redis-cli
```

If Redis was successfully installed, it will open a Redis prompt as:

```
127.0.0.1:6379>
```

If the prompt is visible, Redis is correctly installed and ready for use.

Further, we can do a command **ping** which should respond as **PONG**. If it does, then everything is working fine.

```
127.0.0.1:6379> ping  
PONG
```

Installing Redis Server on Ubuntu / Linux

To install the Redis on Ubuntu, we need **lsb-release**, **curl**, and **gpg**. If these are not available already, it can be installed using **apt**.

```
sudo apt install lsb-release curl gpg
```

Now, we need to add the repository where the **redis** binaries are available.

```
curl -fsSL https://packages.redis.io/gpg | sudo gpg --dearmor  
-o /usr/share/keyrings/redis-archive-keyring.gpg  
echo "deb [signed-by=/usr/share/keyrings/redis-archive-  
keyring.gpg] https://packages.redis.io/deb $(lsb_release -cs)  
main" | sudo tee /etc/apt/sources.list.d/redis.list
```

Once the repositories are added, we can install Redis:

```
sudo apt-get update  
sudo apt-get install redis
```

After this, server can be started using:

```
sudo systemctl start redis
```

Once the server is started, similar to Mac OS, we can verify using **redis-cli**.

Installing Redis Server on Rocky (RHEL-based)

We can simply use the **dnf** package manager to install Redis.

```
sudo dnf update -y  
sudo dnf install -y redis
```

Once installed, the service can be started as:

```
sudo systemctl status redis
```

Finally, we can verify using **redis-cli**.

Pros and Cons of Caching

Before we proceed to use Redis for our purpose, it is important to know about pros and cons that caching brings in. Caching is a crucial technique but it comes with certain advantages and disadvantages, which we should keep in mind while implementing it in the application.

Pros of Caching

Some of the major benefits of caching are as follows:

- **Performance improvement:** Caching significantly reduces the time needed to access data, which makes the responses to the end user faster. Thus, it provides better and faster application performance and an improved user experience.
- **Reduced load:** If implemented properly, most of the data is now served from cache which reduces the load on the application. This way, the application can serve more requests and operate more efficiently.
- **Cost effectiveness:** With caching in place, a single server can handle a larger number of requests efficiently. Consider if a server without caching was able to handle 100 requests per minute, and with caching, it can serve 1000 requests per minute. We do not need to put in more servers when the user load increases. Not only server costs, but the operational costs are also lowered due to decreased load and reduced bandwidth consumption.
- **Reduced network traffic:** The caching server can be installed locally to the application or on another server. If done locally, it can reduce the network traffic. However, it must be seen if both application and caching server can stay on the same machine, without making the resource consumption high, such as memory and CPU.

Cons of Caching

Some cons to be kept in mind and to be careful while doing development are:

- **Stale data:** It is important to ensure cache consistency. If the data is old, there should be effective cache invalidation strategies in place to remove old and invalid data from cache. Consider an example where you update your profile by updating your phone number. The application had already cached the user entity, which keeps your profile data in cache. When there is an update for the cached entity, the old data must be removed from cache and the new data must be placed in it.

It is very important to be careful while storing data in cache. At all points where the data might not be valid anymore, it should either be removed or replaced with the updated data. Sometimes, when data update is not always possible, the TTL kind of feature can be utilized. TTL stands for time to live. Usually, all caching systems provide this feature. This allows us to set a time for which the data should be in cache. Once the time expires, the data will be invalidated automatically.

- **Resource consumption:** If there is a lot of data in the application, it is important to decide what we are caching. Only the data which is needed most frequently should be cached. Otherwise, memory, disk space, and other resources can be a concern in a resource-constrained environment.
- **Cache Miss:** When we try to access some data and it is not found in cache, then it is called a Cache Miss. It is important to know how it is handled. If data is not found in cache, the application must revert to the original data source. Also, when the data is fetched from the original source, it should be placed in cache to serve future requests faster.

Handling a situation like Cache Miss is critical, as it may be expensive in terms of both time and resources. If there are too many requests in place and if there are too many Cache Miss, it will lead to an inefficient application.

- **Data synchronization issues:** For distributed environments, keeping the cached data and original data source as synchronous can be a challenge. It gets bigger if there are multiple caches.
- **Complexity:** Overall, implementing caching can add more complexity to the system. It may also lead to additional development efforts as

well as testing and maintenance efforts.

Using Redis for Caching

In our project management system, so far, we have implemented several modules: users, roles, projects, and tasks. Consider this application being in use by an organization of 100,000 users. At the start of the office, typically 9 am, people are going to login and access their projects to learn about their tasks and make progress. If we cache a few things, the response time would improve and the user experience would be better.

There are many strategies to caching such as on demand caching, and proactive caching. Proactive caching would be when we cache something at the application start without any user requests. This caching is done anticipating future requests. On the other hand, whenever something is accessed and it is a cache miss situation, caching at this moment would be on-demand. In case of on-demand caching, initially there would not be any records, and an object would be cached only after a cache miss.

In our application, we could do a mix of both strategies. When the application starts, we can cache all users, roles, projects, and their tasks. There could be some specific cases when the data we want to cache is not a straightforward database query result, for example, if you want a count of projects and similarly counts of tasks in all projects, respectively. These values would be results of some function which we can cache. In this section, we will see how we can cache both types of data we are interested in caching.

Overall, we want to store the data as:

- **Objects:** essentially JSON objects of projects, tasks, users, and so on.
- **Numbers:** counts of projects, tasks, users, and so on.

There can be more.

Updating Project Dependencies

First of all, we need to update the project dependencies, that is, node modules in our case. The only package that we need is Redis. Let us install Redis using `npm install`:

```
npm install redis
```

It is better to install type definitions for Redis as well so that while doing development we get proper code hints.

```
npm install --save-dev @types/redis
```

After Redis package is available, we can use it as following example:

```
import { createClient, RedisClient } from 'redis';
const client: RedisClient = createClient();
client.on('connect', () => {
  console.log('Connected to Redis');
});
client.set('key', 'value', (err, reply) => {
  if (err) throw err;
  console.log(reply); // OK
});
client.get('key', (err, reply) => {
  if (err) throw err;
  console.log(reply); // value
});
client.quit();
```

In the preceding example, we are importing the `createClient` function and `RedisClient` for type. Next, a client is created using `createClient()` function call and using `client.on()` function we try to connect to the Redis server.

In the example discussed, we used callbacks, but we could also use `async-
await`.

Using `client.set()`, we can set a key-value pair. This will store the key-value to the Redis server acting as a caching server here. When needed, we can get the value of the key using `client.get()`.

Here, we are storing a simple string '`value`', and if we want to store an object, we will have to convert the object to string using `JSON.stringify()` and store it as string. All of this is because Redis does not support JSON objects as values straightforwardly. To make Redis store JSON objects, we need to set up a Redis Module named `RedisJSON`. This module provides native JSON capabilities to Redis.

The **RedisJSON** module is available at GitHub:
<https://github.com/RedisJSON/RedisJSON>.

We need to either clone this repository or download the repository zip file. Before continuing further, ensure that Rust is installed.

If not available, rust can be installed using:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

After this step completes, we can verify the installation by checking the rust version.

```
rustc --version
```

Once installed, we can begin building the **RedisJSON**. Navigate to the downloaded repository and run the following command:

```
cargo build -release
```

This will create the module in **target/release/librejson.so**. If the file is missing, then something went wrong and try building again.

We need to modify the Redis configuration typically located at **/etc/redis/redis.conf** or **/usr/local/etc/redis.conf** and add enable to the **redisjson.so** module.

On linux

```
loadmodule /path/to/redisjson.so
```

On Mac

```
loadmodule /path/to/rejson.dylib
```

Once the module is loaded, restart Redis to enable it.

Cache Utility

Since we're dealing with a large number of entities to cache, using a cache utility class can be beneficial. This class would ideally provide functions for both setting and retrieving cached values.

Once we have the cache utility class, we can simply import it where we need and use the functions directly. Let us create a file **cache_util.ts** which would have the following code:

```
// cache_util.ts
import * as redis from 'redis';
```

```

export class CacheUtil {

  // redis client instance
  private static client = redis.createClient();

  constructor() {
    CacheUtil.client.connect();
  }

  public static async get(cacheName: string, key: string) {
    try {
      const data = await
        CacheUtil.client.json.get(` ${cacheName}:${key}`);
      return data;
    } catch (err) {
      console.error(`Error getting cache: ${err}`);
      return null;
    }
  }

  public static async set(cacheName: string, key: string,
  value) {
    try {
      await CacheUtil.client.json.set(` ${cacheName}:${key}`, '.', value);
    } catch (err) {
      console.error(`Error setting cache: ${err}`);
    }
  }

  public static async remove(cacheName: string, key: string) {
    try {
      await CacheUtil.client.del(` ${cacheName}:${key}`);
    } catch (err) {
      console.error(`Error deleting cache: ${err}`);
    }
  }
}

```

The function **set()** can be used to set a key-value in cache and the **get()** function can be used to retrieve the values. Notice how we are using

`client.json.get()` and `client.json.set()` to actually get and set the values. Both set and get functions are made static so that we can simply use those without initializing the class every time.

We can now initialize the class once and use it everywhere. We need to do so in order to connect the Redis client to the server. Without making a call to `CacheUtil.client.connect()` the application will throw an error as: The client is closed.

Now, when the cache utility is ready, let us initialize it in `main.ts`.

```
// main.ts
// initialise the cache utility
new CacheUtil();
```

This will make a call to the constructor and connect the client to the Redis server.

Caching Entities

We previously discussed two entity caching approaches: on-demand and proactive. For on demand caching, we can update all of our controllers' functions to first check if the data needed is in the cache or not. The following example shows how the `getOneHandler` of `users_controller` which is responsible for returning a user for a given user id:

```
// users_controller.ts
public async getOneHandler(req: Request, res: Response): Promise<void> {
  if (!hasPermission(req.user.rights, 'get_details_user')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }

  // check user is in cache
  const userFromCache = await CacheUtil.get('User',
  req.params.id);
  if (userFromCache) {
    res.status(200).json({ statusCode: 200, status: 'success',
    data: userFromCache });
  }
}
```

```

        return;
    } else {
        // get user from db
        const service = new UsersService();
        const result = await service.findOne(req.params.id);
        if (result.statusCode === 200) {
            delete result.data.password;
            // set user in cache
            CacheUtil.set('User', req.params.id, result.data);
        }
        res.status(result.statusCode).json(result);
        return;
    }
}

```

In the function, we are making a call to **CacheUtil.get()** for cacheName '**User**' and the key is the **user_id** which is supplied in the request parameters. There is a chance that the requested user is not present in the cache so it must be checked if the returned value **userFromCache** is null or not. If it is null, then we should take the regular course and fetch the user from the database. If a user is present in the database, then we should also save that to the cache. The following line making a call to **CacheUtil.set()** is doing that for us:

```
CacheUtil.set('User', req.params.id, result.data);
```

This way, all of the functions can be updated to check cache before making an actual database query. For cases when delete api is called, the value in cache must also be removed. If the value is not removed, future API calls to get the user by **user_id** will return a value which does not exist in the database anymore.

The **delete** function can be updated as:

```
public async deleteHandler(req: Request, res: Response): Promise<void> {
    if (!hasPermission(req.user.rights, 'delete_user')) {
        res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
        return;
    }
}
```

```

const service = new UsersService();
const result = await service.delete(req.params.id);

// remove user from cache
CacheUtil.remove('User', req.params.id);

res.status(result.statusCode).json(result);
return;
}

```

A call to **CacheUtil.remove()** will remove the user from cache.

Caching using this on-demand approach ensures the frequently used data stays in cache and the data which is not used does not fill up the cache. However, in this case, there will always be a cache miss when the data is requested for the first time.

[Building Cache at Startup](#)

Sometimes, it is good to fill the cache when the application starts. This may help in preventing some cache miss since the data requested will be there in the cache. Let us update **UsersUtil** to add a function **putAllUsersInCache()** which will fetch all of the users from the database and put them in the cache.

```

// function to put all users in cache
public static async putAllUsersInCache() {
  const userService = new UsersService();
  const result = await userService.findAll({});
  if (result.statusCode === 200) {
    const users = result.data;
    users.forEach(i => {
      CacheUtil.set('User', i.user_id, i);
    });
    console.log(`All users are put in cache`);
  }else{
    console.log(`Error while putAllUsersInCache() =>
    ${result.message}`);
    console.log(result);
  }
}

```

```
}
```

This function is making a call to **findAll()** using **userService** to get all of the users from the database and if the result of the query is a success, putting the users in the cache using **forEach**.

We can call this function from **main.ts**.

```
// Proactive cache update
setTimeout(() => {
  UsersUtil.putAllUsersInCache();
}, 1000 * 10 );
```

When an application starts, it may require time to establish connections to databases, caching servers, and perform other initializations. It is wise to allow sufficient time for these connections and initializations to complete before commencing other operations to ensure smooth functioning of the application. Hence, we have put some delay (10 seconds here) using **setTimeout**.

Similar to **UsersUtil**, other entity utils can be modified to add a function and then we can make a call to those functions from **main.ts**. It is not necessary that those functions be called from the **main.ts** file. We can put those elsewhere, for example, **CacheUtil** and call **CacheUtil** from **main.ts** to **init** the caching. Likewise, it can be another function than **putAllXXToCache**.

Consideration when Using Redis

Despite being a powerful in-memory data store, there are some challenges when it comes to using Redis. The following points discusses some of the challenges:

- **Memory Limitation**

Since it is an in-memory data store, it offers faster access but is limited by the system capacity. If you have a large dataset and the target is to minimize the cost then it could be a drawback.

- **Data Security**

Redis, by default, is not encrypted. Although, it supports a simple password based authentication but does not have built-in support for

SSL/TLS encryption. Securing data for REST APIs requires additional tools.

- **Limited ways to Query data**

Redis querying capabilities are limited compared to a traditional database system. However, by implementing some additional logic at application level, complex queries can be performed.

- **Single-threaded nature**

Redis servers are single-threaded in nature. Most of the machines these days are multi-core and Redis cannot utilize more than one core. However, newer Redis versions put slow queries to separate threads but the requests to Redis servers are still handled by a single thread only.

- **Hosting Cost on Cloud**

All major cloud platforms offer Redis as a service. AWS **Elasticache**, Azure Cache for Redis are some examples. These offerings can be expensive for large datasets.

If Redis does not perform well for your use case, other options such as Memcached, Apache Kafka, and others can also be explored.

Conclusion

Caching can improve the user experience and greatly enhance the performance of an application while also making it more stable and scalable. In this chapter, we got familiar with the concepts of caching along with setting up Redis, a popular choice for caching for any size of the application. We learned and implemented the two caching strategies: on demand and proactive caching.

In the next chapter, we will implement the notification module while learning another aspect of Redis: Message Queue.

Multiple Choice Questions

1. What is the primary goal of implementing caching in an application?
 - a. To increase data processing time.

- b. To store user passwords securely.
 - c. To reduce the time needed to access data.
 - d. To increase network traffic.
2. What is a significant benefit of caching for handling database queries?
- a. It increases the load on the application server.
 - b. It helps in avoiding database queries by fetching data from the cache.
 - c. It decreases the application's performance.
 - d. It consumes more bandwidth.
3. Which of the following is not a pro of caching?
- a. Performance improvement.
 - b. Reduced load on the application.
 - c. Increased network traffic.
 - d. Cost effectiveness.
4. What does TTL stand for in the context of caching?
- a. Time To Launch
 - b. Time To Live
 - c. Total Time Limit
 - d. Time To Load
5. What is a '*cache miss*'?
- a. When data is successfully retrieved from the cache.
 - b. When data is not found in the cache.
 - c. When the cache is fully utilized.
 - d. When the cache fails to save data.
6. What is a key challenge in caching for distributed environments?
- a. Simplifying user interfaces.
 - b. Reducing the number of users.
 - c. Keeping cached data and original data source synchronized.

- d. Decreasing the server costs.
7. What is an advantage of building cache at startup?
- a. It increases the cache size unnecessarily.
 - b. It ensures frequently requested data is immediately available.
 - c. It slows down the application startup.
 - d. It requires less development effort.
8. Which software is used for caching in the described scenario?
- a. MySQL
 - b. Redis
 - c. MongoDB
 - d. Oracle

Answers

- 1. c
- 2. b
- 3. c
- 4. b
- 5. b
- 6. c
- 7. b
- 8. b

Further Readings

<https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/>

<https://redis.io>

<https://redis.io/docs/data-types/json/>

<https://www.npmjs.com/package/redis>

OceanofPDF.com

CHAPTER 8

Notification Module

Introduction

Communication is the heart of a process which involves teamwork.

Whenever any activity happens in the project, a notification is sent to the respective users. A developer gets a notification of a task being assigned. A tester gets a notification when the task is moved to testing. A notification is sent to all users watching a particular task whenever a new comment is added to it.

Notifications can be of several types. Email, SMS, and inside user-interface are the most common. This chapter will continue our development path to add a notification module.

Structure

In this chapter, we will discuss the following topics:

- Understanding Notification Module
- Implementing Queue
- Notifying about the New Task

Understanding Notification Module

Notifications serve as a critical element throughout the lifecycle of the project, informing the respective stakeholders about the ongoing changes and activities. When a new task is assigned to a developer, an email notification can alert with highlights of the task, ensuring that the developer is informed and updated timely. A QA team member can receive an email about a task being moved to the testing phase.

A notification plays an important role in a project's progress. A notification that is received timely, speeds up the communication among the teams.

The medium of notification can depend on many factors, such as urgency, criticality of communication, nature of the project, and so on.

Typically, there are three types of notifications: Email, SMS, and in-app or inside the user interface notification (in case of websites). In this chapter, we will focus on Emails but the process will also lay down the foundation for implementation of other types of notifications.

Let us start the implementation by creating a new file in **src/util** as **notification_util.ts**. This file would contain a class which will hold utility methods to send notifications.

```
// Path: src/utils/notification_util.ts
export class NotificationUtil {
```

This class will be our single point from where all types of notifications can be sent. From anywhere in the code, we can simply make a call to this class. Let us add the functionality for sending an email. We already implemented a function in the **email_util.ts** class. That function can act as a reference.

For sending an email, we will use a node package, nodemailer.

```
import * as nodemailer from 'nodemailer';
```

Let us add the constructor to the util **NotificationUtil** as:

```
// Path: src/utils/notification_util.ts
import * as nodemailer from 'nodemailer';
export class NotificationUtil {
  // nodemailer transporter instance
  private static transporter;

  constructor(config) {
    if (!config) {
      throw new Error('Config not provided');
    }
    if (!NotificationUtil.transporter) {
      NotificationUtil.transporter = nodemailer.createTransport({
        service: 'gmail',
        auth: {
          user: config.email_config.user,
          pass: config.email_config.password
      })
    }
  }
}
```

```
        }
    });
}
}
}
```

In the preceding code, we have a constructor which takes a config object as the only argument. We will use this config to retrieve the necessary config for sending an email such as SMTP server username, password, and so on.

We have created a transporter object as a private static member of the class. In the context of Node.js and **nodemailer**, a transporter is an object which encapsulates the email sending functionality. Simply put, it is a way to send email using Node.js in which we do not have to worry about the low level details of the process. A transporter object is usually created once and reused. Hence, we have the object created at class level. In the constructor, we first check if this is already initialized or not. If the object is not initialized, we do it.

In our case, we are using gmail as a server but other providers and generic SMTP servers can also be used. To know more about the **nodemailer**, visit the website <https://nodemailer.com/>.

Further, we can add our function to send emails. For sending an email, we need the sender email address, recipient email address, subject, email body. Since the sender's email address is unlikely to be changed for every email, let us add that as a private static variable of the class and we can set that inside the constructor.

```
private static from: string;
```

In the constructor, we can add the following line after creating the transporter :

```
NotificationUtil.from = config.email_config.from;
```

Let us define our function to receive these values as arguments.

