Mastering MEAN Stack

Build full stack applications using MongoDB, Express.js, Angular, and Node.js

Pinakin Ashok Chaubal

bpb

More books: https://t.me/booksforits

Mastering MEAN Stack

Build full stack applications using MongoDB, Express.js, Angular, and Node.js

Pinakin Ashok Chaubal

www.bpbonline.com

More books: https://t.me/booksforits

Copyright © 2023 BPB Online

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 BPB Online 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.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55510-525

www.bpbonline.com

More books: https://t.me/booksforits

Dedicated to

*My wife **Manasi***
*My sons **Arnav** & **Yash***
&
My parents who supported me throughout the journey of this book

More books: https://t.me/booksforits

About the Author

Pinakin Chaubal is a PMP, ISTQB Foundation, and HP0-M47 QTP11 certified IT professional with over 23 years in Performance testing with LoadRunner & JMeter. He has worked in the fields of Software Testing & Quality Automation for various projects with demonstrative success in applying test methodologies & QA processes, software defect tracking, and new deployment environments & automation with automated test tools & frameworks across multi-national companies. He has also gained global onsite exposure of 6 years in the USA and 8 months in Hong Kong & China.

More books: https://t.me/booksforits

About the Reviewer

Mohd Monis is an accomplished full-stack developer currently working as a technical lead, bringing over 8 years of invaluable industry experience. His expertise primarily lies in frontend development, where he excels in analyzing and synthesizing complex business problems.

He actively engages in fruitful conversations about emerging technology trends with his peers. Additionally, he has recently developed a keen interest in creating design mockups.

More books: https://t.me/booksforits

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my wife Manasi.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

More books: https://t.me/booksforits

Preface

Building full stack applications is a complex task that requires a comprehensive understanding of the latest technologies and programming languages. MongoDB, Express, Angular and Node.js are powerful tools that have become increasingly popular in the field of enterprise development.

This book is designed to provide a comprehensive guide to building enterprise applications using the MEAN stack. It covers a wide range of topics, including the basics of MongoDB, Angular and advanced concepts such as object-oriented programming, and the use of the Node.JS for building robust and scalable applications.

Throughout the book, you will learn to buld a MEAN stack for a blog site. The reader is taken step by step through the learning journey. The reader gets to know various concepts in application development and deployment.

This book is intended for developers who are new to building full stack applications. It is also helpful for experienced developers who want to expand their knowledge of these technologies and improve their skills in building robust and reliable applications.

With this book, you will gain the knowledge and skills to become a proficient developer in the field of full-stack development using the MEAN stack. I hope you will find this book informative and helpful.

Chapter 1: Fundamentals of Full Stack Development and the MEAN Stack - This chapter will focus on the concept of full stack development, it's importance and how the MEAN stack is a quick way to develop such applications. We will understand why the MEAN stack is so irresistible for developers. We will also see a glimpse of the application that we will be developing. Understand how the MEAN stack helps in fast, efficient and scalable development. We will understand the difference between the traditional multi-threaded web server and the single-threaded Node web server, learn what blocking/non-blocking code is, and understand the Express framework. We move on to MongoDB as the database for our stack and eventually we look at the Angular frontend framework which will be used in designing the face of our application. We will look at what TypeScript is and also look at the use of Bootstrap. We will be introducing Git as our version control system. We will take a look at AWS as our hosting service. We will then take a look at how the different MEAN components interact with each other. We will

More books: https://t.me/booksforits

get introduced to Docker and Kubernetes and understand their role in deploying MEAN applications..

Chapter 2: Architectural Design of Our Sample Application - This chapter will focus on the various design considerations for a MEAN stack. We will explore various options available. We will start with a basic architecture and then move on to the desired Microservices architecture in which we can have separate docker containers for the Posts service, Comments service, the Express and Node.js framework and the database. This way our system will be fault tolerant and resilient.

Chapter 3: Installing the Components - In this chapter, we will be looking at the Node.js and Angular needed for our MEAN project. We will be installing all these on Windows, although the installation remains largely the same for any operating system. We will create the project folder and initialize it. We will install Angular CLI and then create a new Angular project within the parent folder. We will then install Angular material inside the Angular project folder. We will also look at the generated Angular project and package.json file. We will then check the installed dependencies in node_modules folder. Eventually, we will take a look at the generated app folder and the various files within it.

Chapter 4: Creation of the Frontend Using Angular - In this chapter, we will be building our frontend. This will be a bare-bones frontend which will be integrated with MongoDB and the server framework later. We will start with a bird's eye view of how angular works by customizing the current Angular application.

Chapter 5: Addition of Node.js and Ideas for Integration- This chapter will explore the backend server part of our MEAN framework which comprises of Node.js [Open-source, cross-platform backend JavaScript runtime environment that runs on V8 engine built for Chrome] Express JS [Backend web-application framework for Node.js]. We will start with the basics of building backend logic using Express and Node.js. In this process, we will make use of Postman[API platform for building and using APIs.] Postman simplifies the API lifecycle and streamlines collaboration so that better APIs can be created faster. We will also see two main HTTP verbs for building the backend code which is GET and POST.

Chapter 6: Handling Authorization - With this chapter, we start the development of the first piece of our framework which is the Authorization piece. We will be making use of Typescript throughout this book for the development of the backend part of our code. This chapter will incorporate certain best practices for the

More books: https://t.me/booksforits

development of code. We will be using Google Cloud environment for developing the application, since many times due to creation of many Kubernetes pods, older systems may crash. One can only create a single node cluster on a local desktop or laptop. We can create a multi node cluster when we develop on Google Cloud. We will be creating a docker container for our Auth service and will also make use of Google cloud shell as our Integrated Development Environment.

Chapter 7: Creating the Posts Service and NATS Streaming Integration - In this chapter, we start with the introduction to the common module. As a part of the common module, we create a folder called common and move the Authentication related logic to this folder. We will create an NPM module of the common module. This common module can be included in all services of our project. We then move to the development of the Posts service. This is followed by an introduction to the NATS streaming server, and we create a simple Publisher and Listener which have a simple NATS implementation. We will see how to publish messages and listen to messages. The NATS Streaming code will be moved to the common module. We create a Posts service and integrate it with the NATS streaming server. We then test our integration with Postman

Chapter 8: Introducing Automated Testing - Here, we will be looking at how to incorporate automated testing in our framework using Supertest. Supertest is a library which will test all our APIs without the need to bring up a Kubernetes cluster. The Supertest library can be used by itself or with Mocha. In this book, we will focus on Supertest with the JEST framework.

Chapter 9: Integrating the Comments Service - In this chapter, we will look at the concepts behind building and integrating the comments service. Concepts such as sub-documents and references with regard to MongoDB will be introduced. We will see the pros and cons of using sub-documents and learn the advantages of references.

Chapter 10: Creating the Comments Service - In this chapter, we will look at the creation of the comments service and see how we can use Supertest to run automated tests to test our comments service.

Chapter 11: Implementing the Frontend - This chapter will wrap up things by implementing the front end. We will look at how to go about creating the frontend with Angular. We will also learn about the various pieces of an Angular application like components and services. We will see what the app component is and how to plug in the various components in the app component.

More books: https://t.me/booksforits

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/tdi84xj

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Mastering-MEAN-Stack**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with 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@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.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 BPB books and eBooks.

More books: https://t.me/booksforits

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 **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

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 BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Table of Contents

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Chapter 1

Fundamentals of Full Stack Development and the MEAN Stack

Welcome to the exciting world of full stack development. This book will be divided into three parts. The first part will make the user understand what full stack development is and what the MEAN stack is. The second part will explore the installation of the different MEAN stack components, the creation of the front-end using Angular, adding Node.js and connecting with Angular, and the addition of the MongoDB (NOSQL) database. This part will take care of the **Create, Read, and Delete** part of **CRUD**. The third part will cover the Update part of CRUD and routing, image upload, and pagination. The fourth part will cover adding authentication, authorization, error handling, optimization, and deployment. It is going to be a very exciting journey, so let's get started and understand what full stack development all is about. We will then understand the fundamentals of the MEAN stack.

This chapter will focus on the concept of full stack development, its importance and how the MEAN stack is a quick way to develop such applications. We will understand why the MEAN stack is so irresistible for developers. We will also see a glimpse of the application that we will be developing. Understand how the MEAN stack helps in fast, efficient, and scalable development. We will understand the difference between the traditional multi-threaded web server and the single-threaded Node web server, learn what blocking/non-blocking code is, and understand the Express framework. We move on to MongoDB as the database for our stack and eventually, we look at the Angular frontend framework which will be used in designing the face of our application. We will look at what TypeScript is and look at the use of Bootstrap. We

More books: https://t.me/booksforits

will introduce Git as our version control system. We will look at AWS as our hosting service. We will then look at how the different MEAN components interact with each other. We will get introduced to Docker and Kubernetes and understand their role in deploying MEAN applications.

Structure

This chapter will predominantly focus on the following topics:

- What is full stack development?
- Introduction to the MEAN stack
- Our sample application
- Introduction to Node.js
- Introduction to Express.js
- Introduction to MongoDB
- Introduction to Angular
- Introduction to Typescript and Bootstrap
- Introduction to Git as a version control system
- Interaction between components of the MEAN stack
- Introduction to Docker and Kubernetes as deployment tools

Objectives

After reading this chapter, the readers will understand what full stack development all is about. They will also get introduced to frameworks that support full stack development. They will understand what the MEAN stack is, and they will also get a glimpse of the sample application that we will be building throughout this book. They will learn about the underlying components of the MEAN stack and understand how the various components interact together. They will get introduced to Docker and Kubernetes and see how Docker simplified the build and deployment process.

What is full stack development?

Full stack development refers to the development of both frontend (client-side) and backend (server-side) parts of a web application. It is the art of designing complete web applications or websites. It comprises the development of frontend, backend, and database.

More books: https://t.me/booksforits

Full stack development can consist of the development of whole web applications using components like HTML, CSS, Java/JavaScript/Python, and a database of choice. JavaScript offers two major choices when it comes to designing web applications which are: The MEAN stack and the MERN stack. These stacks comprise frameworks and libraries.

Using full stack development, developers can design complete end-to-end applications. Such applications typically have 3 parts which are frontend, backend, and database. Let's see each of these in a nutshell. We will be expanding on each of these throughout the book.

Frontend

What we call the frontend here is the visible part of the web application or website which is responsible for providing user experience. The user interacts directly with this part of the application.

The frontend languages can be any one of the following:

- **HyperText Markup Language (HTML)**: HTML is used to design web pages using a markup language. HTML is a combination of Hypertext and Markup language. Hypertext is used to establish a link between the web pages. The text within the tag is defined by the Markup language and this defines the structure of a web page.
- **Cascading Style Sheets (CSS)**: CSS is a language that simplifies the process of making web pages attractive. CSS helps to apply styles to web pages and CSS aids to do this independent of the HTML for each page.
- **JavaScript**: JavaScript is a scripting language which is used to make the site interactive for the user.

The frontend frameworks and libraries can be any of the following:

- **Angular**: Angular is an open-source front-end framework, written in JavaScript, mainly used to develop **single-page web applications** (**SPAs**). It converts static HTML to dynamic HTML. It is available to be used and changed by anyone. It extends HTML attributes with directives where data is attached with HTML.
- **ReactJS**: React is a declarative JavaScript library created by Facebook for building user interfaces. ReactJS is an open-source, front-end library responsible for the front- end of the application.
- **Bootstrap**: Bootstrap is a free and open-source collection to create responsive websites and web applications. It is the most popular framework for

More books: https://t.me/booksforits

developing responsive websites which are very popular nowadays in the mobile world.

Backend

What we refer to as backend here is the server-side development of a web application or website which focuses on the functionality of the website. It is also responsible for managing the database through queries and APIs using client-side commands.

The backend languages can be any one of the following:

- **Java**: Java is one of the most popular and extensively used programming languages and platforms known for its high scalability. Moreover, Java components are easily available.
- **Python**: The Python programming language allows the developer to work quickly and integrate systems efficiently.
- **JavaScript**: JavaScript can be used as a back-end programming language in addition to its being a front-end programming language.

The backend frameworks and libraries can be any of the following:

- **Express**: Express.js, or Express, is a back-end web application framework for Node.js, which is free and open-source software under the MIT License. It is specifically designed for building web applications and APIs. It is one of the popular frameworks for node.js among developers.
- **Django**: Django is a high-level web framework built in Python that encourages rapid development and clean, pragmatic design. It takes care of many of the problems of web development so that the developer can focus on writing her app without starting from scratch. It's free and open-source software.
- **Spring**: The Spring framework is an application framework and inversion of control (IOC) container designed for the Java platform. The core features of the framework can be used by any Java application, but extensions are available for building web applications on top of the Java EE platform.

Database

A database is a collection of data that is interrelated which helps in easy and efficient retrieval, insertion, and deletion of data. Data in a database is organized in the form of tables, views, schemas, reports, and so on.

Few of the popular databases used in full stack development are MongoDB, Oracle, MySQL, etc. Let's take a brief look at these:

More books: https://t.me/booksforits

- **MongoDB**: MongoDB is the most popular NoSQL database used in the MEAN and MERN stacks. It is an open-source document-based database. The term *NoSQL* means 'non-relational'. This means that MongoDB isn't based on the relational database structure and provides a totally different mechanism for the storage and retrieval of data.
- **Oracle**: Oracle database is the collection of data that is interrelated. The purpose of this database is to store and retrieve information based on a SQL query.
- **MySQL**: MySQL is a relational database management system like Oracle which is Open Source. An interesting fact about its name is that it is a combination of *My*, the name of co-founder *Michael Widenius's* daughter, and *SQL*, the abbreviation for Structured Query Language.

Introduction to the MEAN stack

MEAN (**MongoDB, Express.js, Angular (or Angular), and Node.js**) is a free and open-source JavaScript software stack for developing dynamic websites and web applications. Since all components of the MEAN stack support code written in JavaScript, MEAN applications can be written in a single language for both server-side and client-side execution environments.

The MEAN stack is often compared to other popular web development stacks such as the LAMP stack, the components of the MEAN stack are higher-level and include a web application presentation layer. An important thing to note is that the MEAN stack does not include an operating system layer.

The acronym MEAN was created by *Valeri Karpov*. He put forward the term in a 2013 blog post. The logo concept was initially created by Austin Anderson for the original MEAN stack LinkedIn group. This logo is a collection of the first letter of each component of the MEAN acronym.

Our sample application

We will build a blogging application which will have features such as:

- A user can sign-up for the blogging application.
- A user can log in to the system.
- The logged-in user can create a blog.
- The user who created the blog can modify and delete the blog.
- Other users can view the blog.
- Other users can post comments against a blog.

More books: https://t.me/booksforits

Over the course of this book, we will build this application step by step so that the concepts are easier to understand. As with any other project, we start by introducing the components of the MEAN stack.

Let us see what Node.js is.

Introduction to Node.js

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment which runs on the V8 engine. It executes the JavaScript code outside the boundaries of a web browser. Node.js allows JavaScript to write command line tools and server-side scripts (for example, running scripts on the server side to produce dynamic web page content before the page is sent to the web browser). Consequently, Node.js represents a *JavaScript everywhere* concept and facilitates web application development using a single programming language, rather than using different languages for server-side and client-side scripts.

Although **`.js`** is the standard filename extension for JavaScript code, the name *Node.js* is not a particular JavaScript file and is simply the name of the product. Node.js is based on an event-driven architecture capable of performing asynchronous I/O. These design choices aim to optimize scalability and throughput in web applications with many input/output operations, as well as for real-time web applications.

Node.js is an asynchronous event-driven JavaScript runtime designed to build scalable network applications. This is different from the common concurrency model which employs OS threads. Networking using a Thread-based mechanism is comparatively inefficient and difficult to use. Node.js users don't need to worry about process dead-locking problems since there are no locks involved. Functions in Node.js do not perform I/O directly leaving the process unblocked. Because of this non-blocking feature, it is reasonable to develop scalable systems in Node.js.

Let's move to Express next.

Introduction to Express.js

Express.js, or Express as it is usually referred to, is a backend web application framework for Node.js. It is a free and open-source software under the MIT License. Designed for building web applications and APIs, Express has been called the de facto standard server framework for Node.js. Express is a flexible web application framework for Node.js that provides a robust set of features for building web and mobile applications. With a huge collection of HTTP utility methods and middleware, the developer can create a robust API quickly and with considerable ease.

Express provides a thin layer of required web application features, without disturbing Node.js features that developers love and are aware of. Express can be utilized for

More books: https://t.me/booksforits

routing in the web application and writing middleware. We will see how to do this as we get into the details of Express when we start using Express while building our application.

Now, let us see what MongoDB is.

Introduction to MongoDB

MongoDB is a document database created for ease of development and scalability. MongoDB comes as both a Community and an Enterprise version. The community edition of MongoDB has the source available and free to use. MongoDB Enterprise, which is available as part of the MongoDB Enterprise Advanced subscription, includes robust support for MongoDB deployment. MongoDB Enterprise also has enterprise-focused features such as LDAP and Kerberos support, auditing, and on-disk encryption.

A record in MongoDB is called a document. A document is a data structure comprising field and value pairs. MongoDB documents are quite similar to JSON objects which are made up of Key-Value pairs. The values of fields may include complex data such as other documents, arrays, and arrays of documents.

A sample MongoDB document is as follows:

```
{
    name: "Joe",
    age: 30,
    gender: "Male",
    interests: ["Maths","Science"]
}
```

A few of the advantages of using documents are:

- Documents refer to native data types in several programming languages.
- Embedded documents and arrays lessen the need for expensive joins.
- Fluent polymorphism is supported by dynamic schema.

Next, we move on to Angular.

Introduction to Angular

Angular is a platform and framework for creating single-page client applications using HTML and TypeScript. Angular, being written in TypeScript, implements functionality as a set of TypeScript libraries that the developers import into their applications.

More books: https://t.me/booksforits

The basic building blocks of the Angular framework are made up of Angular components that are organized into **`NgModules`** which collect related code into something known as functional sets; an Angular application is made up of a set of **`NgModules`**. An application always has at a minimum a root module used for bootstrapping and has many more feature modules.

Components define views. Views are sets of screen elements that Angular can select from and modify according to the program logic and related data. Components make use of services. Services provide specific functionality which is not directly related to views. Service providers can be injected as dependencies into components. This makes the code modular, efficient, and reusable.

Modules, components, and services are Typescript classes that make use of decorators which mark their type and provide metadata that tells Angular how to make use of them. An application's components typically define several views that are arranged hierarchically. Angular provides the Router service that helps to define navigation paths among views. The router provides in-browser navigational capabilities.

Introduction to TypeScript and Bootstrap

In this section, we will take a look at TypeScript and Bootstrap. Let's start with an introduction to TypeScript.

Introduction to TypeScript

TypeScript is a programming language created and maintained by the Microsoft team. Being a strict syntactical superset of JavaScript, it adds optional static typing to the language. As TypeScript is a superset of JavaScript, existing JavaScript programs are also supported by TypeScript.

TypeScript can be used to develop JavaScript applications for both the client side and the server side (as with Node.js or Deno). TypeScript supports definition files that can contain type information of existing JavaScript libraries, similar to C++ header files that can describe the structure of existing object files. This enables other programs to use the values defined in the files similar to statically typed TypeScript entities. TypeScript headers for the Node.js basic modules allow the development of Node.js programs within TypeScript.

Let’s take a look at Bootstrap next.

Introduction to Bootstrap

Bootstrap is a front-end toolkit that focuses on the simpler development of web pages. The main purpose of adding it to a web project is to apply readily available

More books: https://t.me/booksforits

choices of color, size, font, layout, and other UI elements to that project. Once added to a project, Bootstrap provides basic styles for all HTML elements. The result is a uniform appearance for HTML elements across various web browsers. Developers can take advantage of CSS classes defined in Bootstrap to additionally customize the appearance of their webpage.

Bootstrap includes several JavaScript components as jQuery plugins. They provide additional UI elements such as dialog boxes, carousels, and tooltips. Every Bootstrap component consists of an HTML structure, CSS declarations, and may include JavaScript code. They also provide additional functionality to some existing user interface elements, including an auto-complete function for text boxes.

The most important components of Bootstrap are its layout components, as they affect entire web pages. The basic layout component is referred to as *Container*, since every other element in the page is placed in it. Developers have the choice of a fixed-width container and a fluid-width container.

The fixed-width container uses one of the five predefined fixed widths, depending on the size of the screen displaying the page:

- Smaller than 576 pixels
- 576–768 pixels
- 768–992 pixels
- 992–1200 pixels
- Larger than 1200 pixels

The fluid-width container always fills the width of the web page. Next, we will see what GIT is.

Introduction to Git as a version control system

Git is a software used for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code in a team during software development. Git was created by *Linus Torvalds* in 2005 to facilitate the development of the Linux kernel. Eventually, other kernel developers contributed to this initial development. As is common with most other distributed version control systems, every Git directory on any computer is a complete repository with history and complete version-tracking abilities, independent of network access to a centralized server.

More books: https://t.me/booksforits

Let's take a look at how the various components of the MEAN stack interact with each other.

Interaction between components of the MEAN stack

The components of the MEAN stack work together as shown here:

- The client (Angular) makes a request.
- The Node.js parses the client's request.
- A call is made by the Express framework to the MongoDB database to fetch records.
- If records exist, they are returned back to Express by MongoDB.
- The response is passed on to Angular by Node.js.
- Angular displays the response in a presentable format.

We now move on to the final part of this chapter which is Docker and Kubernetes.

Introduction to Docker and Kubernetes as deployment tools

The final chapter of this book is about deploying the completed application to AWS. To smooth out the build deployment process, we will be making use of Docker and Kubernetes. Docker packages an application and its dependencies in a virtual container that can run on any computer, be it Linux, Windows, or macOS. Docker makes use of containers and images which we will explore throughout this book and in the final chapter.

Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management. Originally designed by Google, it is now maintained by the Cloud Native Computing Foundation. Its aim is to provide a *platform for automating deployment, scaling, and operations of application containers across clusters of hosts*. There are many components that make up Kubernetes and we will explore this throughout the book and in the final chapter.

With this, we complete the introductory chapter on the MEAN stack.

More books: https://t.me/booksforits

Conclusion

This was the introductory chapter where we got introduced to what full stack development is. We looked at the components of a typical full stack application. We then got introduced to the MEAN stack and then we looked at the various components of the MEAN stack. The rest of this book is dedicated to learning each chapter in depth and building our sample application using these components in an integrated manner. We took a look at TypeScript and Bootstrap that help us create the application code of the MEAN stack. We looked at Git for version control and build and deployment tools like Docker and Kubernetes.

In the next chapter, we will understand the complete architecture of our MEAN stack that we will be using to build our sample application. We will understand how the various components of our architecture will talk to each other.

Questions

1. Explain what full stack development means and the few software stacks available for this purpose.
2. Explain the different components of the MEAN stack.
3. What do you mean by NoSQL databases?
4. How does MongoDB store data?
5. What is the role of Docker and Kubernetes?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Chapter 2
Architectural Design of Our Sample Application

Any project in its entirety begins with its design. A good architectural design simplifies the job of a developer. A good design makes maintaining the project easier. We have various options as far as the design considerations go. There is the most basic design where we have all components of the MEAN stack together, increasing the risk of failure when a disaster happens. We then have the design where there are different project folders for the backend, frontend, and database. We can even think of the MEAN stack as a three-tier architecture where there are separate tiers for the frontend, backend, and server. This is a slightly better alternative. A good design would allow scaling if the load is increased on the system.

This chapter will focus on the various design considerations for a MEAN stack. We will explore various options available. Custom architecture can also be used to facilitate good design principles. We will start with a basic architecture and then move on to the desired microservice architecture in which we can have separate Docker containers for the posts service, comments service, the Express and Node.js framework, and the database. This way our system will be fault tolerant and resilient.

Structure

This chapter will predominantly focus on the following topics:

- What are single-page applications?

More books: https://t.me/booksforits

- What are multi-page applications?
- Single-page v/s multi-page applications
- Common MEAN stack architecture
- Going with the event-driven approach
- Best practices for designing an API
- Working with the singleton pattern
- Working with the observer pattern
- Designing a flexible architecture
- Differences between the architecture of an application in development and production

Objectives

After reading this chapter, the readers will understand what a single-page application is and how to build an application with the MEAN stack, which is scalable, fault tolerant, and resilient. They will understand how to architect the most basic architecture and build on that. They will understand the need for Docker to make our architecture flexible enough to make it production ready.

Let us start with the most common MEAN stack architecture.

What are single-page applications?

A single-page application works inside a browser and does not require full-page reloading while using it. We are so accustomed to using SPAs in our day-to-day life that we are not aware that the page we are seeing is a single-page application. Some examples are Gmail, Google Maps, GitHub, or the Angular website.

SPAs provide an amazing user experience by simulating an environment in the browser itself. There are no page reloads or the overheads of extra waiting time. Users visit just one web page which in turn loads all other content using JavaScript — which the application relies on.

In a SPA request, markup and data are independent of each other. This has been made possible by advanced JavaScript frameworks like AngularJS, Ember.js, Meteor.js, and Knockout.js.

Single-page makes the user comfortable by providing a single view where full-page reloads don't happen. Single-page applications come with their pros and cons which are listed as follows.

More books: https://t.me/booksforits

Advantages of SPA

There are many advantages to using SPAs which are listed as follows:

- SPA is fast, as most of the resources (HTML+ CSS+ JS) are loaded just once in the span of the application. Only data is transmitted between the server and the client.
- SPAs make the development process simpler as there is no need to write code to render content on the server. The developer can usually start development from a local file without running any server.
- SPA applications do not reload the page on every request so it adds an advantage of using caching at the client side (using local storage/indexed DB) where data can be stored and is/can be accessed throughout the application helping developers to work offline.

As with any application, a SPA also has disadvantages attached to it which are listed next.

Disadvantages of SPA

A few of the disadvantages that can be considered for SPAs are listed here:

- Since these SPAs (client-side frameworks) are heavy and require other scripts to be downloaded to the browser, it slows down the download process and affects the initial downloads.
- A SPA is less secure compared to traditional applications since it enables attackers to inject client-side scripts into web applications due to cross-site scripting.
- Applications built with SPA frameworks can slow down due to memory leaks.

What are multi-page applications?

Multiple-page applications follow the *traditional* approach where every page is served as a new request from the server to the browser. The size of these applications can be huge and tends to have many levels of UI due to the amount of content they have. Due to the introduction of AJAX, we don't have to worry about the amount of data transferred between the client and server. AJAX allows you to refresh only particular portions of the application. On the flip side, it is more difficult to develop than a single-page application as it adds more complexity.

Let's look at the advantages of multi-page applications.

More books: https://t.me/booksforits

Advantages of multi-page applications

The advantages of multi-page applications are listed as follows:

- Multi-page applications provide the users with a visual path of navigation in the application. Menu navigations are lesser compared to a SPA.
- Multi-page applications are **Search Engine Optimization** (**SEO**) friendly. Since the application can be optimized for one keyword per page, it becomes easy to rank for different keywords.

Let us look at the disadvantages of multi-page applications.

Disadvantages of multi-page applications

The disadvantages of multi-page applications are listed as follows:

- There is a tight coupling between front-end and back-end development.
- The development is more complex as the developer needs to utilize frameworks for either the client or the server side. This increases development time.

Single-page v/s multi-page applications

When it comes to deciding whether to use SPA or MPA, it is a matter of choice. Sometimes with a lot of menus as in the case of a shopping application, one might think of using multi-page applications. If the menus or content of your application is such that you can crunch the entire application to make it a single page, then an SPA is the way to go.

In many banking applications, we need to use MPA since we know that the number of menus will be huge. In addition to this, the security of a multi-page is higher than single-page applications like in the case of banking applications. We will be implementing a single-page application in this book.

Let us now learn the most common MEAN stack architecture which gets mostly implemented.

Common MEAN stack architecture

The frequent method most people use to develop their MEAN stack is by using Express, Node.js, and MongoDB to build the REST API and Angular to build the front end. REST stands for Representational State Transfer which is nowadays considered more as an architectural style rather than a protocol. API stands for Application Programming Interface whose responsibility is to make various applications talk to

More books: https://t.me/booksforits

each other. In the web world such an API can be a list of URLs for **`GET`**, **`POST`**, **`PUT`**, and **`DELETE`** using which data in a database can be fetched, inserted, updated, and deleted. Such a REST API is stateless which means that the API does not store the state of the application. A REST API is a stateless interface to the database where you can fetch, store, update, and delete data using either an API testing tool like Postman or by developing a front-end in Angular, React, or Vue.js.

The pictorial representation of such an architecture is shown here:

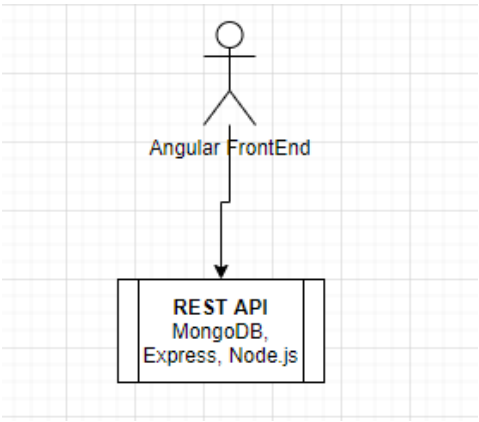

***Figure** 2.1: A basic MEAN architecture*

Such an application has basic architecture and can be used for API implementation. If the load gets increased and the need arises to scale such an architecture on the API side, additional servers are needed for hosting Express, Node.js, and MongoDB and all three (Express, Node.js, and MongoDB) will be duplicated. If we need to have one more instance of the database, then we can create instances of Express and Node.JS when there is no necessity to scale both.

Going with the event-driven approach

We can decide to have an event-driven architecture for our project in which we have different services like post service, comment service, lookup service, and a custom event service. Let us take a look at how operations will be handled.

When a new post is created, an event with a post object will be sent over to a custom **`EventQueue`** system. This **`EventQueue`** will in turn echo this event to the Lookup, Post, and Comment Services. The Post Service will consume this event and add it to the appropriate post in the database. Similarly, when a new comment is created, the JSON containing the comment ID and information along with the ID of the post to which this particular comment belongs will be passed to all the three services which are post service, comment service, and lookup service. The post service will consume this event and add it to the appropriate post.