```
public async sendEmail(to: string, subject: string, body: string) {
    try {
        const mailOptions = {
            from: NotificationUtil.from,
            to: to,
```

```

        subject: subject,
        html: body
    };

    const status = await
NotificationUtil.transporter.sendMail(mailOptions);
    if (status?.messageId) {
        return status.messageId;
    } else {
        return false;
    }
} catch (error) {
    console.log(`Error while sendEmail => ${error.message}`);
    return false;
}
}

```

For the `sendEmail` function, we receive just what we need — `to` (recipient email address), `subject`, and `body` of the email. We create an object `mailOptions` using these four values and finally send email using `transporter.sendEmail()`. If this function successfully sends an email, we will receive a message Id which we can return, otherwise, we will return a `false` boolean value indicating that something did not go right.

This completes our email sending feature with the help of `NotificationUtil`. We can now use this function instead of the `sendMail` function from `email_util.ts`.

We are using the `sendMail` function from `email_util` for the `forgotPassword` function in `UserController`. Let us change that to use `NotificationUtil` by importing the `util` in `UserController`.

```
import { NotificationUtil } from
'../../utils/notification_util';
```

Now, we can modify the `forgotPassword` function to use notification `util`. We need to replace the following line:

```
const emailStatus = await sendMail(mailOptions.to,
mailOptions.subject, mailOptions.html);
```

The line replacing would be as follows:

```
const emailStatus = await
NotificationUtil.sendEmail(mailOptions.to,
mailOptions.subject, mailOptions.html);
```

Instead of **sendMail**, we are using **NotificationUtil.sendEmail** function. Rest everything remains the same.

We can also remove the unused **import**

```
import { sendMail } from '../utils/email_util';
```

We also need to initialize the **NotificationUtil** from our **main.ts** file.

```
new NotificationUtil(config);
```

The config can be imported as:

```
import * as config from '../server_config.json';
```

Implementing Queue

In a large organization there would be a good number of people using project management software. From each user, there would be tons of activities and some of those activities would require an email notification to be sent to other users. If there are a lot of emails to be sent, it is better to handle the communication using a queue. A queue is a mechanism where each call to send email would be added and the queue will be processed separately along with a failure mechanism.

Using a queue in this manner is a good approach for the following reasons:

- **Improved performance and efficiency:** Sending an email directly can be resource-intensive and may slow down the primary operation. A queue would allow decoupled handling of the email sending process and the response times would be faster.
- **Scalability:** As the number of users grows, activities and number of emails to be sent would grow significantly. A queue can handle the increased load. There can be different strategies to handle a queue, for example, separate notification server, separate worker, and so on.
- **Reliability:** If an email fails to send, we can easily retry after sometime, if this is carried out through a queue. The failure can be due to many reasons such as server issues and network problems. The

queue can be designed to handle failures and be equipped with a mechanism to retry.

- **Asynchronous processing:** Queues can be made asynchronous so that the application does not have to wait while email is being sent and we receive a response from the email server.

These are some of the few key reasons why it is a good idea to use queues for sending emails.

There can be incidents when we need immediate response to an email sending call and without that we cannot ensure reliability. One such example is the **forgotPassword** function which we implemented. In this case, as soon as the user provides their email, we verify things at the backend and send an email. The user would expect an email immediately in his/her mailbox.

In such cases, we can also use hybrid methods for email notifications. In our application, we will also do the same thing. For actions needing immediate email, we will use the **sendEmail** function from **NotificationUtil** and for the actions which can wait, such as notifying users about a new comment on the task, or when a new task is created, or when a task is moved in the workflow (for example, **Backlog** → **ToDo** → **Dev-Complete** → **Ready-to-test** → **Closed**), and so on, we will use the queue.

Let us move to implement the queue in our **NotificationUtil**. For the queue, we need to store the queued objects somewhere outside of the application so that if for any reason the application fails, we do not lose the objects in the queue. For this purpose, we can use Redis.

Using Redis for Queue

Redis is a powerful in-memory data store that supports necessary data structures and features to implement a basic queue system. The following code snippet shows how a queue can be implemented and used:

```
const redis = require('redis');
const client = redis.createClient();

// Adding a job to the queue
client.lpush('emailQueue', JSON.stringify({
```

```

from: 'pms-support@pms.com',
to: 'pmsbook2023@gmail.com',
subject: 'Welcome to PMS',
text: 'Welcome to PMS. We are happy to have you on board.'
});

// Processing jobs from the queue
const processJob = () => {
  client.brpop('emailQueue', 0, (err, reply) => {
    if (err) {
      // Handle error
    } else {
      const job = JSON.parse(reply[1]);
      // Process job
      console.log('Processing job:', job);
      // Continue processing next job
      processJob();
    }
  });
};

// Start processing
processJob();

```

In this code, we created a client using `redis.createClient()` at first. This client can be used to push jobs to the queue using `lpush`. This call will maintain the records for our processing later. While processing the jobs, we can fetch the items from the same queue using `brpop` and process. This can be used in any functionality and not just for sending email. The `lpush` and `brpop` functions are specific to Redis.

In the preceding code, there is a comment added for handling the error. There can be many ways to handle the error while processing the queue. Once we use `client.brpop` it will remove the message from the queue and make it available for processing. If processing of the removed message was a failure, it must be handled in a proper manner.

One way to handle this situation can be to retry the processing again after some time. In this case, we need to save the failed message somewhere so that it can be processed later. Another way can be to simply add a log using

console.log and notify the respective stakeholder. If there is an alarm system in place an alert can also be raised.

The approach is simple, we queue something using **lpush** and retrieve using **brpop** for processing. This implementation helps us to process emails but lacks error handling, retries, maintenance, and much more of what a sophisticated node package such as Bull can provide.

We will use the Bull node package for our implementation. Bull is a popular Node.js library used for handling background jobs and job queues. Bull is built on top of Redis.

Some of the key features of Bull are robustness, job scheduling, concurrency control, retry mechanism, event driven, rate limiting, persistence using redis, and so on. Further information about the package is available at <https://optimalbits.github.io/bull/> .

For our use case, where we want to queue emails, process the queue, handle failures, and improve efficiency, Bull is an excellent choice. Let us start with the implementation by installing and then adding Bull to our application.

```
npm install bull
```

After installing Bull, we can import it in the **NotificationUtil**.

```
import Queue from 'bull';
```

We need to create a **queue** for emails:

```
private static emailQueue = new Queue('emailQueue',  
'redis://127.0.0.1:6379');
```

We also need to add a function which can be called from other parts of the application to enqueue an email job.

```
// Function to enqueue email tasks  
public static async enqueueEmail(to: string, subject: string,  
body: string) {  
  // Enqueue the email task  
  await NotificationUtil.emailQueue.add({  
    to,  
    subject,  
    body  
  });
```

```
}
```

This is all for enqueueing an email. We just need to use the `emailQueue.add()` function to add the object which contains necessary information we need while sending the email.

We need to add logic to process the queue asynchronously. For smaller applications the logic to handle the queue can be part of the application itself. However, when applications grow bigger it is better to put the queue handling as a separate **worker/process**.

Let us add a new file for queue workers as `queue_worker.ts` in a new directory `workers`. File path would be `src/workers/queue_worker.ts`.

```
import Queue from 'bull';
import { NotificationUtil } from '../utils/notification_util';
export class QueueWorker {
  private static emailQueue = new Queue('emailQueue',
  'redis://127.0.0.1:6379');
  constructor() {
    console.log('Initializing QueueWorker');
  }
  public beginProcessing() {
    QueueWorker.emailQueue.process(async (job) => {
      try {
        const { to, subject, body } = job.data;
        const responseEmail = await
          NotificationUtil.sendEmail(to, subject, body);
        if (!responseEmail) {
          // handle error
        }
        console.log(`Email sent to ${to}`);
      } catch (error) {
        // handle error
        console.error(`Failed to send email: ${error.message}`);
      }
    });
  }
}
```

The preceding class defines a function to begin processing of the `emailQueue` defined as a static member of the class. In the class, while processing jobs to send email we just make a call to `NotificationUtil.sendEmail()` function.

Handling Failures

It can happen that sometimes emails are not sent and we need to retry. The `QueueWorker` should be capable of handling such incidents. Let us modify the `beginProcessing` function to add support for failed jobs.

```
public beginProcessing() {
    QueueWorker.emailQueue.process(async (job) => {
        // existing logic ..
    });

    QueueWorker.emailQueue.on('failed', async (job, err) => {
        // Retry the job
        console.log(`Retrying job for ${job.data.to}`);
        await job.retry();
    });
}
```

The `QueueWorker.emailQueue.on` function call with '`failed`' status gets executed whenever there is a failed job. In this case, we can retry the job using `job.retry()`.

There can be cases, when a failed job fails many more times. Basically, it is never going to succeed. Such cases should be handled with a check on how many attempts were made to run the same task. To fix this, we first need to define a max attempt count.

```
private static MAX_ATTEMPTS = 4;
```

Now, we need to modify the function again to check if the number of attempts made by a job exceeds the `MAX_ATTEMPTS` or not.

```
QueueWorker.emailQueue.on('failed', async (job, err) => {
    if (job.attemptsMade >= QueueWorker.MAX_ATTEMPTS) {
        // Handle the final failure
        console.error(`Job permanently failed for ${job.data.to}:
${err.message}`);
    }
});
```

```

} else {
  // Retry the job
  console.log(`Retrying job for ${job.data.to}`);
  await job.retry();
}
});

```

This will retry the failed job for four times and then exit if it does not succeed in four attempts. We should handle the case when the job finally fails. This can be done by some other kind of notification, proper reporting in logs, or recording the incident in a persisted mode, for example, in a database table.

Notifying About the New Task

Till this point, we have a basic implementation of the queue for email notifications. We can now use it for other parts of the application. As an example, let us try to add notifications for all members of a project whenever there is a new task created.

Let us first add the following function in **ProjectsUtil**:

```

public static async getProjectByProjectId(project_id: string)
{
  const projectService = new ProjectsService();
  const project = await projectService.findOne(project_id);
  return project.data;
}

```

To send a notification, we need to first get the users of the project and we can get it from the project object. After this, let us update the **addHandler()** in **TaskController** inside **task_controller.ts** file.

```

public async addHandler(req: Request, res: Response): Promise<void> {

  if (!hasPermission(req?.user?.rights, 'add_task')) {
    res.status(403).json({ statusCode: 403, status: 'error',
    message: 'Unauthorised' });
    return;
  }
}

```

```
try {
  // Create an instance of the TaskService
  const service = new TasksService();

  // Extract task data from the request body
  const task = req.body;

  // Get the project
  const project = await
ProjectsUtil.getProjectByProjectId(task.project_id);

  //check if the provided project_id is valid
  const isValidProject = project ? true : false;
  if (!isValidProject) {
    // If user_ids are invalid, send an error response
    res.status(400).json({ statusCode: 400, status: 'error',
      message: 'Invalid project_id' });
    return;
  }

  // Check if the provided user_id is valid
  const isValidUser = await
UsersUtil.checkValidUserIds([task.user_id]);

  if (!isValidUser) {
    // If user_ids are invalid, send an error response
    res.status(400).json({ statusCode: 400, status: 'error',
      message: 'Invalid user_id' });
    return;
  }

  // If user_ids are valid, create the task
  const createdTask = await service.create(task);
  res.status(201).json(createdTask);

  // task is created, now send email to the user
  const userIds = project.user_ids;

  // for each user_id, enqueue an email task
  for (const userId of userIds) {
    const user = await UsersUtil.getUserById(userId);
```

```

    if (user) {
      await NotificationUtil.enqueueEmail(
        user.email,
        'New Task Created',
        `A new task has been created with the title
        ${task.title} and description ${task.description}`);
    }
  }

} catch (error) {
  // Handle errors and send an appropriate response
  console.error(`Error while addUser => ${error.message}`);
  res.status(500).json({ statusCode: 500, status: 'error',
  message: 'Internal server error' });
}
}

```

The function we need at `UserUtil` to get user by `user_id` is -

```

public static async getUserById(user_id: string) {
  const userService = new UsersService();

  const queryResult = await userService.findOne(user_id);
  if (queryResult.statusCode === 200) {
    const user = queryResult.data;
    return user;
  }
  return null;
}

```

The logic of sending email can also be moved to a separate function. We can add a **utility** class for Tasks inside `tasks_controller.ts` similar to projects and users.

```

export class TaskUtil {

  // Notify the users of the project that a change
  public static async notifyUsers(project, task) {
    if (project) {
      const userIds = project.user_ids;
      for (const userId of userIds) {

```

```

        const user = await UsersUtil.getUserById(userId);
        if (user) {
            await NotificationUtil.enqueueEmail(
                user.email,
                'New Task Created',
                `A new task has been created with the title
                ${task.title} and description ${task.description}`
            );
        }
    }
}
}
}

```

The preceding function takes a project and task object and notifies all users. We can make a call to this function from the **addHandler**. The updated **addHandler()** function would be:

```

public async addHandler(req: Request, res: Response): Promise<void> {

    if (!hasPermission(req?.user?.rights, 'add_task')) {
        res.status(403).json({ statusCode: 403, status: 'error',
        message: 'Unauthorised' });
        return;
    }

    try {
        // Create an instance of the TaskService
        const service = new TasksService();

        // Extract task data from the request body
        const task = req.body;

        // Get the project
        const project = await
        ProjectsUtil.getProjectByProjectId(task.project_id);

        //check if the provided project_id is valid
        const isValidProject = project ? true : false;
        if (!isValidProject) {

```

```

    // If user_ids are invalid, send an error response
    res.status(400).json({ statusCode: 400, status: 'error',
    message: 'Invalid project_id' });
    return;
}

// Check if the provided user_id is valid
const isValidUser = await
UsersUtil.checkValidUserIds([task.user_id]);

if (!isValidUser) {
    // If user_ids are invalid, send an error response
    res.status(400).json({ statusCode: 400, status: 'error',
    message: 'Invalid user_id' });
    return;
}

// If user_ids are valid, create the task
const createdTask = await service.create(task);
res.status(201).json(createdTask);

// Notify the users of the project that a new task has been
created
await TaskUtil.notifyUsers(project, task);

} catch (error) {
    // Handle errors and send an appropriate response
    console.error(`Error while addUser => ${error.message}`);
    res.status(500).json({ statusCode: 500, status: 'error',
    message: 'Internal server error' });
}
}

```

We can re-use the same function for update and delete handlers by a simple modification. We can pass an argument ``action`` which takes one value out of `add`, `update`, `delete`. Based on this action, we can modify the subject and content of the email. Following is the updated function:

```

public static async notifyUsers(project, task, action) {
    if (project) {
        const userIds = project.user_ids;

```

```

let subject = '';
let body = '';
if (action === 'add') {
  subject = 'New Task Created';
  body = `A new task has been created with the title
  ${task.title} and description ${task.description}`;
} else if (action === 'update') {
  subject = 'Task Updated';
  body = `A task has been updated with the title
  ${task.title} and description ${task.description}`;
} else if (action === 'delete') {
  subject = 'Task Deleted';
  body = `A task has been deleted with the title
  ${task.title} and description ${task.description}`;
}
for (const userId of userIds) {
  const user = await UsersUtil.getUserById(userId);
  if (user) {
    await NotificationUtil.enqueueEmail(
      user.email,
      subject,
      body
    );
  }
}
}
}

```

Here, we are making the subject and body of the email based on the action provided. We also need to modify the call accordingly:

```
// Notify the users of the project that a new task has been
created
```

```
await TaskUtil.notifyUsers(project, task, 'add');
```

Similar to **addHandler**, we can make call from **updateHandler()** and **deleteHandler()** **deleteHandler()** as:

```
await TaskUtil.notifyUsers(project, task, 'update');
await TaskUtil.notifyUsers(project, task, 'delete');
```

Considerations while Implementing Queues

There is a set of challenges to consider while implementing a queue. Let us discuss a few of those here:

- **Scalability**

When it comes to handling high volumes of messages, ensuring the smooth handling can be challenging. In such cases, using multiple queues (spread across different machines) to balance the load could be helpful.

- **Latency and Throughput**

Throughput is the number of messages processed in a given timeframe. When there is a high volume of messages to process and the application flow is critical it becomes vital to ensure that the latency is within permissible limits. Higher latencies can be a bottleneck in real-time applications. To tackle such situations requires optimizations at application, queue and network level to achieve desirable throughput.

- **Fault Tolerance and Recovery**

There can be cases when due to an issue in the application the queue stops processing the messages. The messages must still be processed after the application has recovered from error. There should be a mechanism so that the messages in queue (waiting to be processed) are persisted or kept safe.

There are other considerations, for example, ordering and consistency of delivery of messages, security, monitoring while implementing the queue system. With help of careful design of the application these problems can be avoided, or in worst case, mitigated.

Conclusion

This chapter introduced a new concept 'Queue' which is very important to handle processing of various types of data. In our case, we used it for email notifications. The **NotificationUtil** and **QueueWorker** together implemented the queue mechanism with the help of the **Bull**, a **Node.js** library built on top of Redis.

In the next chapter, we will learn how an application can be built for production and be deployed on real servers. We will also learn to obfuscate the code so that, if it falls in wrong hands, it will not reveal everything and make the job harder for an outsider to crack.

Multiple Choice Questions

1. What is the primary purpose of notifications in a project lifecycle?
 - a. To provide entertainment.
 - b. To inform stakeholders about changes and activities.
 - c. To gather feedback.
 - d. To schedule meetings.
2. How does a queue contribute to the scalability of email sending in a project management system?
 - a. By limiting the number of users.
 - b. By reducing the number of emails sent.
 - c. By handling increased load effectively.
 - d. By sending all emails immediately.
3. What is a key advantage of a queue in terms of reliability for email notifications?
 - a. It guarantees email delivery on the first attempt.
 - b. It simplifies email content.
 - c. It allows for retrying failed email sends.
 - d. It uses less server resources per email.
4. Why is Redis chosen for implementing the queue in NotificationUtil?
 - a. For its complexity.
 - b. Because it is an in-memory data store with suitable features.
 - c. Solely for cost-saving purposes.
 - d. For its slow processing speed.
5. What is the purpose of a transporter in Node.js and nodemailer?

- a. To store email templates.
 - b. To encapsulate the email sending functionality.
 - c. To manage database connections.
 - d. To encrypt email content.
6. How does a timely received notification affect team communication?
- a. It has no significant impact.
 - b. It slows down communication.
 - c. It speeds up communication.
 - d. It complicates communication.
7. What factors influence the medium of notification?
- a. Developer's preference.
 - b. Time of the day.
 - c. Urgency, criticality of communication, nature of the project.
 - d. Cost of the notification system.
8. How is the transporter object typically used in a Node.js application?
- a. Created for each email sent.
 - b. Created once and reused.
 - c. Only used for receiving emails.
 - d. Initialized in every function call.

Answers

1. b
2. c
3. c
4. b
5. b
6. c
7. c
8. b

Further Readings

<https://nodemailer.com/>

<https://redis.io/>

<https://www.npmjs.com/package/redis>

<https://redis.com/glossary/redis-queue/>

https://en.wikipedia.org/wiki/Message_queue

OceanofPDF.com

CHAPTER 9

Testing API

Introduction

Node.js is a versatile runtime environment that allows developers to run JavaScript on the server-side. When it comes to testing REST APIs, Node.js offers numerous libraries and tools to streamline the process. With Node.js, you can write automated tests that interact with your RESTful APIs, send HTTP requests, and verify responses. It provides a flexible and scalable platform for running API tests, making it an excellent choice for both unit and integration testing.

Node.js also allows you to leverage various testing frameworks and libraries, such as Mocha, Chai, Jest, and Supertest. These tools simplify the creation of test suites, assertion checks, and test runners for your API endpoints. In addition, Node.js's asynchronous nature is well-suited for making HTTP requests and handling asynchronous responses, which is crucial in testing APIs. This asynchronous capability ensures that your tests can efficiently handle multiple requests and responses simultaneously.

In this chapter, we will explore the process of creating test cases and performing API validation.

Structure

In this chapter, we will discuss the following topics:

- Overview of Unit Testing
- Mocha Framework
- Define Test Cases
- Verify Developed API

Overview of Unit Testing

Unit testing is a fundamental practice in software development that ensures that individual units of code are working as intended. In the context of a REST API, this means testing each endpoint and the associated business logic separately to verify that they respond correctly to different inputs and situations.

Unit testing is just one part of testing a REST API. It complements other types of testing, such as integration testing (testing how different parts of the API work together) and end-to-end testing (testing the entire application from the user's perspective).

Unit testing is a critical practice in software development, but it is highly specific to the codebase that you are working with. When writing unit test cases, it is important to cover a wide range of scenarios to ensure your code functions correctly. Here are some key points to consider when writing unit tests:

- **Test Case Structure:**

Describe what the test case is testing using `describe` blocks. Create individual test cases using `it` blocks. Structure your tests logically, covering various aspects of your code.

- **Test Data:**

Include test data or mock data that covers different input scenarios. Include edge cases, boundary values, and typical inputs to validate your code's behavior in different situations.

- **Assertions:**

Use assertions provided by your testing framework (for example, Chai, Jest, Jasmine) to check that the code produces the expected results. Verify that the actual results match the expected results.

- **Error Handling:**

Ensure your code handles errors or exceptions correctly. Test scenarios where exceptions or errors are expected to be thrown.

- **Code Coverage:**

Aim for good code coverage, ensuring that as much of your code as possible is executed by your tests. Use code coverage tools to identify untested code paths.

- **Mocks and Stubs:**

Use mocks and stubs to simulate external dependencies like databases, APIs, or services. Ensure that your code interacts correctly with these dependencies.

- **Positive and Negative Testing:**

Test with positive scenarios where everything works as expected. Test with negative scenarios, where things may go wrong, and the code handles errors correctly.

- **Regression Testing:**

Periodically run your unit tests to catch regressions when new code changes are introduced.

- **Automation**

Unit tests are automated, meaning they can be run automatically by a testing framework without manual intervention.

- **Fast Execution**

Unit tests are designed to be fast, so they can be run frequently during development. This quick feedback loop helps catch issues early in the development process.

Remember that unit tests should focus on a single unit of code (a function, method, or a small component) and should be fast to execute. Writing comprehensive unit tests helps identify and resolve issues early in the development process, leading to more robust and maintainable code.

End-to-End Testing, on the other hand, assesses the entire application from start to finish. It simulates real user scenarios. It essentially tests how a user would interact with the application and includes real database connections, network connections, and more. If there is any other application needed to be connected, it connects that as well.

In this chapter, for testing features, we will connect to a real database to fetch data. When we use real database connection in tests which ideally would be unit tests, we move towards a more integrated testing approach. Sometimes, it may be referred to as integration tests.

Mocha Framework

Mocha is a popular JavaScript test framework for Node.js and web browsers. It provides a versatile and feature-rich environment for writing and running

test cases for your JavaScript applications. Mocha is often used in conjunction with assertion libraries like Chai for making assertions in test cases.

Mocha is well-regarded for its flexibility and wide adoption among developers for various reasons:

- **Ease of Use:**

Mocha's syntax is easy to learn and write, making it accessible for both beginners and experienced developers.

- **Support for Various Test Styles:**

Mocha supports different test styles such as BDD (Behavior-Driven Development), TDD (Test-Driven Development), and QUnit.

- **Asynchronous Testing:**

Mocha has built-in support for testing asynchronous code, allowing you to use `callbacks`, `promises`, or `async/await`.

- **Hooks:**

Mocha provides hooks like `before`, `after`, `beforeEach`, and `afterEach` to set up and tear down test fixtures.

- **Reporter System:**

Mocha offers a range of built-in reporters for generating test reports and results in different formats, as well as custom reporter support.

- **Parallel Test Execution:**

Mocha can run tests in parallel, which can significantly reduce test execution time for large test suites.

- **Test Suites and Nested Descriptions:**

You can organize your tests into hierarchical suites and describe blocks for better structure and readability.

- **Timeouts:**

Mocha allows you to set timeout limits for individual tests or test suites, helping identify slow or blocking tests.

- **Test Skips and Exclusivity:**

You can skip or focus on specific tests or suites using `.skip` and `.only`.

- **Browser and Node.js Support:**

Mocha can be used both in Node.js and in web browsers.

Mocha's flexibility, extensive ecosystem, and active community make it a popular choice for testing JavaScript applications, ranging from small libraries to large complex projects.

To use Mocha, you typically install it as an npm package and write your test cases in JavaScript or a testing framework like Chai. Mocha provides a command-line interface for running tests, and it can be integrated into Continuous Integration (CI) pipelines to automate testing.

[Installing Mocha and Chai](#)

Mocha is a testing framework, and Chai is an assertion library often used together for testing. Install them as development dependencies in our project through cmd enter the following command with root of project directory

```
$ npm install mocha chai @types/mocha @types/chai --save-dev
```

After you have successfully installed Chai, let us explore how to utilize it. Chai serves as a versatile assertion library, effectively functioning as a plugin for your testing needs. **Chai** offers three primary styles: "**expect**," "**should**," and "**assert**." You can use any of these, but "**expect**" is the most popular choice.

```
const chai = require('chai');
const expect = chai.expect;

// Expect a value to be equal to another value
expect(5).to.equal(5);

// Expect an array to include a specific element
expect([1, 2, 3]).to.include(2);

// Expect a value to be a certain data type
expect('Hello').to.be.a('string');

// Expect an object to have a property
expect({ name: 'John' }).to.have.property('name');
```

You can chain various methods to create complex assertions. **Chai** provides a wide range of assertion methods to check equality, check for the existence of properties, and more. Chai is often used in conjunction with testing frameworks like Mocha.

Let us create one directory with the name as **tests** in **src** directory of the project with **test.spec.ts** file. The test file extension must be **.spec.ts** so that tooling can identify it as a file with tests (also known as a spec file). As per the following directory structure you can make test files.

```
tests/
├── user/
│   └── user.spec.ts
├── project/
│   └── project.spec.ts
├── task/
│   └── task.spec.ts
└── common/
    └── utility.spec.ts
└── mocha.opts
```

You can create a Mocha configuration file (**mocha.opts**) if you want to specify Mocha options. This file is optional, but it can be handy for configuring Mocha behavior. Here is a simple example:

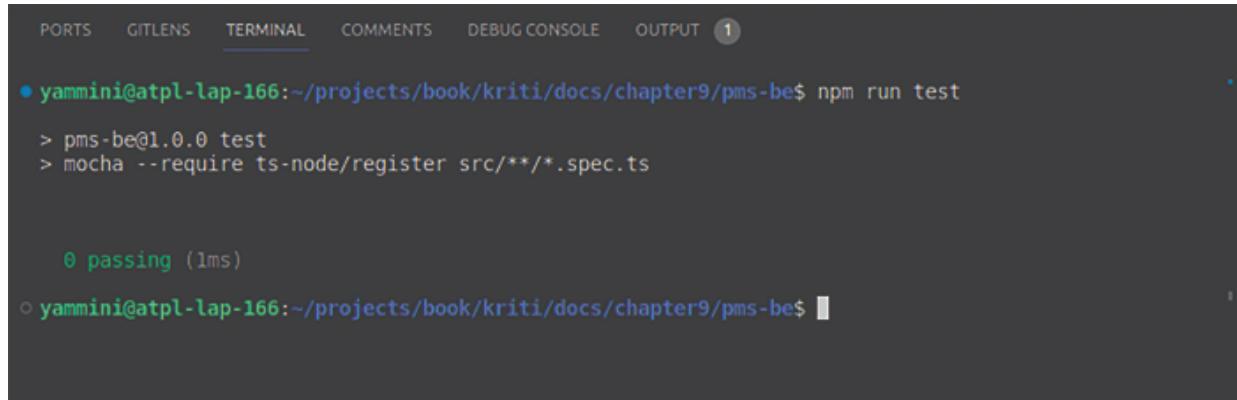
```
--require ts-node/register
--require chai/register-assert
--require chai/register-expect
--require chai/register-should
--require chai-http/register
```

In the **tsconfig.json** file add **"types": ["express", "./src/custom.d.ts"]** in **compilerOptions**.

In the **package.json** file add test script as follows:

```
"scripts": {
  "test": "mocha --require ts-node/register src/**/*.spec.ts
          src/**/*/*.spec.ts
",
  ...
}
```

Currently, no test cases are defined, so it will display 0 passing tests.



```
• yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$ npm run test
> pms-be@1.0.0 test
> mocha --require ts-node/register src/**/*.spec.ts

  0 passing (1ms)

○ yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$
```

Figure 9.1: Run Test Script

Defining a Test Case

After the configuration is added, let us define first basic test case in **utility.spec.ts** file with the following code:

```
import chai from 'chai';
import chaiHttp from 'chai-http';
chai.use(chaiHttp);import { describe, it } from 'mocha';
// Use Chai with Chai HTTP
const expect = chai.expect;
describe('Array', function () {
  describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      expect([1, 2, 3].indexOf(4)).to.equal(-1);
    });
  });
});
```

This test case checks if the **indexOf** function correctly returns **-1** when the value is not present in the array. Chai expect function is used for clear and readable assertions in your test cases. If the expectation is met, the test will pass; otherwise, it will fail and provide feedback on what went wrong.

Run the test in **cmd** with entering **\$ npm run test** which gives following output :

```
Array
  1) #indexOf()
```

✓should return -1 when the value is not present

```
docs > chapter9 > pms-be > src > tests > ts utility.spec.ts > ...
1  import chai from 'chai';
2  import chaiHttp from 'chai-http';
3  chai.use(chaiHttp);
4
5  import { describe, it } from 'mocha';
6  // Use Chai with Chai HTTP
7  const expect = chai.expect;
8  describe('Array', function () {
9      describe('#indexOf()', function () {
10          it('should return -1 when the value is not present', function () {
11              expect([1, 2, 3]).indexOf(4)).to.equal(-1);
12          });
13      });
14  });
15
```

PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1

• **yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be\$** npm run test

```
> pms-be@1.0.0 test
> mocha --require ts-node/register src/**/*.spec.ts
```

```
Array
#indexOf()
✓ should return -1 when the value is not present

1 passing (8ms)
```

○ **yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be\$**

Figure 9.2: Utility Success Test Case

If you change expect line with **expect([1, 2, 3]).indexOf(3)).to.equal(-1);** that fails the test case and gives following output with red mark:

```
Array
1) #indexOf()
  should return -1 when the value is not present
```

```

docs > chapter9 > pms-be > src > tests > ts utility.spec.ts > describe('Array') callback > describe('#indexOf()') callback
1  import chai from 'chai';
2  import chaiHttp from 'chai-http';
3  chai.use(chaiHttp);
4
5  import { describe, it } from 'mocha';
6  // Use Chai with Chai HTTP
7  const expect = chai.expect;
8  describe('Array', function () {
9    describe('#indexOf()', function () {
10      it('should return -1 when the value is not present', function () {
11        |   expect([1, 2, 3]).indexOf(3).to.equal(-1);
12        | });
13      });
14    });
15

PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1

❶ yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$ npm run test
> pms-be@1.0.0 test
> mocha --require ts-node/register src/**/*.spec.ts

Array
  #indexOf()
    1) should return -1 when the value is not present

  0 passing (10ms)
  1 failing

  1) Array

```

Figure 9.3: Utility Fail Test Case

This is a very basic test case to get started. Now, let us dive into more test cases by connecting our application and configuring it for execution.

Configuring the Application

We need to make some changes in **express_server.ts** file. Since we want to connect **express** app in test cases, we need to export it. Here is the updated code:

```

import express from 'express';
import * as bodyParser from 'body-parser';
import { IServerConfig } from './utils/config';
import * as config from '../server_config.json';
import { Routes } from './routes';

export class ExpressServer {

  private static server = null;

```

```

public server_config: IServerConfig = config;
public app;
constructor() {
  const port = this.server_config.port ?? 3000;
  // initialize express app
  this.app = express();
  this.app.use(bodyParser.urlencoded({ extended: false }));
  this.app.use(bodyParser.json());

  this.app.get('/ping', (req, res) => {
    res.send('pong');
  });

  const routes = new Routes(this.app);
  if (routes) {
    console.log('Server Routes started for server');
  }
}

ExpressServer.server = this.app.listen(port, () => {
  console.log(`Server is running on port ${port} with pid =
  ${process.pid}`);
});
}

//close the express server for safe on uncaughtException
public closeServer(): void {
  ExpressServer.server.close(() => {
    console.log('Server closed');
    process.exit(0);
  });
}
}

```

Similarly, when dealing with test cases, it is necessary to connect to the database. Replace the code in the **db.ts** file with the following:

```

import { DataSource, Repository } from 'typeorm';
import { IServerConfig } from './config';
import * as config from '../../../../../server_config.json';
import { Roles } from '../components/roles/roles_entity';
import { Users } from '../components/users/users_entity';

```

```
import { Projects } from
'../components/projects/projects_entity';
import { Tasks } from '../components/tasks/tasks_entity';
import { Comments } from
'../components/comments/comments_entity';

export class DatabaseUtil {
  private server_config: IServerConfig = config;
  private static connection: DataSource | null = null;
  private repositories: Record<string, Repository<any>> = {};

  constructor() {
    this.connectDatabase();
  }

  /**
   * Establishes a database connection or returns the existing
   * connection if available.
   * @returns The database connection instance.
   */
  public async connectDatabase(): Promise<DataSource> {
    try {
      if (DatabaseUtil.connection) {
        return Promise.resolve(DatabaseUtil.connection);
      } else {
        const db_config = this.server_config.db_config;
        const AppSource = new DataSource({
          type: 'postgres',
          host: db_config.host,
          port: db_config.port,
          username: db_config.username,
          password: db_config.password,
          database: db_config.dbname,
          entities: [Roles, Users, Projects, Tasks,
          Comments, Files],
          synchronize: true,
          logging: true,
          poolSize: 10
        });
    }
  }
}
```

```
        await AppSource.initialize();
        DatabaseUtil.connection = AppSource;
        console.log('Connected to the database');
        return DatabaseUtil.connection;
    }

} catch (error) {
    console.error('Error connecting to the database:', error);
}
}

/**
 * Get the repository for a given entity.
 * @param entity - The entity for which the repository is
 * needed.
 * @returns The repository instance for the entity.
 */
public getRepository(entity) {
    try {
        // Check if a valid database connection is available
        if (DatabaseUtil.connection) {
            const entityName = entity.name;

            // Check if the repository instance already exists, if
            // not, create it
            if (!this.repositories[entityName]) {
                this.repositories[entityName] =
                    DatabaseUtil.connection.getRepository(entity);
            }
            return this.repositories[entityName];
        }
        return null;
    } catch (error) {
        console.error(`Error while getRepository =>
        ${error.message}`);
    }
}
}
```

As shown in the code, we have introduced a **promise** to ensure that the test cases run only after the database connection is established. This helps prevent potential errors that could occur otherwise.

In case we want to mock the database connection, there are libraries available to do so. One such library is **sinon**, <https://sinonjs.org/>.

Sinon is a testing library used for creating spies, stubs, and mocks in JavaScript tests, rather than specifically for mocking databases. It can be used to intercept and simulate behavior in functions, methods, or any kind of operation in your codebase, including database operations, API requests, or any other external service interactions. This makes it incredibly useful for writing unit and integration tests where you need to isolate the part of the code being tested.

For this chapter, we are interested in checking our API against a real database. Hence, we do not need to use a mock database connection. However, for the sake of completion, an example is added at the end of this chapter.

Now, replace following code in **utility.spec.ts** file to export app in test cases:

```
import { DatabaseUtil } from '../../utils/db';
import { ExpressServer } from '../../express_server';
import chai from 'chai';
import chaiHttp from 'chai-http';
chai.use(chaiHttp);

import { describe, it } from 'mocha';
// Use Chai with Chai HTTP
const expect = chai.expect;
let app, expressServer;

before(async () => {
  const databaseUtil = new DatabaseUtil();
  await databaseUtil.connectDatabase();
  expressServer = new ExpressServer();
  app = expressServer.app;
});

// Close the server after all tests are done
after(function (done) {
  expressServer.closeServer(done);
```

```
});  
export { app };
```

It imports necessary modules like `DatabaseUtil`, `ExpressServer`, and the testing libraries (`chai` and `chai-http`). `chai.use(chaiHttp)` configures chai to work with HTTP requests, enabling you to make HTTP requests and perform assertions on their responses.

Hooks

In the context of testing frameworks like Mocha, "`before`" and "`after`" are known as test hooks. They are used to set up and tear down the testing environment. Here is what they do:

Before Hook (before): This hook is executed before any test cases within a test suite (defined using `describe`) are run. It is typically used for setting up the environment or any common context needed for the tests. For example, you might use it to establish a database connection, initialize variables, or start a server.

```
before(() => {  
  // Set up the testing environment  
});
```

After Hook (after): This hook is executed after all test cases within a test suite have run. It is commonly used for cleaning up the environment, releasing resources, or performing any necessary actions after the tests are completed. For example, you might use it to close a database connection, shut down a server, or perform `cleanup` tasks.

```
after(() => {  
  // Clean up the testing environment  
});
```

Here is how the "`before`" and "`after`" hooks fit into the test lifecycle:

Before All Tests: The "`before`" hook is executed before any of the test cases within the suite are run. It is a one-time setup for the entire suite.

Run Test Cases: All the test cases (it blocks) within the suite are executed.

After All Tests: The "`after`" hook is executed once all the test cases in the suite have been completed. It is a one-time `cleanup` step for the suite.

These hooks are useful for ensuring a consistent and clean test environment for each test suite. They help avoid code repetition and make it easier to manage resources like database connections, servers, or other setup and teardown tasks.

There are also `beforeEach()`, and `afterEach()` hooks that will execute before each test case and after each test case respectively.

We can explore hook with another example such as payment create one `payment.spec.ts` file with following code :

```
import { expect } from 'chai';
import { describe, it } from 'mocha';

class Payment {
  private amount: number;
  private method: string;

  constructor(amount: number, method: string) {
    this.amount = amount;
    this.method = method;
  }

  processPayment(): string {
    // Simulate payment processing
    return `Payment of ${this.amount} processed via
    ${this.method}`;
  }
}

describe('Payment', () => {
  let payment: Payment;

  // Before hook: This will run before the test suite before(() => {
  //   console.log('Setting up payment processing...');
  //   // Perform setup tasks, e.g., initialize payment gateway
  //   payment = new Payment(100, 'Credit Card');
  // });

  // After hook: This will run after the test suite after(() => {
  //   console.log('Tearing down payment processing...');
  // });
})
```

```

// Perform teardown tasks, e.g., close payment gateway
connection
payment = null!;
});

// Test case
it('should process payment successfully', () => {
  // Act
  const result = payment.processPayment();

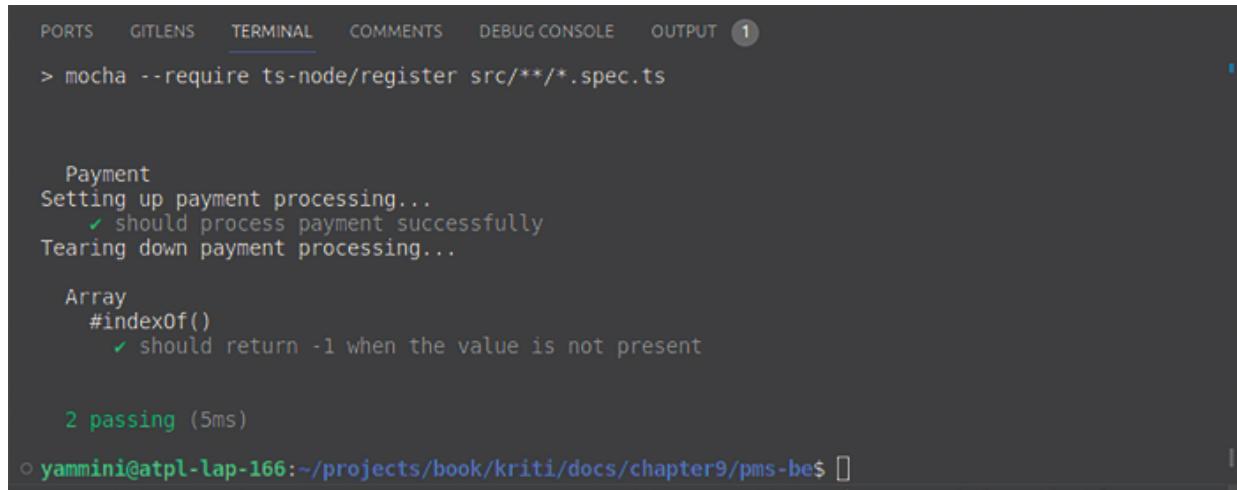
  // Assert
  expect(result).to.equal('Payment of 100 processed via Credit
Card');
});
});

```

In this example:

- We have a `Payment` class with a `processPayment` method that simulates processing a payment transaction.
- We use Mocha's before hook to perform setup tasks before the test suite. This includes initializing payment processing, such as setting up a payment gateway.
- We use Mocha's after hook to perform teardown tasks after the test suite. This includes closing the payment gateway connection.
- We define a single test case to verify that the payment is processed successfully.
- Before the test suite runs, the "**Setting up payment processing...**" message will be logged, indicating that payment processing is being set up.
- After the test suite runs, the "**Tearing down payment processing...**" message will be logged, indicating that payment processing is being torn down.

Now run the test script in the terminal and it will display following output:



```
PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1
> mocha --require ts-node/register src/**/*.spec.ts

Payment
Setting up payment processing...
  ✓ should process payment successfully
Tearing down payment processing...

Array
  #indexOf()
    ✓ should return -1 when the value is not present

2 passing (5ms)
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-bes
```

Figure 9.4: Hook Example with Payment

Verifying APIs through Test cases

In the world of software development, ensuring that your application's APIs work as intended is of paramount importance. To achieve this, we rely on test cases, which are a structured approach to validating the functionality, correctness, and performance of APIs. Verifying APIs through test cases involves systematically testing various aspects of your API to guarantee that it behaves as expected.

Login Test

Create a **"user.spec.ts"** file within the **"tests"** directory, specifically within the **"user"** subdirectory, and include the following code:

```
import chai from 'chai';
import chaiHttp from 'chai-http';
chai.use(chaiHttp);

import { describe, it } from 'mocha';
// Use Chai with Chai HTTP
const expect = chai.expect;
import { app } from '../common/utility.spec';
let authToken; // Declare a variable to store the
authentication token

describe('Login API', () => {
```

```

it('should return a success message when login is successful', (done) => {
  chai.request(app) // Replace 'app' with your Express app
  instance
    .post('/api/login')
    .send({ email: 'yamipanchal1993@gmail.com', password: 'Abc@123456' })
    .end((err, res) => {
      expect(res).to.have.status(200);
      expect(res.body).to.have.property('status').equal('success');
      authToken = res.body.data.accessToken; // Save the
      authentication token
      done();
    });
  });
  it('should return an error message when login fails', (done) => {
    chai.request(app)
      .post('/api/login')
      .send({ email: 'yamipanchal1993@gmail.com', password: 'wrongpassword' })
      .end((err, res) => {
        expect(res).to.have.status(400);
        expect(res.body).to.have.property('message').equal('Password is not valid');
        done();
      });
    });
  });
  export { authToken };
}

```

Run the test case in `cmd` through the `npm run test` which provides the following output.

```

> pms-be@1.0.0 test
> mocha --require ts-node/register 'src/**/*spec.ts'

```

Login API

```

1) "before all" hook in "{root}"
  Connected to the database
  ✓> should return a success message when login is successful
  (75ms)
  ✓ should return an error message when login fails (46ms)

```

```

Connected to the database
Router : RoleRoutes - Connected
Router : UserRoutes - Connected
Router : ProjectRoutes - Connected
Router : TaskRoutes - Connected
Router : CommentRoutes - Connected
Router : FileRoutes - Connected
Server Routes started for server
  Payment
    1) "before all" hook in "{root}"
Setting up payment processing...
  ✓ should process payment successfully
Tearing down payment processing...

  Array
    #indexOf()
    ✓ should return -1 when the value is not present

  Login API
  query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE email = 'yamipanchal1993@gmail.com'
    ✓ should return a success message when login is successful (63ms)
  query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE email = 'yamipanchal1993@gmail.com'
    ✓ should return an error message when login fails

  Server closed
  ○ yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$ 

```

Figure 9.5: Login Test Case

Here, we define two test cases for Login api in that the first has valid data and the second has the wrong password.

- **describe('Login API', () => { ... });**: This line defines a test suite using the describe function. In this case, it is a suite named "Login API" that groups related test cases together.
- **it('should return a success message when login is successful', (done) => { ... });**: Within the test suite, there is an individual test case defined using the it function. This test case has a description that explains what it is testing, which is that it should return a success message when the login is successful. The (done) function is

passed as an argument, indicating that this is an asynchronous test, and the done function is used to signal the completion of the test.

- **chai.request(app):** This line uses the chai-http library to make an HTTP request to an Express.js app. app should be replaced with the actual instance of your Express app.
- **.post('/api/login'):** This line specifies that it is a POST request to the '/api/login' endpoint. It is likely that this endpoint is responsible for handling user login.
- **.send({ email: 'yamipanchal1993@gmail.com', password: 'Abc@123' }):** Here, the code sends a JSON object in the request body with the email and password values for the login attempt.
- **.end((err, res) => { ... }):** This is the **callback** function that gets executed when the HTTP request is completed. It receives two parameters: err for any errors that might occur during the request, and res for the response from the server.
- **expect(res).to.have.status(200);:** This line uses Chai's expect assertion to check if the HTTP response has a status code of 200, which typically indicates a successful request.
- **expect(res.body).to.have.property('status').equal('success') ;:** This line checks that the response body contains a property named 'status' with the value 'success'. It is a common way to check if an API response indicates a successful operation.
- **authToken = res.body.data.accessToken;:** If the login is successful, this line extracts the authentication token from the response and stores it in the authToken variable. This token is often used for subsequent authenticated requests.
- **done();:** Finally, the **done** function is called to indicate that the test has completed.

This code is a test case for login API returns a successful response with the expected status code and message. It also captures the authentication token for further testing, typically for authenticated routes.

List of User Test

In the same file, add the following test case code to verify the List of user API.

```
describe('GET List of Users', () => {
  it('should return array with status code 200', (done) => {
    chai.request(app)
      .get('/api/users')
      .set('Authorization',
        `Bearer ${authToken}`) // Pass the token in the headers
      .end((err, res) => {
        // console.log(res);
        expect(res).to.have.status(200);
        expect(res.body).to.have.property('data').to.be.an('array')
      );
      done();
    });
  });
});

Output
> pms-be@1.0.0 test
> mocha --require ts-node/register 'src/**/*.spec.ts'

Login API
1) "before all" hook in "{root}"
  Connected to the database
  ✓ should return a success message when login is successful
  (75ms)
  ✓ should return an error message when login fails (46ms)

GET List of Users
  ✓ should return array with status code 200

Server closed
```

```

docs > chapter9 > pms-be > src > tests > users > ts user.spec.ts > describe('GET List of Users') callback > it('should return array with status code 200')
36  describe('GET List of Users', () => {
37    it('should return array with status code 200', (done) => {
38      .get('/api/users')
39      .set('Authorization', `Bearer ${authToken}`) // Pass the token in the headers
40      .end((err, res) => {
41        // console.log(res);
42        expect(res).to.have.status(200);
43        expect(res.body).to.have.property('data').to.be.an('array');
44        done();
45      });
46    });
47  });
48
49  export { authToken };

```

PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1

```

GET List of Users
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'yamini'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles".role_id IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users"
✓ should return array with status code 200

Server closed
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$ 

```

Figure 9.6: User List Get API Testcase

Here are the details of the key parts of the proceeding code:

- **describe('GET List of Users', () => { ... })**: This line defines a new test suite with the description "GET List of Users". This suite groups together related test cases that are concerned with retrieving a list of users.
- **it('should return array with status code 200', (done) => { ... })**: Within the test suite, there is an individual test case defined using the **it** function. The test case description specifies that it should return an array with a status code of 200. The **(done)** function is used to indicate that this is an asynchronous test, and the **done** function will be called when the test is complete.
- **.get('/api/users')**: This line specifies that it is a GET request to the '/api/users' endpoint. This endpoint is likely responsible for fetching a list of users.

- `.set('Authorization', Bearer ${authToken})`: Here, the code is setting an **"Authorization"** header in the HTTP request. It includes an authentication token in the header, typically in the format **"Bearer <token>"**. This is a common way to authenticate API requests. **authToken** is expected to contain the token obtained during a successful login (as shown in your previous code snippet).
- `.end((err, res) => { ... })`: This is the **callback** function that gets executed when the HTTP request is completed. It receives two parameters: **err** for any errors that might occur during the request, and **res** for the response from the server.
- `expect(res).to.have.status(200)`: This line uses Chai's **expect** assertion to check if the HTTP response has a status code of 200, which typically indicates a successful request. This is the common way to ensure that the server responded with a 200 **OK** status for a successful **GET** request.
- `expect(res.body).to.have.property('data').to.be.an('array')`: This line checks that the response body contains a property named **'data'** and that this property's value is an array. This is a way to verify that the response contains a list of users in the form of an array.
- `done()`: Finally, the **done** function is called to signal that the test has completed.

This code is a test case for an Express.js API endpoint that tests whether a GET request to fetch a list of users returns the expected status code (**200**) and verifies that the response contains an array of user data. It also includes the authorization token in the request header for authentication, assuming that the **authToken** variable holds a valid token obtained from a login request.

Add User Test

In the same file, add the following test case that verifies add user API.

```
describe('ADD User', () => {
  it('should return with status code 201', (done) => {
    chai.request(app)
      .post('/api/users')
      .set('Authorization',
```

```

`Bearer ${authToken}`) // Pass the token in the headers
.send({
  'fullname': 'Super Admin',
  'username': 'pms-admin1',
  'email': 'admin@pms1.com',
  'password': 'Admin@pms1',
  'role_id': 'dbda47e4-f843-4263-a4d6-69ef80156f81'
})
.end((err, res) => {
  expect(res).to.have.status(201);
  done();
});
});

it('should return with status code 409', (done) => {
  chai.request(app)
    .post('/api/users')
    .set('Authorization',
      `Bearer ${authToken}`) // Pass the token in the headers
    .send({
      'fullname': 'Super Admin',
      'username': 'pms-admin1',
      'email': 'admin@pms1.com',
      'password': 'Admin@pms1',
      'role_id': 'dbda47e4-f843-4263-a4d6-69ef80156f81'
    })
    .end((err, res) => {
      expect(res).to.have.status(409);
      expect(res.body).to.have.property('message').equal('Key
(username)=(pms-admin1) already exists.');
      done();
    });
  });
});

Output
> pms-be@1.0.0 test
> mocha --require ts-node/register 'src/**/*.spec.ts'
Login API

```

1) "before all" hook in "{root}"

Connected to the database

- ✓ should return a success message when login is successful (75ms)
- ✓ should return an error message when login fails (46ms)

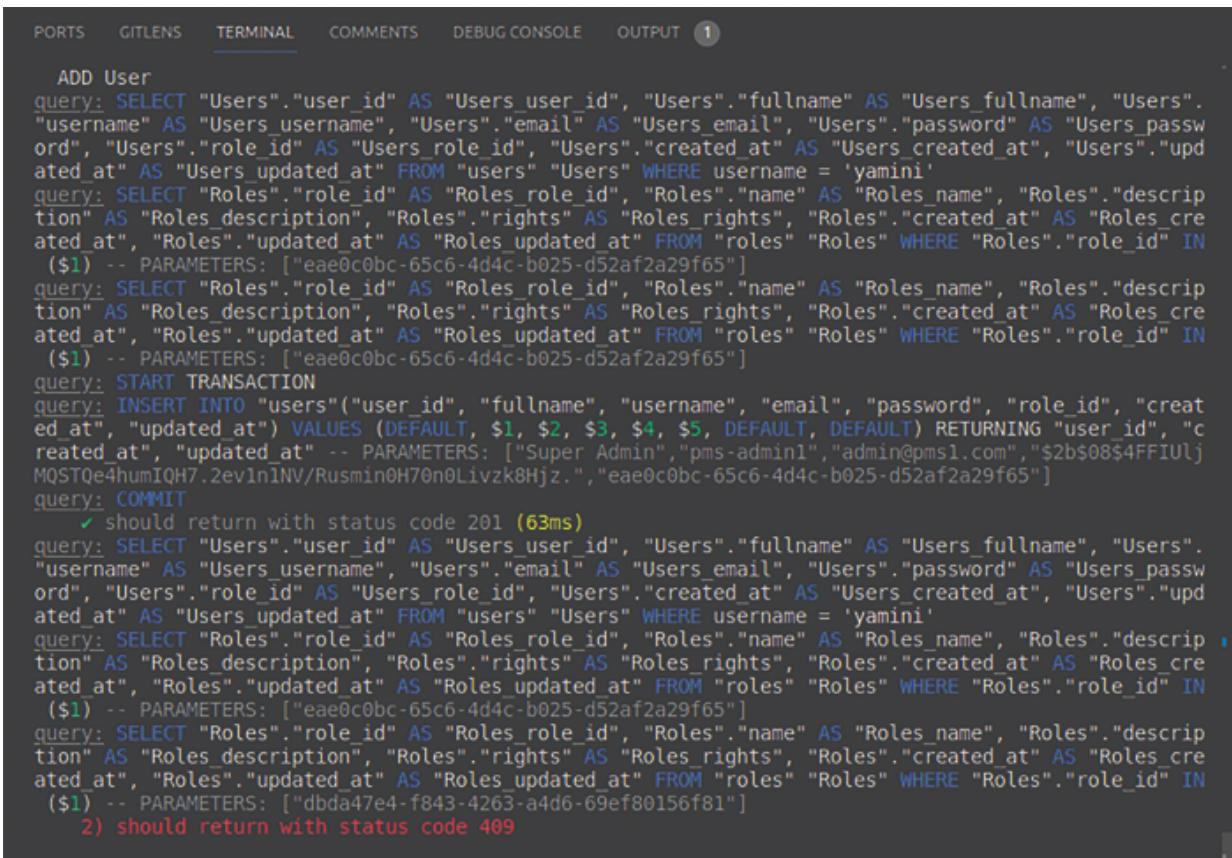
GET List of Users

- ✓ should return array with status code 200

ADD User

- ✓ should return array with status code 201
- ✓ should return with status code 409 (54ms)

Server closed



```
PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1

ADD User
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'yamini'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: START TRANSACTION
query: INSERT INTO "users"("user_id", "fullname", "username", "email", "password", "role_id", "created_at", "updated_at") VALUES (DEFAULT, $1, $2, $3, $4, $5, DEFAULT, DEFAULT) RETURNING "user_id", "created_at", "updated_at" -- PARAMETERS: ["Super Admin", "pms-admin1", "admin@pms1.com", "$2b$08$4FFIULjMQSTQe4humIQH7.2evln1NV/Rusmin0H70n0Livzk8Hjz.", "eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: COMMIT
  ✓ should return with status code 201 (63ms)
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'yamini'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["dbda47e4-f843-4263-a4d6-69ef80156f81"]
  2) should return with status code 409
```

Figure 9.7: Add User API Test Case

Here are the details of the key parts of the proceeding code:

- **describe('ADD User', () => { ... })**: This line defines a test suite with the description "ADD User". This suite groups together related test cases that are concerned with adding new users.

- The first test case:

- `it('should return with status code 201', (done) => { ... })`;: This test case description indicates that it is testing whether adding a user should return a status code of **201** (Created). The `(done)` function is used to indicate that this is an asynchronous test, and the `done` function will be called when the test is complete.
- `.post('/api/users')`: It's a **POST** request to the '`/api/users`' endpoint, presumably used to add a new user.
- `.set('Authorization', Bearer ${authToken})`: This line sets an "`Authorization`" header in the HTTP request, including the authentication token for authorization.
- `.send({ ... })`: The code sends a JSON object in the request body with user information, including the user's fullname, username, email, password, and role IDs. This represents the data you are trying to add.
- `.end((err, res) => { ... })`;: The `callback` function that gets executed when the HTTP request is completed.
- `expect(res).to.have.status(201)`;: This line uses Chai's `expect` assertion to check if the HTTP response has a status code of 201, indicating that the user creation was successful.
- `done()`;: Finally, the `done` function is called to signal that the test has completed.

- The second test case:

- `it('should return with status code 409', (done) => { ... })`;: This test case description indicates that it's testing whether adding a user with the same username should return a status code of **409** (Conflict).
- The structure of this test case is similar to the first one, with the main differences being the expected status code and the additional checks:
- `expect(res).to.have.status(409)`;: This line checks if the HTTP response has a status code of **409**, indicating a conflict.

- `expect(res.body).to.have.property('message').equal('Key (username)=(pms-admin1) already exists.');`: This line verifies that the response body contains a specific message indicating that the provided username already exists.
- `done();`: As before, the done function is called to signal the completion of the test.

This code contains two test cases. The first test case checks if a new user is successfully added with a 201 status code, while the second test case checks if a conflict (409 status code) is returned when attempting to add a user with an existing username. The second test case also verifies the presence of an error message in the response.

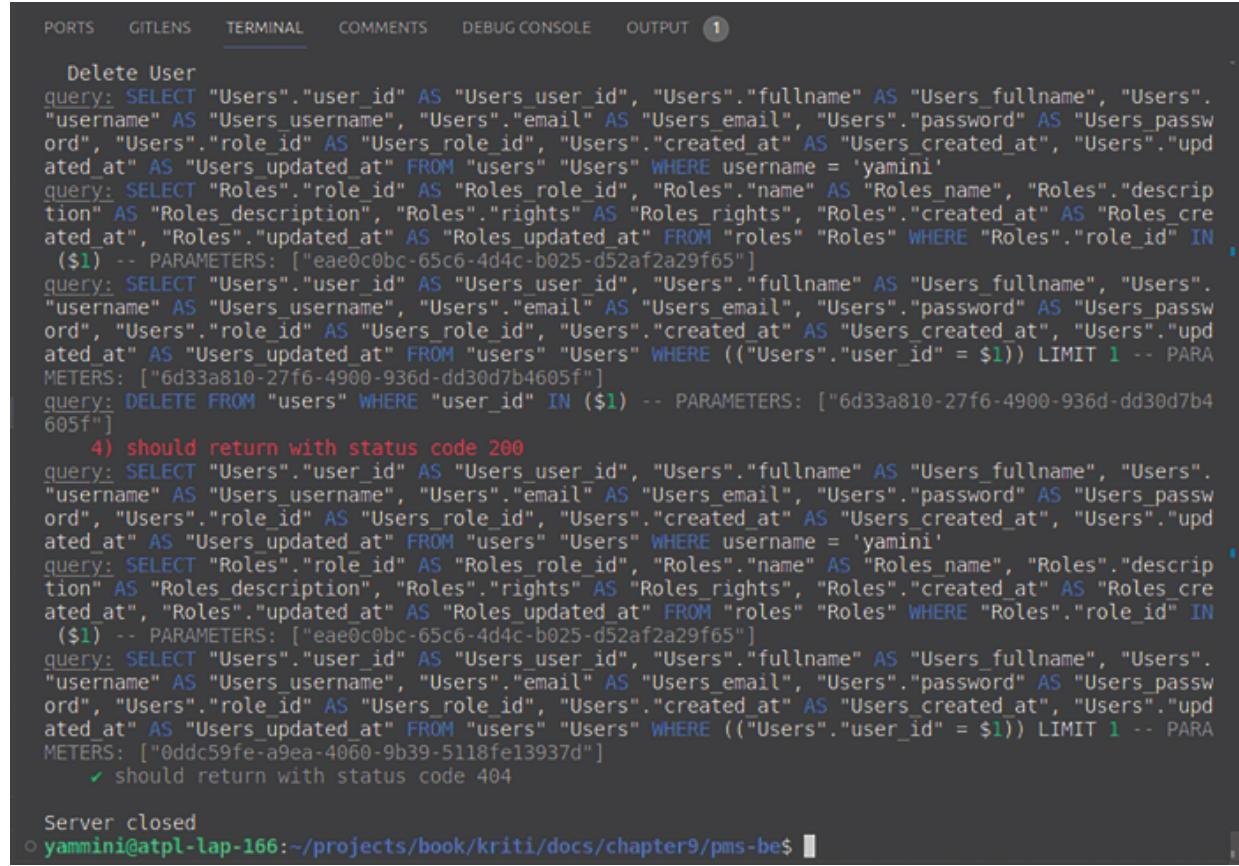
Delete User Test

Within the existing file, include the following test case code to validate the functionality of the user deletion API.