More books: https://t.me/booksforits

When a post is updated, an event with a post object will be sent over to a custom **`EventQueue`** system. This **`EventQueue`** will in turn echo this event to the lookup, post, and comment services. The post service will consume this event and update the appropriate post. Similarly, when a comment is updated, the JSON containing the comment ID and information along with the ID of the post to which this particular comment belongs will be passed to all the three services which are the post service, comment service, and lookup service. The post service will consume this event and update the appropriate post.

A similar pattern follows for the delete operation as well. Let's now move on to best practices for designing an API.

Best practices for designing an API

The single best practice while designing a flexible API is to separate out the services in its individual container. This way, it is easier to scale up or down in case there is a fluctuation of load. Separation of concerns is also considered good when designing a big microservice architecture. You might go granular to the level where you have a separate microservice for generating logs or something similar.

All services should be loosely coupled. This is a very important point to remember for microservice architecture. In the upcoming chapters, we will follow the best practices for microservices in NodeJS.

Next, we take a look at the singleton pattern. We can use the singleton pattern while dealing with database operations.

Working with the singleton pattern

We use the singleton pattern when we need to ascertain that there is only one object instantiated. Therefore, instead of creating a new object every time, we need to make sure that the constructor was called only once and then we reuse the instance.

We can achieve this by creating our class to have the following:

- Hidden (private) constructor.
- The Public **`getInstance`** method that calls the constructor which in turn returns the instance of the class.

Next, we will take a look at the Observer pattern.

More books: https://t.me/booksforits

Working with the Observer pattern

Observers can subscribe to an object to keep track of any potential state change that might occur to that object. Observers can unsubscribe as well, stopping in their entirety to watch the object under consideration.

In objected-oriented programming, the observer pattern needs interfaces, concrete classes, and hierarchy. In NodeJS, everything becomes simpler. The observer pattern is already built into the system and is available through the **`EventEmitter`** class.

`EventEmitter` allows us to register functions as listeners. This will be called when a specific event is fired. The Observer patterns will be understood in greater detail in the upcoming chapters.

We now move to the next topic of designing a flexible MEAN stack architecture.

Designing a flexible architecture

Over the course of this book, we will build a microservice architecture that is flexible enough. There will be no hard dependencies. We have also decided to have an event-driven architecture for our MEAN stack. This event-driven architecture will consist of the following:

- **Event queue**: This will provide a queuing mechanism for the post and comment objects that will be created by the post and comment service, respectively. The post and comment services will be on separate Docker containers to facilitate scaling.
- **Post service**: This will be responsible for creating, updating, and deleting posts. The update request would simply modify posts and keep the associated comments intact.
- **Comments service**: This will be responsible for creating, updating, and deleting comments related to posts and hence the requests would contain the parent **`PostId`**. The update request would simply modify the associated comments of the post.
- **Lookup service**: This will be accessed by the front-end. The front-end will not access the post or comment service directly. It will only access the lookup service.

These are the initial services to start with. A few more services will be added along the way. Let us take a look at the comparison between the architectures of development and production.

More books: https://t.me/booksforits

Differences between the architecture of an application in the dev and prod environment

Coming to the comparison between the architectures of development and production, we can list the following differences:

Development	Production
May contain an API gateway	Always beneficial to have an API gateway
May contain a load balancer	Mandatory to have a load balancer
Live reload may be present	Live reload should be present

***Table 2.1**: Comparison between the architectures of development and production*

Conclusion

This was a theoretical chapter on the architectural design that we will be following in this book. We understood what single-page and multi-page applications are and explored the difference between them. We took a look at the common stack architecture and discussed which architecture is suitable for project needs. We saw the best practices of designing an API followed by a basic understanding of the singleton and observer patterns. We saw how to design a flexible architecture and finally, we took a look at the major differences between the architecture of the dev and prod environment.

In the next chapter, we will understand the installation of the various components of the MEAN stack. We will explore the cloud option of MongoDB. We will also take a look at how a NATS streaming server can be beneficial for message queuing.

Questions

1. Explain what single-page applications are.
2. Explain what multi-page applications are and how they differ from single-page applications.
3. Explain the basic architecture of the MEAN stack.
4. What is meant by the Observer pattern?
5. Create an architecture diagram for Production-Grade applications.

More books: https://t.me/booksforits

Chapter 3
Installing the Components

The previous chapter focused on the architectural design of our application. We took a brief look at the singleton and observer patterns. As we saw, we will be following an event-driven architectural pattern to implement microservices in our project.

In this chapter, we will look at the Node.js and Angular needed for our MEAN project. We will install all these on Windows, although the installation remains largely the same for any operating system. We will create the project folder and initialize it. We will install Angular CLI and then create a new Angular project within the parent folder. We will then install Angular material inside the Angular project folder. We will also look at the generated Angular project and **`package.json`** file. We will then check the installed dependencies in the **`node_modules`** folder. Eventually, we will take a look at the generated app folder and the various files within it.

Structure

This chapter will predominantly focus on the following topics:

- Installing the Node.js runtime
- Creating the project folder
- Initializing the project with the **`npm init`** command

More books: https://t.me/booksforits

- Installing the Angular CLI
- Creating a new Angular project
- Installing Angular material
- Verify that Angular works
- Exploring the project structure
- Scanning the **`package.json`** file
- The **`node_modules`** folder
- Looking at the app folder generated by Angular
- How does an Angular application start?

Objectives

After reading this chapter, the reader will understand how to get his local development environment set up for the MEAN stack application. Since our next chapter focuses on Angular development, we will get an Angular project set up first. As far as the *E* (Express) and *M* (MongoDB) components are concerned, we will install Express using npm and we will use MongoDB in the cloud. We will install Express later.

Let's start with the installation of Node.js. We will use VS Code as our code editor. It is freely available and can be downloaded from the Microsoft website.

Installing the Node.js runtime

We will see the installation of the Node.js runtime for Windows, Mac, and Linux. Let's start with Windows.

Installing the Node.js on Windows

In order to install Node.js, visit the following URL **https://nodejs.org/en/download/**, which displays the page shown as follows:

More books: https://t.me/booksforits

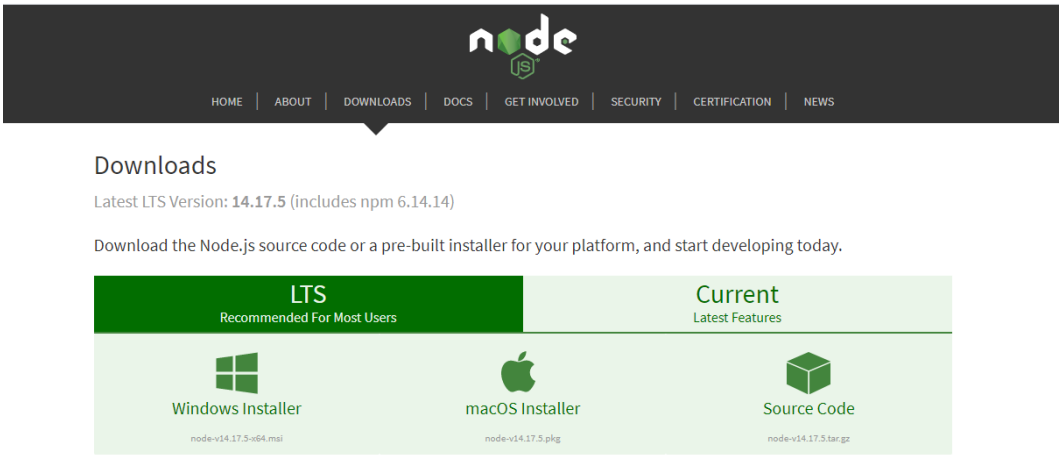

***Figure 3.1**: Node.js download*

Download the Windows installer as an MSI file or zip version. If the MSI file is downloaded, then double click on the **`.msi`** file and follow the instructions.

Verify the installation by typing the following commands in the command prompt:

```
> node --version and
> npm --version
```

You should see an output similar to the one shown here:

```
C:\Users\pinakin>node --version
v14.17.5

C:\Users\pinakin>npm --version
6.14.14
```

***Figure 3.2**: Verifying the Node version*

If the ZIP file is downloaded, simply extract the ZIP to a convenient location and mention the node folder path in the **`PATH`** environment variable.

Installing the Node.js runtime on macOS

In order to install the Node.js runtime, visit the following URL:

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

Download the macOS installer as a pkg file. Double click on the **`.pkg`** file and follow the instructions. Verify the installation by typing the commands shown above and verify the output in a terminal. Please note that the node and npm versions might be different from what I have.

More books: https://t.me/booksforits

Installing the Node.js runtime on Linux

In order to install the Node.js runtime on a Linux machine, we will consider an AWS EC2 Linux instance. The following steps apply to any Linux machine. SSH into the Linux instance and type the following commands:

1. Install **Node version manager** (**nvm**) by typing the following at the command line:

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/
   install.sh | bash
   ```

2. Activate nvm by typing the following at the command line:

   ```
   . ~/.nvm/nvm.sh
   ```

3. Use nvm to install the latest version of Node.js by typing the following at the command line:

   ```
   nvm install node
   ```

4. Verify the installation by typing the following commands and verify the output in a terminal:

   ```
   node -v and
   ```

   ```
   npm -v
   ```

The output obtained will be as follows:

```
[ec2-user@ip-172-31-30-102 ~]$ node -v
v16.6.2
[ec2-user@ip-172-31-30-102 ~]$ npm -v
7.20.3
```

***Figure 3.3**: Verifying node and npm version on Linux*

Creating the project folder

Moving forward, we will be working with a Windows 10 system. Now, create a folder anywhere on the Windows system and name it **`MEAN`**. This folder will house our Angular application, to begin with, but we will modify the contents of this folder as and when required. This folder will contain a sub-folder for our Angular application in the upcoming sections of this chapter.

Now that we have our folder (**`MEAN`**), we will run the following command inside our root folder (**`MEAN`**) to generate the **`package.json`** file:

```
> npm init
```

More books: https://t.me/booksforits

Hit *Enter* to select the **`default`** options. It will create the **`package.json`** file for us. The sample contents get listed as shown in the following output. Type **`y`** finally:

```
package name: (mean)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\pinakin\MEAN\package.json:

{
  "name": "mean",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) y
```

***Figure 3.4**: npm init command to generate the package.json file*

If we look at the contents of our MEAN directory, we can see something as shown in the following figure. We see the generated package.json file below when we fetch the directory listing:

```
C:\Users\pinakin\MEAN>dir
 Volume in drive C is Windows
 Volume Serial Number is 7481-75CC

 Directory of C:\Users\pinakin\MEAN

14-08-2021  10:35    <DIR>          .
14-08-2021  10:35    <DIR>          ..
14-08-2021  10:35               200 package.json
               1 File(s)            200 bytes
               2 Dir(s)  911,848,947,712 bytes free
```

***Figure 3.5**: Directory structure*

More books: https://t.me/booksforits

Installing the Angular-CLI

We now have Node and NPM installed and that makes things a lot simpler now. Using npm, we will install Angular CLI using the following command:

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

The output obtained would be as follows:

```
C:\Users\pinakin\MEAN>npm install -g @angular/cli
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request
/issues/3142
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.ra
ndom() in certain circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random
 for details.
C:\Users\pinakin\AppData\Roaming\npm\ng -> C:\Users\pinakin\AppData\Roaming\npm\node_modules\@angular\c
li\bin\ng

> @angular/cli@12.2.1 postinstall C:\Users\pinakin\AppData\Roaming\npm\node_modules\@angular\cli
> node ./bin/postinstall/script.js

+ @angular/cli@12.2.1
added 237 packages from 181 contributors in 58.964s
```

***Figure 3.6**: Angular CLI installation*

After the installation of Angular CLI, next, we will generate an Angular project called **Angular-Mean**.

Creating a new Angular project

Now, we will generate a new Angular project using the following command:

```
>ng new Angular-Mean
```

The output obtained would be as follows:

More books: https://t.me/booksforits

```
C:\Users\pinakin\MEAN>ng new Angular-Mean
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE Angular-Mean/angular.json (3075 bytes)
CREATE Angular-Mean/package.json (1074 bytes)
CREATE Angular-Mean/README.md (1057 bytes)
CREATE Angular-Mean/tsconfig.json (783 bytes)
CREATE Angular-Mean/.editorconfig (274 bytes)
CREATE Angular-Mean/.gitignore (604 bytes)
CREATE Angular-Mean/.browserslistrc (703 bytes)
CREATE Angular-Mean/karma.conf.js (1429 bytes)
CREATE Angular-Mean/tsconfig.app.json (287 bytes)
CREATE Angular-Mean/tsconfig.spec.json (333 bytes)
CREATE Angular-Mean/src/favicon.ico (948 bytes)
CREATE Angular-Mean/src/index.html (297 bytes)
CREATE Angular-Mean/src/main.ts (372 bytes)
CREATE Angular-Mean/src/polyfills.ts (2820 bytes)
CREATE Angular-Mean/src/styles.css (80 bytes)
CREATE Angular-Mean/src/test.ts (788 bytes)
CREATE Angular-Mean/src/assets/.gitkeep (0 bytes)
CREATE Angular-Mean/src/environments/environment.prod.ts (51 bytes)
CREATE Angular-Mean/src/environments/environment.ts (658 bytes)
CREATE Angular-Mean/src/app/app.module.ts (314 bytes)

CREATE Angular-Mean/src/app/app.component.html (24585 bytes)
CREATE Angular-Mean/src/app/app.component.spec.ts (974 bytes)
CREATE Angular-Mean/src/app/app.component.ts (216 bytes)
CREATE Angular-Mean/src/app/app.component.css (0 bytes)
√ Packages installed successfully.
```

***Figure 3.7**: Creating a new Angular project*

Now, we see that we have a folder created called Angular-Mean. Navigate to the folder and install Angular Material.

Installing Angular Material

It provides community-maintained and accessible components for everyone to implement. The components are well-tested, documented, and reliable. Angular Material provides straightforward APIs with consistent cross-platform behavior.

It provides tools that help developers build their custom components with interaction patterns. These components are customizable within the bounds of the Material Design specification. Install Angular Material using the following command:

```
> ng add @angular/material
```

More books: https://t.me/booksforits

The following log gets generated, and it will spin up an Angular application of **http://localhost:4200**:

```
C:\Users\pinakin\MEAN\Angular-Mean>ng add @angular/material
i Using package manager: npm
√ Found compatible package version: @angular/material@12.2.1.
√ Package information loaded.

The package @angular/material@12.2.1 will be installed and executed.
Would you like to proceed? Yes
√ Package successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink        [ Preview: https://m
aterial.angular.io?theme=indigo-pink ]
? Set up global Angular Material typography styles? Yes
? Set up browser animations for Angular Material? Yes
UPDATE package.json (1140 bytes)
√ Packages installed successfully.
UPDATE src/app/app.module.ts (423 bytes)
UPDATE angular.json (3239 bytes)
UPDATE src/index.html (579 bytes)
UPDATE src/styles.css (181 bytes)
```

Figure 3.8: Adding Angular Material

Verifying that Angular works

The last thing to do is verify that Angular does indeed work. Type the following command:

```
> ng serve
```

The following logs get generated:

```
C:\Users\pinakin\MEAN\Angular-Mean>ng serve
- Generating browser application bundles (phase: setup)...Compiling @angular/core : es2015 as esm2015
Compiling @angular/common : es2015 as esm2015
Compiling @angular/platform-browser : es2015 as esm2015
Compiling @angular/platform-browser-dynamic : es2015 as esm2015
Compiling @angular/animations : es2015 as esm2015
Compiling @angular/animations/browser : es2015 as esm2015
Compiling @angular/platform-browser/animations : es2015 as esm2015
√ Browser application bundle generation complete.

Initial Chunk Files   | Names         |      Size
vendor.js             | vendor        |   2.33 MB
polyfills.js          | polyfills     | 508.67 kB
styles.css, styles.js | styles        | 458.41 kB
main.js               | main          |  55.40 kB
runtime.js            | runtime       |   6.62 kB

                      | Initial Total |   3.34 MB

Build at: 2021-08-14T06:38:32.737Z - Hash: 0430ceab8ce4af8a66ee - Time: 11789 ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhos
t:4200/ **
```

Figure 3.9: Running an Angular project

More books: https://t.me/booksforits

As indicated, navigate to **http://localhost:4200** and you will see the following page:

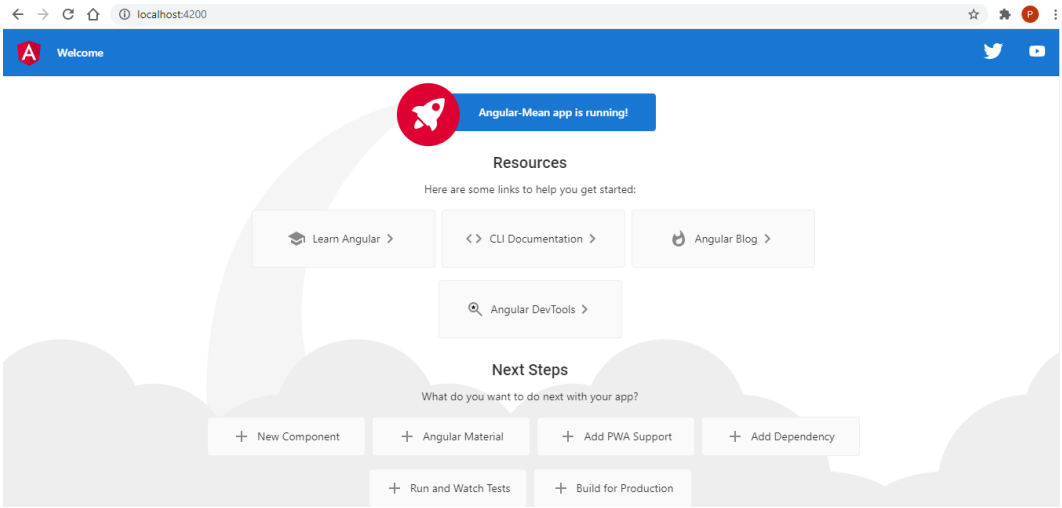

***Figure 3.10**: Verifying the Angular application of browser*

We will now take a look at the project structure.

Exploring the project structure

After the installations we did in earlier sections, we will now take a look at the project structure that got created. We will take a look at the Visual Studio code project structure:

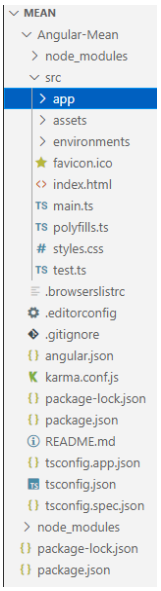

***Figure 3.11**: Project folder structure*

More books: https://t.me/booksforits

From the preceding screenshot, we can see that there are various files generated. For now, let's focus on the two **package.json** files. There is one file at the root of the **MEAN** folder and one at the root of the **Angular-Mean** folder. The one at the root of the **MEAN** folder was created when we did a **npm init** and the one at the root of the **Angular-Mean** was created when we did a **ng new** inside the **MEAN** folder.

An **app** folder was created when we did an **ng new**. This app folder is our **Main Module**. We will see what modules are in the chapter for Angular. We also see two **node_modules** folders, one at the root of the **MEAN** folder and one at the root of the **Angular-Mean** project. These folders got created when we did an **npm install -g @angular/cli** in the **MEAN** folder and **ng add @angular/material** in the **Angular-Mean** folder.

Let's now look at the **package.json** files that got created.

Scanning the package: json file

There are two **package.json** files generated for our project. One is for the main **MEAN** project, and one is for the **Angular-Mean** project. Let's first study the package. The JSON file for the **MEAN** project.

The following is the **package.json** file for the **MEAN** project:

```
{
  "name": "mean",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@angular/cli": "^12.2.1"
  }
}
```

More books: https://t.me/booksforits

Note that we have various sections. However, we will focus on the dependencies section. We can see that we have an Angular CLI dependency. When we deploy our project, this file will be very important since we will just run the command **`npm install`** and it will take all the dependencies listed in the **`dependencies`** section and install them in the local **`node_modules`** folder.

The scripts section will have all the scripts to be executed at the time of executing the project. We will make changes to the scripts section in *Chapter 7: Update Functionality and Implementation of Routing*.

A similar **`package.json`** exists for the **`Angular-Mean`** folder which has the dependencies for the Angular Material as follows:

```
"dependencies": {
    "@angular/animations": "~12.2.0",
    "@angular/cdk": "^12.2.1",
    "@angular/common": "~12.2.0",
    "@angular/compiler": "~12.2.0",
    "@angular/core": "~12.2.0",
    "@angular/forms": "~12.2.0",
    "@angular/material": "^12.2.1",
    "@angular/platform-browser": "~12.2.0",
    "@angular/platform-browser-dynamic": "~12.2.0",
    "@angular/router": "~12.2.0",
    "rxjs": "~6.6.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.11.4"
  }
```

Since we have mentioned about the **`node_modules`** folder, let's take a look at that next.

The node_modules folder

The **`node_modules`** is the folder where all dependencies get downloaded which are listed in **`package.json`** file. If we don't have a **`node_modules`** folder but have a **`package.json`** file with dependencies, then on running **`npm install`**, all the listed

More books: https://t.me/booksforits

dependencies will be downloaded in the **node_modules** folder. An excerpt of the **node_modules** folder is shown here:

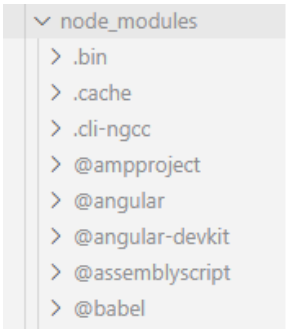

***Figure 3.12**: node_modules folder*

When we do version control of this project, we should not check-in the **node_modules** folder to GIT. The **package.json** file generates this folder for us.

Let's explore the **app** folder and the **app.component** HTML file next.

Looking at the app folder

Let's take a look at one of the main folders of our project which is the **app** folder.

Inside the **src/** folder, there is a folder called **app/**. The **app/** folder houses the front-end data and logic. This folder contains the Angular components, templates, and styles. We will take a look at each file in the **app/** folder and understand its responsibility:

- **app.component.ts**: The logic for the application's root component, namely **AppComponent**, is defined here. The view associated with **AppComponent** becomes the root of the view hierarchy as components and services are added to the application. A view hierarchy is a tree of related views that can be acted on as a single unit. The root view acts as a component's host view.
- **app.component.html**: The HTML template associated with the **AppComponent** is contained in this file.
- **app.component.css**: This defines the stylesheet for the **AppComponent**.
- **app.component.spec.ts**: This houses the unit test for the **AppComponent**.
- **app.module.ts**: The **root** module named **AppModule** is defined here. It tells Angular how the application is to be assembled. Initially, only the **AppComponent** is declared. As more components are added to the **app**, they must be declared here.

More books: https://t.me/booksforits

Let's now see how an Angular application starts.

Starting an Angular application

Let's understand how an Angular application gets started. There exists a file called **`main.ts`** which acts as a starting point for the Angular application. The **`main.ts`** file is as shown here:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-
dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

In the **`main.ts`** file, we see that there is a **`bootstrap`** method, and this **`bootstrap`** method is responsible for starting the Angular application. It refers to **`AppModule`** which looks in the **`app/`** folder. Lets' take a look at the **`app.module.ts`** file:

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

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/
animations';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
```

More books: https://t.me/booksforits

```
    BrowserModule,
    BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

It can be seen in the **app.module** file that there is a **bootstrap** array, which is a list of components. Next, comes the **app.component.ts** file shown as follows:

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular-Mean';
}
```

We can see from the **app.component.ts** file that the component selector is **app-root** and the template file to be used is **app.component.html**. The content of this file will be displayed on the screen. The **app.component.html** is a huge file containing a lot of tags which render the page. If you right click on the browser and click on **View source**, the file that is displayed is the **index.html**. Let's take a look at the **index.html** page next.

Listed below is the **index.html** file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularMean</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

More books: https://t.me/booksforits

```
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link href="https://fonts.googleapis.com/
css2?family=Roboto:wght@300;400;500&display=swap" rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
</head>
<body class="mat-typography">
  <app-root></app-root>
</body>
</html>
```

It can be seen that we have a tag called **`<app-root>`**. This is the place where the content of app.component.html gets displayed. We will start changing all these files from the next chapter.

With this, we conclude this chapter.

Conclusion

In this chapter, we took a look at the installation of Node.js and Angular in this project. We created the project folder. We initialized Node.js in the folder. We then went ahead and installed Angular CLI and generated a new Angular project. Inside this Angular project, we installed Angular material. We then verified that Angular does in fact work with the current setup. We then explored the project structure followed by a look at the **`package.json`** file. We took a look at the **`node_modules`** folder followed by the app folder. Finally, we saw how an Angular application starts.

In the next chapter, we will take a detailed look at the creation of our front end using Angular and Bootstrap. We will see the benefits of Angular Material for designing a beautiful UI.

Questions

1. Explain how to initialize a Node.js application.
2. Explain how to install Angular using npm.
3. Explain the role of the **`app`** folder.
4. Explain the role of **`package.json`**.
5. Explain how an Angular application starts.

More books: https://t.me/booksforits

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Chapter 4
Creation of the Frontend Using Angular

In *Chapter 3, Installing the Components,* we focused on the installation of the required software for the MEAN application. We started with the installation of the Node.js runtime. We created an empty folder on the local machine and initialized it with **`npm`**. We then installed the Angular CLI and created a new Angular project using the **`ng new`** command. We installed Angular Material here and verified that Angular does in fact work. We took a look at the folder structure by understanding the **`package.json`** file, **`node_modules`** folder, and **`app`** folder. Finally, we saw how an Angular application starts.

In this chapter, we will be building our frontend. This will be a bare-bones frontend which will be integrated with MongoDB and the server framework later. We will start with a bird's eye view of how angular works. We will customize the current Angular application.

Structure

In this chapter, we will predominantly focus on:

- Understanding the folder structure
- Working with components
- Creation of the NavBar component
- Routing in Angular

More books: https://t.me/booksforits

- Few routing directives
- Implementing the home page
- Implementing the register page
- Getting input from the user
- Angular Material

We will next check the objectives of this chapter.

Objectives

After reading this chapter, the reader will be able to create the front-end application using Angular and Angular Material. This will be a bare-bones front-end application that will be plugged into the database and the Node.js server later. The reader will be able to understand the component structure of Angular as we design the front end.

Let's start by getting a basic understanding of the folder structure that was created for us when we executed the **`ng new`** command.

Understanding the folder structure

We now open the project in Visual Studio Code and check the project structure. We can see several files that get created. We also see an **`src`** folder, as shown in the following figure:

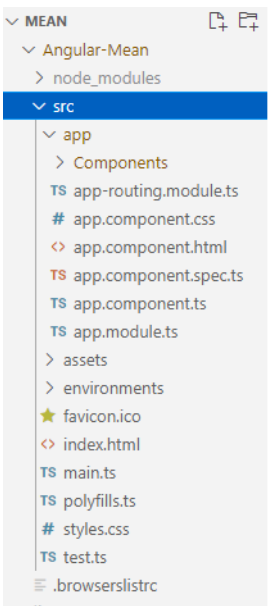

***Figure 4.1**: The folder structure*

More books: https://t.me/booksforits

Let's go inside the **`src`** folder and check what got created. The **`Components`**, **`Environments`**, **`Assets`** folders and the **`app-routing.module.ts`** (refer to *Figure 4.2*) have been created manually which we will see in the later part of this chapter:

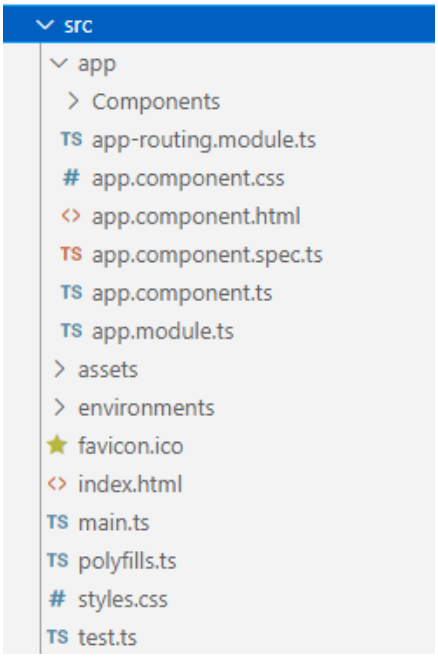

***Figure 4.2**: Important components of the folder structure*

We see **`app.component.ts`**, **`app.component.html`**, **`app.module.ts`** that were created for us. We already took a look at these three files in *Chapter 3: Installing the Components*.

Working with components

Angular applications are made up of components. Components are the basic UI building blocks of an Angular app. An Angular app consists of a tree of Angular components. Angular components are nothing but a subset of directives that are always associated with a template. Unlike other directives, only one component can be instantiated for a given element in a template. Each component is made of:

- An HTML template that declares what to render on the page.
- A Typescript class that contains logic.
- A CSS selector that defines how to style the template of the component.

First, let us look at the high-level view of what a component structure is.

More books: https://t.me/booksforits

General component structure of an Angular app

The structure of a webpage typically consists of a header, footer, left pane, and right pane. In some applications, the right and left panes may not be there but the header and footer are commonly seen.

The structure of a typical web page is as follows:

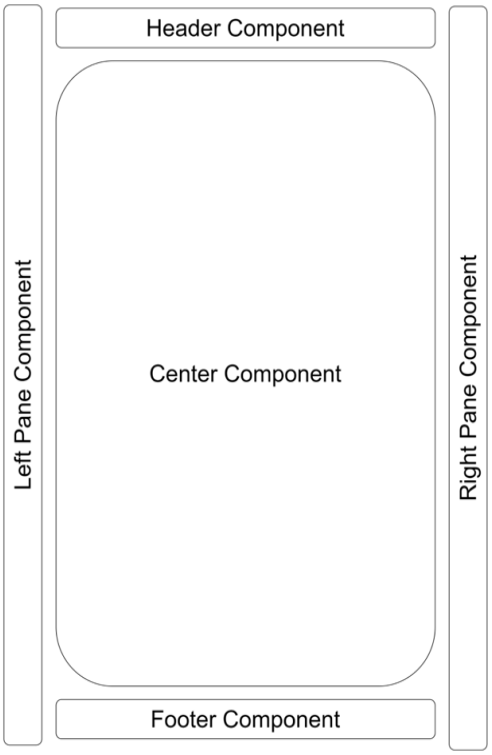

***Figure 4.3**: The component structure*

An Angular app can consist of various sections as displayed in the preceding figure and all sections can be populated with the content independently.