```
describe('Delete User', () => {
  it('should return with status code 200', (done) => {
    chai.request(app)
      .delete('/api/users/0ddc59fe-a9ea-4060-9b39-5118fe13937d')
      .set('Authorization',
        `Bearer ${authToken}`) // Pass the token in the headers
      .end((err, res) => {
        expect(res).to.have.status(201);
        done();
      });
  });

  it('should return with status code 404', (done) => {
    chai.request(app)
      .delete('/api/users/0ddc59fe-a9ea-4060-9b39-5118fe13937d')
      .set('Authorization', `Bearer ${authToken}`) // Pass the
      token in the headers
      .end((err, res) => {
        expect(res).to.have.status(404);
        done();
      });
  });
});
```

```
});  
});  
Output  
> pms-be@1.0.0 test  
> mocha --require ts-node/register 'src/**/*.spec.ts'  
Login API  
1) "before all" hook in "{root}"  
  Connected to the database  
  ✓ should return a success message when login is successful  
(75ms)  
  ✓ should return an error message when login fails (46ms)  
GET List of Users  
  ✓ should return array with status code 200  
ADD User  
  X should return array with status code 201  
  ✓ should return with status code 409 (54ms)  
Delete User  
  ✓ should return with status code 200  
  ✓ should return with status code 404  
Server closed
```



```

PORTS GITLENS TERMINAL COMMENTS DEBUG CONSOLE OUTPUT 1

Delete User
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'yamini'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE ((Users.user_id = $1)) LIMIT 1 -- PARAMETERS: ["6d33a810-27f6-4900-936d-dd30d7b4605f"]
query: DELETE FROM "users" WHERE "user_id" IN ($1) -- PARAMETERS: ["6d33a810-27f6-4900-936d-dd30d7b4605f"]
4) should return with status code 200
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE username = 'yamini'
query: SELECT "Roles"."role_id" AS "Roles_role_id", "Roles"."name" AS "Roles_name", "Roles"."description" AS "Roles_description", "Roles"."rights" AS "Roles_rights", "Roles"."created_at" AS "Roles_created_at", "Roles"."updated_at" AS "Roles_updated_at" FROM "roles" "Roles" WHERE "Roles"."role_id" IN ($1) -- PARAMETERS: ["eae0c0bc-65c6-4d4c-b025-d52af2a29f65"]
query: SELECT "Users"."user_id" AS "Users_user_id", "Users"."fullname" AS "Users_fullname", "Users"."username" AS "Users_username", "Users"."email" AS "Users_email", "Users"."password" AS "Users_password", "Users"."role_id" AS "Users_role_id", "Users"."created_at" AS "Users_created_at", "Users"."updated_at" AS "Users_updated_at" FROM "users" "Users" WHERE ((Users.user_id = $1)) LIMIT 1 -- PARAMETERS: ["0ddc59fe-a9ea-4060-9b39-5118fe13937d"]
✓ should return with status code 404

Server closed
yammini@atpl-lap-166:~/projects/book/kriti/docs/chapter9/pms-be$
```

Figure 9.8: Delete User API Testcase

In this code, Add User test case displays fail because same user is already in database. Similarly, the Delete User test case might fail if the user you're trying to delete doesn't exist in the database when the test runs.

- The first test case:

- **it('should return with status code 200', (done) => { ... })**: The description of this test case indicates that it is testing whether a successful user deletion should return a status code of **200 (OK)**. The **(done)** function is used to indicate that this is an asynchronous test, and the **done** function will be called when the test is complete.
- **.delete('/api/users/0ddc59fe-a9ea-4060-9b39-5118fe13937d')**: It is a **DELETE** request to a specific endpoint **'/api/users/'** with a user identifier (for example, **'0ddc59fe-a9ea-4060-9b39-5118fe13937d'**) in the URL. This typically

represents the action of deleting a specific user by their unique identifier.

- `.set('Authorization', Bearer ${authToken})`: This line sets an "Authorization" header in the HTTP request, including the authentication token for authorization.
 - `.end((err, res) => { ... })`: The `callback` function that gets executed when the HTTP request is completed.
 - `expect(res).to.have.status(201)`: There is a potential issue in this line. It checks if the HTTP response has a status code of **201**, but the test case description implies that it should be expecting a **200** status code. This line should be corrected to `expect(res).to.have.status(200)`.
 - `done()`: Finally, the `done` function is called to signal that the test has completed.
- The second test case:
 - `it('should return with status code 404', (done) => { ... })`: The description of this test case indicates that it is testing whether attempting to delete a user that does not exist should return a status code of **404 (Not Found)**. The structure of this test case is similar to the first one, with the main difference being the expected status code:
 - `expect(res).to.have.status(404)`: This line checks if the HTTP response has a status code of 404, indicating that the requested user was not found.
 - `done()`: As before, the `done` function is called to signal the completion of the test.

This code contains two test cases for testing the user deletion functionality through an Express.js API. The first test case checks if a user is successfully deleted with a **200** status code, and the second test case checks if attempting to delete a non-existent user results in a **404** status code.

In this manner, you have the flexibility to create distinct test scenarios for various sections within your application, including user management, project handling, task management, and comment features. For each of these APIs, you can define specific test cases. Additionally, you can extend your testing suite by incorporating cases that verify authentication failures, such as

testing for a **401** status code when authentication is missing and a 403 status code when authorization is not granted for certain API endpoints.

Mocking Database Connection

To mock the database connection, we can use the popular library — **sinon**. Let us install the library with npm.

```
npm install sinon -save-dev
```

Consider if we have a function to get the user for a given id as:

```
// db.ts
async function getUserById(user_id:string) {
  // actual logic to fetch the user from database
}
module.exports = { getUserById };
```

We can use sinon to mock this behavior -

```
const sinon = require('sinon');
const { expect } = require('chai');
const db = require('../db');
describe('getUserById', function() {
  it('should return mocked user data', async function() {
    // Create a stub for getUserById
    const mockUser = { id: 1, name: 'Alice M' };
    const stub = sinon.stub(db,
      'getUserById').resolves(mockUser);

    // Call the function (which is now stubbed)
    const user = await getUserById(1);

    // Verify the function returned the mocked data
    expect(user).to.deep.equal(mockUser);

    // Restore the original function
    stub.restore();
  });
});
```

In the preceding example, **sinon.stub()** is used to replace the actual function **getUserById()** with a version that returns a promise which

resolves to the `mockUser`. This test does not connect to an actual database. This way it ensures that the test is isolated and repeatable. `stub.restore()` at the end restores the original function.

Conclusion

In this chapter, the test cases presented in the code examples provide a comprehensive approach to testing various aspects of an Express.js API application. These test cases cover different scenarios which include Login and user management APIs. These tests are an essential part of ensuring the reliability, security, and correctness of the API, and they can help identify and address issues early in the development process.

By systematically testing the application, you can increase its robustness and enhance the overall quality of your software.

In the next chapter, we will learn how to build and deploy our application.

OceanofPDF.com

CHAPTER 10

Building and Deploying Application

Introduction

After completing the development phase, it becomes essential to enable global access to the application. The final and crucial step for any application is to build and deploy it to a centralized location, ensuring widespread availability and usability to the end users. When deploying a Node.js application, there are various servers and platforms available to host and run your application. The choice depends on factors such as scalability, ease of use, performance, and specific requirements of the project.

In this chapter, we will delve into building and securely deploying the application, employing the most widely used deployment processes in demand.

Structure

In this chapter, we will discuss the following topics:

- Code Obfuscation
- Building the Application
- Deploying the Application

Code Obfuscation

Code obfuscation is a technique used to transform source code into a form that is more difficult to understand or reverse engineer, while maintaining its original functionality. The purpose of code obfuscation is not to enhance the security of the code significantly but rather make it more challenging for someone to comprehend or decompile the code.

Common Techniques

There are some common techniques used in code obfuscation as follows:

- **Renaming Variables and Functions:**

Obfuscators change the names of variables, functions, and classes to meaningless or random strings, making it harder to understand the purpose of each element.

- **Control Flow Obfuscation:**

This involves restructuring the control flow of the program, such as using **goto** statements or introducing redundant code, to make it less predictable and harder to follow.

- **String Encryption:**

Literal strings in the code are encrypted or encoded, and then decrypted or decoded at runtime. This makes it more challenging for someone to understand the string values by simply inspecting the code.

- **Code Splitting:**

Breaking down functions into smaller pieces or splitting them across multiple files, making it harder to comprehend the entire flow of the program.

- **Dummy Code Insertion:**

Introducing irrelevant or redundant code snippets that do not affect the program's functionality but add complexity for a human reader trying to understand the code.

- **Constant Value Obfuscation:**

Changing the representation of numerical constants or using mathematical operations to obscure their actual values.

- **Anti-Debugging Techniques:**

Embedding code that detects debugging attempts and alters the behavior of the program, making it more difficult for reverse engineers to analyze the code in a debugger.

- **Code Compression:**

Reducing the overall size of the code by compressing it, making it harder to read and analyze.

Let us delve into the practical aspects of code obfuscation and explore its details hands-on.

Obfuscating an entire TypeScript project involves applying code obfuscation to all TypeScript files, including their dependencies. The process can be a bit involved and requires careful consideration of the build process, dependencies, and potential impact on the project.

There are many ways in which a code can be obfuscated. The most popular tools among all are **javascript-obfuscator**, **UglifyJs**, **webpack**, and others. We will be using **javascript-obfuscator** to obfuscate the code.

[Installing Required Dependencies](#)

Install the JavaScript obfuscation library, **javascript-obfuscator**, to integrate it into the project. Open the terminal from root directory of project and paste following command:

```
$ npm install javascript-obfuscator --save-dev
```

[Creating an Obfuscation Script](#)

Create a tool directory in the root folder and add script **obfuscate.js** file with following code that will obfuscate each generated JavaScript file.

```
// obfuscate.js
/* eslint-disable no-undef */
/* eslint-disable @typescript-eslint/no-var-requires */
const JavaScriptObfuscator = require('javascript-obfuscator');
const fs = require('fs');
const path = require('path');
const jsonObfuscatorModule = require('json-obfuscator');
const sourceDirectory = 'dist/src'; // Update with your actual
output directory
const obfuscatedDirectory = 'build'; // Output directory for
obfuscated code

const obfuscateFile = (filePath) => {
  const code = fs.readFileSync(filePath, 'utf8');
  const obfuscatedCode = JavaScriptObfuscator.obfuscate(code, {
    compact: true,
    controlFlowFlattening: true
    // ... other obfuscation options
  });
}
```

```

const obfuscatedFilePath = path.join(obfuscatedDirectory,
path.relative(sourceDirectory, filePath));
fs.mkdirSync(path.dirname(obfuscatedFilePath), { recursive:
true });
fs.writeFileSync(obfuscatedFilePath,
obfuscatedCode.getObfuscatedCode(), 'utf8');
};

const processDirectory = (directoryPath) => {
const files = fs.readdirSync(directoryPath);
files.forEach((file) => {
const filePath = path.join(directoryPath, file);
if (fs.statSync(filePath).isDirectory()) {
processDirectory(filePath);
} else if (path.extname(filePath) === '.js') {
obfuscateFile(filePath);
}
});
};

processDirectory(sourceDirectory);

```

Here,

- **sourceDirectory** specifies the directory containing the original JavaScript files. You should update this with the actual output directory of JavaScript files compiled from Typescript files.
- **obfuscatedDirectory** specifies the directory where obfuscated files will be saved.
- **obfuscateFile(filePath)** reads the content of a JavaScript file, obfuscates it using **javascript-obfuscator**, and writes the obfuscated content to the specified location.
- **processDirectory(directoryPath)** recursively processes files in a directory, invoking obfuscateFile for each JavaScript file found.

So when this script will run, a new build directory will be created that contains JavaScript obfuscated code.

This provides a summary of the obfuscation process, which will be applied in the subsequent build and execution of the code.

Downside of Code Obfuscation

Code obfuscation makes the source code harder to read and understand. However, there are some downsides of it.

Obfuscation can add extra layers of complexity to the code which can impact the performance at run time. Since the code is obfuscated, it is harder to read and while debugging it becomes challenging to trace the problem source. This adds extra time for debugging and troubleshooting.

For future versions of the source code, obfuscation is done again every time new code is released. Obfuscation process adds additional characters and structures to the code which leads to larger file sizes.

One last point to discuss here is security. Obfuscation may give a false sense of code security. It is true that the code afterwards is harder to read but it can be reverse-engineered. Obfuscation does not fix any security vulnerabilities in the code and if someone reverse-engineers the obfuscated code, the vulnerabilities can be exploited.

While skilled hackers may reverse-engineer the code, obfuscation acts as a deterrent to casual or less-skilled hackers. Even for skilled hackers it requires additional efforts to reverse-engineer.

Building the Application

Building an application refers to the process of transforming the source code of an application into a format or structure that can be executed or run by a computer. The build process involves compilation and transpilation. TypeScript compilation is the process of translating TypeScript source code into JavaScript code making it compatible with various JavaScript runtime environments.

Open the terminal from the root directory and execute the following command that compiles the code:

```
$ tsc
```

This command compiles all TypeScript files in the application based on the configuration provided in **tsconfig.json**. It includes compiler options with `'rootDir'` set to `'src'` that specifying the root directory of input files and `'outDir'` set to `'dist'` that specifying the output directory for compiled files.

After successful compilation, TypeScript generates equivalent JavaScript files in the specified output directory **dist**.

Now we update **package.json** as follows, add scripts for **build**, **start** and **test** so it will be easy to run with npm.

```
"scripts": {  
  "test": "mocha --require ts-node/register  
  'src/**/*.spec.ts'",  
  "start": "node dist/src/main.js",  
  "build": "tsc"  
}
```

After defined scripts in **package.json** execute it with npm run in the terminal. For example, **npm run build**, **npm run start** or **npm run test**. Here, we run application with **js** after compilation but while developing we can use **tsc --watch**.

In the advanced phase of application building, we will develop a script that generates a compressed binary file for the application. This file will encapsulate the JavaScript source code with obfuscation and handle the installation of necessary Node packages.

Create **mkpackage.sh** file with following code in root directory of application.

```
npm install  
./node_modules/.bin/tsc  
rm -rf binaries/*  
mkdir -p build  
mkdir -p binaries  
rm -rf build/*  
node tools/obfuscate.js  
cd build  
echo "Getting git version info.."  
export VER=1.0  
echo exports.version=\"1.0\" > ver  
echo exports.version_long=\"$VER\" >> ver  
echo "Copying things.."  
cp ../package.json .
```

```
echo "Doing compression .. "
tar czf pms_be_${VER}.tgz *.js package.json components routes
tests utils
mv pms_be_${VER}.tgz ..../binaries/.
cd ..
echo "Created file pms_be_${VER}.tgz"
```

This script has a set of commands for building and packaging a Node.js application. Let us break down each part:

- **npm install**:
Install the Node.js dependencies specified in the `package.json` file.
- **./node_modules/.bin/tsc**:
Invokes the TypeScript compiler (`tsc`) located in the `node_modules` directory to transpile TypeScript code into JavaScript. The `./node_modules/.bin/` prefix is used to run the locally installed TypeScript compiler.
- **rm -rf binaries**:
Removes all files and subdirectories from the `binaries` directory.
- **mkdir -p build** and **mkdir -p binaries**:
Creates the `build` and `binaries` directories if they do not exist.
- **rm -rf build**:
Clears all files and subdirectories from the `build` directory.
- **node tools/obfuscate.js**:
Executes a Node.js script located at `tools/obfuscate.js`. This script likely performs obfuscation of JavaScript code. The details of this script are not provided, but it seems to be a custom script for obfuscating the application's JavaScript source code.
- **cd build**:
Changes the current working directory to the `build` directory.
- **echo "Getting git version info.."**:
Outputs a message indicating that the script is retrieving Git version information.
- **export VER=1.0**:
Exports the variable `VER` with the value `1.0` for use in subsequent commands.

Sets an environment variable named `VER` to the value `1.0`.

- `**`echo exports.version=\"1.0\" > ver` and `echo exports.version_long=\"$VER\" >> ver`:**`

Creates a file named `ver` with version information. It exports the application version as well as a long version string.

- `**`echo "Copying things.."`:**`

Outputs a message indicating that files are being copied.

- `**`cp ../package.json .`:**`

Copies the `package.json` file from the parent directory (`..`) to the current (`build`) directory.

- `**`echo "Doing compression .. "`:**`

Outputs a message indicating that compression is in progress.

- `**`tar czf pms_be_$VER.tgz *.js package.json components routes tests utils`:**`

Creates a compressed tarball (`pms_be_\$VER.tgz`) containing specific files and directories (JavaScript files, `package.json`, `components`, `routes`, `tests`, `utils`).

- `**`mv pms_be_$VER.tgz ../binaries/.`:**`

Moves the created tarball to the `binaries` directory.

- `**`cd ..`:**`

Changes the working directory back to the parent directory.

- `**`echo "Created file pms_be_$VER.tgz"`:**`

Outputs a message indicating the successful creation of the tarball.

This script seems to automate various tasks involved in building and packaging a Node.js application. It handles dependency installation, TypeScript compilation, obfuscation, versioning, and compression into a tarball for distribution. The resulting tarball is stored in the `binaries` directory with a name reflecting the application version.

This binary file will be deployed on the server further.

Deploying the Application

There are different types of servers used to deploy applications and here we will use the most recommended way to deploy node application on AWS instance. Deploying a Node.js application on Amazon Web Services (AWS) offers several advantages, making it a popular choice for many developers and businesses. AWS has a rich ecosystem of services that complement Node.js application development.

AWS Server Setup

Creating an Amazon EC2 instance for hosting a Node.js application involves several steps. Following is a step-by-step guide on how to create an EC2 instance and deploy a Node.js application on it.

Signing in to the AWS Management Console

Amazon provides a free tier account for one year so anyone can create that account and use different services which are free. If anyone has that account, you can directly login, or else create an account and Sign in with your AWS account credentials.

Once you sign in, choose a region that is geographically closer to your users to reduce latency and improve the responsiveness of your application.

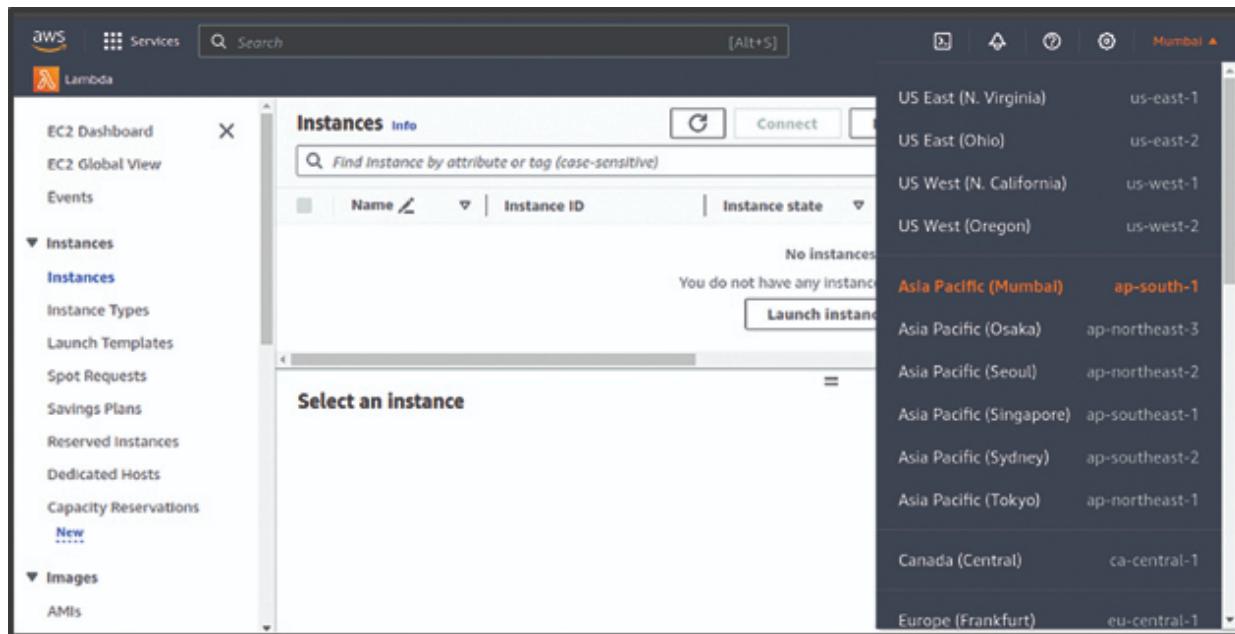


Figure 10.1: Choose Region in AWS

Here, we are choosing the Asia Pacific (Mumbai) Region.

[Navigating to EC2](#)

In the AWS Management Console, navigate to the "**Services**" dropdown. Under the "**Compute**" section, select "**EC2**."

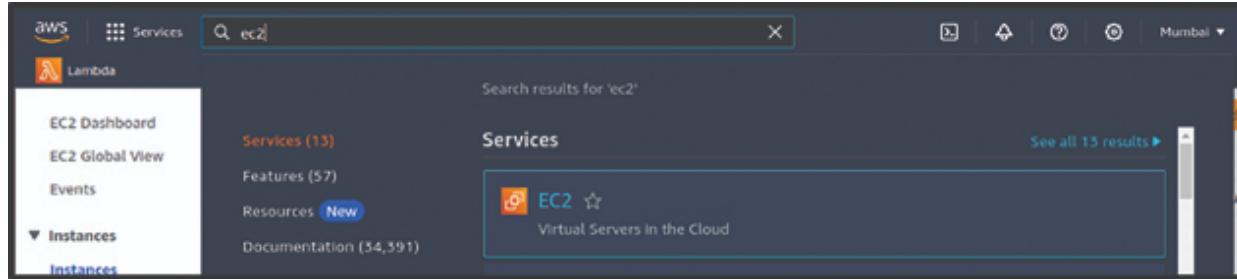


Figure 10.2: Select EC2 Instance

[Choosing an Amazon Machine Image \(AMI\)](#)

Select an Amazon Machine Image (AMI) based on your requirements. For a basic Node.js application, you can choose an Amazon Linux AMI. Select the desired instance type based on your application's resource needs. The default option is often suitable for small applications.

When you select an AMI, consider the following requirements you might have for the instances that you want to launch.

For a small Node.js application, you typically do not need a high-powered or expensive EC2 instance. You can choose a cost-effective instance type that meets the requirements of your application. Here are a few EC2 instance types that are suitable for small Node.js applications:

- **Memory Requirements:** Choose an instance type that provides enough memory for your Node.js application. The `t3.micro` and `t2.micro` instances, for example, come with 1 GB of memory.
- **CPU Requirements:** For small applications, a burstable performance instance like `t3.micro` may be sufficient. If you have specific CPU requirements, consider other instance types.
- **Storage:** Determine the storage capacity needed for your application. The instance types mentioned above come with Elastic Block Store (EBS) storage, and you can adjust the size based on your requirements.

- **Network Performance:** For small applications, the default network performance of these instances should be sufficient. If you expect high network traffic, you might need to consider higher-performance instances.

For example:

- **t3.micro:** This is a low-cost, general-purpose burstable instance type. It is suitable for applications with variable workloads that do not require sustained high CPU performance.
- **t2.micro:** Similar to the **t3.micro**, the **t2.micro** is a low-cost, burstable instance type. It is a previous generation instance, but it can still be suitable for small applications with light workloads.

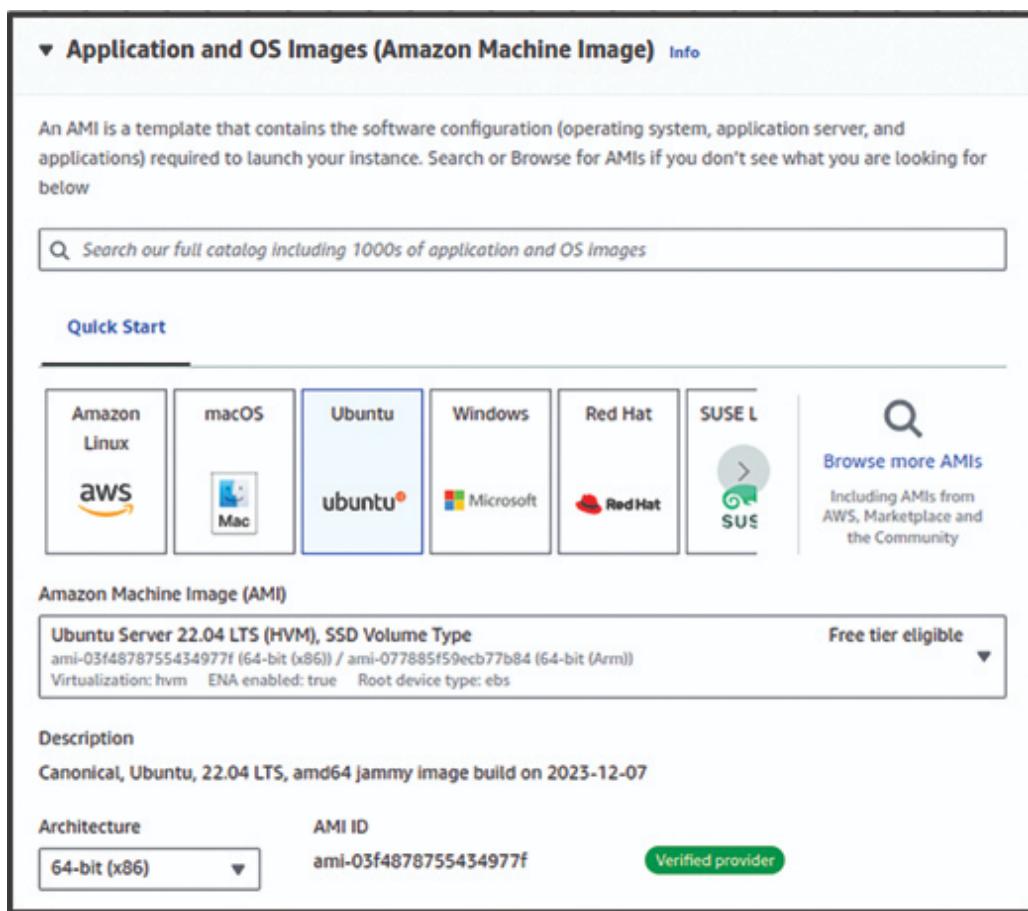


Figure 10.3: Select AMI

Keep in mind that AWS offers a Free Tier, allowing you to use a limited amount of resources, including **t2.micro** instances, at no cost for the first 12 months, so here we are choosing **t2.micro**.

Key Pair Generation

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance. A key pair is a set of security credentials that consists of a private key and a public key.

Key pairs are used for secure access to your Amazon EC2 instances. When you launch an EC2 instance, you specify a key pair, and the public key is placed on the instance while the private key is kept secure. Click "**Create Key Pair**," give it a name, and download the private key (.pem) file.

Once you download a .pem file, give permission to that file from the terminal.

```
$ sudo chmod 400 PMS_KEY.pem
```

Using a key pair (.pem) file is crucial for secure access to your EC2 instances. Ensure that you follow best practices for key management and security. If you lose your private key, you might lose access to the instances associated with that key pair. Always store your private key securely and avoid sharing it with unauthorized users.

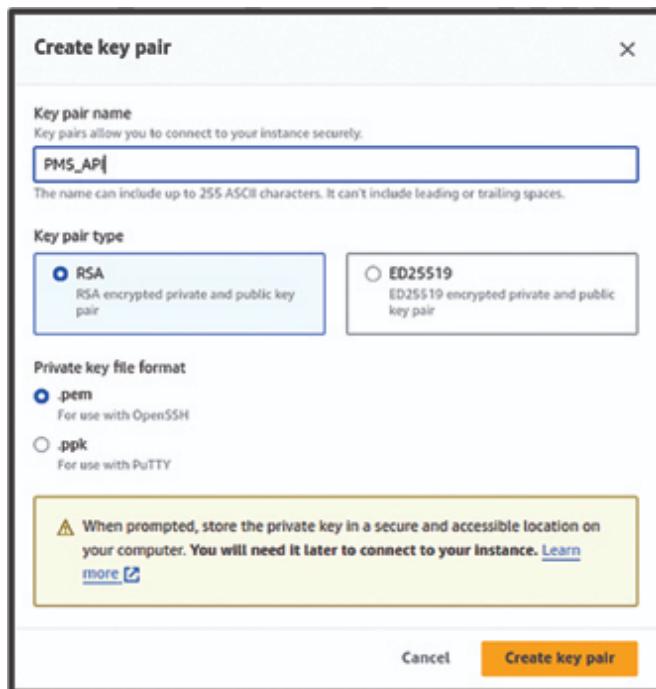


Figure 10.4: Key Pair Generate

Network Settings

Configure security groups to control inbound and outbound traffic to your instance.

Configuring rules allows SSH access for administration and HTTP/HTTPS access for your web application. Always follow security best practices and restrict access to only the necessary ports and IP ranges as follows.

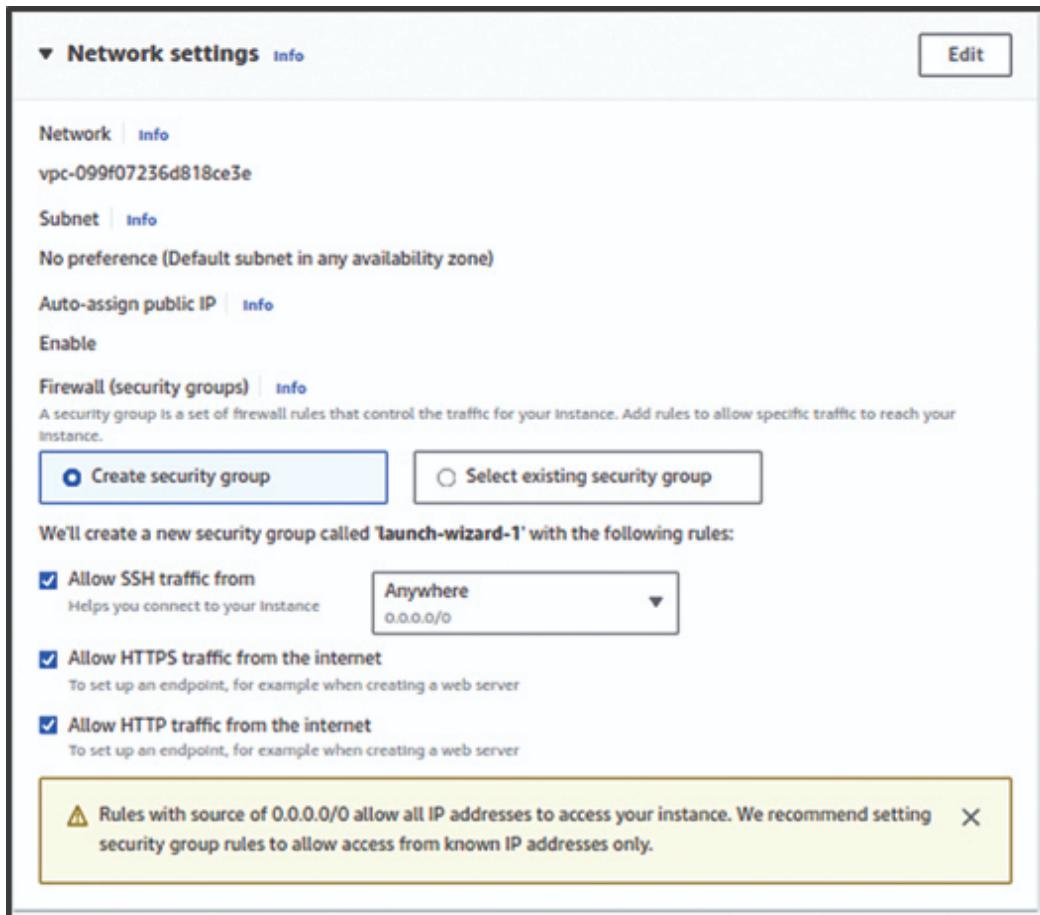


Figure 10.5: Network Settings

Configuring Storage

In the "Add Storage" step, you can configure the storage settings.

Root Volume: This is the root volume where the operating system is installed. You can specify the size (in GiB) and choose the storage type (for example, General Purpose SSD, Provisioned IOPS SSD, Magnetic).

Add New Volume: You can add additional volumes if needed. Click "Next" when done.

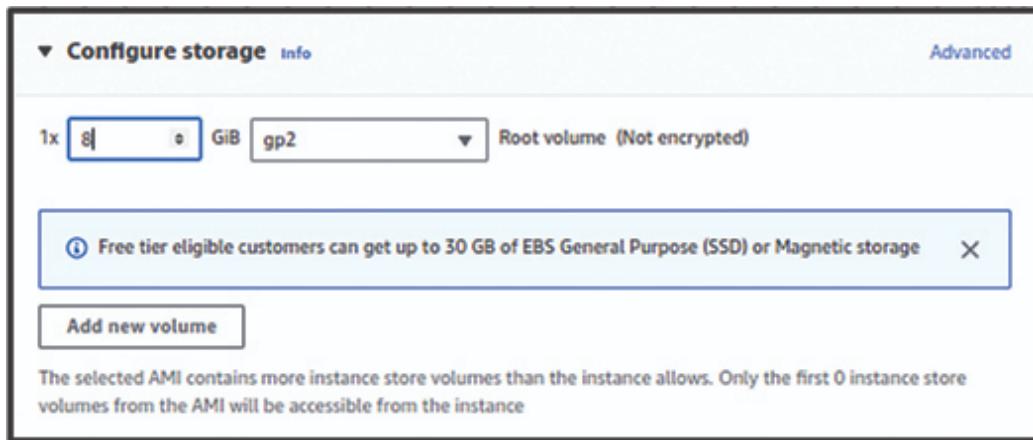


Figure 10.6: Configure Storage

Here, we are selecting 8GB storage for Node.js applications, so no additional volume.

Launching an Instance

Configure instance details such as the number of instances, network settings, and storage. The default settings are typically sufficient for a basic Node.js application.

Configure storage settings based on your application's data requirements. The default settings are usually suitable for simple applications. Review your configuration settings to ensure they are correct. Click the "Launch" button.

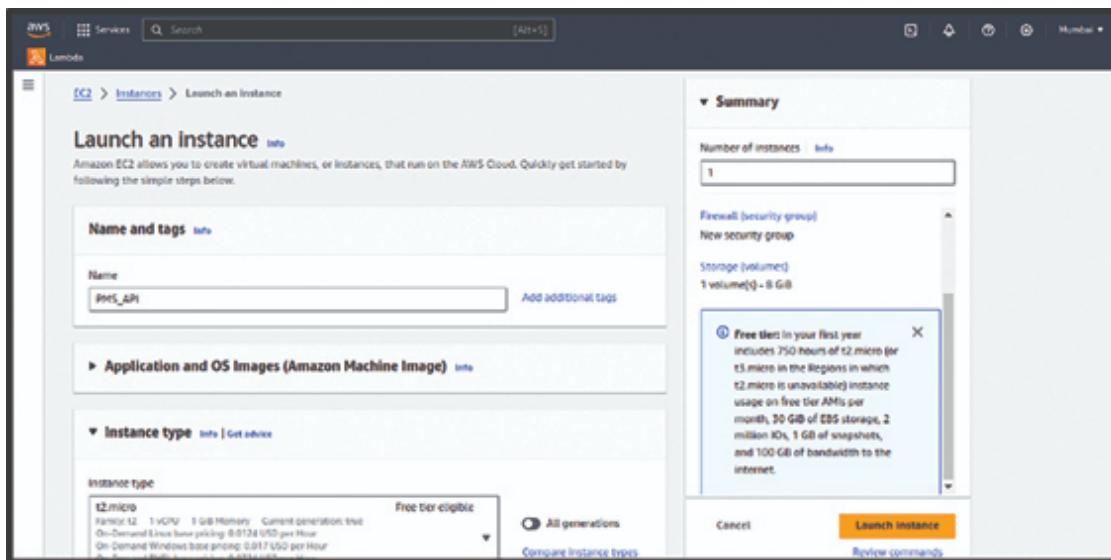


Figure 10.7: EC2 Launch Instance

On successfully launching an instance you can view a list of instances with state Running.

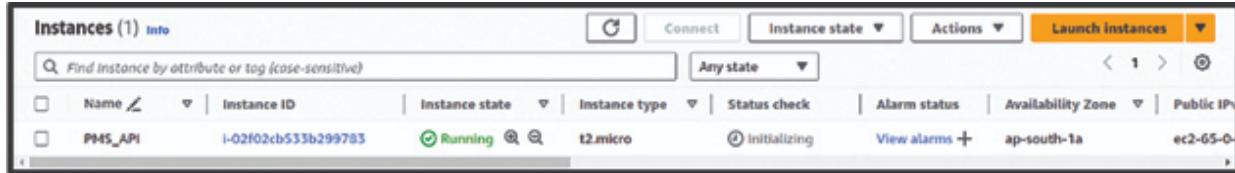


Figure 10.8: Running Instance

Clicking the Instance Id reveals all details such as public and private IP address, hostname, platform details. You can also edit the instance, if needed.

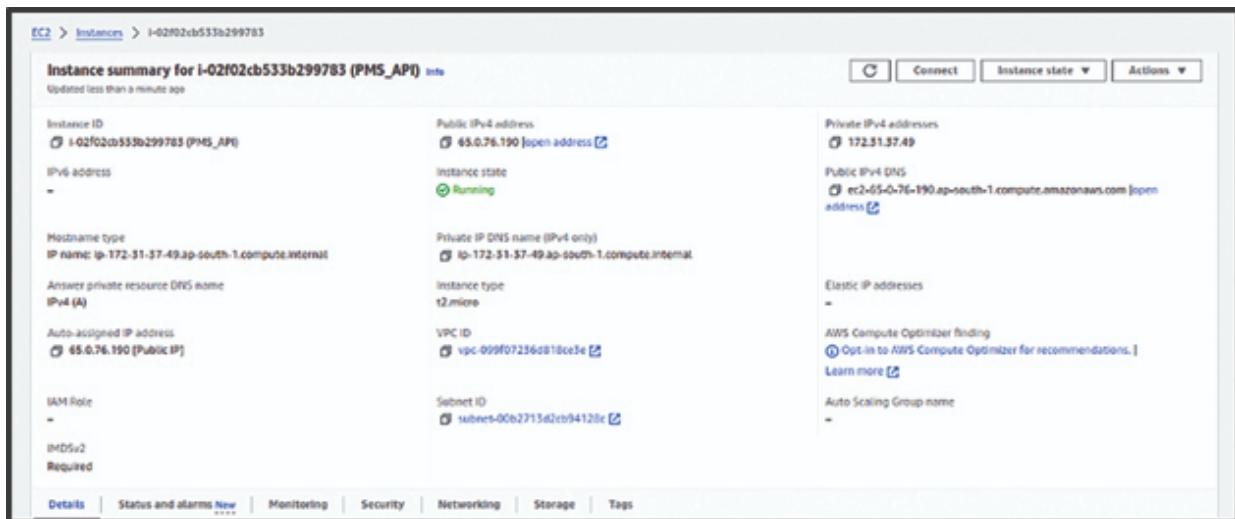


Figure 10.9: Instance Details

Now, it is time to connect that server either from **aws** directly or from **ssh** as described previously. Click the connect button to initiate the connection.

Connecting the Server

We have already saved the .pem file to connect to the server through ssh. Open the terminal in the directory where the .pem file is located and enter the following command:

```
$ sudo ssh -i YourKeyName.pem ec2-user@YourPublicIPAddress
```

Replace **YourKeyName.pem** with the path to your private key file and **YourPublicIPAddress** with the public IP address of your EC2 instance.

For example, \$ sudo ssh -i PMS_KEY.pem ubuntu@65.0.76.190

On successfully connecting with the server, you find the following output on terminal.