Each of the components will have its own folder structure. This facilitates the reuse of the components time and again. We can even be more granular by putting parts of these components into individual components. We can put a textbox for example in its own component. Remember that the right level of granularity is required to reduce the complexity of our application.

Let us look at the component structure that we will implement.

More books: https://t.me/booksforits

Component structure for our application

The components for our sample application will consist of the following:

- Home and register component
- Nav Bar component

At the first load of the web page, there will be a blank area in which other components will be loaded when the corresponding menu from **`Nav Bar`** is clicked. The **`Nav Bar`** will look as shown in the following figure. *Figure 4.4* contains links for **Home**, **register**, **Profile**, **Blog**, **Login**, and **Logout**. We also have a brand for our **Student Blog** as shown in the following figure:

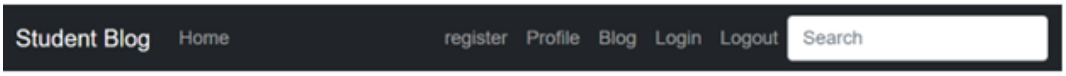

***Figure 4.4**: The Bootstrap Navbar*

We will implement the **Home** and **register** screens in this chapter since these can be created as placeholder pages without the need for a database. We will simply put a **`placeholder`** text for the profile page but we will fully implement the profile page when we learn about authentication.

The mockup screen for creating a blog will look something similar to the following figure:

***Figure 4.5**: The Mockup of Create Post form5*

Next, let's move on to get some content on the screen

More books: https://t.me/booksforits

Creation of Nav Bar component

Till now we have the core **app** module with us which was added automatically when we did an **ng new**. Let's create our first component which is the **Nav Bar** component.

Navigate to the app folder and type the command as follows:

```
> ng generate component NavBar
```

Four files get generated as follows:

***Figure 4.6**: The component structure of NavBar*

When the NavBar component gets created by the preceding command, the corresponding entries of this component get added to the **app**:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/
animations';
import {AppRoutingModule} from './app-routing.module';
import {HttpClient, HttpClientModule} from '@angular/common/http';
import { NavBarComponent } from './Components/nav-bar/nav-bar.component';

@NgModule({
  declarations: [
    AppComponent,
    NavBarComponent],
  imports: [
    BrowserModule,
    HttpClientModule,
  ],
```

More books: https://t.me/booksforits

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

We now take a closer look at the component TypeScript and HTML files that got created. For any component, we need to make changes to only the TypeScript and HTML files.

Let's take a look at the TypeScript file first

Looking at the nav-bar.component.ts

The **`nav-bar.component.ts`** file that is listed below imports **`Component`** and **`OnInit`** from Angular Core. We have the **`@Component`** decorator next which indicates that this is an **`Angular`** component. This takes in an object as a parameter. In the following example, we have **`selector`**, **`templateUrl`**, and **`styleUrls`** properties. The **`selector`** indicates how this component should be accessed at other places in the HTML file. In our case (it will be accessed in the **`app.component.html`**). This way we plug the **`Nav Bar`** into the **`App`** component. The template URL indicates the HTML file corresponding to this TypeScript file:

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

@Component({
  selector: 'nav-bar',
  templateUrl: './nav-bar.component.html',
  styleUrls: ['./nav-bar.component.css']
})
export class NavBarComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

Let's take a look at the **`App`** component HTML snippet where we plug our **`NavBar`**:

```
<nav-bar></nav-bar>
```

More books: https://t.me/booksforits

It is a simple tag which should match the selector in the NavBar component TypeScript file. Let's have a look at the component HTML file next.

Looking at the nav-bar.component.html

The **NavBar** component file is a simple template taken from Bootstrap.com and modified as per our needs. The main things to be noted here are the sections for **Profile**, **Home**, and **Register**. Here we see two new things, that is, **routerLink** and **routerLinkActive**:

```
<nav class="navbar navbar-expand-sm navbar-dark bg-dark mt-4">
  <div class="container-fluid">
      <a class="navbar-brand" routerLink="/home">Student Blog</a>
      <button type="button" data-toggle="collapse" data-target="#navbar-
menu" class="navbar-toggler">
           <span class="navbar-toggler-icon"></span></button>
      <div class="collapse navbar-collapse" id="navbar-menu">
          <ul class="navbar-nav mr-auto">
              <li [routerLinkActive]="['active']"
[routerLinkActiveOptions]="{exact:true}"><a class="nav-link"
routerLink="/home">Home</a></li>
          </ul>
  <ul class="navbar-nav">
    <li [routerLinkActive]="['active']"
[routerLinkActiveOptions]="{exact:true}"><a class="nav-link"
routerLink="/register">register</a></li>
    <li [routerLinkActive]="['active']"
[routerLinkActiveOptions]="{exact:true}"><a class="nav-link"
routerLink="/profile">Profile</a></li>
    <li><a href="#" class="nav-link">Blog</a></li>
    <li><a href="#" class="nav-link">Login</a></li>
    <li><a href="#" class="nav-link">Logout</a></li>
          </ul>
          <form>
              <input type="text" placeholder="Search" class="form-
control">
          </form>
      </div>
  </div>
</nav>
```

More books: https://t.me/booksforits

Now, we have a few new attributes/directives with us which are **`routerLink`**, **`routerLinkActive`**, and **`routerLinkActiveOptions`**; we will understand what these are in terms of Routing in Angular.

Routing in Angular

In a single-page app, rather than going out to the server to get a new page, the page that gets displayed to the user is constructed by showing or hiding portions of the view that correspond to particular components, users are shown different views as they perform different tasks in the application.

The Angular Router is used to handle the navigation from one view to the next. The Router enables navigation by interpreting a browser URL as an instruction to change the displayed view.

To implement Routing in our project, we will add an app routing module so that the Routing mechanism is in a single place. The module is a container for the different parts of an application. The module is a container for the application controllers. Controllers always belong to a particular module.

Routing module

We will create the common routing module next. This file is manually created and as discussed in the subsequent steps, we will see how to create this file and where it should be placed. We will also understand the need for creating a separate **`routing.module.ts`** file.

The code of the **`Routing`** module is shown below. We import our components along with the **`Routes`** and **`RouterModule`** from Angular Router. As shown in the following code snippet, we have the **`HomeComponent`**, **`ProfileComponent`**, and **`RegisterComponent`** imported. These components will be created using the **`ng g c <component name>`** command. Once created, these components should be added here along with the routes:

```
import { NgModule } from '@angular/core';
import { Routes,RouterModule} from '@angular/router';
import {HomeComponent} from './Components/home/home.component'
import {ProfileComponent} from './Components/profile/profile.component'
import {RegisterComponent} from './Components/register/register.
component'
```

Next, we have our routes defined as follows. We have routes for **Profile** and **register**. Next, we see a path with a blank entry. This is the case when the user navigates to **http://localhost:4200**. When the user navigates to **http://localhost:4200/profile**, he

More books: https://t.me/booksforits

gets redirected to the profile page and when he navigates to **http://localhost:4200/register**, he gets redirected to the Register page. Remember **Profile** and **register** are both components. ****** indicate any other route:

```
const appRoutes: Routes = [
  {path: 'profile', component: ProfileComponent},
  {path: 'register', component: RegisterComponent},
  {path: '', component: HomeComponent},
  {path: '**', component: HomeComponent}


]
```

Then, we come to the **@NgModule**. The **@NgModule** annotation defines the module.

Here are a few characteristics of **NgModule**:

- Components, directives, and pipes that are part of the module can be listed in the declarations array.
- Other modules can be imported by listing them in the imports array.
- Services that are part of the module can be listed in the providers array.

This declarative grouping is useful for organizing our view and documenting which functionality is closely related. We finally export the **RouterModule**. The **forRoot()** method creates a **NgModule** that contains all the directives, the given routes, and the **Router** service itself:

```
@NgModule({
  declarations: [],
  imports: [RouterModule.forRoot(appRoutes)],
  providers: [],
  bootstrap: [],
  exports:[RouterModule]
})
export class AppRoutingModule { }
```

Since we are in the **Router** section, let's understand the **RouterLink**, **RouterLinkActive**, and **RouterLinkActiveOptions** next.

Few Routing directives

In this section, we look at the following three common routing directives:

More books: https://t.me/booksforits

- **`RouterLink`**
- **`RouterLinkActive`**
- **`RouterLinkActiveOptions`**

Let's start with the **`RouterLink`** directive.

The RouterLink directive

When the **`RouterLink`** directive is applied to an element in a template, it converts that element to a link that initiates the navigation to a route. One or more components are opened up in one or more **`<router-outlet>`** locations on the **`app.component.html`** when the user clicks on any menu or button on the page.

A sample **`RouterLink`** is as follows:

```
<li [routerLinkActive]="['active']"
[routerLinkActiveOptions]="{exact:true}"><a class="nav-link"
routerLink="/register">register</a></li>
```

We see two more options **`[routerLinkActive]`** and **`[routerLinkActiveOptions]`**.

The RouterLinkActive directive

This directive creates a visual distinction for elements associated with an active route. For example, the preceding code highlights the word **register** when the router activates the associated route. This becomes possible due to the addition of an *active* CSS class.

The RouterLinkActiveOptions directive

The **`RouterLinkActiveOptions`** determines the options to be configured to determine whether a link is active. These options are passed to the **`Router.isActive()`** function which determines whether a URL is activated. The **`{exact:true}`** option highlights the link only if an exact match is found.

Let's move ahead and implement the **`Home`** page.

Implementing the Home page

As indicated earlier, the two files that we need to consider are the **`component.ts`** and **`component.html`**. First of all, we will generate the **`Home`** page component using the following command:

```
> ng generate component home
```

More books: https://t.me/booksforits

This generates the required files. Let's look at the home component files

Looking at the Home component's TypeScript code

The following code is the **Home** component Typescript code:

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


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


  constructor() { }


  ngOnInit(): void {
  }


}
```

The template file is **home.component.html** which we will see next.

Looking at the Home component's HTML code

While running the code to generate the component, CLI generates the required files which also contains the HTML file. If the HTML file is opened, it will either be empty or will contain ** Home works **. The following code is the **Home** Component's HTML code:

```
<div class="jumbotron text-center">
  <h1>Student Blogging Application</h1>
  <p class="lead">Welcome to the Student Blogging application by
<strong>Pinakin Chaubal</strong></p>
  <div>
```

More books: https://t.me/booksforits

```
    <a routerLink="/register" class="btn btn-primary">Registration</a>
    <a href="#" class="btn btn-default">Login</a>
  </div>
</div>
```

Following this, we have the code for putting **Information** icons to give a beautiful look and feel. We offer the user the option to register and login:

```
<div class="row">
  <div class="col-sm-6 col-md-4">
    <div class="thumbnail">
      <img src="https://image.flaticon.com/icons/png/512/4786/4786245.
png" alt="placeholder image" width="140" height="140">
      <div class="caption">
        <h3>Register for free</h3>
        <p>Register for free on the site and start blogging</p>
      </div>
    </div>
  </div>


  <div class="col-sm-6 col-md-4">
    <div class=thumbnail> <img src="https://image.flaticon.com/icons/
png/512/417/417777.png" alt="Generic placeholder image" width="140"
height="140">
      <div class=caption>
        <h3>Login Securely</h3>
        <p>Login Securely to the blogging application</p>
      </div>
    </div>
  </div>
```

We also give the user option to create, update, and delete his own blogs as well as view other students' blogs:

```
  <div class="col-sm-6 col-md-4">
    <div class=thumbnail> <img src="https://image.flaticon.com/icons/
png/128/4786/4786243.png" alt="Generic placeholder image" width="140"
height="140">
```

More books: https://t.me/booksforits

```
      <div class=caption>
        <h3>Create, update and delete Blogs</h3>
        <p>This app lets you create, update and delete blogs</p>
      </div>
    </div>
  </div>
  <div class="col-sm-6 col-md-4">
    <div class=thumbnail> <img src="https://image.flaticon.com/icons/
svg/417/417768.svg" alt="Generic placeholder image" width="140"
height="140">
      <div class=caption>
        <h3>View</h3>
        <p>View Blogs</p>
      </div>
    </div>
  </div>

  <div class="col-sm-6 col-md-4">
    <div class=thumbnail> <img src="https://image.flaticon.com/icons/
svg/417/417754.svg" alt="Generic placeholder image" width="140"
height="140">
      <div class=caption>
        <h3>Dashboard</h3>
        <p>View the dashboard</p>
      </div>
    </div>
  </div>
  <div class="col-sm-6 col-md-4">
    <div class=thumbnail> <img src="https://image.flaticon.com/icons/
svg/417/417764.svg" alt="Generic placeholder image" width="140"
height="140">
      <div class=caption>
        <h3>Add, update and delete comments</h3>
        <p>This app lets you add, update and delete your comments</p>
      </div>
```

More books: https://t.me/booksforits

```
    </div>
  </div>
```

In the following code, by clicking on the **Registration** button, the user gets redirected to the registration page:

```
<a routerLink="/register" class="btn btn-primary">Registration</a>
```

Next, we move on to exploring the typescript and template files for the **Register** page.

Implementing the Register page

Let us look at what is involved in the creation of the **Register** page. We will look at the TypeScript and Template code for the **Register** page.

Looking at the Register component's TypeScript code

Here is the **Register** component's typescript code. We make use of Reactive forms in Angular for this. The reactive forms module needs to be imported in **`app.module.ts`**. We have **`FormBuilder`**, **`FormGroup`**, and **`Validators`** for this purpose. **`FormBuilder`** provides a syntax that reduces boilerplate code to build complex forms:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
```

A **`FormGroup`** aggregates the values of each child **`FormControl`** element into a single object. The control name is the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the whole group becomes invalid.

Along with **`FormControl`** and **`FormArray`**, **`FormGroup`** is one of the three fundamental building blocks that are used to define forms in Angular.

Here the form consists of **`email`**, **`username`**, **`password`**, and **`confirm`**:

```
@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  form: FormGroup;
```

More books: https://t.me/booksforits

```
  createForm(){
    this.form = this.formBuilder.group({
      email: '',
      username: '',
      password: '',
      confirm: ''
    })
  }
  constructor(
    private formBuilder: FormBuilder
  ) {
    this.createForm()
  }

  ngOnInit(): void {
  }

}
```

We will now look at the template code.

Looking at the Register component's template code

We finally look at the following template code for our **Registration** component. This is a very simple data entry form for registration with four fields **UserName**, **Email**, **Password**, and **Confirm Password**:

```
<!-- Registration Form -->
    <form [formGroup]="form">
    <!-- Username Input -->
    <div class="form-group">
      <label for="username">Username</label>
      <div>
        <input type="text" name="username" class="form-control"
```

More books: https://t.me/booksforits

```
autocomplete="off" placeholder="*Username" formControlName="username"/>
      </div>
    </div>

    <!-- Email Input -->
    <div class="form-group">
      <label for="email">Email</label>
      <div>
        <input type="text" name="email" class="form-control"
autocomplete="off" placeholder="*Email" formControlName="email"/>
     </div>
    </div>

    <!-- Password Input -->
    <div class="form-group">
      <label for="password">Password</label>
      <div>
        <input type="password" name="password" class="form-control"
autocomplete="off" placeholder="*Password" formControlName="password" />
      </div>
    </div>

    <!-- Confirm Password Input -->
    <div class="form-group">
      <label for="confirm">Confirm Password</label>
      <div>
        <input type="password" name="confirm" class="form-
control" autocomplete="off" placeholder="*Confirm Password"
formControlName="confirm" />
      </div>
    </div>

    <!-- Submit Input -->
    <input type="submit" class="btn btn-primary" value="Submit" />
  </form>
  <!-- Registration Form /-->
```

More books: https://t.me/booksforits

We will next make our registration form accept input from the user.

Getting input from the user

We will next get values from the user and for this, we will add the following at the end of the register template. This is known as **string interpolation** and is used to display values entered on the screen:

```
<p>Username: {{form.controls.username.value}}  </p>
<p>Email: {{form.controls.email.value}}  </p>
<p>Password: {{form.controls.password.value}}  </p>
<p>Confirm: {{form.controls.confirm.value}}  </p>
```

Now, when we type anything in the boxes, the text appears as shown in *Figure 4.7*:

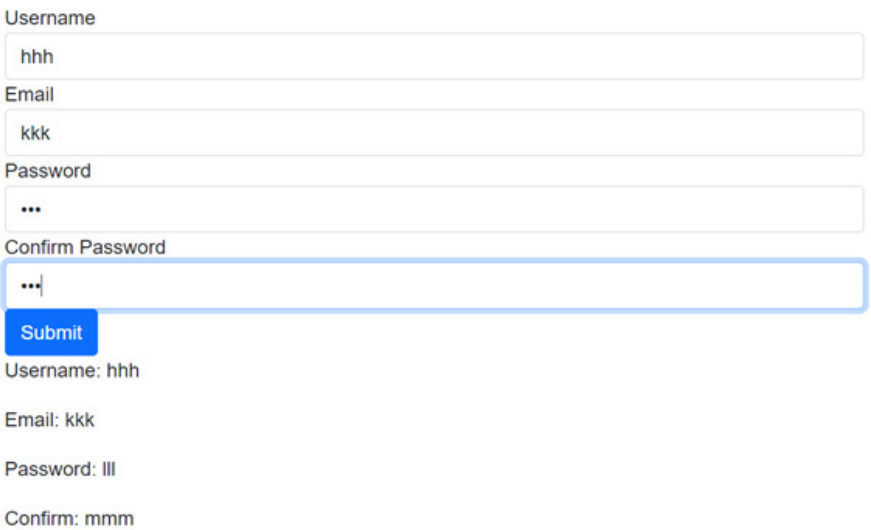

***Figure 4.7**: Data binding in action*

This is data binding in action. Next, we move to the final topic of this chapter which is Angular Material.

Angular Material

Angular Material provides internationalized and accessible components which are well-tested to ensure reliability and performance. It has simple APIs with cross-platform behavior. It provides tools that help developers build their custom components. It is customizable within the bounds of the Material Design specification. It has been built by the Angular team to integrate seamlessly with Angular.

More books: https://t.me/booksforits

We will see more on Angular Material in the upcoming chapters.

Conclusion

In this chapter, we took a look at the creation of the front end using Angular components, routers, and reactive forms. We looked at the folder structure and understood the Angular component structure. Then, we created components for the **NavBar**, **Home**, and **Register**. Then, we added string interpolation to pull values from the text boxes on the **Register** page and displayed those on the screen. Finally, we looked at Angular Material. With this, we have a few components which could be developed independently of the database and server.

In the next chapter, we will take a detailed look at adding the Node.js server and connecting it with the Angular frontend.

Questions

1. Explain the folder structure of an Angular application.
2. Explain the component structure of Angular.
3. Explain what is meant by a template file.
4. Explain what is meant by a TypeScript file.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Chapter 5
Addition of Node.js and Ideas for Integration

We saw how to design the **NavBar**, **Home**, and **Register** pages in *Chapter 4: Creation of Frontend Using Angular*. We took a look at the app components: the main part of our Angular application and also other components that are children of the **App** component. We took a look at what routing in Angular is and looked at the various routing-related directives. We also implemented a few routes. We saw how we could take input from a user and display it on the screen using interpolation, that is, **{{}}**. Eventually, we took a look at Angular Material and how it can be beneficial in developing an Angular application.

This chapter will explore the backend server part of our MEAN framework which comprises Node.js [Open-source, cross-platform backend JavaScript runtime environment that runs on V8 engine] Express.js [back-end web-application framework for Node.js]. We will start with the basics of building backend logic using Express and Node.js. In this process, we will make use of Postman [an API platform for building and using APIs.] Postman simplifies the API lifecycle and streamlines the developer's collaboration so that better APIs can be created faster. We will also see two main HTTP verbs for building the backend code which are **GET** and **POST**.

Structure

This chapter will predominantly focus on the following:

- What is an API?

More books: https://t.me/booksforits

- Verbs used for building REST APIs
- Creation of posts and comments projects
- Testing of the individual services
- Ideas of connecting with frontend

Objectives

After reading this chapter, the reader will be able to implement APIs using Express and Node.js. This will be a bare-bones backend that will be plugged into the database later. The reader will be able to understand the process of creating APIs such that each can be developed independently.

Let's start by getting a basic understanding of what an API is.

Understanding an API

Application Programming Interface (**API**) allows different applications to interact with each other. We will build a REST API in this chapter. The term REST stands for REpresentational State Transfer, an architectural style designed to communicate with web services. REST API is well suited for MEAN stack backend development. There is one more type of API which is the SOAP API. SOAP API is the older of the two. We will create APIs to create and list posts and comments. We will not make use of any database in this chapter. We will simply create an in-memory array to store the posts and comments. The posts and comments will form what are called services. The posts and comments services will be developed individually. In this process, we will use certain HTTP verbs like `GET`, `POST`, `PUT`, and `DELETE`.

Let us study what those verbs will be in the next section.

Verbs used in building REST APIs

HTTP defines a set of request methods which indicate a desired action to be performed on a given resource. Four verbs are used for API development: GET, `POST`, `PUT`, and `DELETE`. These represent the CRUD operations [Create, Read, Update, and Delete]. There are other verbs too like `PATCH`, `OPTIONS`, and `HEAD` but we will focus on the first four verbs. The four verbs are listed here:

- `GET`: The HTTP `GET` method is used to read (or retrieve) a representation of a resource which is a service in our case. In case of a successful response, `GET` returns a representation in XML or JSON and an HTTP response code of 200 (`OK`). We will focus on JSON in this book. In an error case, it often returns a 404 (`NOT FOUND`) or 400 (`BAD REQUEST`). `GET` is idempotent which means that calling it once or 100 times will have the same effect.

More books: https://t.me/booksforits

- **POST**: The HTTP **POST** verb is most often used to create new resources. In case of a successful creation of a resource, the status code of 201 is returned along with a message of **Content Created**.
- **PUT**: The HTTP **PUT** verb is used to update existing resources. In case of a successful update of a resource, the status code of 200 is returned along with a JSON representation of the updated resource. **PUT** is an idempotent resource, that is, it makes updates to the same resource.
- **DELETE**: The HTTP **DELETE** verb is used to delete existing resources. In case of a successful deletion of a resource, the status code of 200 is returned along with a JSON representation of the updated resource. **DELETE** is an idempotent operation, that is, it deletes the same resource.

Creating the posts and comments projects

We will now create the posts and comments project inside the MEAN project. The posts and comments projects will be created in a separate folder under the **MEAN** folder.

Let us start with the posts project.

Creating the posts project

Navigate to the newly created **posts** folder and type the command as follows:

```
> npm init -y
```

Giving **-y** or **-yes** automatically answers **yes** to any prompt that npm might print on the command line. Once this command gets executed, we get the **package.json** file. Now, install **express**, **cors**, and **nodemon** as follows:

```
> npm install express cors nodemon
```

We next turn to **package.json** in the posts project. Note that the default script has been changed to the one shown below. Whenever **index.js** is changed, Node gets restarted due to Nodemon thus saving development time:

```
{
  "name": "posts",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
```

More books: https://t.me/booksforits

```
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "nodemon": "^2.0.12"
  }
}
```

The file that we need to see next is the **index.js** file. In this, we bring in the express and crypto packages. Then, we call the express function "**express()**" and put the new Express application inside the **app** variable (in order to start a new Express application). It's something like creating an object of a class. The **express.json()** function is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on body-parser which is now deprecated. In order to use this, we use it as an argument to the **app.use** function. We then have an object called posts which has been initialized to an empty object. We have the **get** function in which we simply send back the posts array. We have the post function which generates a random hex number as the ID with the help of the **crypto** package. We extract the body of the request into the **title** and **content**. We then fill in the posts object and put the **id** as the subscript. We put in the **id**, **title**, and **content** in the object and then return this object with a status code of 201(**Created**). In the last part, we make the server listen on port **4000**:

```
const express = require('express');
const {randomBytes}=require('crypto');
const app = express();
app.use(express.json());
const posts = {};
app.get('/posts', (req,res)=> {
    res.send(posts);
});
app.post('/posts',(req,res) => {
    console.log('I am in')
    const id = randomBytes(4).toString('hex');
    const {title, content} = req.body;
    posts[id]={
```

More books: https://t.me/booksforits

```
            id,title,content
        };
        res.status(201).send(posts[id])
    });
    app.listen(4000,() => {
        console.log('Listening on 4000')
    });
```

Let us create the comments project next.

Creating the comments project

Navigate to the newly created **`comments`** folder and type the command shown here:

```
> npm init -y
```

Giving **`-y`** or **`-yes`** automatically answers "**`yes`**" to any prompt that npm might print on the command line. Once this command gets executed, we get the **`package.json`** file. Now, install express, **`cors`**, and **`nodemon`** as follows:

```
> npm install express cors nodemon
```

The file that we need to see next is the **`index.js`** file. In this code, we send a specific comment filtered by post ID or a blank array in the **`get`** function and in the **`post`** function, we extract the comments into a variable and then use the **`push`** function to push the **`commentId`** and the comment contents in the **`comments`** array. We push this into the **`commentsByPostId`** object:

```
    const express = require('express');
    const {randomBytes}=require('crypto');
    const app = express();
    app.use(express.json());
    const commentsByPostId = {};
    app.get('/posts/:id/comments', (req,res)=> {
        res.send(commentsByPostId[req.params.id]||[]);
    });
    app.post('/posts/:id/comments',(req,res) => {
        console.log('I am in')
        const commentId = randomBytes(4).toString('hex');
        const  {commentContents}  = req.body;
```

More books: https://t.me/booksforits

```
    const comments=commentsByPostId[req.params.id] || [];
    comments.push({id: commentId,commentContents});
    console.log(comments);
    commentsByPostId[req.params.id]=comments;
    res.status(201).send(comments)
});
app.listen(4001,() => {
    console.log('Listening on 4001')
});
```

Let us now test our services.

Testing of the services

We have created two services, that is, posts service and comments. It's time to test these services next. We start with the posts service.

Testing the posts service

Navigate to the **posts** directory on the command prompt and start the posts service by issuing the command **npm start**.

The console displays the log as shown in *Figure 5.1*:

```
C:\Users\pinakin\MEAN\Posts>npm start

> posts@1.0.0 start C:\Users\pinakin\MEAN\Posts
> nodemon index.js

[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Listening on 4000
```

***Figure 5.1**: Logs for posts service*

Open the Postman tool and first of all, go to the **Headers** section and add a header called **content-type** with the value **application/json**. Select the **POST** option from the dropdown and fill in the URL as **http://localhost:4000/posts**. Add the content as shown in the following figure in the body and click on **Send**.

More books: https://t.me/booksforits

The response received contains the randomly generated **`id`**, the **`title`** of the post, and the **`content`**. By doing this, we have added the post in memory, as shown in *Figure 5.2*:

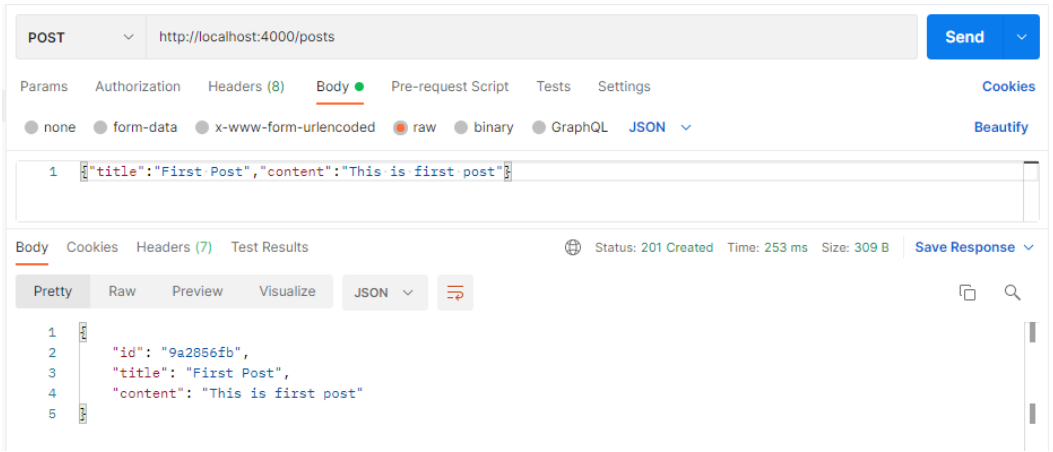

***Figure 5.2**: POST request for the posts service*

Now, we verify the contents of the newly added post by doing a **`GET`** request as shown in *Figure 5.3*:

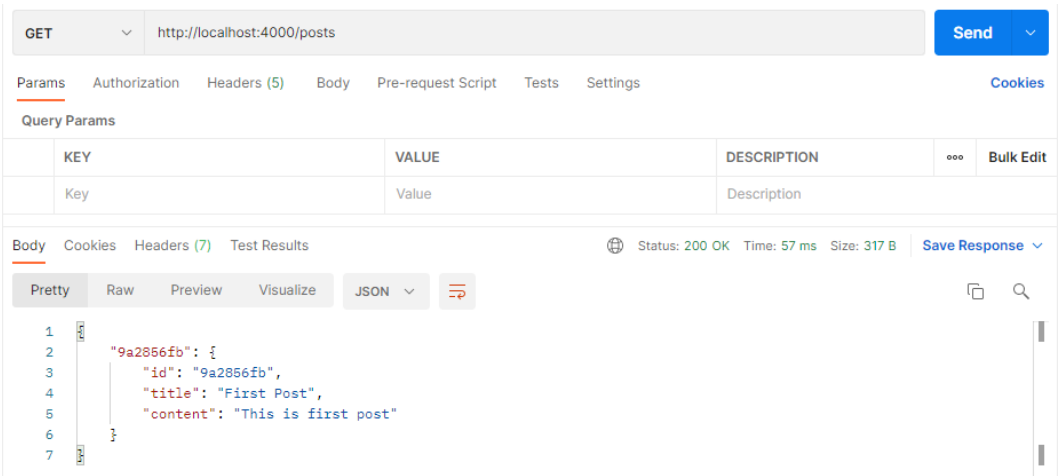

***Figure 5.3**: GET request for the posts service*

Next, we will test the comments service.

More books: https://t.me/booksforits

Testing the comments service

Navigate to the **comments** directory on the command prompt and start the comment service by issuing the command **npm start**. The comments service starts on port **4001**. The console displays the log as shown in *Figure 5.4*:

```
C:\Users\pinakin\MEAN\comments>npm start

> comments@1.0.0 start C:\Users\pinakin\MEAN\comments
> nodemon index.js

[nodemon] 2.0.12
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Listening on 4001
```

***Figure 5.4**: Starting the comments service*

Open the Postman tool and first of all, go to the **Headers** section and add a header called **content-type** with the value **application/json**. Select the **POST** option from the dropdown and fill in the URL as **http://localhost:4001/posts/123/comments**. Note that the comments service is not aware of what posts exist in the system and **123** is a dummy post ID. There will be comments associated with a post ID and hence, we provide the post ID to create comments for that particular post ID. Add the content shown in the following figure in the body and click on **Send**.

The response received contains the randomly generated ID and the content. By doing this, we have added the comments in-memory. We have added two comments for the post ID **123** as shown in *Figure 5.5*:

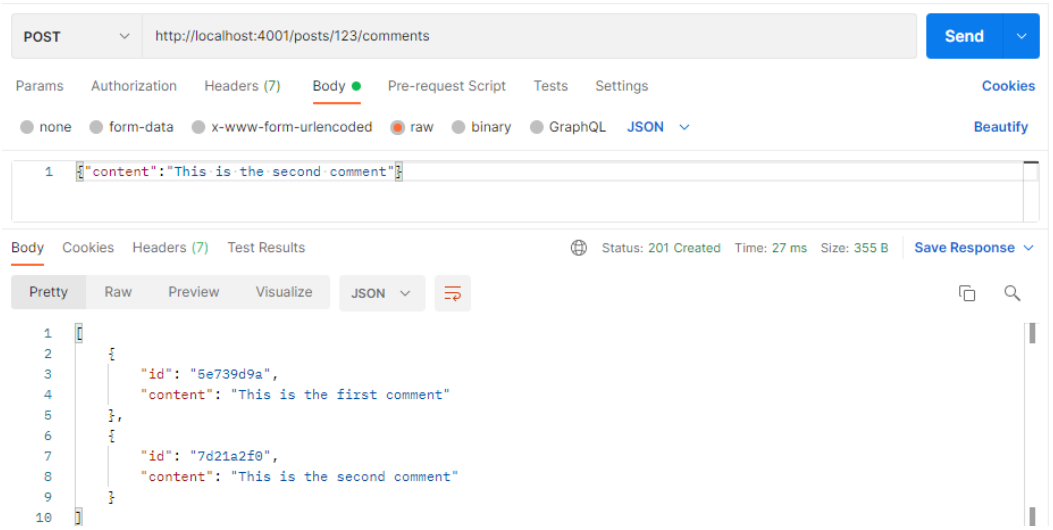

***Figure 5.5**: POST request for the comments service*

More books: https://t.me/booksforits

Let us now fetch the created comments using a **GET** request:

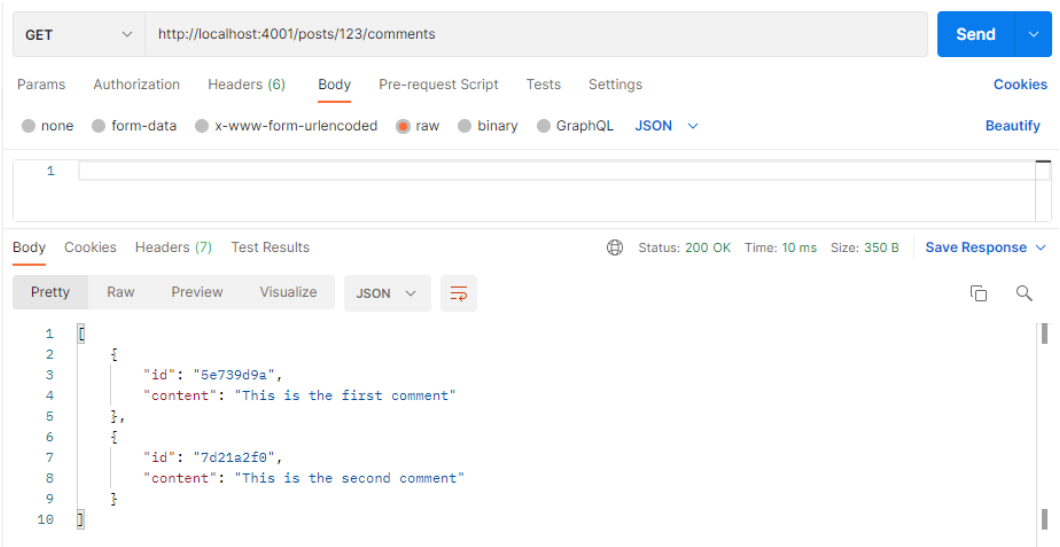

***Figure 5.6**: GET request for the comments service*

Ideas of connecting with frontend

We will connect our backend code to work with the frontend code. We will not connect it directly from Angular to Node. We will generate messages using NATS so that the individual microservices can communicate with each other.

NATS Streaming is a data streaming system powered by NATS. It has been written in the Go programming language. The name of the executable for the NATS Streaming server is nats-streaming-server. NATS Streaming operates seamlessly with the core NATS platform. The NATS Streaming server is an open-source software under the Apache-2.0 license. The NATS Streaming server is actively maintained and supported by Synadia.

A few salient features of the NATS streaming server are as follows:

- **Enhanced message protocol**: The NATS Streaming server implements its own enhanced message format using the Protocol Buffers provided by Google. These messages get transmitted as binary message payloads through the core NATS platform. Thus, they require no changes to the basic NATS protocol.
- **Message/event persistence**: The NATS Streaming server offers configurable message persistence: in-memory, flat files, or database. The storage subsystem makes use of a public interface that allows contributors to develop custom implementations.

More books: https://t.me/booksforits

- **At-least-once-delivery**: NATS Streaming provides the facility of acknowledging messages between publisher and server (for publish operations) and between subscriber and server (confirm message delivery). Messages are persisted by the server in memory or secondary storage (or external storage) and will be delivered again to eligible subscribing clients on a need basis.

We will incorporate the NATS streaming server in the upcoming chapters.

Conclusion

In this chapter, we took a look at the creation of the backend server code using Node.js and Express. We started by looking at what an API is and looked at the various verbs associated with API usage. Then, we built two simple services for posts and comments. We used the `POST` and `GET` verbs in this process. We then tested these with the help of a tool called Postman. Finally, we took a look at ideas for connecting the backend code with the frontend. We introduced the NATS Streaming server and took a brief look at it.

In the next chapter, we will get introduced to MongoDB and update our backend code to integrate with the MongoDB database. We will first take a look at how we can utilize the NATS streaming server in order to talk to the database.

Questions

1. Explain what an API is.
2. Explain the different verbs used in APIs.
3. Write a small API to get a response from Google.com.
4. What is a NATS streaming server?
5. What is Asynchronous communication?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Chapter 6
Handling Authorization

We saw how to design a bare-bones application using Node.js and Express. Without making use of the MongoDB database, we constructed two services: posts and comments. We constructed arrays in memory to store the posts and comments. Using in-memory data storage is good for testing purposes but when it comes to serious full-stack development, one has to utilize the power of a database.

With this chapter, we start the development of the first piece of our framework which is the Authorization piece. We will make use of TypeScript throughout this book for the development of the backend part of our code. This chapter will incorporate certain best practices for the development of code. We will use the Google Cloud environment for developing the application, since many times due to the creation of many Kubernetes pods, older systems may crash. One can only create a single node cluster on a local desktop or laptop. We can create a multi-node cluster when we develop on Google Cloud. We will create a docker container for our Auth service. We will make use of Google cloud shell as our **Integrated Development Environment** (**IDE**).

Structure

This chapter will predominantly focus on the following topics:

- Introducing the authentication service

More books: https://t.me/booksforits

- Introduction to Google Cloud
- Enabling the Google Cloud Build
- Creating an auth docker build
- Creating a **`.dockerignore`** file
- Creating an authentication deployment
- Introducing MongoDB and creating an auth Mongo deployment
- Building a user model
- Creating an Ingress service yaml
- Creating an Ingress load balancer
- Using Skaffold for build automation
- Introduction to middleware
- Introduction to cookies and JSON web tokens
- Password encryption
- Error handling using express-validator
- Creation of abstract class for custom error handling
- Creating subclasses for validation
- Separating the logic for routes
- Creation of Signup, Signin, and CurrentUser routes
- Testing the application using Postman

Objectives

After reading this chapter, the reader will be able to create the backend code for authorization. This code will incorporate industry best practices for the development of Express and Node.js applications. The reader will be able to sign-up as a user, sign-in as a user and get the details of the currently logged-in user and eventually sign out. This may seem like a small task but is extremely code-heavy with lots of concepts. The reader will get a complete understanding of the development of authentication code without the use of a frontend.

Let us start by getting a basic understanding of the authentication service.

More books: https://t.me/booksforits

Introducing the authentication service

Authentication is a critical part of any application development. Authentication is the entry gate of any application and should be made as robust as possible. From this chapter onwards, we will focus only on the backend part for some time. We will implement sign up, sign in, current user and logout functionalities. The current user and logout functionalities require that the user has already signed in.

For preliminary validation of the incoming request, we will make use of an express-validator. An express-validator is a set of express.js middlewares that wraps **`validator.js`**, **`validator`**, and **`sanitizer`** functions. The authentication service will be making use of utility classes like the password class or middleware for handling errors. We will see what middleware is later in this chapter.

We will create a model class for users who will be signing up for using the blogging service. This class will serve as a template for all users. We will see what details have been included in this model class.

We will develop our application on Google Cloud. So, let us see what details are involved in utilizing Google Cloud.

Introduction to Google Cloud

We will take a look at utilizing the cloud environment for our development. Google Cloud is a very good environment which provides the facility to create Kubernetes clusters. So, let us first understand what Kubernetes clusters are.

What is a Kubernetes cluster?

Kubernetes is a portable, extensible, and open-source platform used for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are available widely.

A Kubernetes cluster is a set of machines also called nodes for running containerized applications. The containerized applications are created using Docker and are also known as Dockerized applications.

A cluster, at a minimum, contains a control plane and one or more compute machines also called nodes. The responsibility of the control plane is to maintain the desired state of the cluster. This includes things such as which applications are running and which container images they use. Nodes are responsible for running the applications and workloads.

Before one can create a Kubernetes cluster, he should have a Google Cloud account. So, let's create one next.

More books: https://t.me/booksforits

Creating a Google Cloud account

In order to use the Google Cloud platform, one needs to create a Google Cloud free tier account. This account is valid for 3 months and need not pay anything. The user simply has to have an active credit card.

Perform the following steps as mentioned to create a Google account:

1. Go to **cloud.google.com**, and click on **Get started for free**.
2. On the page that appears, there is an option to use an existing Google account or create a new one.
3. Select the country and agree to the terms and conditions. Click on **Agree and Continue**.
4. Enter the **Credit/Debit** card number, **Expiry date**, and **Cardholder name**.
5. Click on **Start my free trial**.

The dashboard similar to the one shown in the following figure should be visible:

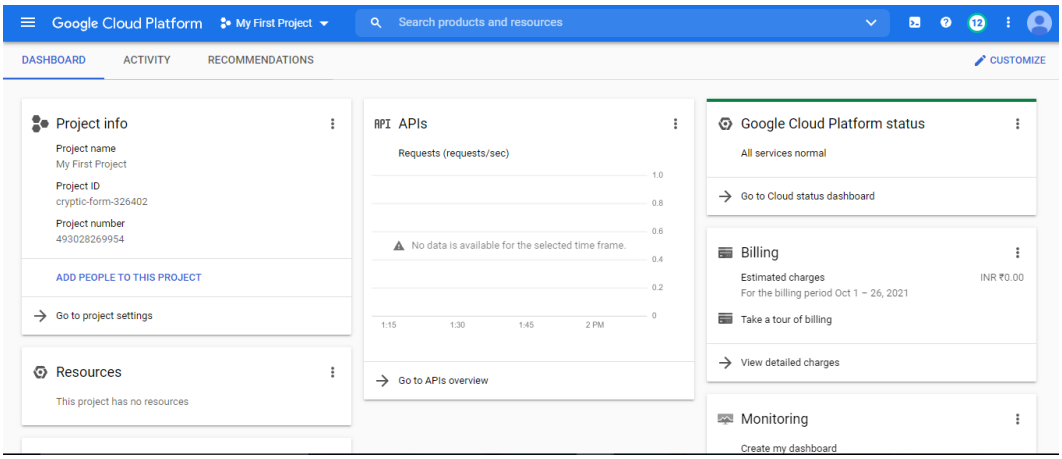

***Figure 6.1**: Google cloud dashboard*

More books: https://t.me/booksforits

Creating a new project

The first step in development with Google Cloud is the creation of a project. Click on the dropdown highlighted below:

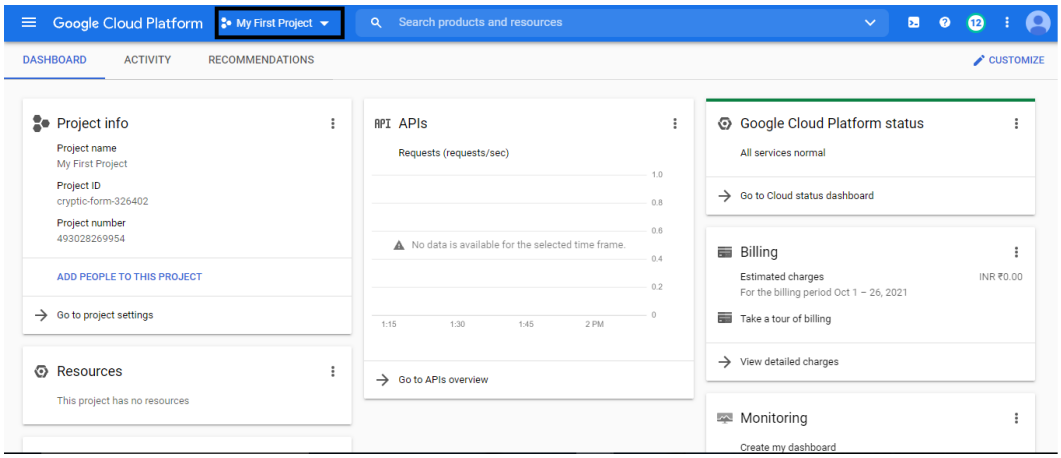

***Figure 6.2**: Creating a new project*

On the screen that appears, click on **New Project** as shown in *Figure 6.2*:

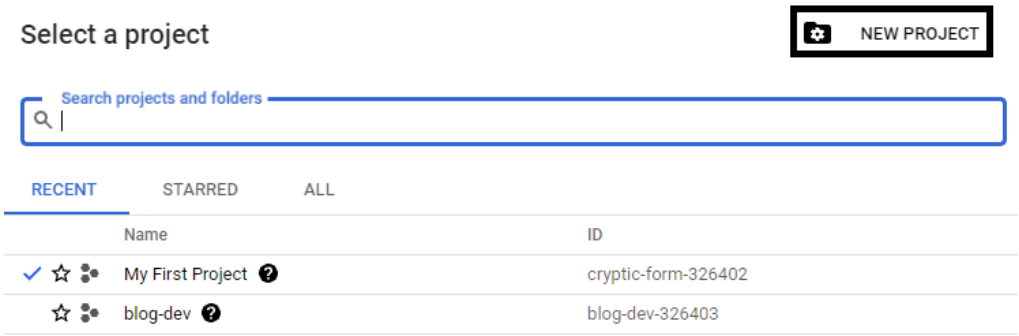

***Figure 6.3**: New project link*

More books: https://t.me/booksforits

Enter the project name and click on **Create**:

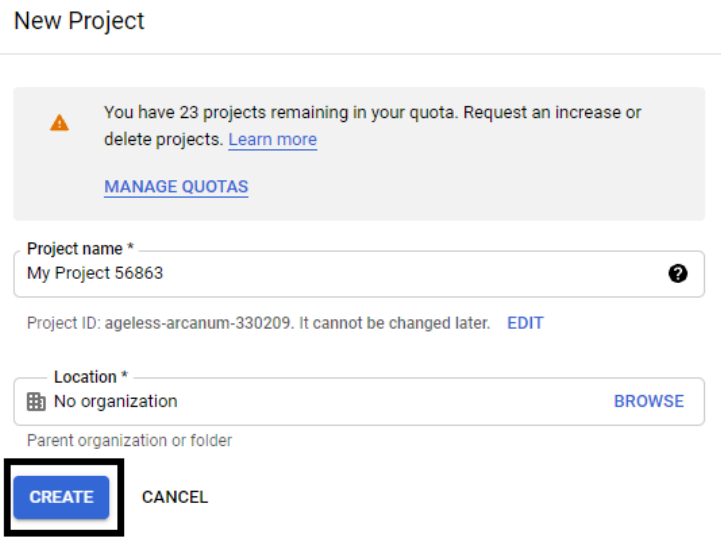

***Figure 6.4**: Creating the project*

With this, the project is ready. Select the project. The next step is to set up a Kubernetes cluster.

Setting up a Kubernetes cluster

In order to run the auth docker container, we require a Kubernetes cluster. On the Dashboard page shown in the following figure, click on the **hamburger** menu followed by **Kubernetes Engine** followed by **Clusters** as shown in *Figure 6.5:*

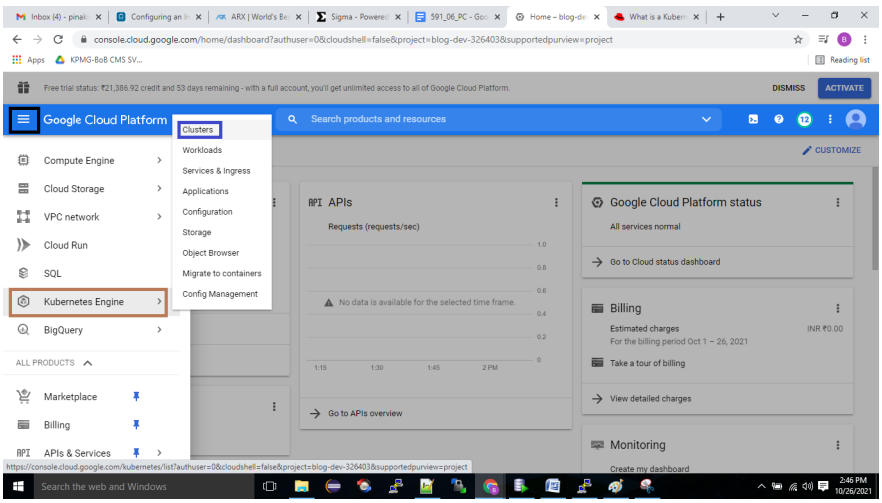

***Figure 6.5**: Setting up a new Kubernetes cluster*

More books: https://t.me/booksforits

On the page that appears click on **CREATE** as shown in *Figure 6.6*:

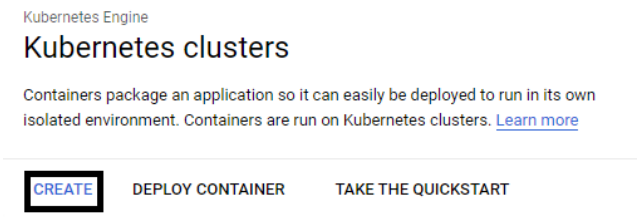

***Figure 6.6**: Create a button for the new cluster*

Select **CONFIGURE** for GKE Standard as shown in *Figure 6.7*:

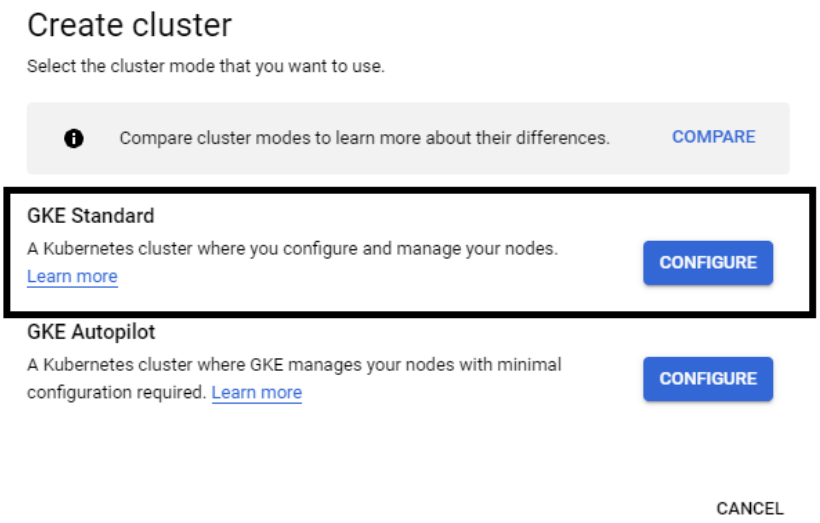

***Figure 6.7**: Selecting the GKE standard cluster*

Name the cluster as shown in *Figure 6.8*:

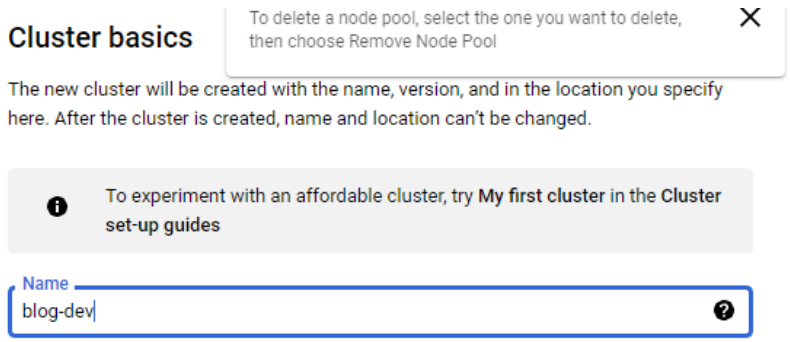

***Figure 6.8**: Naming the cluster*

More books: https://t.me/booksforits

Select the control plane version as the static version as shown here:

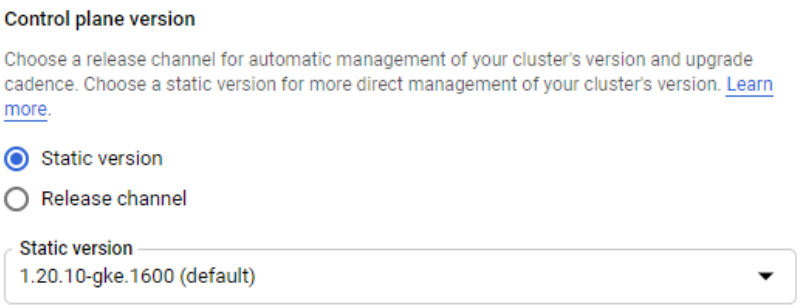

***Figure 6.9**: Selecting the static control plane version*

Click on the default-pool in the left pane and verify that the cluster consists of 3 nodes as shown in *Figure 6.10*:

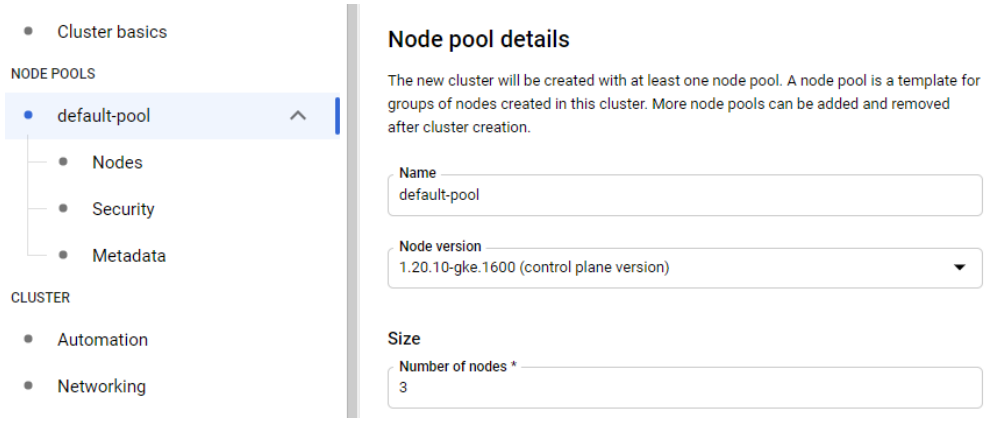

***Figure 6.10**: Looking at the default-pool*

Click on **Nodes** as shown above and select **GENERAL-PURPOSE**, **Series** as **N1** and **Machine type** as **g1-small** as shown in *Figure 6.11* and click on **CREATE**:

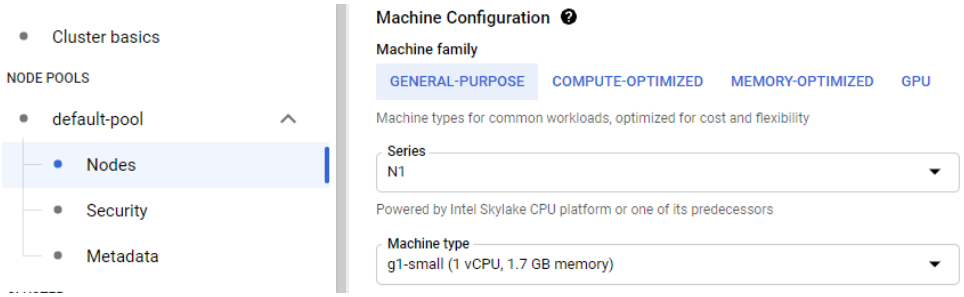

***Figure 6.11**: Selecting the N1 series g1-small nodes*

More books: https://t.me/booksforits

After some time, the 3-node cluster gets created:

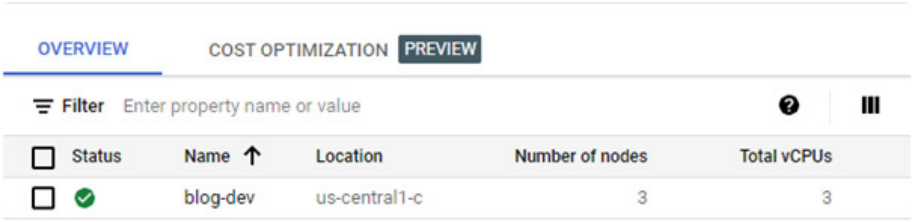

***Figure 6.12**: Successful creation of the cluster*

We will be doing a build of our code using Google Cloud Build. Click on Cloud Build and Enable the Cloud Build.

Now that we have created a cluster and enabled Cloud Build, let's build the Kubernetes artefacts. We start with the creation of a docker container for the auth service.

Creating an auth docker build

Instead of creating Docker containers from the command line, we will make use of a file called Dockerfile to create the build artefacts. On the Google Cloud dashboard, activate the cloud shell by clicking on the highlighted button shown in *Figure 6.13*:

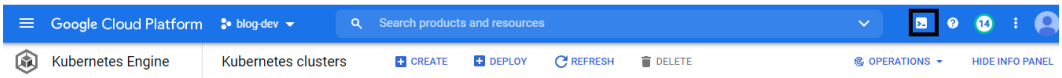

***Figure 6.13**: Accessing the cluster*

At the bottom of the screen, a terminal opens up as shown below. Click **Open Editor** as shown below:

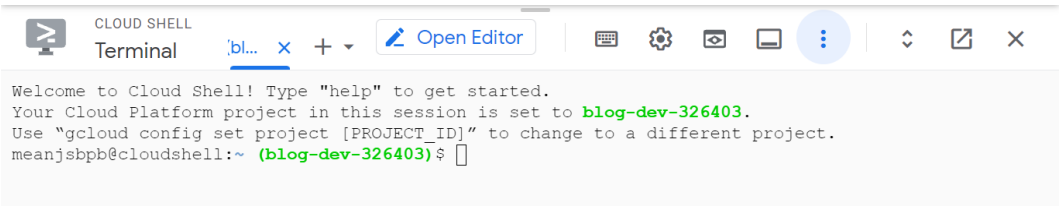

***Figure 6.14**: The terminal*

More books: https://t.me/booksforits

The cloud shell editor opens as shown in *Figure 6.15*:

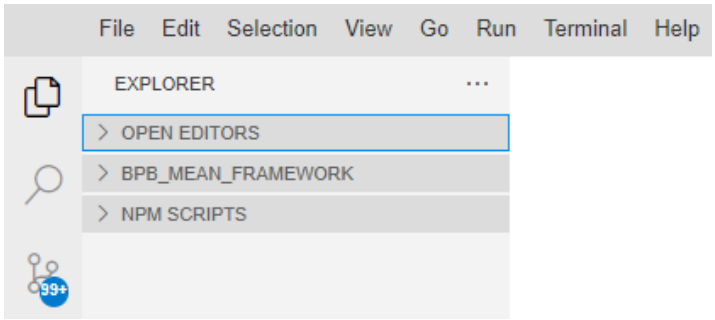

***Figure 6.15**: The cloud shell editor*

Right click in the left pane and create a new folder and name it **`BPB_MEAN_Framework`**. Under this folder, create another folder and name it **`auth`**. Right inside the **`auth`** folder, create a file and name it **`Dockerfile`** with no extension. This is the file that contains instructions to build an image of an individual micro-service.

Put the contents shown below in the Dockerfile:

```
1. FROM node:alpine
2. WORKDIR /app
3. COPY package.json ./
4. RUN npm install
5. COPY . .
6. CMD ["npm","start"]
```

What we are creating above is a Docker container. Docker and Kubernetes come pre-installed with Google Cloud shell and that's the reason why we use Google Cloud shell for our development.

Let us look at the preceding file line-by-line:

(Please note that the line numbers have been inserted for explanation purposes. At the time of execution, remove the line numbers and execute)

1. From the Docker hub, download the **`node: alpine`** image.
2. Create a working directory on the container called **`app`**.
3. In the **`app`** directory, copy the **`package.json`** from the **`root`** directory on Cloud Shell.
4. Execute the **`npm install`** to install the required dependencies.
5. Copy the rest of the files over to the container.

More books: https://t.me/booksforits

6. At the time of Container creation execute the **`npm start`** command.

Notice that we are copying over all the files from the directory on Cloud Shell to the container. This will copy the **`node_modules`** folder that has already been created on Cloud Shell to the container. This makes the container bulky. To avoid copying over the **`node_modules`** folder, we have something called a **`.dockerignore`** file that we look at next

Creating a .dockerignore file

Create a file called **`.dockerignore`** under auth and place the contents shown below:

```
node_modules
```

This will avoid copying over the **`node_modules`** folder to the container since this will make the Container bulky.

Our next step is to create a deployment for authentication.

Creating an authentication deployment

Under the **`root`** folder, create a new folder **`infra`** and inside this, create a folder **`k8s`**. Inside the **`k8s`** folder, create a **`yaml`** file called **`auth-depl.yaml`**. Place the contents as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
```

More books: https://t.me/booksforits

```
          image: us.gcr.io/blog-dev-326403/auth:project-latest
          env:
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_KEY
---
apiVersion: v1
kind: Service
metadata:
  name: auth-srv
spec:
  selector:
    app: auth
  ports:
    - name: auth
      protocol: TCP
      port: 3100
      targetPort: 3100
```

There are many new things. The two top-level parents are the deployment and service. These both can be in independent files. Here, we have placed the deployment and service in one file for ease of understanding. Let us go through the key elements of a deployment and a service

Key elements of a deployment

The key elements of a deployment yaml are given below:

- **`apiVersion`**: This indicates which version of the Kubernetes API is being used to create this object. In our case, we use **`apps/v1`**.
- **`kind`**: This indicates what kind of object is to be created. In this case, it is a deployment.
- **`metadata`**: This is the data that helps to identify the object uniquely. In this case, it is **`auth-depl`**.
- **`spec`**: This is the state that is desired for the object. This is the specification of the Pod to be created. The **`spec`** consists of the specification for the Pods:

More books: https://t.me/booksforits

- **`replicas`**: This indicates how many Pods will be created. In this case, we have 1.
- **`selector`**: This decides which Pods to consider. Here, we are specifying that we need to consider Pods having a label as **`app: auth`**. For this, **`matchlabels`** is specified with the key as **`app`** and value as **`auth.`**
- **`template`**: This describes the Pod template and has the following subfields:
    - **`metadata`** which has the label's app**`: auth`** which states that this Particular Pod has the label of the **`app: auth`**.
    - **`spec`** which specifies the details of the containers. This has the following subfields:
        - **`containers`**: This is a list and has details of the container and the subfields mentioned below:
            - **`name`**: This specifies the name of the container.
            - **`image`**: This specifies the image from which the container is to be built.
            - **`env`**: This is a list and specifies the environmental variables for the container:
                - **`name`**: This specifies the name of the environmental variable.
                - **`valueFrom`**: This specifies the location and consists of the field mentioned below:
                    - **`secretKeyRef`**: This has a name and a key.

Next, we will see the key elements of a deployment yaml.

Key elements of a service

Service can be defined as an abstract way to expose an application running on a set of Pods as a network service. With Kubernetes, one need not modify his application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods the facility to have their IP addresses and a single DNS name for a set of Pods, and also load-balance across them.

The key elements of a service yaml are given below:

- **`type`**: It indicates the type of service to be used. The default is **`ClusterIP`** and we are using that here. With **`ClusterIP`**, the pods are not exposed to the outside world. They are only accessible within the cluster.

More books: https://t.me/booksforits

- **apiVersion**: It indicates which version of the Kubernetes API is being used to create this object. In our case we use **v1**.
- **kind**: This indicates what kind of object needs to be created. In this case, it is a Service.
- **metadata**: This is data that helps to identify the object uniquely. In this case, it is **auth-srv**.
- **Spec**: This is the state that is desired for the object. This is the specification of the Service to be created.
- **Selector**: This indicates which pods are to be selected. In this case, pods with the label of the **app: auth** are selected.
- **Ports**: This is a list of ports that have a name, the protocol to be used is TCP, which lists the port on which the pods can be accessed and the **targetport** is the port on the container itself.

Now that we have explored the auth deployment, we will move on to understand what MongoDB is and we will also create a container for the MongoDB deployment.

Introducing MongoDB and creating an auth Mongo deployment

Our first module signup will utilize a database for storing the user's email and password. For this, we will make use of the MongoDB database. MongoDB is a NoSQL database that stores data as Collections and Documents. Collections are similar to tables in a relational database in MongoDB and Documents are similar to a table row in a relational database.

We will create a separate DB instance for each of our services, and we will also have a separate deployment for it.

Create a new file in the **Infra/k8s** folder and name it **auth-mongo-depl**. Here is the deployment yaml for MongoDB for auth service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-mongo-depl
spec:
  replicas: 1
  selector:
```

More books: https://t.me/booksforits

```
    matchLabels:
      app: auth-mongo
  template:
    metadata:
      labels:
        app: auth-mongo
    spec:
      containers:
        - name: auth-mongo
          image: mongo
---
apiVersion: v1
kind: Service
metadata:
  name: auth-mongo-srv
spec:
  selector:
    app: auth-mongo
  ports:
    - name: db
      protocol: TCP
      port: 27017
      targetPort: 27017
```

All the elements in the yaml shown above are already known to you. Things to be noted here are the image name which is Mongo, this will be pulled from **`DockerHub`** and the port to be used; in this case, is **`27017`** which is the default port for MongoDB. The name mentioned in the port is just for logging purposes.

We have our Mongodb deployment ready. We will now move to the creation of what is called a **`Mongoose`** user model. A Mongoose user model simply means the entire collection of users.

Let's build our user model.

Building a user model

In this section, we will build the mongoose user model. We will build this model piece by piece and understand what each piece signifies:

More books: https://t.me/booksforits

1. Import **mongoose** and the **Password** class. We will see the **password** class later in this chapter:

```
import mongoose from 'mongoose';

import {Password} from '../services/password';
```

2. Define an interface for the **UserAttribs** attributes. The user should have a valid **email** address and should provide a **password**. Both the attributes are of type **string**:

```
interface UserAttribs{

    email:string;

    password:string;

}
```

3. Create another interface which extends the **mongoose** model. In this interface, we create a function called build for creating our user by providing **email** and **password** in the form of **UserAttribs** and the build returns us a user document (represented by the **UserDoc** interface which is mentioned next:

```
interface UserModel extends mongoose.Model<UserDoc>{

    build(attribs: UserAttribs):UserDoc;

}
```

4. Create an interface for the user document which extends **mongoose.Document** with **email** and **password** of type **string**:

```
interface UserDoc extends mongoose.Document {

    email: string;

    password: string;

}
```

5. Create a schema called **userSchema** which states that **email** and **password** are both of type **String** and are required fields. We have also changed the response structure using **toJSON** which has the **transform** function which accepts a document and a return object called **ret**. MongoDB returns **_id**, **__v** in the response. To make the response uniform, we delete the **_id** after assigning it to the **id**. We also delete the **password** and **__v** fields. We also have the Mongoose **pre save** function. This function will be executed when we try to save a document to the database. We will check to see whether the user's password in the user document has been modified. If it is then hash the password and assign it to the user's password again. Call the **done()** function to indicate that our work related to the comparison is complete:

More books: https://t.me/booksforits

```
const userSchema = new mongoose.Schema({
 email:{
     type:String,
     required: true
    },
 password:{
     type:String,
     required: true
    }

}, {
    toJSON: {
        transform(doc, ret){
            ret.id = ret._id;
            delete ret._id;
            delete ret.password;
            delete ret.__v;

        }
    }
});


userSchema.pre('save',async function(done){
    if (this.isModified('password')){
       const hashed = await Password.toHash(this.get('password'));
        this.set('password',hashed);
    }
    done();
});
```

6. Use a static function bound to the schema which we will use for creating a new user with the help of user attributes:

```
userSchema.statics.build = (attribs:UserAttribs)=>{
```

More books: https://t.me/booksforits

```
    return new User(attribs);
}
```

7. Create a **mongoose.model** with the use of **userSchema** and assign it to a variable **User**:

```
const User = mongoose.model<UserDoc, UserModel>('User',userSchema);
```

8. Export the **User**:

```
export { User};
```

With the user model in place, we can now proceed to create the Ingress Service. Remember, our service does not have a type and it defaults to a **ClusterIP** service which can be accessed only inside the cluster and the outside world does not have access to it.

Let's learn how to create an Ingress service. But before that, we will create the **index.ts** file.

Creating index.ts

We will create **index.ts** in this section. As a pre-requisite, type the following in the **auth** directory:

```
> npm install express-async-errors
```

The **index.ts** is the file which is the central part of our framework:

1. Import **express**, **express-async-errors**, **{json}**, **mongoose** and **cookieSession**:

```
import express from 'express';
import 'express-async-errors';
import {json} from 'body-parser';
import mongoose from 'mongoose';
import cookieSession from 'cookie-session';
```

2. Import the different routes for **current-user**, **signin**, **signout**, and **signup**. We also import the error handler middleware and the **Not Found error**:

```
import {currentUserRouter} from './routes/current-user';
import {signinRouter} from './routes/signin';
import {signoutRouter} from './routes/signout';
import {signupRouter} from './routes/signup';
import {errorHandler} from './middlewares/error-handler';
```

More books: https://t.me/booksforits

```
import {NotFoundErr} from './errors/not-found-err';
```

3. Create an **Express** application and inform Express to trust the connection if behind a proxy. Select the port as **3100** or take the value from **process.env.PORT** environment variable:

```
const app=express();
app.set('trust proxy', true);
const port = process.env.PORT || 3100;
```

4. The **json()** function is a built-in middleware function. It parses incoming requests with JSON payloads. The **app.use()** function is used to mount the middleware function(s) that is mentioned at the path which is being specified. **cookieSession** is a simple cookie-based session middleware:

```
app.use(json());
app.use(cookieSession({
    signed: false,
    secure: true
})
);
```

5. Mount the routes as follows:

```
app.use(currentUserRouter);
app.use(signinRouter);
app.use(signoutRouter);
app.use(signupRouter);
```

6. If the user navigates to a path that is not specified in any of the routes, then throw a custom **NotFoundErr**. We will see the error creation a bit later:

```
app.all('*',async(req,res)=>{
    throw new NotFoundErr();
});
```

7. Mount the error handler middleware:

```
app.use(errorHandler);
```

8. Create an **async** function that checks for the presence of a JWT key environment variable, connect to the **MongoDB** instance, and start listening to incoming requests:

More books: https://t.me/booksforits

```
const startup = async()=>{
    if(!process.env.JWT_KEY){
        throw new Error('Jwt key must be defined');
    }
    try{
    await mongoose.connect('mongodb://auth-mongo-srv:27017/auth');
    console.log('Connected to Mongo DB');
    } catch(err){
        console.error(err);
    }
    app.listen(port, ()=>{
    console.log('Listening on port 3100',port);
    });

}
```

9. Invoke the **`startup`** function:

```
startup();
```

Our next task is to create an Ingress service to access our pods in the cluster.

Creating an Ingress service yaml

Create a new file inside the **`infra/k8s`** folder and name it **`ingress-srv.yaml`**. Let's understand what goes into an Ingress Service yaml:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: 'true'
spec:
  rules:
```

More books: https://t.me/booksforits

```
- host: blog.dev
  http:
    paths:
      - path: /api/users/?(.*)
        backend:
          serviceName: auth-srv
          servicePort: 3100
```

We will see what next things are present in this yaml:

- **Rules**: These are rules for our Ingress service. It has a host which is **`blog.dev`** and under HTTP, it has a list of paths. If the user navigates to any path **`/api/users/*`**, then the service **`auth-srv`** on port **`3100`** is going to handle this request.

Now that we have the yaml setup, we will create the Ingress Load Balancer for 3 nodes through the command line.

Creating an Ingress load balancer

In the Google Cloud terminal, navigate to the project and follow the given steps:

1. Initialize **`gcloud`** using the command **`gcloud init`**.
2. In the first step, select 1 for reinitializing or initializing the configuration.
3. Next, select the **`meanjsbpb@gmail.com`** account.
4. Next, select the **Cloud project** to use.
5. Next, select the default **Compute region and zone**.
6. Run the command **`gcloud container clusters get-credentials blog-dev`** to fetch the cluster credentials.
7. Execute the command **`kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.3/deploy/static/provider/cloud/deploy.yaml`** to configure Ingress-NGINX

More books: https://t.me/booksforits

8. Go to the console and navigate to Networking-Network Services-Load Balancing:

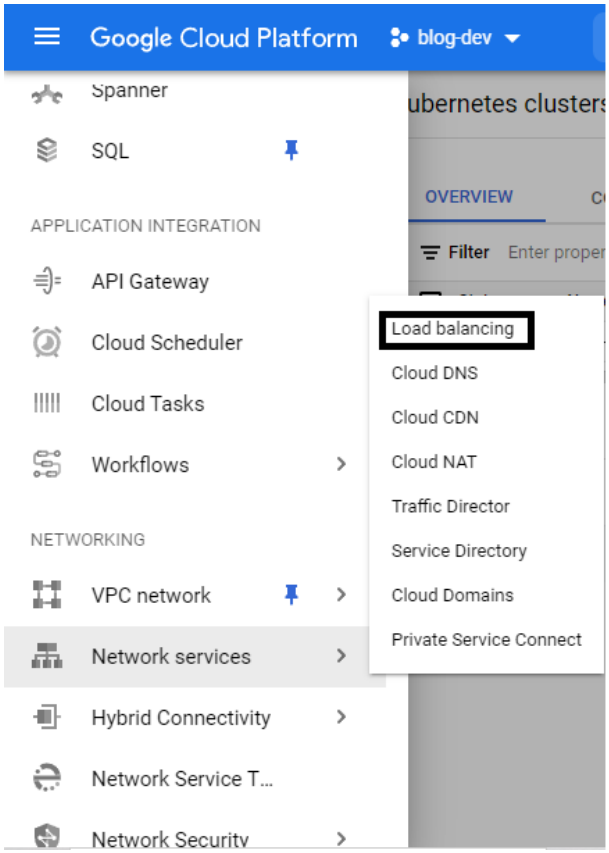

***Figure 6.16**: Load Balancer menu*

9. See the target pool with three instances. Click on it:

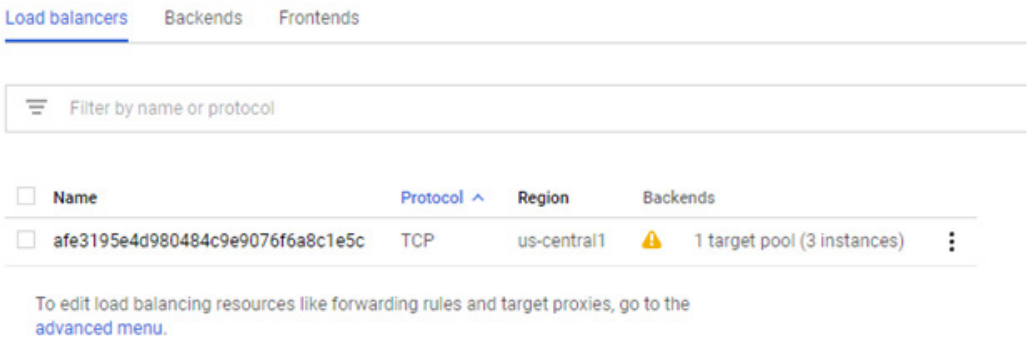

***Figure 6.17**: Ingress Load Balancer*

More books: https://t.me/booksforits

10. Once the name is clicked, the list of three nodes gets displayed as shown in the following screenshot:

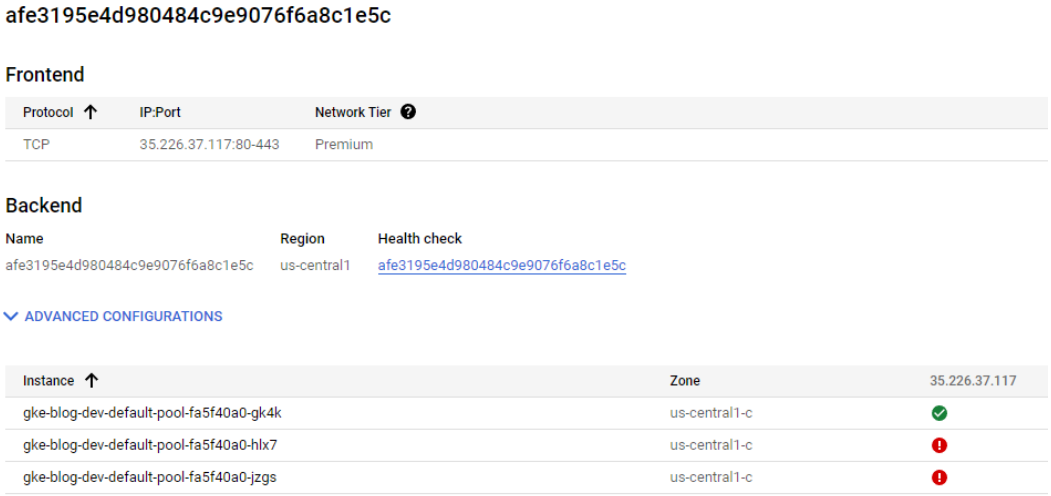

***Figure 6.18**: Nodes available and their health*

Note down the IP under Frontend and open the **`hosts.ini`** file at **`c:\windows\system32\drivers\etc\hosts.ini`**. Administrative mode is required for this:

```
35.226.37.117blog.dev
```

With this, the Ingress LoadBalancer setup is complete. We now move to a different topic where we will see how to create a Kubernetes secret. We saw that we included the **`JWT_KEY`** above for the JSON web token. We will be storing the Key in a Kubernetes secret.

Creating a Kubernetes Secret

A Secret is an object that contains sensitive data like a password, a token, or a key. Such information can be put in a Pod specification or directly in a container image. Using a Secret means that we need not include confidential data in our application code which can be accessed by hackers.

Secrets can be created independently of the Pods that use them. So, there is less risk of the Secret (along with its data) being exposed during the creation, viewing, and editing of Pods. Kubernetes, and applications in the cluster, can take additional precautions with Secrets, such as avoiding writing confidential data to persistent storage.

More books: https://t.me/booksforits

The way to create a secret is by specifying the command shown below under the project folder:

```
kubectl create secret generic jwt-secret --from-literal=JWT_KEY=asdf
```

We now have our infrastructure setup. As a final part of the Infrastructure setup, let's look at a tool called Skaffold which is used for Build Automation.

Using Skaffold for build automation

In order to use skaffold, install it from **https://skaffold.dev/docs/install/**. Google Cloud Shell has Skaffold pre-installed. Under the **root** directory, create a file called **Skaffold.yaml**:

```
apiVersion: skaffold/v2alpha3
kind: Config
deploy:
  kubectl:
    manifests:
      - ./infra/k8s/*
build:
  #local:
   # push: false
  googleCloudBuild:
    projectId: blog-dev-326403
  artifacts:
    - image: us.gcr.io/blog-dev-326403/auth
      context: auth
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: 'src/**/.ts'
            dest: .
```

More books: https://t.me/booksforits

The **skaffol.yaml** file has the following sections:

- **apiVersion**: This indicates the version of the configuration.
- **Kind**: This should always be config.
- **Deploy**: This describes how images are deployed.
    - **kubectl**: This uses a client-side **kubectl apply** to deploy manifests.
        - **manifests**: This lists the yaml files to be used to create the artefacts.
- **Build**: This describes how images are built.
    - **googleCloudBuild**: This describes how to do a remote build using Google Cloud Build.
        - **projectId**: This mentions the Google Cloud project ID.
    - **artifacts**: The images we are going to build. This is a list.
        - **image**: Name of the image to be built.
        - **context**: Directory containing the sources of the artefacts.
        - **docker**: Describes an artifact built from a Dockerfile.
        - **sync**: This indicates that if a change is made in a synced file, then the update is done in following two ways
            - Done directly in the container and if a change is made in a non-synced file,
            - A full build gets triggered

Now, we have Skaffold configured, we will move to understand what middleware is.

Introduction to middleware

Middleware functions are functions having access to the request object (**req**), the response object (**res**), and the next middleware function in the request-response cycle. These functions are used to modify **req** and **res** objects for tasks like adding response headers, parsing the request bodies, and so on.

There are two types of middleware which are route-specific middleware and global middleware. The route-specific middleware is mentioned in the **get** method after the route and a global middleware is specified using **app. use(middleware_name)**. When we invoke a middleware function, the order is of importance. For invoking a middleware from another middleware, simply use the **next()** function.

More books: https://t.me/booksforits

There is a very specific middleware which is called the error handling middleware and it contains the error object before **req**, **res** objects or next parameters. Whenever there is an error thrown, the error handling middleware will be invoked.

Introduction to cookies and JSON web tokens

Cookies provide a way to store data in a user's browser. This data can be a user's name, email, and so on. Any time a user signs up for a website, a cookie is created and sent back to the browser where it is stored. Any time the user visits the website, this cookie is sent along with the request and this way, the server knows that the same person is visiting the website. A cookie can have a name, value, and zero or more attributes which are nothing but name/value pairs.

JSON web token, on the other hand, is an open standard that securely transmits information between the client and the server as a JSON object. In our case, the JSON web token will be contained in a cookie. JWTs are signed and thus the sender can be verified. The signature of the JSON web token can be calculated using the header and payload of JWT and we can be sure that it has not been tampered with.

This is just a gentle introduction to cookies and JWT. We will see the structure and how to decode those later.

One very important concept is encrypting the password which we will now explore.

Password encryption

We imported the **password** class while creating the user model. Now is the time to see the actual implementation of the **Password** class:

1. Import the **scrypt** and **randomBytes** from the crypto built in the node library and **promisify** from the util library. Scrypt is the hashing function that we will utilize. The only problem with **scrypt** is that it is based on callback. Since the two functions we are implementing are using the **async** await functionality, we will use **promisify** to convert a callback into a promise-based function:

```
import {scrypt, randomBytes} from 'crypto';

import {promisify} from 'util';
```

2. Use the **promisify** function to convert the callback-based implementation into a promise-based implementation:

```
const scryptAsync = promisify(scrypt);
```

More books: https://t.me/booksforits

3. Use the static **toHash** function to convert the password that is passed **in.salt** is used to add random bits to a password before it's hashing. When we use **scrypt**, we get a buffer which is like an array with some raw data inside. TypeScript will not know what **buf** is and hence, we explicitly tell TypeScript that **buf** is of type **Buffer**. Eventually, we return a template **string** after converting **buf** to a string of hexadecimal characters and the salt appended to it with the help of a '**.**'(dot):

```
static async toHash(password: string){
        const salt = randomBytes(8).toString('hex');
       const buf = (await scryptAsync(password,salt,64)) as Buffer;
        return `${buf.toString('hex')}.${salt}`;
}
```

4. Create another function called **comparePass** which accepts a stored password and a supplied password and compares it. Since the **storedPassword** is a combination of the hashed password and the salt concatenated by a '**.**', we use the **split** function to separate out the two. Perform the password hashing process on the supplied password. Eventually, return a Boolean value of true if the hexadecimal representation of the buffer obtained from the password hashing process is equal to the **hashedPassword** part of the **storedPassword**:

```
static async comparePass(storedPassword: string, suppliedPassword:
string){
        const [hashedPassword,salt]=storedPassword.split('.');
         const buf = (await scryptAsync(suppliedPassword,salt,64))
as Buffer;


        return buf.toString('hex')===hashedPassword;

}
```

Since we now have password hashing in place, we will move on to error handling.

Error handling using express-validator

Rather than implementing error handling using user-built validations, we will make use of a library called **express-validator**. Navigate to the auth directory and execute the following command:

```
> npm install express-validator
```

More books: https://t.me/booksforits

We will now create a series of **middleware** classes. We start with the **error-handler.ts**.

Understanding an error-handler

Create a folder called **middleware** and create a file called **error-handler.ts**. Perform the following steps:

1. Import the **Request** and **Response** interfaces and **NextFunction** from express. Import **CustomErr** from **custom-err** which we will create in the **errors** directory in some time:

```
import {Request, Response, NextFunction} from 'express';

import {CustomErr} from '../errors/custom-err';
```

2. Export the **errorHandler** middleware which accepts arguments of type **Error**, **Request**, **Response**, and **NextFunction**. This middleware checks for the **err** object and if it is the instance of **CustomErr**, then we return the status in the response by picking the **statusCode** from the **response** object and sending back an object containing errors which are extracted from the **serializeErr()** function of the err object. If the **err** object is not an instance of **CustomErr**, then we send back a **400 error** and an object with the message '**Something went wrong**':

```
export const errorHandler = (
    err:Error,
    req:Request,
    res:Response,
    next:NextFunction
    ) => {

        if(err instanceof CustomErr){

              return res.status(err.statusCode).send({errors:err.
serializeErr()});
        }


        res.status(400).send({
            errors: [{message: 'Something went wrong'}]
        });
}
```

More books: https://t.me/booksforits

We will now understand the current-user functionality.

Understanding the current-user

Create another file in the **middleware** directory and name it **current-user.ts**. Perform the following steps:

1. Import **Request**, **Response**, and **NextFunction** from express and **jwt** from **jsonwebtoken**:

```
import {Request,Response,NextFunction} from 'express';
import jwt from 'jsonwebtoken';
```

2. Information contained in a payload consists of an **id** and **email**:

```
interface UserPayload {
    id: string;
    email: string;
}
```

3. We add in the information about what a **request** object is. We specify that the **currentUser** is of type **UserPayload**:

```
declare global {
    namespace Express {
        interface Request {
            currentUser?: UserPayload;
        }
    }
}
```

4. Export a function called **currentUser** which accepts 3 arguments of type **Request**, **Response**, and **NextFunction**. A check is done to see if there is a **session** object and if there exists a **jwt** property on that object. If there isn't, then move on to the next middleware in the chain. If we get past this point, the extract the payload from the Json Web Token and assign it to **currentUser**:

```
export const currentUser=(req: Request,
    res: Response,
    next: NextFunction
```

More books: https://t.me/booksforits

```
 ) => {
    if(!req.session?.jwt){
        return next();
    }

    try{
            const payload=jwt.verify(req.session.jwt,process.env.
JWT_KEY!) as UserPayload;
        req.currentUser = payload;
    } catch(err){

    }
    next();
};
```

Let us now move on to the middleware for validating a request.

Understanding validate-request.ts

We will now see how to validate our incoming request. We create another middleware for this purpose in the **middleware** directory and name it **validate-request.ts**.

Perform the following steps to construct this middleware:

1. Import **Request**, **Response**, and **NextFunction** from **express**, **validationResult** from **express-validator**, and **RequestValidationErr** from **request-validation-err** from the **errors** directory:

   ```
   import {Request,Response,NextFunction} from 'express';
   import {validationResult} from 'express-validator';
   import {RequestValidationErr} from '../errors/request-validation-
   err';
   ```

2. Export a function called **validateRequest** which takes in the **Request** and **Response** objects and **NextFunction**. Verify the incoming request for errors. If the errors array is empty, then throw a new **Request Validation Error** with the errors array as an argument. Finally, call the next middleware:

   ```
   export const validateRequest = (req: Request, res: Response,next:
   NextFunction) => {
   ```

More books: https://t.me/booksforits

```
    const errors = validationResult(req);


    if (!errors.isEmpty()){
        throw new RequestValidationErr(errors.array());
    }
    next();
};
```

Let's now move on to the middleware for **`require-auth`**. This is required to block unauthorized access to certain routes.

Understanding require-auth.ts

We will now see how to require authorization for certain routes. We create another middleware for this. Create a file called **`require-auth.ts`** in **`middleware`** directory. Perform the following steps to create this middleware:

1. Import the **`Request`**, **`Response`**, and **`NextFunction`** from **`express`**:

   ```
   import {Request,Response,NextFunction} from 'express';
   import {NotAuthErr} from '../errors/not-auth-err';
   ```

2. Export a function called **`requireAuth`** which accepts the **`Request`**, **`Response`**, and **`Next`** function. This function verifies where the **`currentUser`** exists on the request and if not, it throws a **`NotAuthErr`**. Finally, call the next **`Middleware`**:

   ```
   export const requireAuth = (req:
   Request,res:Response,next:NextFunction)=> {
       if (!req.currentUser){
           throw new NotAuthErr();
       }
       next();
   }
   ```

With this, we have seen all the middlewares. It's time to take a look at the implementations of the **`Errors`** that we throw.

Let's start with the parent class for all errors. It will be an abstract class.

More books: https://t.me/booksforits

Creating an abstract class for custom error handling

Create a new file in the **errors** directory and name it **custom-err.ts**. Perform the following steps to construct this class:

1. Export the abstract class **CustomErr**. This class extends the built-in class **Error**. Define an abstract property called **statusCode** of type number. Define a constructor with a parameter message of type **string**. We call the superclass constructor using this message. The next piece of code is to be used whenever we extend any built-in class. Finally, we have the **serializeErr()** function that returns an object containing a message and the optional erroneous field:

```
export abstract class CustomErr extends Error{
    abstract statusCode: number;

    constructor(message:string){
        super(message);
        Object.setPrototypeOf(this,CustomErr.prototype);

    }

    abstract serializeErr(): {message: string,field?: string}[];
}
```

We will now see how we can create subclasses from **custom-err** for error handling.

Creating subclasses for validation

Let's create the different errors that will be required by our application. We will start with **request-validation-err.ts**.

Understanding request-validation-err.ts

We trigger the **request-validation-err error** when the incoming request has errors in it. Create a file called **request-validation-err.ts** in the **errors** directory and follow the given steps to construct the **request-validation-err**:

More books: https://t.me/booksforits

1. Import the **validationError** from **express-validator** and **CustomErr** from **custom-err**:

```
import {ValidationError} from 'express-validator';
import {CustomErr} from './custom-err';
```

2. Export a class called **RequestValidationErr** which extends **CustomErr**. In this class, define a property **statusCode** with value **400**. In the constructor, invoke the parent constructor with appropriate parameters and then, we write the mandatory statement which has to be present whenever we extend a built-in class. We then have the **serializeErr** function that return an object for errors with an error message and the field in error:

```
export class RequestValidationErr extends CustomErr{
    statusCode=400;
    constructor(public errors: ValidationError[] ){
        super('Invalid request parameters');
      Object.setPrototypeOf(this,RequestValidationErr.prototype);
    }
    serializeErr(){
        return this.errors.map(err=> {
            return {message: err.msg,field:err.param};
        })
    }
}
```

We will now take a look at the **database-connection** error.