```
yammini@atpl-lap-166:~/projects/book/kriti$ sudo ssh -i PMS_KEY.pem ubuntu@65.0.76.190
[sudo] password for yammini:
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 6.2.0-1017-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 System information as of Sat Jan 27 17:20:39 UTC 2024

 System load:  0.0          Processes:           103
 Usage of /:   20.8% of 7.57GB  Users logged in:    1
 Memory usage: 22%          IPv4 address for eth0: 172.31.37.49
 Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Sat Jan 27 16:39:01 2024 from 103.81.92.145
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-37-49:~$
```

Figure 10.10: Connected Server On Terminal

Deploying Code on Server

Upon establishing a successful connection to the server, proceed to install essential packages like Node.js and PostgreSQL. Adjust the **server_config.js** file to reflect the appropriate configurations for port, database, and email settings.

Since we are using Linux (Ubuntu) here, the steps to install Node.js can be recalled from [Chapter 1, Introduction to Node.js](#). For PostgreSQL, the official website - <https://www.postgresql.org/> can be followed for installation as also mentioned in [Chapter 4, Planning the App](#).

Additionally, ensure that the specified port, for instance, **8080**, is open in the AWS security group by configuring the inbound rules to accommodate the requirements of our Node.js application.

pm2 is a process manager for Node.js applications that allows you to manage and deploy Node.js applications in a production environment. It provides various features, such as process monitoring, automatic restarts, and clustering, to ensure the reliability and performance of your applications.

Install **pm2** on the EC2 instance and run the following command in the terminal.

```
$npm install -g pm2
```

Let us create a **deploy.sh** script with the following code in the root directory of the project. This script facilitates the transfer of binary code from our local server to the pre-built remote server and executes specific commands to launch the application on the server.

```
#!/bin/bash

# Define SSH key and other deployment details
SSH_KEY="..../PMS_KEY.pem"
USER="ubuntu"
HOST="43.205.144.240"
PROJECT_DIR="/home/ubuntu/PMS"
HOME_DIR="/home/ubuntu"
CONFIG_FILE="server_config.json"

process_pms_server(){
    echo "----"
    rm binaries/*
    sh mpackage.sh
    echo ""
    echo "File created - "
    echo "binaries/"
    search_dir=binaries
    for entry in "$search_dir"/*.tar.gz
    do
        echo "- \t$entry"
    done
    echo ""

    echo "==> Removing SensorApp Backend Server . <=="
    echo "=====
```

```

ssh -i "$SSH_KEY" $USER@$HOST "rm $HOME_DIR/pms_be*.tgz ;
mkdir -p PMS; chmod +w $PROJECT_DIR;"

echo "==> Transferring SensorApp Backend Server . <=="
echo "=====
scp -i "$SSH_KEY" binaries/pms_be*.tgz $USER@$HOST:$HOME_DIR/
|| exit 1
scp -i "$SSH_KEY" server_config.json $USER@$HOST:$HOME_DIR || exit 1
echo "==> Extracting PMS Backend Server. <=="
echo "=====
# Commands to be executed on the remote server
commands=(
  "tar xf pms_be*.tgz -C $PROJECT_DIR;"
  "cd $PROJECT_DIR;"
  "npm install;"
  "pm2 restart main.js --name pms-api;"
  "pm2 save;"
)
ssh -i "$SSH_KEY" $USER@$HOST "${commands[*]}"
}

echo "Do you wish to deploy PMS Backend Server?"
select yn in "Yes" "No"; do
  case $yn in
    Yes)
      process_pms_server
      break;;
    No)
      echo "Deployment cancelled."
      break;;
    *)
      echo "Invalid option. Please choose 1 for Yes or 2 for No."
      ;;
  esac
done
echo ""
echo ""

```

```
echo "====="
echo "==> The Deployment finished <=="
echo "=====
```

- **#!/bin/bash:** Specifies that the script should be interpreted using Bash.
- **Configuration:** Define variables for SSH key, user, host, directories, and configuration file.
- **Function - process_sensorapp_backend_server:** Removes existing binary files. Executes the script (**mkpackage.sh**) to create the binary. Displays the created files in the binaries directory.
Removes existing server files, creates directories, and sets permissions on the remote server.
Transfers binary files and the configuration file to the remote server.
Extracts and deploys the PMS Backend Server on the remote server using pm2.
- **User Confirmation:** Prompts the user for deployment confirmation using a select statement.
- **Completion Message:** Displays a message indicating the completion of the deployment process.

This script automates the deployment process of a Node.js application to a remote server using **pm2** for process management.

Run the script from the terminal with the root directory of the project.

```
$ ./deploy.sh
```

```

* yammini@atpl-lap-166:~/projects/book/kriti/pms-be$ ./deploy.sh
Do you wish to deploy PMS Backend Server?
1) Yes
2) No
#? 1
----

up to date, audited 529 packages in 1s

80 packages are looking for funding
  run `npm fund` for details

4 moderate severity vulnerabilities

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
Getting git version info..
Copying things..
Doing compression ..
Created file pms_be_1.0.tgz

File created -
binaries/
- \tbinaries/*.tar.gz

==> Removing SensorApp Backend Server . <==
=====
==> Transferring SensorApp Backend Server . <==
=====
pms_be_1.0.tgz                                100%   51KB  1.0MB/s  00:00
server_config.json                            100%  402   23.2KB/s  00:00
==> Extracting PMS Backend Server. <==
=====

up to date, audited 497 packages in 2s

80 packages are looking for funding
  run `npm fund` for details

3 moderate severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
Use --update-env to update environment variables
[PM2] Applying action restartProcessId on app [main.js](ids: [ 0 ])
[PM2] [pms-api](0) ✓
[PM2] Saving current process list...
[PM2] Successfully saved in /home/ubuntu/.pm2/dump.pm2

=====

==> The Deployment finished <=
=====
```

Figure 10.11: Output of Deployment Script

Using **pm2** logs you can monitor the application. Upon successful execution, the application will display the following output:

\$ pm2 logs