Understanding database-connection-err.ts

Create a file called **database-connection-err.ts** in the **errors** directory and follow the given steps:

1. Import **CustomErr** from **custom-err**:

```
import {CustomErr} from './custom-err';
```

2. Export the class **DatabaseConnectionErr** which extends **CustomErr**. Define a property **statusCode** with a value of **500**. Put an appropriate reason:

```
export class DatabaseConnectionErr extends CustomErr{
    statusCode=500;
```

More books: https://t.me/booksforits

```
        reason = 'Error connecting to database';
        constructor(){
            super('Error connecting to database');
          Object.setPrototypeOf(this,DatabaseConnectionErr.prototype);
        }

    serializeErr(){
        return [
            {message: this.reason}
        ]
    }
    }
```

We will now see what a bad request error is.

Understanding bad-request-err.ts

Create a file called **bad-request-err** in the **errors** directory and follow the given steps:

1. Import the **CustomErr** class from **custom-err** in the **errors** directory:

```
import {CustomErr} from './custom-err';
```

2. Export the **BadRequestErr**. This class extends the **CustomErr** class. Apart from setting the status code, it has the serializeErr function that returns the object containing the error message:

```
export class BadRequestErr extends CustomErr{
    statusCode = 400;

    constructor(public message: string){
        super(message);
        Object.setPrototypeOf(this, BadRequestErr.prototype);
    }
    serializeErr(){
        return [{message: this.message}];
    }
}
```

More books: https://t.me/booksforits

Next, we will see the **Not Found** error.

Understanding not-found-err.ts

Create a file called **not-found-err** in the **errors** directory and follow the given steps:

1. Import the **CustomErr** class from **custom-err** in the **errors** directory:

```
import {CustomErr} from './custom-err';
```

2. Export the **NotFoundErr**. This class extends the **CustomErr** class. Apart from setting the status code, it has the **serializeErr** function that returns the object containing the error message:

```
export class NotFoundErr extends CustomErr{
    statusCode=400;
    constructor(){
        super('Route not found');
        Object.setPrototypeOf(this,NotFoundErr.prototype);
    }

    serializeErr(){
        return [{
            message: 'Not Found'
        }];
    }
}
```

Finally, we will see the **no-auth-err.ts** which sends an error message if we try to access a route that requires the user to be signed in. Let us take a look at the **no-auth-err.ts**.

Understanding no-auth-err.ts

Create a file called **no-auth-err** in the **errors** directory and follow the given steps:

1. Import the **CustomErr** class from **custom-err** in the **errors** directory:

```
import {CustomErr} from '../errors/custom-err';
```

More books: https://t.me/booksforits

2. Export the **NoAuthErr**. This class extends the **CustomErr** class. Apart from setting the status code, it has the **serializeErr** function that returns the object containing the error message:

```
export class NotAuthErr extends CustomErr{
    statusCode=401;
    constructor(){
        super('Not Authorized');
        Object.setPrototypeOf(this, NotAuthErr.prototype);
    }
    serializeErr(){
        return [{message: 'Not Authorized'}]
    }
}
```

We have now most of the code ready for the auth service. We will now take a look at how to set the routes for the auth service.

Separating the logic for routes

In order to keep our code modular, we will maintain a separate folder for routes and place the routes for **Signup**, **Signin**, and **CurrentUser** in the folder.

We simply include the routes in **index.ts** and this solves the purpose of routing using various paths.

Creating Signup, Signin and CurrentUser routes

Create a folder called routes and create four files for **Signup** (**signup.ts**), **Signin** (**signin.ts**), **CurrentUser** (**current-user.ts**), and **Signout** (**signout.ts**).

We start with the signup route.

Signup route

This is the route that the user utilizes for signing up with the application:

1. Import **express**, **validateRequest**, **Request**, **Response**, **body**, **jwt**, **User**, and **BadRequestErr** from the paths shown below:

More books: https://t.me/booksforits

```
import express from 'express';
import {validateRequest} from '../middlewares/validate-request';
import {Request,Response} from 'express';
import {body} from 'express-validator';
import jwt from 'jsonwebtoken';
import {User} from '../models/user'
import {BadRequestErr} from '../errors/bad-request-err';
```

2. Create an Express Router:

```
const router = express.Router();
```

3. Create a post request for **`/api/users/signup`** and pass in validations for the email and password. Password should be between 4 and 20 characters. If there are any requests found in the incoming request, then the **`validateRequest`** middleware will throw an error and exit out. If there are no errors found in the incoming request, the **`req.body`** is deconstructed, and the email and password are extracted out. We use the email to find out whether the email already exists in the database. We throw a **`BadRequestErr`** if the email has already been taken. If the email does not already exist, then invoke the **`build`** function in the User model to construct a user and save it to the database using the **`save()`** function:

```
router.post('/api/users/signup',[
    body('email')
    .isEmail()
    .withMessage('Email must be valid'),
    body('password')
    .trim()
    .isLength({min: 4, max: 20})
    .withMessage('Password must be between 4 and 20 characters')
],
    validateRequest,

    async(req: Request,res: Response)=>{

    const {email,password} = req.body;
```

More books: https://t.me/booksforits

```
const existingUser = await User.findOne({email});

if (existingUser){
   throw new BadRequestErr('Email is in use');
}

const user = User.build({
    email,password
})
await user.save();
```

4. Create the JSON web token payload using the ID and email along with the key stored in the environment variable **`JWT_KEY`**:

```
const userJwt = jwt.sign({
    id: user.id,
    email: user.email
},
    process.env.JWT_KEY!
);
```

5. Set the Json web token in the **`session`** property of the request object and send back the status code of 201 (**`Created`**) along with the created user. Finally, export the router:

```
    req.session = {
        jwt: userJwt
    };
    res.status(201).send(user);
});

export {router as signupRouter};
```

Let us see the code for Signing in.

More books: https://t.me/booksforits

Signin route

Once the user has signed up, a JWT token gets created. While signing in, pass the JWT token. Now, let's take a look at the steps for Signing in:

1. Import **`express`**, **`Request`**, and **`Response`** from **`express`**, **`body`** from **`express-validator`**, **`jwt`** from **`jsonwebtoken`**, User from the **`user model`** class, **`validateRequest`** from **`middleware`** directory, **`BadRequestErr`** from **`errors`** directory, and **`Password`** from the **`services`** directory:

```
import express from 'express';
import {Request,Response} from 'express';
import {body} from 'express-validator';
import jwt  from 'jsonwebtoken';
import {User} from '../models/user'
import {validateRequest} from '../middlewares/validate-request';
import {BadRequestErr} from '../errors/bad-request-err';
import {Password} from '../services/password';
```

2. Create the express router:

```
const router = express.Router();
```

3. Invoke the **`post`** method on the router and put the path as **`/api/users/signin`** followed by validations on the email and password which are passed on to the **`validateRequest`** middleware. Extract the email and password from the body of the request. Find the user with the help of an email. If he is not an existing user, throw a **`Bad request`** error. Compare the password with the one stored in the database. If the passwords don't match, throw a Bad request error else fetch the JWT for this user and store it in the session. Send back the user that has been found with a status of **`200`**. Finally, export the router as **`signinRouter`**:

```
router.post('/api/users/signin',[
    body('email')
    .isEmail()
    .withMessage('Email must be valid'),
    body('password')
    .trim()
    .notEmpty()
```

More books: https://t.me/booksforits

```
    .withMessage('You must supply password')
]
,validateRequest,
async (req: Request,res: Response)=>{
    const {email,password} = req.body;

    const existingUsr = await User.findOne({email});
    if (!existingUsr){
        throw new BadRequestErr('Invalid Credentials');
    }

    const passwordsMatching = await Password.comparePass(
        existingUsr.password,
        password
    );
    if (!passwordsMatching){
        throw new BadRequestErr('Invalid Credentials');
    }

     const userJwt = jwt.sign({
        id: existingUsr.id,
        email: existingUsr.email
    },
        process.env.JWT_KEY!
    );

    req.session = {
        jwt: userJwt
    };
    res.status(200).send(existingUsr);
   }
);

export {router as signinRouter};
```

More books: https://t.me/booksforits

CurrentUser route

Once the user has signed in, we will verify the Signed in user. Follow the given steps to implement the **`CurrentUser`** route:

1. Import **`express`** from **`express`** and **`currenUser`** from the **`middlewares`** directory:

```
import express from 'express';
import {currentUser} from '../middlewares/current-user';
```

2. Create an express router:

```
const router = express.Router();
```

3. Create a **`get`** method for the **`/api/users/currentuser`** path. Using the **`currentuser`** middleware, return the object with the current user or null. Finally, export the router as **`currentUserRouter`**:

```
router.get('/api/users/currentuser',currentUser,(req,res)=>{
    res.send({currentUser: req.currentUser} || null);
});

export {router as currentUserRouter};
```

We now will move to the final piece of code for authentication and that is the signout route

Signout route

This is a very simple route. Let's take a look and understand the **`signout`** logic:

1. Import **`express`** from **`express`**:

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

2. Create an express router:

```
const router = express.Router();
```

3. Create a post method for **`/api/users/signout`** and make the session null and send back an empty object. Export the router as **`signoutRouter`**:

```
router.post('/api/users/signout',(req,res)=>{
    req.session=null;
    res.send({});
```

More books: https://t.me/booksforits

```
});

export {router as signoutRouter};
```

With this, we complete the code for authentication. Now, let's do a quick test using Postman.

Testing the application using Postman

Execute the following command:

```
> skaffold dev
```

The console output is shown in *Figure 6.19*:

```
Waiting for deployments to stabilize...
 - deployment/auth-depl is ready. [1/2 deployment(s) still pending]
 - deployment/auth-mongo-depl is ready.
Deployments stabilized in 2.523 seconds
Press Ctrl+C to exit
Watching for changes...
[auth]
[auth] > auth@1.0.0 start
[auth] > ts-node-dev src/index.ts
[auth]
[auth] [INFO] 03:10:46 ts-node-dev ver. 1.1.8 (using ts-node ver. 9.1.1, typescript ver. 4.4.4)
[auth] Connected to Mongo DB
[auth] Listening on port 3100 3100
```

***Figure 6.19**: Starting the application with Skaffold*

For signing up, open Postman, navigate to **https://blog.dev/api/users/signup** and put **`content-type`** as **`application/json`**:

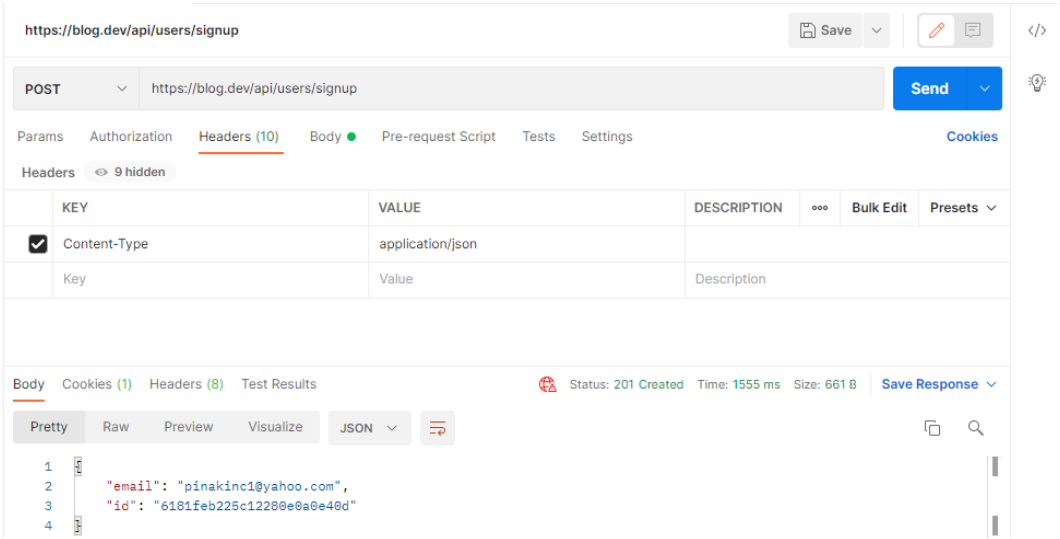

***Figure 6.20**: Testing the signup functionality*

More books: https://t.me/booksforits

After signing up, get the cookie as shown in *Figure 6.21*:

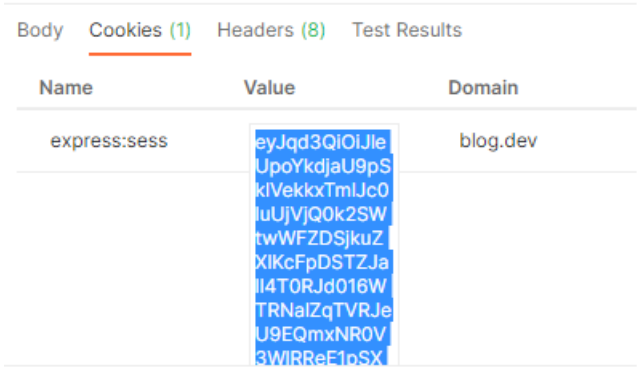

***Figure 6.21**: Verifying the cookie generation*

The next step is signing in. For that, navigate to **https://blog.dev/api/users/signin**:

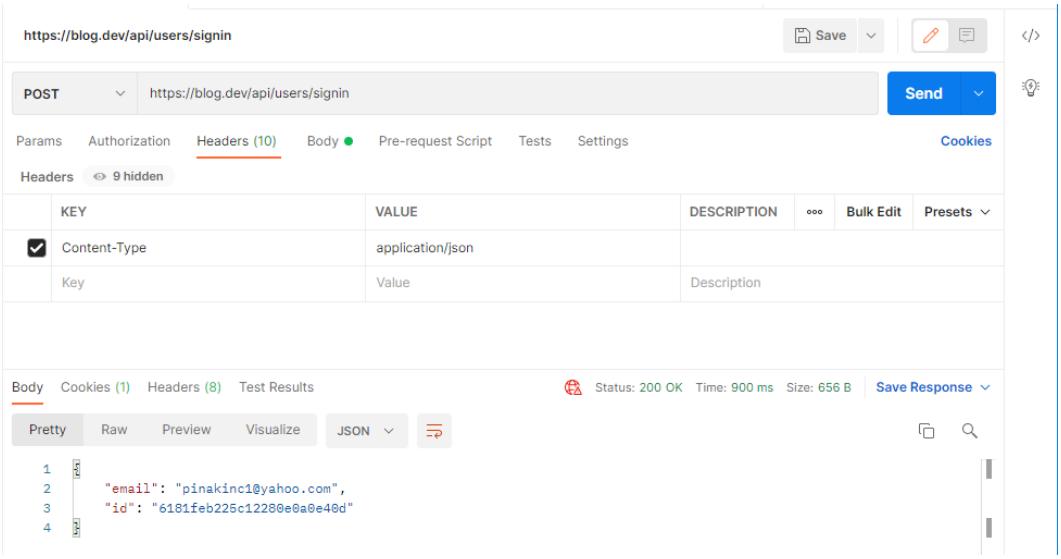

***Figure 6.22**: Sign-in*

More books: https://t.me/booksforits

The next step is to find the current user:

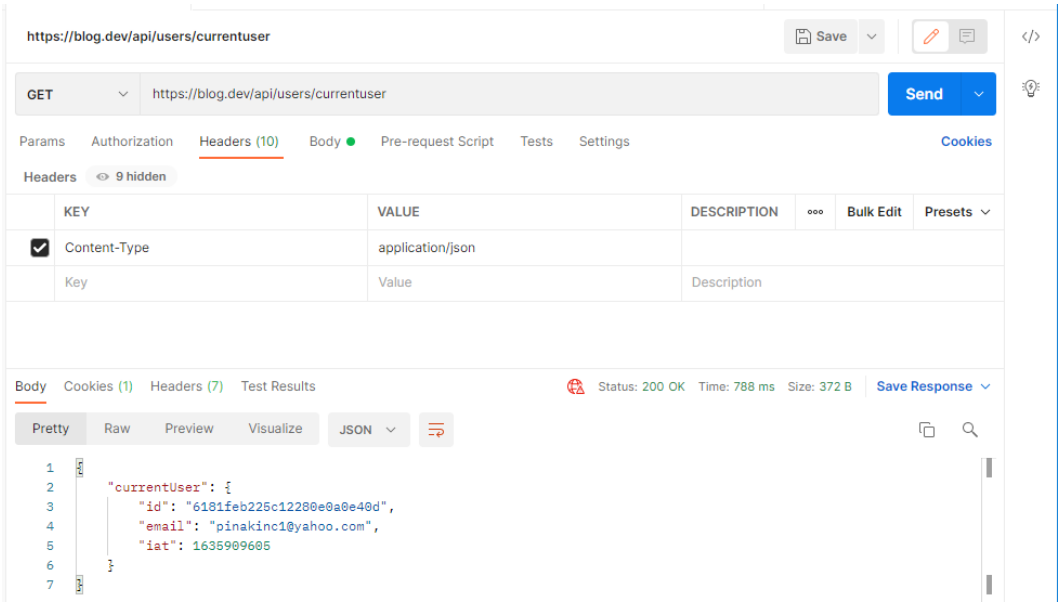

***Figure 6.23**: Fetching the current user*

The last step is signing out, as shown here:

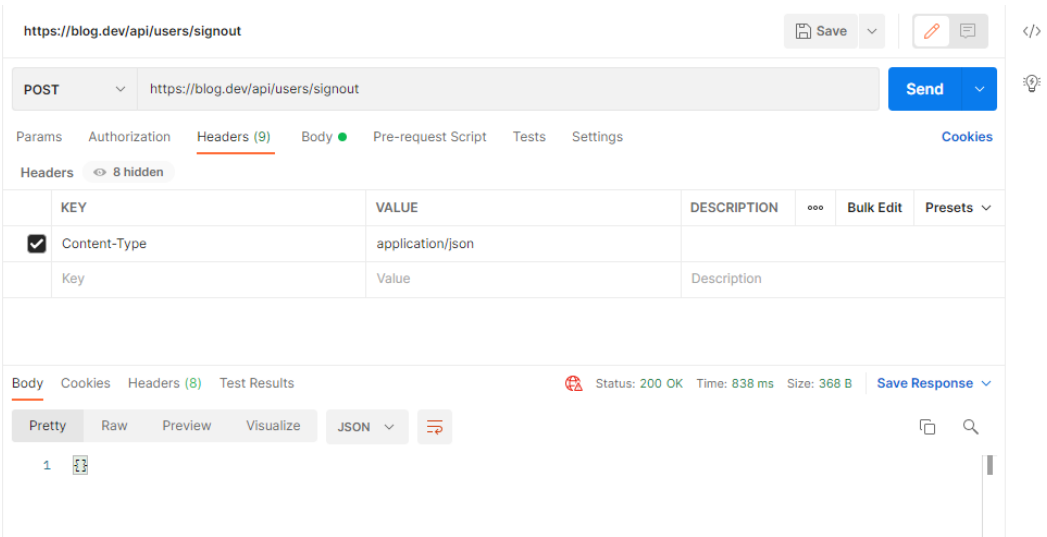

***Figure 6.24**: Signout functionality*

Now, if we try to find the current user, we get back an empty object. We have seen the complete creation of the auth service. With this, we conclude the chapter.

More books: https://t.me/booksforits

Conclusion

In this chapter, we looked at the creation of the auth service. There were a lot of activities involved in the creation of the auth service. We made use of the Google Cloud environment for creating the clusters and building our code. Google cloud shell was used as the IDE. We looked at the end-to-end workflow starting from Sign up, Sign in, Current user and Sign out. We took a tour of the error-handling process and created an error-handling middleware and a folder for errors. We also created the ingress load balancer with three nodes and successfully tested the application using Postman.

In the next chapter, we will start with the backend code for the Posts and Comments service and also take a look at how we can utilize the NATS streaming server.

Questions

1. Explain what middleware is.
2. Explain how error handling has been done in this chapter.
3. Explain what a cookie is.
4. Explain the use of JSON web tokens.
5. What is the use of async await syntax?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Chapter 7
Creating the Posts Service and NATS Streaming Integration

In the previous chapter, we saw how to construct an authentication service and constructed the Sign-up, Sign-in, and Sign-out flows and tested these flows with Postman.

In this chapter, we start with the introduction to the common module. As a part of the **common** module, we will create a folder called **common** and move the Authentication related logic to this folder. We will create an NPM module from the **common** folder. This common module can be included all services of our project. We then move to the development of the Posts service. This is followed by an introduction to the NATS streaming server, and we will create a simple Publisher and Listener which have a simple NATS implementation. We will see how to publish messages and listen to messages. The NATS streaming code will be moved to the common module. We create a Posts service and integrate it with the NATS streaming server. We then test our integration with Postman.

Structure

This chapter will predominantly focus on the following topics:

- Introducing the **common** module
- Creating a GIT repository for the **common** module
- Publishing the **common** module to NPM

More books: https://t.me/booksforits

- Installing required packages in the **`common`** module
- Changes to **`package.json`** and **`tsconfig`**
- Moving the authentication logic in the **`common`** module
- Installing the **`common`** module in the **`auth`** folder
- Standard process for new services
- The Posts service
- Creating the Posts deployment YAML
- Creating the Posts Mongo DB deployment YAML
- Making changes to the Skaffold YAML
- Looking at the Auth deployment YAML
- Looking at the Auth **`index.ts`**
- Creating the Posts service
- Creation of the **`nats-wrapper`** class
- Creation of the Posts model
- Introduction to the NATS streaming server
- Creating the NATS deployment file
- Creating a basic publisher and listener ts files
- Understanding the **`BaseListener`** and **`PostCreatedListener`**
- Understanding the **`BasePublisher`**, **`PostCreatedPublisher`**, and **`PostUpdatedPublisher`**
- Understanding the **`PostCreatedEvent`** and the **`PostUpdatedEvent`**
- Understanding the **`Subjects`** enum
- Updating the **`common`** module
- Testing the publisher and listener
- Testing out the Posts service using Postman

Objectives

After reading this chapter, the reader will be able to create an **`NPM`** module out of the shared code in the **`common`** folder. We will also understand the backend code for

More books: https://t.me/booksforits

the Posts service and the integration of the Posts service with the NATS Streaming server. We will understand how a simple Publisher and Listener works for the NATS Streaming server. We will also understand how events work by executing the Posts service from Postman.

Let us start by creating the **common** module.

Introducing the common module

To facilitate code reuse, we will create an NPM package with all the reusable code in the **common** folder. This package will be hosted on NPM and can be installed in any project when needed. The code in this folder will be written in TypeScript and published as JavaScript. Let us spring into action by creating a folder for the reusable code.

Create a folder for **common** and navigate to the folder as follows:

```
/BPB_Mean_Framework/$ mkdir common
/BPB_Mean_Framework/$ cd common
```

Once in the **common** folder, create the **package.json** using the following command:

```
/BPB_Mean_Framework/common/$ npm init -y
```

The **package.json** file is shown here:

```
{
  "name": "@pcblog/common",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

We will now create a Git repository for the **common** module.

More books: https://t.me/booksforits

Creating a GIT repository for the common module

The next step is to initialize a new Git repository for the **common** module as follows:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ git init
Initialized empty Git repository in /home/meanjsbpb/BPB_MEAN_Framework/common/.git/
```

***Figure 7.1**: Initializing a GIT repository*

Add and commit the code to the repository as follows:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ git add .
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ git commit -m "First Commit"
[master (root-commit) 3727f6b] First Commit
 1 file changed, 12 insertions(+)
```

***Figure 7.2**: Committing changes to the repository*

We will now publish the **common** module to the NPM registry.

Publishing the common module to NPM

In order to publish code to NPM, we need to first log in to NPM. We need to create an account on **npmjs.com**.

The next step is to log in to NPM as shown in the following screenshot:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ npm login
npm notice Log in on https://registry.npmjs.org/
Username: pinakinc
Password:
Email: (this IS public) chaubalpinakin@gmail.com
Logged in as pinakinc on https://registry.npmjs.org/.
```

***Figure 7.3**: Logging in to NPM*

Once we log in successfully, we publish this code to the NPM registry as shown in the following screenshot:

More books: https://t.me/booksforits

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ npm publish --access public
npm notice
npm notice 📦 @pcblog/common@1.0.0
npm notice === Tarball Contents ===
npm notice 229B package.json
npm notice === Tarball Details ===
npm notice name:          @pcblog/common
npm notice version:       1.0.0
npm notice filename:      @pcblog/common-1.0.0.tgz
npm notice package size:  256 B
npm notice unpacked size: 229 B
npm notice shasum:        67a6965b8c544c913fd2811952897341eb078a15
npm notice integrity:     sha512-iZaOE0le4lZ0H[...]AMqaxfagKi3QA==
npm notice total files:   1
npm notice
+ @pcblog/common@1.0.0
```

***Figure 7.4**: Publishing to NPM*

Let us view the published package on NPMJS as shown in the following screenshot:

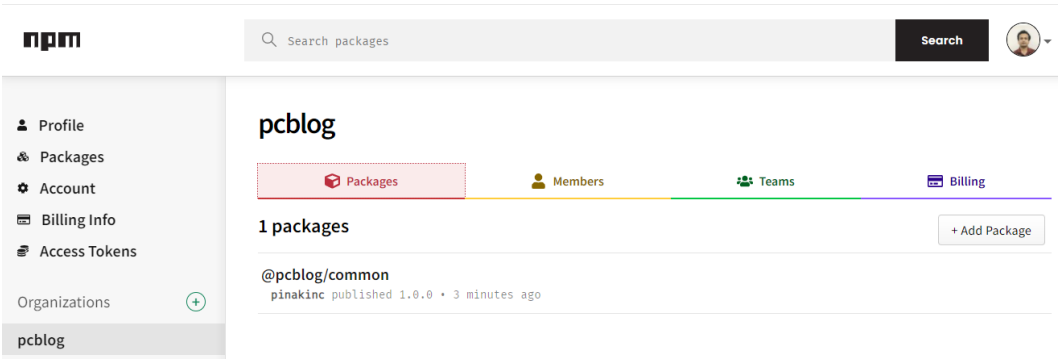

***Figure 7.5:**common package on NPM*

Now that we have published the initial package on NPM, we will move on to installations in the **common** folder.

Installing required packages in the common module

We will now install TypeScript in the **common** module as shown in the following screenshot:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ npm install typescript

added 1 package, and audited 3 packages in 2s

found 0 vulnerabilities
```

***Figure 7.6**: Installing TypeScript*

More books: https://t.me/booksforits

The next step is to create the **tsconfig** file inside the **common** folder as shown in the following screenshot:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/common$ npx tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true
```

Figure 7.7: *Creating a tsconfig file*

Install the package **del-cli** as a dev dependency using the following command:

```
/BPB_Mean_Framework/common/$ npm install del-cli –save-dev
```

Making changes to package.json and tsconfig

In the **package.json** file, we need to make changes so that the **build** folder is deleted before a fresh build and then do a cleanup using **npm run** clean as well as trigger the TypeScript compiler. When the **common** module is imported into some other module, the **index.js** should be picked from the **build** folder and that is indicated in the **main** tag. The **types** tag indicates where the **main** type definition file is and the files tag specifies which files have to be present in the published version of the **common** folder which in our case are the entire contents of the **build** folder.

One more thing to note is that we must follow certain steps to publish a module to npm. The steps are as follows:

1. Add the changes to the **staging** folder of git using **git add**.
2. Do a **git commit**.
3. Increment the version in **package.json**.
4. Build and publish the module.

We will need to revisit the **common** module time and again and hence to speed up the development process, we combine these steps and place it in a tag called **pub** which stands for publish. Please remember that this particular change is only for development purposes and we will never use this in production.

Let us see the changes made to **package.json.**

More books: https://t.me/booksforits

Making changes to package.json

Make the changes to the **main**, **scripts**, **types**, and **files** in the **script** section as shown in the following code:

```
{
  "name": "@pcblog/common",
  "version": "1.0.0",
  "description": "",
  "main": "./build/index.js",
  "scripts": {
    "clean": "del ./build/*",
    "build": "npm run clean && tsc"
    "pub": "git add . && git commit -m \"Updates\" && npm version patch
&& npm
            run build && npm publish"

  },
  "types": "./build/index.d.ts",
  "files": [
    "build/**/*"
  ],
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Let us see the changes made to **tsconfig.json**.

Changes to tsconfig

In **tsconfig.json**, make changes as follows:

```
"declaration": true,
"outDir": "./build",
```

TypeScript generates a type-definition **.d.ts** file from the **.js** file if the declaration is set to true and **outDir**. It will specify the **build** directory inside the **common** folder in which **.js** files will be placed after compilation.

More books: https://t.me/booksforits

When we commit the **common** module to git, the **build** folder and **node_modules** folder should not get committed and that is what is specified in the **.gitignore** file. Create a new file with the name **.gitignore** (Please note the **.** before **gitignore**) and mention the **node_modules** and build inside the file.

We will now move the **errors** and **middlewares** folders from the **Auth** directory to the **common** module.

Moving the authentication logic in the common module

For reuse purpose, we will now move the **errors** and **middlewares** folders from the **Auth** folder to the **common** folder.

The next step is to create an **index.ts** in the **src** folder in the **common** module and export the files in the **errors** and **middlewares** folder so that other folders can access these files.

The **index.ts** file is shown in the following code:

```
export * from './errors/bad-request-err';
export * from './errors/custom-err';
export * from './errors/database-connection-err';
export * from './errors/not-auth-err';
export * from './errors/not-found-err';
export * from './errors/request-validation-err';
export * from './middlewares/current-user';
export * from './middlewares/error-handler';
export * from './middlewares/require-auth';
export * from './middlewares/validate-request';
```

Install the following modules in the **common** folder:

- **express**
- **express-validator**
- **cookie-session**
- **jsonwebtoken**
- **@types/cookie-session**
- **@types/express**
- **@types/jsonwebtoken**

More books: https://t.me/booksforits

Do an **npm run pub** to build, commit, and publish to the NPM registry. Next, we will install the **common** module in the **Auth** service.

Installing the common module in the auth folder

We have published a working version of the **common** module to NPM. If we take a look at the **Auth** folder, it contains a lot of broken **import** statements.

To fix these, we will install the **common** module in the Auth service as shown here:

```
/BPB_Mean_Framework/auth/$ npm install @pcblog/common
```

In the **routes** folder in **Auth**, update the import in current-user route as shown here:

```
import {currentUser} from '@pcblog/common';
```

Update the import in the sign-in route as shown here:

```
import {validateRequest, BadRequestErr} from '@pcblog/common';
```

Update the import in the sign-up route as shown here:

```
import {validateRequest,BadRequestErr} from '@pcblog/common';
```

Update the import in **index.ts** as shown here:

```
import {errorHandler,NotFoundErr} from '@pcblog/common';
```

This completes the setup of the **common** folder. We move on to the Posts service now which will take care of CRUD operations for the blog posts.

Standard process for new services

We will follow a standard process for developing new services for our application. The steps are as follows:

1. Create **package.json**.
2. Install required dependencies.
3. Create **Dockerfile**.
4. Create **index.ts** to serve as an execution starting point.
5. Build the image and push it to the Docker hub.
6. Create a **deployment** and a **service**.

More books: https://t.me/booksforits

7. Update **skaffold.yaml** for syncing files.
8. Create a deployment and service for MongoDB.

The Posts service will have its own database. The posts service will have the following routes defined as shown in the following table titled **Posts Service**:

Posts Service			
Route	**Method**	**Body**	**Objective**
/api/posts	GET	-	Retrieve all posts
/api/posts/:id	GET	-	Retrieve the post with a specific ID
/api/posts	POST	{title: string, content: string}	Create a Post
/api/posts	PUT	{title: string, content: string}	Update a Post

***Table 7.1**: Routes for the Posts service*

The Posts service

We will now start creating the Posts service which will be responsible for creating, updating, deleting, and fetching posts.

We will start by creating a separate folder under the parent project folder.

Creating the Posts folder

Create a folder under the project folder and name it **posts**. Copy the following files in the **posts** folder from the **auth** folder:

- **package.json**
- **package.lock.json**
- **Dockerfile**
- **.dockerignore**
- **tsconfig.json**

Perform a **npm install** to install dependencies. Create a folder called **src** in the **posts** folder and copy the **index.ts** from the auth folder. Update the **package.json** as mentioned:

```
"name": "posts"
```

More books: https://t.me/booksforits

Originally, this will have the name '**auth**' since it was copied from the auth service. We will now update **index.ts**.

Updating the index.ts

The **index.ts** will be updated by following the given steps:

1. Import **express**, **express-async-errors**, **json**, **mongoose**, and **cookieSession** as shown here:

```
import express from 'express';
import 'express-async-errors';
import {json} from 'body-parser';
import mongoose from 'mongoose';
import cookieSession from 'cookie-session';
```

2. Import **errorHandler**, **NotFoundErr**, and **currentUser** as shown here:

```
import {errorHandler,NotFoundErr,currentUser} from '@pcblog/
common';
```

3. Import **createPostRouter**, **showPostsRouter** and **indexPostRouter**, **updatePostRouter** and **natsWrapper** as shown here:

```
import {createPostRouter} from './routes/createPost';
import {showPostsRouter} from './routes/showPosts';
import {indexPostRouter} from './routes/index';
import { updatePostRouter } from './routes/update';
import {natsWrapper} from './nats-wrapper'
```

4. Create an **express** app, set the **trust proxy** to true, and configure the port to be used as shown below:

```
const app=express();
app.set('trust proxy', true);
const port = process.env.PORT || 3100;
```

5. Use different **middlewares** as follows:

```
app.use(json());
app.use(cookieSession({
    signed: false,
```

More books: https://t.me/booksforits

```
    secure: true
})
);

app.use(currentUser);

app.use(createPostRouter);
app.use(showPostsRouter);
app.use(indexPostRouter);
app.use(updatePostRouter)

app.all('*',async(req,res)=>{
    throw new NotFoundErr();
});
app.use(errorHandler);
```

6. Put checks for **JWT_KEY** and **MONGO_URI** as follows:

```
const startup = async()=>{
    if(!process.env.JWT_KEY){
        throw new Error('Jwt key must be defined');
    }

    if(!process.env.MONGO_URI){
        throw new Error('MONGO_URI must be defined');
    }
```

7. Connect to the Nats streaming server and handle the system interrupt signals as follows:

```
    try{
        await natsWrapper.connect('blog','hhdhdhdhd','http://nats-
srv:4222');
        natsWrapper.client.on('close',()=>{
            console.log('NATS connection closed!');
```

More books: https://t.me/booksforits

```
        process.exit();
    });

    process.on('SIGINT',()=>natsWrapper.client.close());
    process.on('SIGTERM',()=>natsWrapper.client.close());
```

8. Connect to MongoDB as follows:

```
    await mongoose.connect(process.env.MONGO_URI);
    console.log('Connected to Mongo DB');
    } catch(err){
        console.error(err);
    }
    app.listen(port, ()=>{
    console.log('Listening on port 3100',port);
    });


}
```

9. Invoke the **`startup`** function as shown here:

```
startup();
```

10. Build the project using the following command only if you are not running the project on Google Cloud:

```
/BPB_Mean_Framework/posts/$ docker build -t pinakinc/posts
```

11. The preceding step is required if you are not running your project on Google Cloud since Skaffold searches for a build on Docker Hub.

The next step will be the creation of the Posts deployment YAML

Creating the Posts deployment YAML

We will now look at the Posts service deployment YAML. The Posts deployment file is required to run the Posts service. We have references to Mongo URI and JWT key which we will keep since the Posts service will make use of these.

More books: https://t.me/booksforits

The Posts service deployment file is shown as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: posts-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: posts
  template:
    metadata:
      labels:
        app: posts
    spec:
      containers:
        - name: posts
          image: us.gcr.io/blog-dev-326403/posts:project-latest
          env:
            - name: MONGO_URI
              value: 'mongodb://posts-mongo-srv:27017/posts'
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_KEY
---
apiVersion: v1
kind: Service
metadata:
  name: posts-srv
```

More books: https://t.me/booksforits

```
spec:
  selector:
    app: posts
  ports:
    - name: posts
      protocol: TCP
      port: 3100
      targetPort: 3100
```

We will now create the MongoDB deployment YAML for the Posts service.

Creating the Posts Mongo DB deployment YAML

Listed here is the Posts MongoDB deployment file. This file has similar contents to auth MongoDB deployment file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: posts-mongo-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: posts-mongo
  template:
    metadata:
      labels:
        app: posts-mongo
    spec:
      containers:
        - name: posts-mongo
          image: mongo
```

More books: https://t.me/booksforits

```
---
apiVersion: v1
kind: Service
metadata:
  name: posts-mongo-srv
spec:
  selector:
    app: posts-mongo
  ports:
    - name: db
      protocol: TCP
      port: 27017
      targetPort: 27017
```

Next, we take a look at the changes made in Skaffold YAML for the Posts service.

Making changes to the Skaffold YAML

Add the following content in **`Skaffold.yaml`**:

```
    - image: us.gcr.io/blog-dev-326403/posts
      context: posts
      docker:
        dockerfile: Dockerfile
      sync:
        manual:
          - src: 'src/**/.ts'
            dest: .
```

This entry is similar to the auth entry as can be seen from the preceding code listing. Along with this, add the following piece of code to **`ingress-srv.yaml`**:

```
          - path: /api/posts/?(.*)
            backend:
              serviceName: posts-srv
              servicePort: 3100
```

More books: https://t.me/booksforits

Next, we need to make similar changes for the Auth Service.

Looking at the Auth deployment YAML

We will now take a look at the Auth deployment file and put environment variables in place of hard-coded URLs.

The environment variable **`MONGO_URI`** shown in the following code displays how we use the **`MONGO_URI`** environment variable for the MongoDB URL.

There are no changes made to the service as shown here:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: us.gcr.io/blog-dev-326403/auth:project-latest
          env:
            - name: MONGO_URI
              value: 'mongodb://auth-mongo-srv:27017/auth'
            - name: JWT_KEY
              valueFrom:
                secretKeyRef:
```

More books: https://t.me/booksforits

```
                name: jwt-secret
                key: JWT_KEY
---
apiVersion: v1
kind: Service
metadata:
  name: auth-srv
spec:
  selector:
    app: auth
  ports:
    - name: auth
      protocol: TCP
      port: 3100
      targetPort: 3100
```

We will now take a look at changes made to the **`index.ts`** for the Mongo URI.

Looking at the Auth index.ts

For the sake of brevity, we have only given the code snippet that needs changes for the Mongo URI environment variable.

The code fragment is mentioned here:

```
const startup = async()=>{
    if(!process.env.JWT_KEY){
        throw new Error('Jwt key must be defined');
    }
    if(!process.env.MONGO_URI){
        throw new Error('MONGO_URI must be defined');
    }
    try{
    await mongoose.connect(process.env.MONGO_URI);
    console.log('Connected to Mongo DB');
```

More books: https://t.me/booksforits

```
    } catch(err){
        console.error(err);
    }
    app.listen(port, ()=>{
    console.log('Listening on port 3100',port);
    });

}
```

Rest of the code in the Auth service remains unchanged. We will now move on to the code for the Posts service.

Creating the Posts service

In this section, we will create the route handlers for the Posts service. We start with the creation of a new post.

Creating a new Post

We will create a new Post and save it to the corresponding database. At the same time, we publish a message to the NATS Streaming server. We shall see the various code snippets from top to bottom.

Import **`express`**, **`Request`**, and **`Response`** from **`express`** so that we can instantiate the Express application and get the appropriate request and response. Import **`body`** from **`express-validator`**. Import r**`equireAuth`** and **`validateRequest`** from the **`common`** package that is on **`npmjs`**. Import the Post model from the **`models`** folder. We will see the Post model creation in a later section. Import the **`PostCreatedPublisher`** from the **`publishers`** folder under **`events`** and import **`natsWrapper`** from the **`src`** folder under **`Posts`**:

```
import express from 'express';
import {Request, Response} from "express";
import {body} from 'express-validator';
import {requireAuth, validateRequest} from '@pcblog/common';
import {Post} from '../models/posts';
import {PostCreatedPublisher} from '../events/publishers/post-created-publisher';
import {natsWrapper} from '../nats-wrapper';
```

More books: https://t.me/booksforits

Create the Express router and assign it to the router:

```
const router = express.Router();
```

Create a post request to **/api/post** and in this request, build the post with **title**, **content**, and the **ID** of the current user. We have enabled authorization using the **requireAuth** middleware and enabled request validation using the **validateRequest** middleware:

```
router.post('/api/posts',requireAuth,[
    body('title').not().isEmpty().withMessage('Title is required'),
    body('content').not().isEmpty().withMessage('Content is required')

],validateRequest,async (req : Request,res : Response) =>
    {
        const {title,content}=req.body;
        const post = Post.build({
            title,
            content,
            userId: req.currentUser!.id,

        });
```

Save the post to the database:

```
        await post.save();
```

Publish the post to NATS Streaming server and send along a status of 201:

```
        new PostCreatedPublisher(natsWrapper.client).publish({
            id: post.id,
            title: post.title,
            content: post.content,
            userId: post.userId
        });
        res.status(201).send(post);
    });
```

Eventually, export the router:

```
export {router as createPostRouter};
```

More books: https://t.me/booksforits

Updating an existing Post

We will now see the code for updating an existing post. We will update an existing post and at the same time, we will publish a message to the NATS Streaming server. We shall see the various code snippets from top to bottom.

Import **express**, **Request**, and **Response** from express in order to instantiate an Express application, send requests and fetch response data. Import **body** from **express-validator**. Import **requireAuth**, **NotFoundErr**, **NotAuthErr**, and **validateRequest** from the **common** package that is on **npmjs**. Import the **Post** model from the **models** folder. We will see the Post model creation in a later section. Import the **PostCreatedPublisher** from the **publishers** folder under **events** and import **natsWrapper** from the **src** folder under **Posts** using the following code:

```
import express, { Request, Response } from 'express';
import {body} from 'express-validator';
import {
    validateRequest,
    NotFoundErr,
    requireAuth,
    NotAuthErr
} from '@pcblog/common';
import {Post} from '../models/posts';
import {PostUpdatedPublisher} from '../events/publishers/post-updated-publisher';
import {natsWrapper} from '../nats-wrapper';
```

Create the Express router and assign it to the router:

```
const router = express.Router();
```

Create a put request on **/api/posts/id**. We require the user to be authenticated before updating a post. Similarly, we have put the check for missing titles and content. For that, we invoke the **validateRequest** middleware. We find the post first and then update it. If any user tries to update someone else's post, we throw an error message as follows:

```
router.put('/api/posts/:id',requireAuth, [
    body('title')
    .not()
```

More books: https://t.me/booksforits

```
    .isEmpty()
    .withMessage('Title is required'),
    body('content')
    .not()
    .isEmpty()
    .withMessage('Content is required')
], validateRequest,
async (req: Request,res: Response)=>{
    console.log(req.params.id);
    const post = await Post.findById(req.params.id);

    if (!post) {
        throw new NotFoundErr();
    }

    if (post.userId !== req.currentUser!.id) {
        throw new NotAuthErr();
    }

    post.set({
        title: req.body.title,
        content: req.body.content
    });
```

If all goes well, save the post, and publish a message to the NATS Streaming server:

```
    await post.save();
    new PostUpdatedPublisher(natsWrapper.client).publish({
        id:post.id,
        title:post.title,
        content: post.content,
        userId: post.userId
    })
```

More books: https://t.me/booksforits

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

Eventually, export the router:

```
export {router as updatePostRouter};
```

Now, we will see how to display all posts that exist in the system.

Displaying all Posts

The code for displaying all posts is comparatively smaller and simpler.

Import **`express`**, **`Request`**, and **`Response`** from **`express`** and the **`post`** model from the **`models`** folder:

```
import express,{Request,Response} from 'express';
import {Post} from '../models/posts';
```

Create the Express router and assign it to the router:

```
const router =express.Router();
```

Create a get request on **`/api/posts`**. First of all, find all the posts and send them to the response:

```
router.get('/api/posts', async (req: Request,res:Response) => {
    const posts = await Post.find({});
    res.send(posts);
});
```

Export the router:

```
export {router as indexPostRouter};
```

Now, we will look into the last route to find a specific post by ID.

Displaying a specific Post

In order to pull a specific post, we pass the ID in the URI. Let's take a look at the code that is involved.

Import **`express`**, **`Request`**, and **`Response`** from **`express`**. Import the **`NotFoundErr`** from the **`common`** module and import the **`Post`** model from the **`models`** folder:

```
import express, {Request, Response} from 'express';
```

More books: https://t.me/booksforits

```
import {NotFoundErr} from '@pcblog/common';
import {Post} from '../models/posts';
```

Create the router and assign to the router:

```
const router = express.Router();
```

Create a get request on **/api/posts/:id**. Find the post using the ID passed in the URI. If the Post is not found, throw a **NotFoundErr** else send the post to the response:

```
router.get('/api/posts/:id',async (req: Request,res: Response) => {
    const post = await Post.findById(req.params.id);
    if (!post){
        throw new NotFoundErr();
    }
    res.send(post);
});
```

Export the router:

```
export {router as showPostsRouter};
```

Next, we will create a **nats-wrapper** for the NATS Streaming server.

Creation of the nats-wrapper class

Let us see what is involved in creating the **nats-wrapper** class.

Import **nats** and Stan from **node-nats-streaming**:

```
import nats, {Stan} from 'node-nats-streaming';
```

Create a class **NatsWrapper**. In this class, create a property **_client** of type **Stan**. Fetch the NATS client and connect:

```
class NatsWrapper {

    private _client?:Stan;

    get client(){
        if(!this._client){
          throw new Error('Cannot access NATS client before connecting');
        }
        console.log('inside nats-wrapper'+this._client);
```

More books: https://t.me/booksforits

```
        return this._client;
    }

    connect(clusterId:string,clientId:string,url:string){

        this._client=nats.connect(clusterId, clientId, {url});
```

Return a promise for the status of the connection and return or reject appropriately as shown in the following code:

```
        return new Promise<void>((resolve,reject)=> {

        this.client.on('connect',()=>{
            console.log('connected to NATS');
            resolve();
        });
        this.client.on('error',()=>{
            console.log('some error connecting to NATS');
            reject();
        });
    });
    }
}
```

Create an instance of **`NatsWrapper`**, assign it, and export it.

```
export const natsWrapper = new NatsWrapper();
```

With this, we complete all the routes concerned with the Post service as well as the NATS wrapper. We will now put in the missing pieces such as the Post model and the NATS Streaming server.

Next, we will see the Post model.

Creation of the Posts model

In this section ,we will create the Posts model. We start by importing mongoose and Mongoose from **`mongoose`**:

```
import mongoose, { Mongoose } from 'mongoose';
```

More books: https://t.me/booksforits

Create an interface for the structure of a Post which includes **title**, **content**, and **userId**:

```
interface PostAttribs {
    title: string;
    content: string;
    userId: string;

}
```

Create an interface that extends **mongoose.Document** and is comprised of the **title**, **content**, and **userId**:

```
interface PostDoc extends mongoose.Document {
    title: string;
    content: string;
    userId: string;

}
```

Create an interface for the model. This will extend **mongoose.Model** which takes in the document created in the previous step:

```
interface PostModel extends mongoose.Model<PostDoc> {
    build(attribs:PostAttribs): PostDoc

}
```

Next, we create a schema which specifies the properties of the **title**, **content**, and **userId**. We also have the **toJSON** tag that has the transform function to remove **_id** from the response:

```
const postSchema = new mongoose.Schema ({
    title: {
        type: String,
        required: true

    },

    content: {
```

More books: https://t.me/booksforits

```
        type: String,
        required: true

    },
    userId: {
        type: String,
        required: true

    },

}, {
    toJSON:{
        transform(doc, ret){
            ret.id = ret._id;
            delete ret._id;
        }
    }
});
```

We build a post using the **`build`** function as follows:

```
postSchema.statics.build = (attribs: PostAttribs)=>{
    return new Post(attribs);
};
```

Finally, we construct the Post model using the **`PostDoc`** and **`PostModel`**:

```
const Post = mongoose.model<PostDoc,PostModel>('Post',postSchema);
```

Finally, export the **`Post`**:

```
export {Post};
```

With the Post model in place, it is time to fill the next missing piece, the NATS streaming server.

Introduction to the NATS streaming server

Software applications and services need to exchange data to fulfill messaging needs between services. NATS is an infrastructure that allows for data exchange, which is

More books: https://t.me/booksforits

in the form of messages. The NATS Streaming server is also called **message-oriented middleware**.

With NATS, application developers can:

- Build distributed and scalable client-server applications with minimal effort.
- Storage and distribution of data in real time.

Developers make use of one of the NATS client libraries in their application code so that they can publish, subscribe, request, and reply between instances of the application or between completely disparate applications. Those applications are referred to as *client applications* or sometimes just as *clients*.

We will incorporate NATS like any other service by creating a NATS deployment file in a folder dedicated to NATS.

Creating the NATS deployment file

The first step for the creation of the NATS service is creating the NATS deployment file. The NATS deployment file is shown here:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats
  template:
    metadata:
      labels:
        app: nats
    spec:
      containers:
        - name: nats
          image: nats-streaming:0.17.0
```

More books: https://t.me/booksforits

```
          args: [
            '-p',
            '4222',
            '-m',
            '8222',
            '-hbi',
            '5s',
            '-hbt',
            '5s',
            '-hbf',
            '2',
            '-SD',
            '-cid',
            'blog'
          ]
---
apiVersion: v1
kind: Service
metadata:
  name: nats-srv
spec:
  selector:
    app: nats
  ports:
    - name: client
      protocol: TCP
      port: 4222
      targetPort: 4222
    - name: monitoring
      protocol: TCP
      port: 8222
      targetPort: 8222
```

More books: https://t.me/booksforits

A few parameters in the NATS Streaming configuration to make a note of are follows:

- **-p** is the port on which the NATS Streaming server is available.
- **-m** is the port on which the NATS Streaming server can be monitored.
- **-hbi** is the interval at which the server sends a heartbeat to the client.
- **-hbt** denotes the time for which the server waits for a heartbeat response.
- **-hbf** denotes the number of failed heartbeats before the server closes the client connection.
- **-SD** is used to enable the STAN debugging output.
- **-cid** indicates the cluster id which in our case is a blog.

Creating a basic publisher and listener ts files

To understand how the NATS Streaming server works, we will try out a simple Publisher and a Listener. The Publisher will publish messages and the Listener will receive and process those. The Publisher and Listener are basic and will serve the purpose of understanding the NATS Streaming server.

Let us start with the Publisher.

The Test Publisher

Create a folder called **nats-test** and create the **src** folder there. Inside the **src** folder, create a file called **publisher.ts**. Install **node-nats-streaming** in the **nats-test** folder,

Import **post-created-publisher** from the **events** folder and **nats** from **node-nats-streaming**:

```
import {PostCreatedPublisher} from './events/post-created-publisher';
import nats from 'node-nats-streaming';
```

Connect to the cluster **blog** with URL **http://localhost:4222**:

```
const stan = nats.connect('blog','abc',{
    url:'http://localhost:4222'});
```

Verify that we are able to connect to NATS:

```
    stan.on('connect',async ()=>{
        console.log('Publisher connected to Nats');
```

More books: https://t.me/booksforits

Create an instance of **`PostCreatedPublisher`** by passing it stan and publish messages with **`ID`**, **`title`**, and **`content`**:

```
        const publisher = new PostCreatedPublisher(stan);
        try {
        await publisher.publish({
            id: '123',
            title: 'New entries from Automation Geek',
            content: 'New entries added',

        });
    } catch (err){
        console.error(err);
    }
    });
```

We will now create a test listener which will listen to events emitted by the Publisher.

The Test Listener

The Test Listener will be created in the same folder as the test publisher and will listen to events emitted by the Publisher. The code in the Listener has been split up for ease of understanding.

Put in the required imports as shown here:

```
import nats from 'node-nats-streaming';
import {randomBytes} from 'crypto';
import {PostCreatedListener} from './events/post-created-listener';
```

Connect to the NATS server on **`4222`** by passing in the cluster ID and URL which in our case is **`http://localhost:4222`**:

```
const stan = nats.connect('blog',randomBytes(4).toString('hex'),{
    url:'http://localhost:4222'
});
```

Start listening for new events by creating a new instance of **`PostCreatedListener`** by passing in the NATS client (**`stan`**) and invoking the **`listen()`** function on it:

```
stan.on('connect',()=>{
```

More books: https://t.me/booksforits

```
    console.log('Listener connected to Nats');
    stan.on('close',()=>{
        console.log('NATS connection closed');
        process.exit();
    });
    new PostCreatedListener(stan).listen();
});
```

Listen to **System** event when they occur:

```
process.on('SIGINT',()=> stan.close());
process.on('SIGTERM',()=> stan.close());
```

As seen in the preceding code, we created a new instance of **PostCreatedListener** passing in the NATS client. The **PostCreatedListener** is a sub class of a class called **BaseListener**.

Let us take a look at the **BaseListener** followed by the **PostCreatedListener**.

Understanding the BaseListener and PostCreatedListener

In the **nats-test** folder, under the **src** folder, create a folder called **events**. Create files called **base-listener**, and **post-created-listener** in the **events** folder.

Let us understand the **base-listener** first.

The base-listener class

Import **Message** and **Stan** from the **node-nats-streaming** library and **Subjects** from the **subjects** enum. We will see how the **Subjects** enum is structured:

```
import { Message, Stan } from 'node-nats-streaming';
import {Subjects} from './subjects';
```

Define an event interface with properties **subject** and **data**. The subject is of type **Subjects** enum and data is of type **any**:

```
interface Event {
    subject: Subjects;
    data: any;
}
```

More books: https://t.me/booksforits

Define an abstract class **`Listener`** which takes in a generic **`T`** which extends the **`Event`** interface. Let us break this class into small pieces and understand what is getting done here:

```
export abstract class Listener<T extends Event> {
```

Define an abstract property called **`subject`** which is of the type **`subject`** from the **`Event`** interface, an abstract property **`queueGroupName`** of type string, an abstract **`onMessage`** function which takes data from the **`Event`** interface and msg of type message, private **`Stan`** client, and a timeout of 5 seconds:

```
  abstract subject: T['subject'];

  abstract queueGroupName: string;

  abstract onMessage(data: T['data'], msg: Message): void;

  private client: Stan;

  protected ackWait = 5 * 1000;
```

Define a constructor function and assign the **`stan`** client to the **`client`** property:

```
  constructor(client: Stan) {
    this.client = client;
  }
```

Define the various subscription options and in this, return the stan client by setting subscription options, delivering all available messages, manual acknowledgement, and the durable name by passing in the queue group name:

```
  subscriptionOptions() {
    return this.client
      .subscriptionOptions()
      .setDeliverAllAvailable()
      .setManualAckMode(true)
      .setAckWait(this.ackWait)
      .setDurableName(this.queueGroupName);
  }
```

Invoke the **`subscribe`** function on the stan client with the **`subject`**, **`queueGroupName`**, and **`subscriptionOptions`**:

```
  listen() {
    const subscription = this.client.subscribe(
      this.subject,
```

More books: https://t.me/booksforits

```
      this.queueGroupName,
      this.subscriptionOptions()
    );
```

Invoke the on function on the subscription with the string **message** and a callback which accepts a **Message**. Print the subject and the **queuegroupname** to the console. Invoke the **parseMessage** by passing in **msg** and assign it to the **parsedData** property. Invoke the **onMessage** with the **parsedData** and **msg**:

```
    subscription.on('message', (msg: Message) => {
            console.log(`Message received: ${this.subject} / ${this.
queueGroupName}`);

      const parsedData = this.parseMessage(msg);
      this.onMessage(parsedData, msg);
    });
  }
```

The **parseMessage** function takes in the msg of the type **Message** and uses a ternary operator to pull out data:

```
  parseMessage(msg: Message) {
    const data = msg.getData();
    return typeof data === 'string'
      ? JSON.parse(data)
      : JSON.parse(data.toString('utf8'));
  }
}
```

Let us look at the **post-created-listener** class next.

The post-created-listener class

Import **PostCreatedEvent** from the **post-created-event** and the **Subjects** from the **subjects** enum:

```
import {PostCreatedEvent} from './post-created-event';
import {Subjects} from './subjects';
```

Export a class called **PostCreatedListener** which extends the **Listener** abstract class by passing a generic of **PostCreatedEvent**:

```
export class PostCreatedListener extends Listener<PostCreatedEvent> {
```

More books: https://t.me/booksforits

Assign the **`PostCreated`** enum entry to the subject and a hard-coded value of **`blog-service`** to **`queueGroupName`**:

```
  subject: Subjects.PostCreated = Subjects.PostCreated;
  queueGroupName = 'blog-service';
```

Invoke the **`onMessage`** function by passing in data from **`PostCreatedEvent`** and a message. Extract out the **`id`**, **`title`**, and **`content`** from the data and console log the output. Eventually, manually acknowledge the message:

```
  onMessage(data: PostCreatedEvent['data'], msg: Message) {
    console.log('Event data!', data);

    console.log(data.id);
    console.log(data.title);
    console.log(data.content);

    msg.ack();
  }
}
```

Let us understand the Base Publisher and **`PostCreatedPublisher`** next:

```
import {Stan} from 'node-nats-streaming';
import {Subjects} from './subjects';

interface Event{
    subject: Subjects;
    data: any;
}
export abstract class Publisher<T extends Event> {
    abstract subject: T['subject'];
    private client: Stan;

    constructor(client: Stan) {
        this.client = client;
    }
```

More books: https://t.me/booksforits

```
    publish(data: T['data']): Promise<void>{
        return new Promise((resolve,reject)=>{
          this.client.publish(this.subject,JSON.stringify(data),(err)=>{
                if (err){
                    return reject(err);
                }

                console.log('Event published to subject',this.subject);
                resolve();
            });

        });
    }

}
```

Understanding the BasePublisher, PostCreatedPublisher and PostUpdatedPublisher

In the **nats-test** folder, under the **events** folder, create files called **base-publisher** and **post-created-publisher**. Let us understand the **base-publisher** first.

The base-publisher class

We will break the base-publisher and understand what goes on inside this class.

Import the **Stan** client from **node-nats-streaming** library and **Subjects** from the **subjects** enum:

```
import {Stan} from 'node-nats-streaming';
import {Subjects} from './subjects';
```

Define an interface called **Event** with the subject of type **Subjects** and data of type **any**:

More books: https://t.me/booksforits

```
interface Event{
    subject: Subjects;
    data: any;
}
```

Create an abstract class **`Publisher`** with generics of an **`Event`** type:

```
export abstract class Publisher<T extends Event> {
```

Extract the **`subject`** and the **`Stan`** client:

```
    abstract subject: T['subject'];
    private client: Stan;
```

Define a constructor function for assigning the client:

```
    constructor(client: Stan) {
        this.client = client;
    }
```

Publish the data by returning a promise and resolving it:

```
    publish(data: T['data']): Promise<void>{
        return new Promise((resolve,reject)=>{
          this.client.publish(this.subject,JSON.stringify(data),(err)=>{
                if (err){
                    return reject(err);
                }

                console.log('Event published to subject',this.subject);
                resolve();
            });

        });
    }

}
```

Next, let us see the **`post-created-publisher`** class next

More books: https://t.me/booksforits

The post-created-publisher class

Create a folder called **publishers** inside **posts/src**. Inside the **posts/src/publishers**, create a class called **post-created-publisher**. The post-created publisher will extend the base publisher.

Import the **Publisher**, the **PostCreatedEvent**, and **Subjects** from **common**. Assign the **PostCreated** enum entry to the **subject** property:

```
import {Publisher,Subjects,PostCreatedEvent} from '@pcblog/common';


export class PostCreatedPublisher extends Publisher<PostCreatedEvent>{
    subject: Subjects.PostCreated = Subjects.PostCreated;
}
```

Let us see the **post-updated-publisher** next.

The post-updated-publisher class

Inside the **posts/src/publishers**, create a class called **post-updated-publisher**. The post-updated publisher will extend the base publisher.

Import the **Publisher**, the **PostCreatedEvent**, and **Subjects** from **common**. Assign the **PostUpdated** enum entry to the **subject** property:

```
import {Publisher,Subjects,PostUpdatedEvent} from '@pcblog/common';

export class PostUpdatedPublisher extends Publisher<PostUpdatedEvent> {
    subject:Subjects.PostUpdated=Subjects.PostUpdated;

}
```

Understanding the PostCreatedEvent and the PostUpdatedEvent

Let us see the **post-created-event** and **post-updated-event** next. Create **post-created-event** and **post-updated-event.ts** under **common/src/events** and also under **nats-test/src**. We start with the **PostCreatedEvent**.