```
/home/ubuntu/.pm2/logs/pms-api-out.log last 15 lines:  
0|pms-api | Router : RoleRoutes - Connected  
0|pms-api | Router : UserRoutes - Connected  
0|pms-api | Router : ProjectRoutes - Connected  
0|pms-api | Router : TaskRoutes - Connected  
0|pms-api | Router : CommentRoutes - Connected  
0|pms-api | Server Routes started for server  
0|pms-api | Connected to the database  
0|pms-api | Server is running on port 8080 with pid = 95988  
0|pms-api | Connected to the database  
0|pms-api | Connected to the database
```

Other Methods For Deployment

Apart from deploying on AWS EC2 instances as we followed in this chapter, there are many other ways to deploy a Node.Js application.

Amazon has other offerings for deployment such as AWS Lambda, AWS Elastic Beanstalk, and Amazon ECS/EKS. AWS lambda is ideal for event driven or microservices architecture applications. If you do not want to be worried about the underlying infrastructure and want to focus on just your application, Elastic Beanstalk is a great choice. It handles the deployment, load balancing, and auto scaling automatically.

Docker containers can also be used to containerize your application and deploy with help of Amazon ECS and EKS.

Similar to Amazon, Google Cloud Platform (GCP) also offers Google App Engine, Kubernetes Engine, and Cloud functions. Microsoft Azure offers Azure App Service, Azure Functions, and Azure Kubernetes Service (AKS).

Heroku is another cloud platform which offers straightforward deployment of Node.js applications. It can automatically detect that your application is a Node.js application and will install the dependencies using NPM.

Conclusion

In this chapter, we delve into making our application code secure using obfuscation, construction, and deployment of a Node.js application, employing the widely recommended processes and adopting a straightforward approach to AWS integration. This involves manual deployment on AWS, but you have the option to explore various methods, including Jenkins, CI/CD, Git pipelines, AWS CodeBuild, and pipelines, among others. These options encompass various DevOps tasks, so we are not enhancing more here.

In summary, after the development phase, it is essential to ensure that your application reaches end-users. Deployment plays a crucial role in this process, as without it, everything remains in a state of readiness but lacks actual utilization.

The next chapter, “*The Journey Ahead*” is the last chapter of the book. We will briefly discuss what we learned and what can be done further.

Multiple Choice Questions

1. What is code obfuscation?
 - a. Encryption technique for securing data.
 - b. Process of transforming source code to make it harder to understand.
 - c. Method for optimizing code performance.
 - d. Technique for compressing code files.
2. What are Anti-debugging techniques ?
 - a. Embedding code for detecting debugging attempts.
 - b. Changing representation of numerical constants.
 - c. Splitting functions into smaller pieces.
 - d. Compressing the overall code size.
3. What TypeScript is primarily used for in Node.js development?
 - a. Database management.
 - b. Code obfuscation.

- c. Server-side scripting.
 - d. Browser compatibility.
4. What is the purpose of the command `tar czf pms_be_$VER.tgz *.js package.json components routes tests utils?`
- a. It installs npm packages.
 - b. It compresses specific files and directories into a tarball.
 - c. It initializes a new Node.js project.
 - d. It extracts files from a tarball.
5. Which npm script is commonly used to run a Node.js application after TypeScript compilation?
- a. `npm run build`
 - b. `npm start`
 - c. `npm install`
 - d. `npm run compile`
6. What is the significance of the variable `$VER` in the command?
- a. It represents the version of Node.js.
 - b. It is used for encryption purposes.
 - c. It is a placeholder for the actual version number.
 - d. It specifies the compression format.
7. What is the purpose of an EC2 instance in AWS?
- a. To store data in the cloud.
 - b. To manage DNS records.
 - c. To deploy and run applications on virtual servers.
 - d. To route traffic to different services.
8. Which file is used as the SSH key for connecting to the remote server in the script?
- a. `PMS_KEY.pem`
 - b. `deploy.sh`
 - c. `server_config.json`

- d. `mkpackage.sh`
9. What does the `process_pms_server` function in the script do?
- a. Installs Node.js packages.
 - b. Deploys the Node.js application.
 - c. Configures the server's security group.
 - d. Uninstalls Node.js from the server.
10. How does **PM2** help in managing Node.js applications?
- a. It automates the installation of Node.js.
 - b. It monitors and manages the processes of Node.js applications.
 - c. It provides a graphical user interface for Node.js development.
 - d. It facilitates code profiling in Node.js.

Answers

- 1. b
- 2. a
- 3. c
- 4. b
- 5. b
- 6. c
- 7. c
- 8. a
- 9. b
- 10. b

Further Reading

<https://www.npmjs.com/package/pm2>

<https://docs.aws.amazon.com/ec2/>

<https://www.jenkins.io/>

<https://www.heroku.com/>

<https://cloud.google.com>

<https://azure.microsoft.com/en-us>

<https://www.npmjs.com/package/javascript-obfuscator>

OceanofPDF.com

CHAPTER 11

The Journey Ahead

Introduction

“The only way to discover the limits of the possible is to go beyond them into the impossible.” – Arthur C. Clarke

Throughout the book, we learned many important concepts and methods of utilizing Node.Js and Express.Js for the API needed for a project management software. With the help of this single example, we designed the necessary APIs for users, projects, tasks, and more. If you followed the chapters and did your own version of APIs, you should have a functioning backend of a PMS.

There is a lot more which can be done, but it cannot be covered in a single book alone. This chapter will try to cover what can be done further, along with best practices which can be followed. In executing all that, you should have pretty decent software at hand.

The journey has just begun.

Structure

In this chapter, we will discuss the following topics:

- The Story so Far
- Next Steps
- Staying Ahead
- Further Readings
- Wrapping Up

The Story So far

Recalling the Node.js demonstration at JSConf 2009 by Ryan Dahl, we began learning about Node.js. Starting from the basics of Node.js and its

beautiful way of working with Event Loop, we learned where it can be used and also discussed some pros and cons.

We learned the installation of Node.js on multiple platforms and created the basic HTTP and HTTPS servers with and without the module cluster. Cluster module helps to harness the capability of the machine and significantly improves performance.

People had been writing the code of Node.js applications in JavaScript the usual way, but then Microsoft released TypeScript in 2012. Being open source, cross-platform, and supporting object oriented programming with its strong type checking made developers' life easier by detecting bugs and mistakes. Therefore, developers started adapting to TypeScript.

Keeping this in mind, [Chapter 2, “Introduction to TypeScript”](#) focused on covering the basics of TypeScript. After the necessary packages installation, a basic application and some tiny examples helped us understand the key concepts of the language.

Next, we learned about Express.js and how it could be used to create API endpoints easily. We installed necessary packages of Express.js, and gained familiarity with the core concepts, along with the pros and cons.

Understanding Node.js applications in Express.js through discrete examples is feasible, but developing a cohesive application provides deeper insight into application planning, design, and execution. Hence we started writing one application to understand it all. In [Chapter 4, Planning the App](#), we started with the planning of the application — a project management system. We did a setup of the project, necessary dependencies, directory structure, and also created the needed database tables along with routes skeleton.

We laid the foundation so that we can begin building the API, a project management software would require.

For any enterprise application, whether it is only backend API or full stack, access management is a must have feature. That is what we did too — by creating APIs for user management, and further enhanced the chapter with a ‘forgot password’ feature. We created a token based authentication system for which we used the `jsonwebtoken` package.

Once we had the users in the system, we began writing the project and task module in [Chapter 6, “REST API for Project and Task Modules”](#). We

covered the basic crud operations for projects and tasks entities, along with assigning to users.

[Chapter 7, “API Caching”](#) focused on caching that is crucial for performance enhancement in high-traffic applications. We implemented Redis for data caching and developed a Cache Util to manage Redis interactions.

In web applications where communication is the key, integrating a notification module is essential. We not only implemented this feature in [Chapter 8, “Notification Module”](#) but also utilized Redis Queues to enhance its efficiency. As an example of using the notification system, we modified the code to automatically send notifications to relevant project members whenever a new task was created.

Needless to say how important unit testing is, we learned the basics of unit testing in [Chapter 9, “Testing API”](#). After necessary configuration, we used the Mocha framework along with Chai to write unit tests.

For the final touch, we delved into a crucial aspect with [Chapter 10, “Building and Deploying Application”](#) while covering a notable concept of code obfuscation.

The journey so far has been an extensive exploration of building a robust application with Node.js and Express.js, covering everything from basics to more advanced topics like caching, unit testing and deployment strategies. This should equip you with the skills and knowledge to create, optimize and deploy efficient, scalable web applications.

Next Steps

There is only a small part of “*what’s possible*” that could be covered in this book. Depending on the context, further development can be achieved. In this section, we will enlist some of the features which can be developed to enhance the application further.

The points being discussed can be thought of — things that can be added further, and the best practices on how it can be done. The following points not only cover what could enhance the application but also the ways to make it scalable and maintainable.

FrontEnd

A project management software is usually a web-application. The days of desktop applications are over. The time now is of collaboration, communication, and most importantly of availability. A web application can be available anywhere in the world. It can be accessed through a laptop, or a mobile phone, and that serves the purpose. Development of the front end was not in the scope of the book. However, let us discuss some aspects of how to get it done.

There are many popular frameworks at the time of writing this book — React, Angular, Vue.js. All of these are capable of making it a dynamic and responsive user interface. Additionally, Bootstrap can help with a uniform and responsive styling.

A project management application could have the following pages in user interface:

- **Dashboard:** To show the project activities, summary of assigned tasks, quick actions such as creating a new task, new project, and more. The counts of all projects, assigned projects, tasks assigned could also be shown as big numbers.
- **Projects:** All of the assigned projects can be listed as a table. Some buttons to perform some actions such as adding a new project, managing project memberships can also be shown for the project manager roles. For each project in the list, users should be able to open the list of all issues, list of outstanding issues, and more. The options can be customized to open the list of issues in testing for QA team members.
- **Users:** Admins can be facilitated with a page to manage users.
- **A User profile Icon:** An icon on the top right which opens the user profile where a user can manage his/her own profile, for example, name, timezone settings, password, notification settings, and more.
- **Reports:** For project managers it could be a useful page to analyze the project status.
- **Server Status:** A page for admin, where current CPU, memory and storage status can be shown.

There could be more, depending on the requirements and scale of the project.

Reporting

Reporting is a crucial aspect of the project management application. It might look like a huge task, but it is not so difficult to implement.

It is important to analyze what kind of reports we are seeking. A set of simple time-series showing the number of active and closed issues to show the progress of the project could be useful. In the case of Agile development, showing project progress as velocity could be important.

A pie chart of the number of issues in each phase— backlog, development, testing, and closed can also be shown. A report of the number of opened issues versus number of closed issues is also a good performance index for a project.

Reporting would need key changes at front end and back end as well. It would be wise to create a dedicated module at the back end to generate reports. Some reports could be generated through a scheduler (for example, generating reports at midnight everyday) to persist in order to avoid doing repetitive processing for every request. Some reports would be on-request and real-time. Abstraction of the reporting module will provide clean implementation.

A data visualization library will be needed to generate reports with charts. Open-source libraries like **D3.js**, **Chart.js**, **Plotly**, **Vis**, and others can be a good option. If cost is not an issue, **Highcharts** and **amCharts** are among the top tools that can be tried.

Applying Machine Learning

There are many opportunities to apply machine learning in a web-application. If you track the user traffic, a surge detection in traffic could help to maintain high-availability by taking appropriate action at the right time.

By tracking user activities and applying machine learning to it, a tailored user-interface could be provided to users for a more engaging application.

Tracking user activities can also help to analyze which actions are taken or which pages are visited more than others. Back end operations such as caching, database queries, and others could be optimized for top actions.

A prediction could be done for project completion date, or sprint performance in case of Agile development. There are numerous such applications which can be done.

Server Monitoring

As the traffic grows, the resource usage grows and it is vital to keep track of the CPU, RAM, and storage of the servers along with database and network. There are ways in which the server can be monitored.

- **Using built-in operating system tools:** Most operating systems, specially servers where applications are usually deployed, offer tools for monitoring resources such as **top**, **htop**, **vmstat**, and so on, for Linux.
- **A separate module inside the application:** The application could be modified to create a module which periodically tracks and stores the resource usage in the database. This data can be shown, visualized in the front end. A separate page could be created. Additionally, alerts can be implemented whenever a resource usage crosses the set threshold. For example, if available RAM is 16 GB, then a threshold can be set to 12 GB. Whenever RAM usage crosses 12 GB, a notification alert can be sent to Admin to take necessary actions.
- **Custom scripts:** Python, Bash, or other languages can also be used to write scripts which can monitor the resource usage of machine and server externally and it can be integrated with an alerting system.
- **Other tools:** Prometheus, complemented with Grafana can be used to see real time statistics of the servers.

Depending on the deployment other tools can be of help. For cloud deployment, there are cloud provider specific tools, for example, AWS Cloudwatch, Azure Monitor, and Google Cloud operations suite.

Security Features

As of now, the application uses a token system to provide authentication and authorization for users. We created `access_token` and `refresh_token` using `jsonwebtoken`. Front end could use the `access_token` for every request and if `access_token` expires, and `refresh_token` is available, it could be used to generate a new `access_token`.

This could be improved with other grant types and features necessary for `OAuth2` implementation.

SSL/TLS Encryption

Necessary support for SSL should be there. This means, if we need to host the application with a url such as `HTTPS://`, it should be possible, although there can be strategies to implement this. One is to make the application configurable to choose between `HTTPS` and `HTTP`. Another one is to use `nginx` in front and use SSL certificates with help of a Certification Authority (CA). Let us Encrypt is a free service and is widely recognized. It is easy to set up and provides a necessary level of encryption for secure communication.

Social Media Login

In case the application is going to be in public domain, it could be nice to implement login via social media platforms such as Google, Facebook, Twitter, and so on, to provide a seamless user registration and login.

Two-Factor Authentication

Every now and then, we see the passwords and other details being hacked from websites with less secure infrastructure. It becomes indispensable to implement Two-Factor Authentication (2FA) to add an additional layer of security.

LDAP Integration

If the application is deployed as a private to an organization, and if the organization has their own LDAP server in place, it could be great to make an application integrated with LDAP for user management. It can streamline the authentication and authorization process within the application.

Container-Based Deployments

Sometimes, it is necessary to deploy the application as containers. Containerization with Docker and its orchestration with help of Kubernetes is the way to go. Together they provide a smooth and scalable deployment.

To enable containerized deployment, you need to create:

- **Dockerfile:** a Dockerfile is needed by Docker for specifying the base image (linux flavor and version needed for running the application), dependencies, variables, commands, and so on.
- **docker-compose.yml:** This is a YAML file that defines how the docker container should behave in production. This is used with Docker Compose, which is a tool for running the multi-container docker applications. With this file, we can also configure the application's service.
- **Kubernetes configs:** We need to write Manifests for deployment, ConfigMaps, and other files to configure Kubernetes to orchestrate the Docker Containers.

Usually when there is a need for this expertise, there are dedicated teams to do this job within the organization. In case you are willing to give it a try, installation of docker, writing a simple Dockerfile, and learning basic commands to run the container is the starting point.

Swagger UI

Since the application currently in place offers APIs, it is important to provide API testing and documentation. Swagger UI provides a web-based user interface where users can see the API endpoints, access the documentation associated with each endpoint, and make a call to it directly from the browser.

Swagger UI is a tool provided by <https://swagger.io>. It is popular due to its ease of use, one place to test endpoints and access documentation, security in place and being real-time and accurate.

It is not something to be shown to website users, for example, project managers and members since they do not need to know the underlying

endpoints. However, for developers and QA team members it is an essential tool.

Staying Ahead

At the time of writing, there are many Large Language Models (LLMs) available, for example, OpenAI's ChatGPT and Google's Gemini. This is not traditional machine learning. This is an era of Generative Artificial Intelligence. AI is not just a tool to predict a value, for example, a house price, or to classify something into categories anymore. AI can also generate content for us. Given a context, AI can provide a paragraph, a page, a book in no time. AI can find errors in a code. Impressively, it can write code too. GitHub's co-pilot is capable of providing in-place code snippets relevant to your application. AI can teach you a topic if you are willing to.

Keeping in mind all of this, it is important to think and pen down how Generative AI can help. Considering our prime example, Project Management System, we could give a basic idea of what we are trying to manage and AI could help with creating the tasks for us. If we provide the complete Software Requirement Specification (SRS) to the AI, it can generate all of the details of tasks as well.

Generative AI can be used in the following key areas (keeping it focused to the PMS):

- **Project planning** : If provided input goals such as what are we trying to achieve with the project, number of resources, timelines and constraints, GenAI can help with a comprehensive project plan, sprints, or milestone planning.
- **Risk Assessment** : If the project status does not meet the milestone timelines, a risk assessment can be provided.
- **Resource Allocation**: If the resources are to be picked from a pool, given the history of resources work in other projects, a decision could be made for picking up the right resource for the project.
- **Preparing Reports**: Based on the current status of the project, automated reports can be generated.

- **Task Prioritization:** Tasks can be given priority at the beginning of the project and the priority can be changed, if needed, based on the project status.
- **Predictive Analysis:** Based on the tasks complexity, status and in which phase it is, milestones status, a project end date of the next milestone date can be predicted.
- **Preparing Lessons Learned:** At the end of the project, a list of lessons learned can be prepared to avoid making mistakes next time.
- **Real-time Sentiment Analysis:** A task is usually commented on by the developers. A real-time sentiment analysis could be done.

There could be a lot more ways GenAI can help.

Further Reading

This book covered Node.js, TypeScript, Express.js, Mocha, Chai, Redis, and some other tools. A book covering many aspects could only touch basic to medium levels of knowledge. For further learning, there are some suggestions for you to try.

Advanced Node.js Development

Topics such as streams, child processes, performance optimizations, and other such advanced topics can be learned further. Moreover, using decorators and design patterns will make your code neat, near zero defect, and maintainable.

Node.js can also be a great choice for applications needing heavy, real-time communication. Technologies such as Socket.io and websockets offer the necessary features.

One popular architecture is microservices. Resources focusing on microservices architectures can be considered for further learning.

Full Stack Development

A browser understands only three things— HTML, CSS, and JavaScript.

Since the underlying language, JavaScript, is the same for back end, for example, Node.js and front end, for example, React, Angular, Vue, and

many more frameworks, getting into the shoes of a full-stack developer becomes rather easier.

After gaining a good understanding of Node.js/TypeScript, getting familiar with React or Angular is a common trend as seen recently.

Test-Driven Development

There are many books that offer guidance on implementing projects using Test-Driven development. The TDD approach, if followed religiously, results in bug free projects. Not only does it allow you to catch the bugs at early stages, but it also lowers the cost of development and maintenance.

In this approach, the test cases are written first and then the code is written in a way that the test would pass. Hence it is vital to write test cases which cover testing for all possible inputs.

There is usually a negative mindset when you talk about TDD. It is mostly about the time and learning needed to write the test cases upfront. However, when studies were performed, overall time and cost of the projects were much less as compared to projects executed without TDD.

Performance Tuning

There would be a point in life when you would have realized that the queries being used in the applications are taking more time than expected. You try to optimize the database using indexes, growing RAM size, and more. However, there is more to database management and optimization than just these tunings.

Application logic is another place where the optimization is done. Finding memory leaks and CPU-hungry logics is critical. The scary part is that developers often forget or do not take care of the optimization at the time of development. Sometimes, stress tests are also following the same path so they do not cover all parts of the code and thus some issues remain unidentified. These issues arise at the production instance, and at that time, it becomes challenging to fix.

Other Topics

Apart from the aforementioned topics, the resources on serverless architecture, DevOps practices, and functional programming in Node.js can also be tried.

Integration of AI/ML, data analysis and visualization libraries is a key to get into a Data Science background.

Another important aspect is the security of your application. Although we have covered topics such as OAuth and JWT, more can be learned on how to secure Node.js applications.

Final Thoughts

As we reach the end of the book, it is important to reflect on the vast landscape that we have traversed. From the basics to building the API powered with authentication, caching and unit tests, we have covered significant ground.

As you progress further, you shall witness the power of JavaScript and will realize why it is becoming the most popular language of all time. The introduction to TypeScript made it more powerful by offering type safety and enhancing the code quality.

Remember that the landscape of web development is ever-changing. Technologies evolve. A decade ago everything was Java, Dot net, and Python. Today JavaScript is the one rocking and being bread and butter for a significant percentage of developers. The developers survey by Stackoverflow.com in 2022 revealed that 67% developers use JavaScript for their work.

As you continue your journey, keep experimenting, keep innovating, keep learning. Most importantly, keep your passion for coding, for learning, for trying new things, alive. After all, the *journey has just begun*.

Index

A

abstract [50](#)
API Application, configuring [308-313](#)
API, test cases
 delete, user [325-328](#)
 login, testing [316-319](#)
 user test, adding [321-325](#)
 user test, list [319-321](#)
Arrow Function, keyword
 new [59](#)
 this [58, 59](#)
AWS Instance
 AMI, utilizing [338](#)
 connection, deploying [345-349](#)
 console, signing [338](#)
 EC2, navigating [338](#)
 key pair, generating [340, 341](#)
 network, setting up [341](#)
 server, connecting [344](#)
 storage, configuring [342](#)

B

Base Controller [153, 154](#)
Base Service [154, 155](#)
bcryptCompare [175](#)
Blocking Scopes, keyword
 const [60, 61](#)
 let [60](#)
body-parser [80](#)

C

Caching [267](#)
Caching, benefits
 cost, effectiveness [270](#)
 load, reducing [269](#)
 network traffic, reducing [270](#)
 performance, improving [269](#)
Caching, disadvantages
 Cache, missing [270](#)
 complexity [271](#)
 data, synchronizing [271](#)

resource, consumption [270](#)
stale data [270](#)
Caching, entities [275-277](#)
Caching Redis, using [271](#)
Callback [62](#)
Code Obfuscation
 about [330](#)
 application, building [334-337](#)
 code, complexity [333, 334](#)
 library, installing [332](#)
 script, directory [332, 333](#)
Code Obfuscation, techniques
 anti-debugging [331](#)
 code, compressing [331](#)
 code, splitting [331](#)
 constant value, using [331](#)
 control flow [331](#)
 dummy code, inserting [331](#)
 string, encrypting [331](#)
 variables, renaming [331](#)

D

Database Designing
 about [106](#)
 comment schema table [110, 111](#)
 project schema table [109](#)
 role schema table [108](#)
 task schema table [109](#)
 user schema table [108](#)
Database Models [128](#)
Database Models, entities
 comment [135-138](#)
 project [131, 132](#)
 role [128, 129](#)
 task [133-135](#)
 user [130, 131](#)
Database, tables
 comment [107](#)
 project [107](#)
 role [107](#)
 task [107](#)
 user [107](#)
data types, categories
 Non-Primitive Data [41](#)
 Primitive data [40](#)

E

EC2 Instance, types

CPU, requirements [339](#)
memory, requirements [339](#)
network, performance [339](#)
storage [339](#)

ECMAScript [57](#)

ECMAScript, features

- Arrow Function [57](#)
- Blocking Scopes [59](#)
- classes [61](#)
- EsLint [67](#)
- modules [65](#)
- object, enhancing [66](#), [67](#)
- parameters [65](#)
- Promises [62](#), [63](#)
- template, supporting [61](#)

encryptString function [175](#)

Error Handling [86](#)

Error Handling, types

- Asynchronous [91-93](#)
- Built-in [86](#), [87](#)
- custom [88-91](#)
- static file, serving [93](#)

EsLint

- about [67](#)
- configuring [67](#)
- installing [67](#)
- running [68](#), [70](#)

Event-Driven Mechanism [12](#)

EventEmitter [12](#)

Express.js

- about [73](#), [74](#)
- application, installing [75](#), [76](#)
- best practices [96-100](#)

Express.js, advantages

- community, collaborating [74](#)
- flexible [74](#)
- minimalist [74](#)
- Node.js, compatibility [74](#)
- scalable [74](#)

Express.js, features

- Error Handling [86](#)
- REST API [77](#)
- templating engines [94-96](#)

Express.js, limitations

- built-in, lack [75](#)
- client-side, applications [75](#)
- inconsistency [75](#)

F

fs.readFile() [15](#)
fs.readFileSync() [14](#)

G

getInstance [155](#)

H

hasPermission [188](#)
HttpException [90](#)

J

JavaScript [32](#)

M

Machine Learning [357](#), [358](#)
microservice architecture [16](#), [17](#)
Mocha Framework [303](#)
Mocha Framework, installing [304-306](#)
Mocha Framework, reasons
asynchronous, testing [303](#)
easy use [303](#)
hooks [304](#)
Node.js, supporting [304](#)
parallel test, executing [304](#)
system, reporting [304](#)
test, exclusivity [304](#)
test styles, supporting [303](#)
test suits, organizing [304](#)
timeout [304](#)
monolithic architecture [15](#), [16](#)

N

Node.Js
about [2](#)
best practices [361](#), [362](#)
clustor module [23-27](#)
code, synchronizing [14](#)
event, programming [13](#)
history [1](#)
HTTP Server, utilizing [19-22](#)
installing [5](#)
Node.Js, advantages
caching [4](#)
cross platform [4](#)

- high, performance [4](#)
- huge, community [4](#)
- scalling [4](#)
- Node.Js, applications
 - data streaming [3](#)
 - IoT Device [3](#)
 - real-time [3](#)
 - single-page [3](#)
- Node.Js, architectures
 - microservice architecture [16, 17](#)
 - monolithic architecture [15, 16](#)
 - serverless architecture [17-19](#)
- Node.Js, disadvantages
 - hell, callback [5](#)
 - library, compatibility [5](#)
 - single, threading [4](#)
- Node.Js, version
 - Linux, installing [8-11](#)
 - MacOS, installing [11](#)
 - Ubuntu, installing [6](#)
- Non-Primitive Data, types
 - Any [42](#)
 - Array [41](#)
 - custom [42](#)
 - Enums [41](#)
 - Object [41](#)
 - Tuple [42](#)
- Notification Module [282-285](#)
- NPM (Node Package Manager) [6](#)
- NVM, concepts
 - environments, isolating [6](#)
 - flexibility [6](#)
 - process, updating [6](#)
 - project, versioning [6](#)
 - version, managing [6](#)
- NVM (Node Version Manager)
 - about [6](#)
 - advantages [6-8](#)

P

- Password Management [201](#)
- Password Management, aspects
 - password, changing [202-205](#)
 - password, recovering [205-211](#)
 - password, reset [211-214](#)
- PMS, key areas
 - lesson, preparing [361](#)
 - predictive, analyzing [361](#)
 - project, planning [361](#)

real-time sentiment [361](#)
reports, preparing [361](#)
resource, allocating [361](#)
risk, assessment [361](#)
task, prioritizing [361](#)
PMS, module
comment module [106](#)
project module [105](#)
task module [105](#)
user module [105](#)
PMS (Project Management System)
about [103](#), [104](#)
cluster module, analyzing [122-125](#)
express server, creating [119](#), [120](#)
project dependency, installing [114-118](#)
roadmap [104](#)
scope [104](#), [105](#)
structure, setting up [112](#)
TypeORM, utilizing [125-128](#)
PostgreSQL
reference link [113](#)
Primitive Data, types
Bigint [41](#)
Boolean [41](#)
Null [41](#)
number [40](#)
String [40](#)
symbol [41](#)
print() method [57](#)
Project Management
Delete Project [232](#), [233](#)
GetAll Project [224-226](#)
GetOne Project [228-230](#)
input, validating [216](#), [217](#)
naming [227](#)
project, adding [219-223](#)
Project Util [234](#)
services [216](#)
Update Project [230-232](#)
Project Management, interface
dashboard [356](#)
profile icon [357](#)
project [356](#)
reports [357](#)
server, status [357](#)
user [357](#)
Project Management, key parts [218](#)
Promises, properties
array, destructuring [64](#)
object, destructuring [64](#)

Q

Queues, challenges
 fault tolerance, recovery [297](#)
 latency [297](#)
 scalability [297](#)
Queues, reasons
 asynchronous, processing [286](#)
 performance, improving [286](#)
 reliability [286](#)
 scalability [286](#)

R

Redis
 about [267](#)
 cache, building [277](#), [278](#)
 cache, utility [273](#)-[275](#)
 failures, handling [290](#), [291](#)
 MacOS, installing [268](#)
 project, dependencies [271](#)-[273](#)
 Queue, implementing [285](#)-[290](#)
 RHEL, installing [269](#)
 setting up [268](#)
 Ubuntu, installing [268](#), [269](#)
Redis, challenges
 cloud, hosting [278](#)
 data, query [278](#)
 data, security [278](#)
 memory, limitation [278](#)
 nature, threading [278](#)
Reporting [357](#)
resolve() [63](#)
REST API
 about [77](#)
 building [80](#), [81](#)
 middleware, supporting [84](#), [85](#)
 Routing [82](#)
REST API, principles
 cacheable [77](#), [78](#)
 Client-Server, architecture [77](#)
 code, demanding [79](#)
 layered, architecture [79](#)
 stateless, designing [77](#)
 uniform interface [78](#)
REST API Routing, method
 Route Handlers [84](#)
 Route Parameters [83](#), [84](#)
 Route Paths [83](#)
Role Management [155](#)

Role Management, functionalities

- GetAll Roles [164, 165](#)
- GetOne Role [166, 167](#)
- Role, adding [160-163](#)
- Role, deleting [169](#)
- Role Service [156, 157](#)
- system, utilizing [170, 171](#)
- updating [167, 168](#)
- validator [157-160](#)

Routes [138](#)

Routes, parts

- Handlers [138](#)
- HTTP, method [138](#)
- parameters [139](#)
- URL Paths [138](#)

Routes, types

- Comment Routes [145-149](#)
- Project Routes [143](#)
- Role Routes [140, 141](#)
- Task Routes [144](#)
- User Routes [141](#)

S

serverless architecture [17-19](#)

Sinon [312](#)

static [50](#)

Swagger UI

reference link [360](#)

T

Task Management

- Add Task [238-242](#)
- Delete Task [255-257](#)
- files, uploading [257-265](#)
- GetAll Task [242-246](#)
- GetOne Task [249-252](#)
- input, validating [235](#)
- service [234, 235](#)
- Task, searching [248](#)
- Task, updating [253-255](#)

Task Management, key parts [237, 238](#)

test case, configuring [306-308](#)

token, security features

- containers, deploying [359, 360](#)
- LDAP, integrating [359](#)
- social media, logging [359](#)
- SSL/TLS, encrypting [359](#)
- two-factor, authenticating [359](#)

TypeScript
about [31](#), [32](#)
building, steps [35-39](#)
installing [34](#)
TypeScript, advantages
bug, detecting [33](#)
cross platform [33](#)
flexible, maintaining [33](#)
IDE, supporting [33](#)
OOP and ES6, supporting [33](#)
open source [33](#)
type, checking [33](#)
TypeScript, disadvantages
application, unsuitable [33](#)
curve, learning [34](#)
time, compilation [34](#)
TypeScript Installation, types
global, approach [34](#)
project-wise [34](#)
TypeScript Modifiers, types
private [48](#)
protected [48](#)
public [47](#)
Static [49](#)
TypeScript OOP, concepts
Abstraction [52](#), [53](#)
Class [43-45](#)
data types [40](#)
Encapsulation [53-55](#)
Inheritance [45-47](#)
Interface [50](#), [51](#)
Modifiers [47](#)
Polymorphism [55-57](#)

U

uniform interface, constraints
HATEOAS [79](#)
manipulation, representing [78](#)
resource, identifying [78](#)
self-descriptive, messaging [78](#), [79](#)
Unit Testing [301](#), [302](#)
Unit Testing, key points
assertions [302](#)
automating [303](#)
code, coverage [302](#)
data, testing [302](#)
error, handling [302](#)
fast, executing [303](#)
mocks, stubs [302](#)

negative, testing [302](#)
regression, testing [302](#)
test case, structure [302](#)
User Management [171](#)
User Management, aspects
authenticating [181-185](#)
authorizing [185-193](#)
delete user [200, 201](#)
GetAll Users [193-195](#)
GetOne User [195-197](#)
input, validating [172, 173](#)
onboarding [174-177](#)
user, service [172](#)
user, signing [179-181](#)
user, updating [197-200](#)

[OceanofPDF.com](#)