The PostCreatedEvent

The **PostCreatedEvent** will enforce restrictions on the type of data received. We are going to receive a **subject** of type **Subjects.PostCreated** and data which comprises **id**, **title**, and **content** which are strings:

More books: https://t.me/booksforits

```
import {Subjects} from './subjects';

export interface PostCreatedEvent {
    subject: Subjects.PostCreated;
    data: {
        id: string;
        title: string;
        content: string;



    };
}
```

This is a comparatively a simple interface which makes use of the **`Subjects`** enum.

Let us see the **`PostUpdatedEvent`** next.

Understanding the PostUpdatedEvent

The **`PostUpdatedEvent`** is very similar to **`PostCreatedEvent`** except for the entry from the **`Subjects`** enum and an additional user **`id`** in the data portion as shown here:

```
import {Subjects} from  './subjects';

export interface PostUpdatedEvent {
    subject: Subjects.PostUpdated;
    data: {
        id: string;
        title: string;
        content: string;
        userId: string;
    };
}
```

Let us take a look at the **`subjects`** enum next.

More books: https://t.me/booksforits

Understanding the subjects enum

We will have two entries in the **subjects** enum as shown here:

```
export enum Subjects {
    PostCreated = "post:created",
    PostUpdated = "post:updated"
}
```

One entry is for the creation of a post and the other is for updates to a post.

We will take a look at the Post updated event next. Since we have all the pieces, it is time to shuffle them.

Updating the common module

Copy the **base-publisher**, **base-listener**, **post-created-event**, **post-updated-event**, and **subjects** enum to the **events** folder of the **common** directory and do an **npm run pub** to publish the changes to the NPM registry.

Go to the **post** folder and do an **npm install @pcblog/common**. This installs the **common** module into the posts service.

Let us do a test of our **publisher.ts** and **listener.ts**.

Testing the publisher and listener

Bring up the Kubernetes cluster by referring to earlier sections. Find the **nats pod** and do a port forwarding to **4222** using the following commands:

```
meanjsbpb@cloudshell:~ (blog-dev-326403)$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
auth-depl-6f86b5d874-wmpbm          1/1     Running   0          2m5
auth-mongo-depl-64856fc95f-btqmh    1/1     Running   0          2m4
client-depl-598f44fbd4-vgq66        1/1     Running   0          2m3
nats-depl-869fc5f45f-bgmvk          1/1     Running   0          2m1
posts-depl-76d6cd9944-bpsmn         1/1     Running   0          2m
posts-mongo-depl-7fcf5f4c67-nnl7b   1/1     Running   0          2m
```

***Figure 7.8**: List the running pods*

Open a new terminal session and run the port forwarding command as shown in the following screenshot:

```
meanjsbpb@cloudshell:~ (blog-dev-326403)$ kubectl port-forward nats-depl-869fc5f45f-bgmvk 4222:4222
Forwarding from 127.0.0.1:4222 -> 4222
```

***Figure 7.9**: Enable Port forwarding*

More books: https://t.me/booksforits

Open a new terminal session and type the command as shown in the following screenshot:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/nats-test (blog-dev-326403)$ npm run listen

> nats-test@1.0.0 listen
> ts-node-dev --rs --notify false src/listener.ts

[INFO] 10:32:28 ts-node-dev ver. 1.1.8 (using ts-node ver. 9.1.1, typescript ver. 4.5
.5)
Listener connected to Nats
```

Figure 7.10: *Executing the listener*

Open a new terminal session and type the following command:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/nats-test (blog-dev-326403)$ npm run publis
h

> nats-test@1.0.0 publish
> ts-node-dev --rs --notify false src/publisher.ts

[INFO] 10:33:08 ts-node-dev ver. 1.1.8 (using ts-node ver. 9.1.1, typescript ver. 4.5
.5)
Publisher connected to Nats
Event published to subject post:created
```

Figure 7.11*: Output of the Publisher*

Switch back to the **`Listener`** window and see the output as shown in the following screenshot:

```
meanjsbpb@cloudshell:~/BPB_MEAN_Framework/nats-test (blog-dev-326403)$ npm run listen

> nats-test@1.0.0 listen
> ts-node-dev --rs --notify false src/listener.ts

[INFO] 10:32:28 ts-node-dev ver. 1.1.8 (using ts-node ver. 9.1.1, typescript ver. 4.5
.5)
Listener connected to Nats
Message received: post:created / blog-service
Event data! {
  id: '123',
  title: 'New entries from Automation Geek',
  content: 'New entries added'
}
123
New entries from Automation Geek
New entries added
```

Figure 7.12*: Output of the Listener*

Now that we have tested the application using the **`Publisher`** and **`Listener`**, let us test it using Postman.

More books: https://t.me/booksforits

Testing out the Posts service using Postman

Follow the given steps to test the Posts service using Postman:

1. Sign up using the following code:

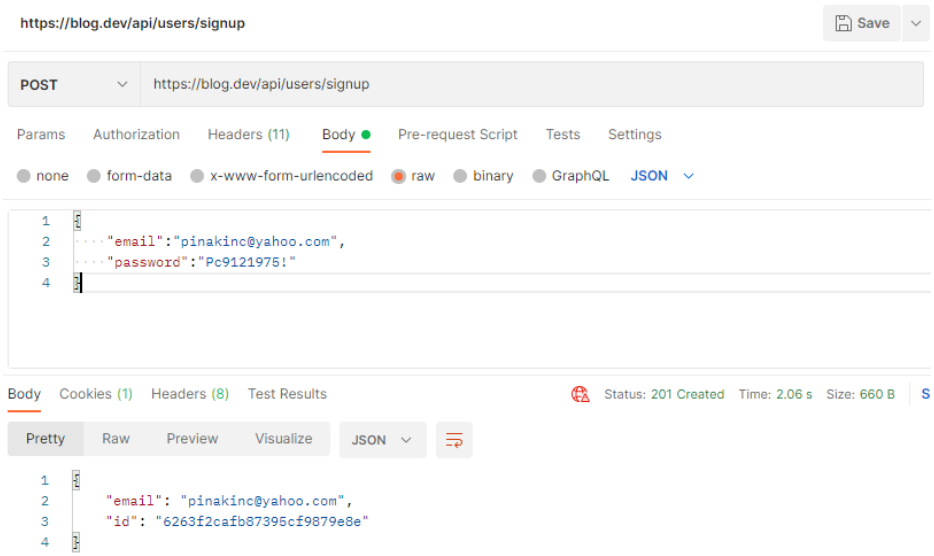

***Figure 7.13**: Signing up using Postman*

2. Sign in using the username/password as shown in the following screenshot:

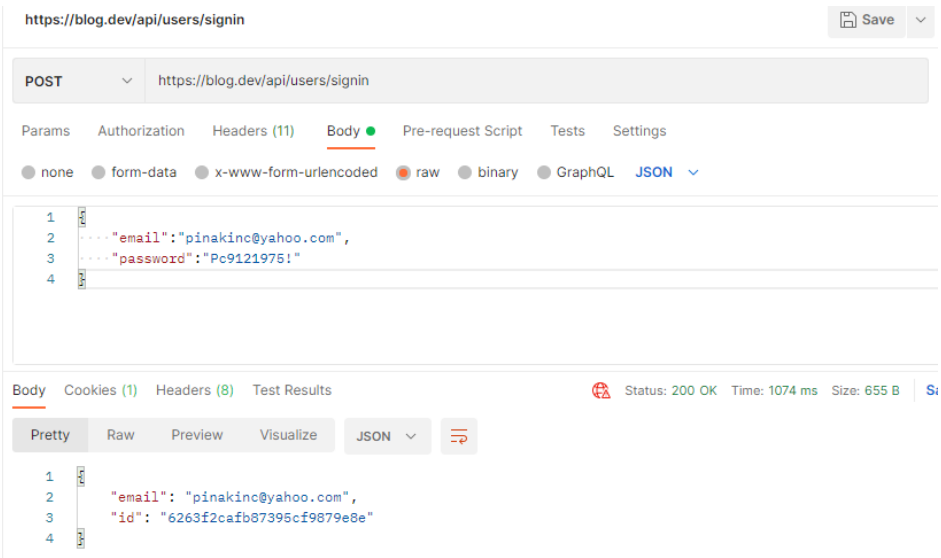

***Figure 7.14**: Signing in using Postman*

More books: https://t.me/booksforits

3. Create 2 new Posts as shown in the following screenshot:

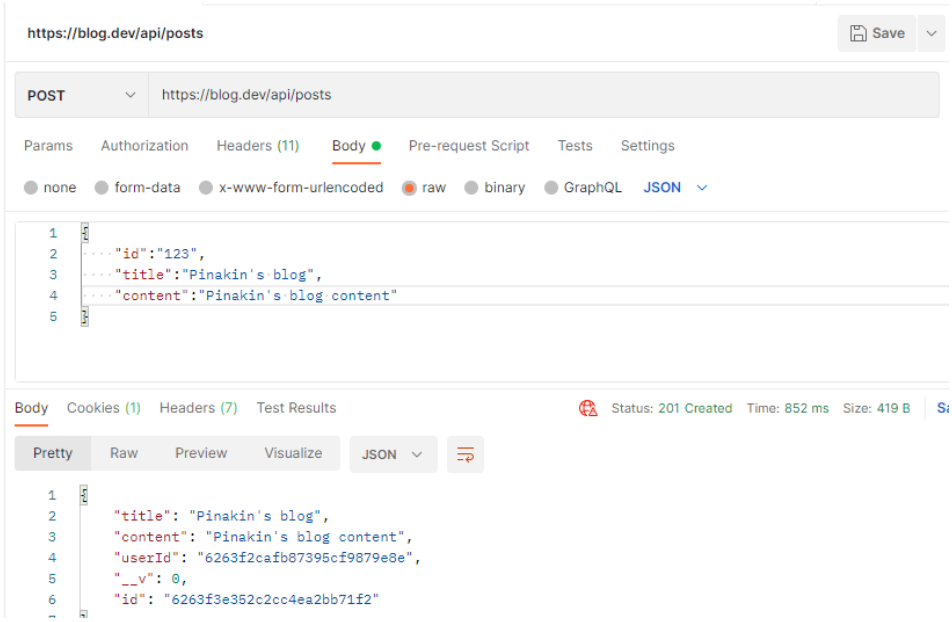

Figure 7.15: *Creating a blogpost in Postman*

In the following screenshot, we have added another post:

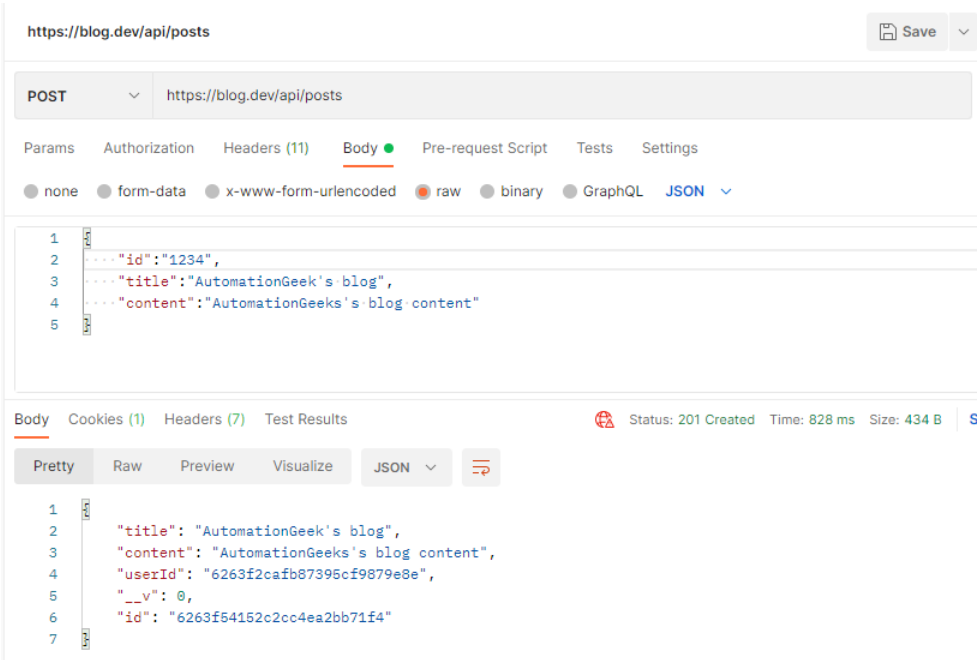

Figure 7.16*: Creating a blogpost in Postman*

More books: https://t.me/booksforits

4. List the available blogs:

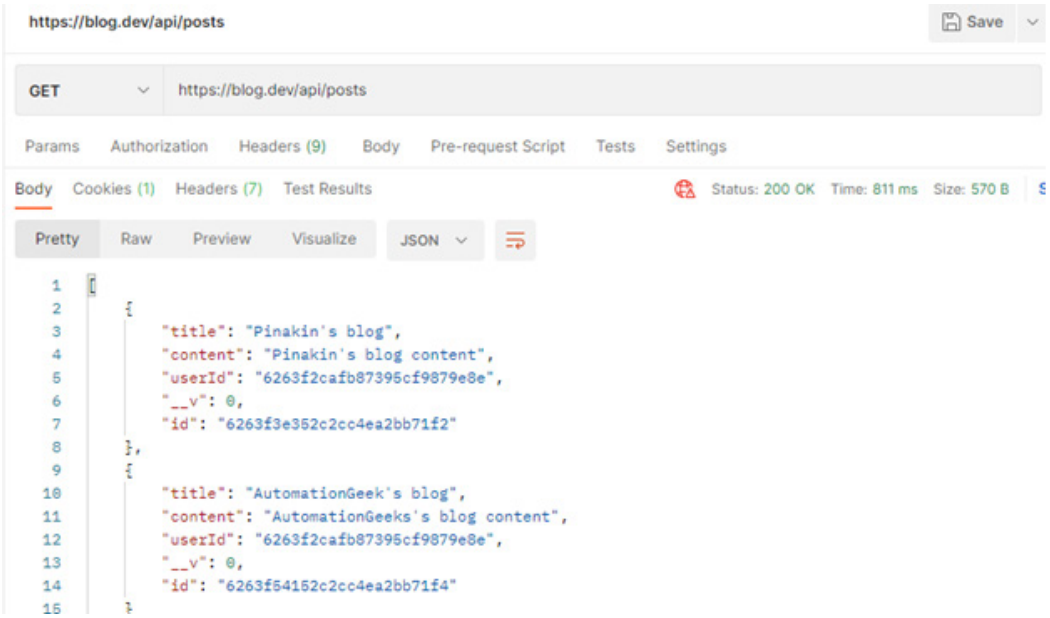

***Figure 7.17**: List all Posts*

5. List the blog by the ID as shown in the following screenshot:

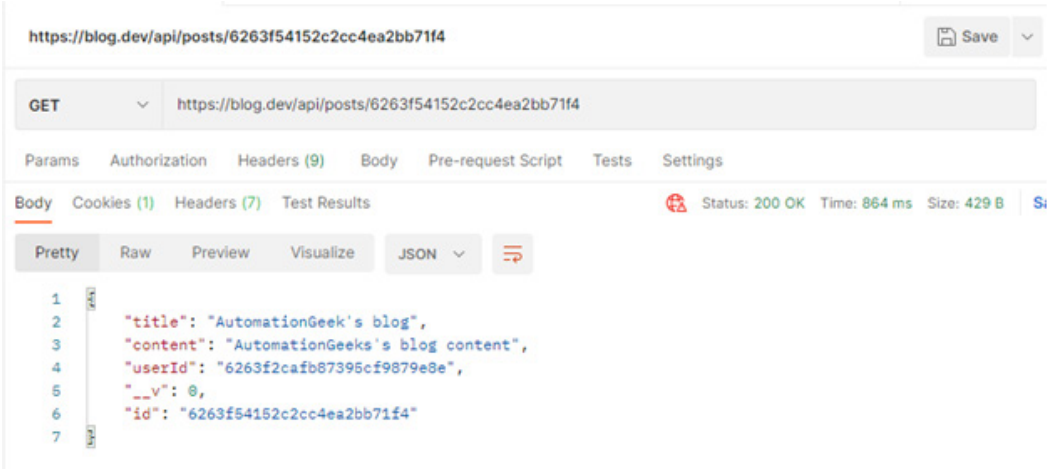

***Figure 7.18**: Fetch a particular Post*

More books: https://t.me/booksforits

6. Update the post by the ID as shown in the following screenshot:

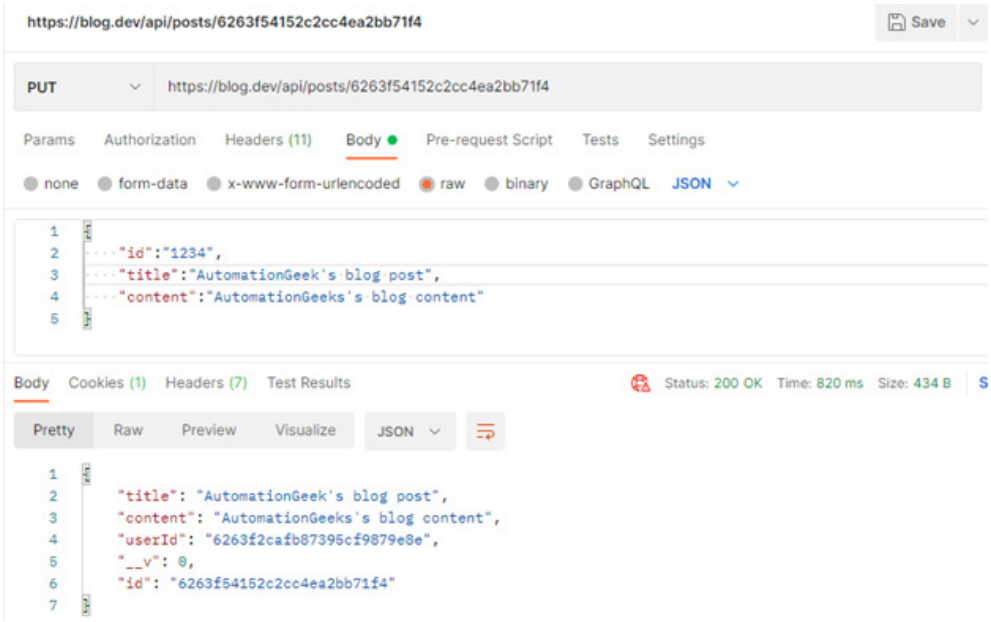

***Figure 7.19**: Updating a post*

7. List all posts again and get the updated blog post as shown in the following screenshot:

***Figure 7.20**: The updated Blogpost*

With this, we come to the end of this chapter.

Conclusion

In this chapter, we looked at the creation of the common module and moved authentication logic pieces to the common module. We developed the Post service to

More books: https://t.me/booksforits

add, update by ID, list all posts, and update posts by ID. We also looked at the NATS Streaming server and developed a sample Publisher and Listener. We incorporated code to integrate the Posts service with NATS and eventually used Postman to add, update, and list posts.

In the next chapter, we will start building the comments service and look at the relationship between the Post and the Comments service.

Questions

1. Explain how to create an NPM Module.
2. Explain the process of changing an NPM module.
3. Explain what is the NATS Streaming server.
4. Explain what is port forwarding.
5. Configure a simple Publisher and Listener using NATS.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Chapter 8
Introducing Automated Testing

Introduction

In the previous chapter, we saw how to build the Post Service and integrate it with the NATS streaming server. The NATS streaming server will be used as a means to stream the various messages that will float around our framework.

Here, we will look at incorporating automated testing in our framework using **`SuperTest`**. **`SuperTest`** is a library that will test all our APIs without bringing up a Kubernetes cluster. The **`SuperTest`** library can be used by itself or with Mocha. In this book, we will focus on **`SuperTest`** with the JEST framework.

Structure

This chapter will predominantly focus on:

- Introducing **`SuperTest`**
- Setting up automated testing
- The **`index.ts`** refactor
- Setup for Auth service
- Designing the tests for the Auth service

More books: https://t.me/booksforits

- The **index.ts** refactor for the POST service
- Executing the tests for the Auth service
- Setup for POST service
- Designing the tests for posts service
- Executing the tests for the Posts Service

Objectives

After reading this chapter, we will learn how to use **SuperTest** for Automated testing of the API code to achieve rapid application development. The **SuperTest** library is such that the setup takes a minimal amount of time. Let us start by learning what **SuperTest** is.

Introducing SuperTest

SuperTest is a Nodejs library that is designed so that developers can test APIs. **SuperTest** extends another library called **Superagent**. **Superagent** is an HTTP client written in JavaScript for the Node.js library and the browser. **SuperTest** can be used as a standalone library or with JavaScript testing frameworks like Jest or Mocha. In this book, we will use **SuperTest** with JEST.

Let us look at how to set up **SuperTest**.

Setting up automated testing

Install dev dependencies as mentioned here:

- **@types/jest**
- **@types/supertest**
- **jest**
- **ts-jest**
- **SuperTest**
- **MongoDB-memory-server**

To set up **SuperTest**, we must follow these steps:

1. Inside the project folder, create a folder called **test**. This folder will contain the setup code for **SuperTest**.
2. Inside the same project folder, in the routes folder, create a folder called **__test__**.

More books: https://t.me/booksforits

3. Please note that the folder must be created with the underscores to make **`SuperTest`** work. All test scripts will be put in this folder.

For reference, please take a look at the following screenshot:

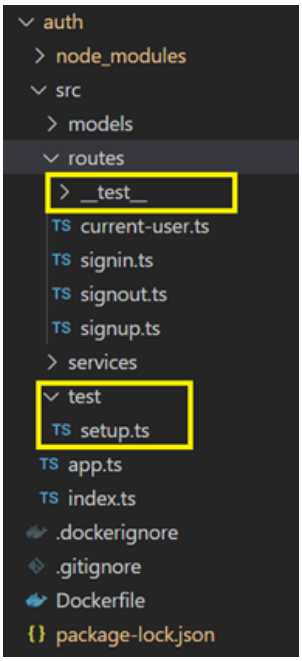

***Figure 8.1**: Looking at the test setup*

The index.ts refactor for Auth service

In order to make **`SuperTest`** work correctly, there are a few changes that need to be made for the **`index.ts`** for both the Auth and POST service. These files are the same as the previous ones, just that the **`index.ts`** has been split into two files. The **`app`** file is imported into the **`index.ts`** file. The modified **`index.ts`** for the auth service is shown here:

```
import mongoose from 'mongoose';
import {auth_app} from './app';
const auth_port = process.env.PORT || 3100;

const startup = async()=>{
    if(!process.env.AUTH_JWT_KEY){
        throw new Error('Jwt key must be defined');
    }
```

More books: https://t.me/booksforits

```
    if(!process.env.AUTH_MONGO_URI){
        throw new Error('MONGO_URI must be defined');
    }
    try{
    await mongoose.connect(process.env.AUTH_MONGO_URI);
    console.log('Connected to Mongo DB');
    } catch(err){
        console.error(err);
    }

    auth_app.listen(auth_port, ()=>{
    console.log('Listening on port 3100',auth_port);
    });

}
startup();
```

Create a new file called **app.ts** and export it. The export is needed so that the **app.ts** can be imported into the **index.ts** file. The **app.ts** file will have the code as follows:

```
import express from 'express';
import 'express-async-errors';
import {json} from 'body-parser';
import cookieSession from 'cookie-session';
import {errorHandler,NotFoundErr,currentUser} from '@pcblog/common';

import {createPostRouter} from './routes/createPost';
import {indexPostRouter} from './routes/index';
import {showPostsRouter} from './routes/showPosts';
import {updatePostRouter} from './routes/update';

const post_app=express();
post_app.set('trust proxy', true);
```

More books: https://t.me/booksforits

```
post_app.use(json());
post_app.use(cookieSession({
    signed: false,
    secure: process.env.NODE_ENV !== 'test'
})
);

post_app.use(currentUser);
post_app.use(createPostRouter);
post_app.use(indexPostRouter);
post_app.use(showPostsRouter);
post_app.use(updatePostRouter);

post_app.all('*',async(req,res)=>{
    throw new NotFoundErr();
});
post_app.use(errorHandler);
export {post_app};
```

After the refactor, we will create another new file called **`setup.ts`** and the individual tests for the auth service.

Setup for Auth service

We will now work on the setup file for the Auth service. Let us split the code and understand each piece. Import request, **`MongoMemoryServer`**, **`mongoose`**, and **`app`** as shown here:

```
import request from 'supertest';
import { MongoMemoryServer } from 'mongodb-memory-server';
import mongoose from 'mongoose';
import { auth_app } from '../app';
```

Declare a global variable that will include a sign-in function that returns a promise:

```
declare global {
    namespace NodeJS {
        interface Global {
```

More books: https://t.me/booksforits

```
            auth_signin(): Promise<string[]>;
        }
    }
};
```

Declare a **beforeAll** function which runs before all tests. This function creates an instance of **MongoMemoryServer**. **MongoMemoryServer** is an in-memory MongoDB database used for automated testing. Get the Mongo URI and connect using the URI:

```
Let auth_mongo: any;
beforeAll(async () => {
    process.env.AUTH_JWT_KEY = 'asdfadfas';

    auth_mongo = new MongoMemoryServer();
    const mongoURI = await auth_mongo.getUri();

    await mongoose.connect(mongoURI);
});
```

Declare a **beforeEach** function which runs before each test where we pull all MongoDB collections and assign them to collections. We then iterate through each and delete the data:

```
beforeEach(async () => {
    const collections = await mongoose.connection.db.collections();

    for (let collection of collections) {
        await collection.deleteMany({});
    }
});
```

Declare an **afterAll** function which runs after all tests where we close the MongoDB collection:

```
afterAll(async () => {
    await auth_mongo.stop();
    await mongoose.connection.close();
});
```

More books: https://t.me/booksforits

The **global.signin** function helps sign into the application using the email and password that is provided here. We make a post request to **/api/users/signup** to sign up for the application by providing the **email** and **password** that is provided. We then expect a return code of **201** since a new user gets created. We finally return a cookie to remember this user:

```
Global.auth_signin = async () => {
    const auth_email = 'test@test.com';
    const auth_password = 'password';

    const auth_response = await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email,auth_password
        })
        .expect(201);

    const auth_cookie = auth_response.get('Set-Cookie');

    return auth_cookie;
};
```

We will next create tests in the **__test__** folder in the **routes** folder.

Designing the tests for the Auth Service

The next step is to design the tests for the signup, sign in, current user, and sign out route handlers.

We then start with the signup route handler test.

Tests for Signup route handler

Create a file called **signup.test.ts** in the **__test__** folder in the **routes** folder. Import **request** and **app** as follows:

```
import request from 'supertest';

import {auth_app} from '../../app';
```

More books: https://t.me/booksforits

Create a test to return **201** on successful signup. Here, we make a post request to **/api/users/signup** and send it a test email and password in the body. We expect the return code to be **201**:

```
It('gives a return code of 201 on successful signup', async ()=> {
    return request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(201);
});
```

Create a negative test to return **400** (**Invalid Request**) when an invalid email is provided. Here, we make a post request to **/api/users/signup** and send it an invalid test email and password in the body. We expect the return code to be **400**:

```
It('gives a return code of 400 with an invalid email', async ()=> {
    return request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'testtestcom',
            auth_password: 'password'
        })
        .expect(400);
});
```

Create a negative test to return **400** (**Invalid Request**) when an invalid password is provided. Here, we make a post request to **/api/users/signup** and send it a test email and an invalid password in the body. We expect the return code to be **400** as shown in the following code snippet:

```
it('gives a return code of 400 with an invalid password', async ()=> {
    return request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
```

More books: https://t.me/booksforits

```
            auth_password: 'pas'
        })
        .expect(400);
});
```

Create a negative test to return **`400`** (**`Invalid Request`**) when a missing email or password is provided. Here, we make a post request to **`/api/users/signup`** and send it a missing password first followed by a missing email in the body. We expect the return code to be **`400`** as follows:

```
it('gives a return code of 400 with missing email and password', async
()=> {
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com'
        })
        .expect(400);
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            password: 'password'
        })
        .expect(400);
});
```

Create a negative test to return **`400`** (**`Invalid Request`**) when we provide a duplicate email for a subsequent signup. Here, we make a post request to **`/api/users/signup`** and send it a valid test email and password in the body. We expect the return code to be **`201`**. We make a subsequent request with the same email and password. We expect **`400`** as the return code in this case:

```
it('duplicate emails should not be allowed', async ()=> {
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
```

More books: https://t.me/booksforits

```
        })
        .expect(201);
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(400);
});
```

Create a positive test to return **201** when a valid email is provided. Here, we make a post request to **/api/users/signup** and send it a valid test email and password in the body. We expect the return code to be **201**. In the end, we expect the cookie to be set:

```
it('cookie should be set after successful signup', async ()=> {
    const auth_response = await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(201);
    expect (auth_response.get('Set-Cookie')).toBeDefined();
});
```

Those were all the tests for the Signup route handler. We now move to the Signin route handler.

Tests for Signin route handler

Create a file called **signin.test.ts** in the **__test__** folder in the **routes** folder.

Import **request** and **app** as follows:

```
import request from 'supertest';
import {auth_app} from '../../app';
```

More books: https://t.me/booksforits

Create a negative test to return **400** when a non-existent email and password are provided. Here, we make a post request to **/api/users/signin** and send it a valid test email and password in the body. Since we have not signed up with that email and password, we expect the return code to be **400** when we try to sign in without signing up first:

```
it('fails when an email that does not exist is supplied', async ()=> {
    await request(auth_app)
        .post('/api/users/signin')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(400);
});
```

Create a negative test to return **400** when an existing email and non-existent password are provided. Here, we make a post request to **/api/users/signup** first and send it a valid test email and password in the body. Thereafter, we make a follow-up post request to **/api/users/signin** with the same user **id** that was used for signing up but with a different password. Since we have not signed up with that password, we expect the return code to be **400** when we try to sign in with an incorrect password:

```
it('fails when a wrong password is supplied', async ()=> {
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(201);
    await request(auth_app)
        .post('/api/users/signin')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password1'
        })
        .expect(400);
});
```

More books: https://t.me/booksforits

Create a positive test to return **201** when an existing email and password are provided for signing in after signup. Here, we make a post request to **/api/users/signup** first and send it a valid test email and password in the body. Thereafter, we make a follow-up post request to **/api/users/signin** with the same user ID and password that was used for signing up. We get **200** as the return code. Finally, we expect the cookie to be defined as:

```
it('sends back a cookie when valid credentials are given', async ()=> {
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(201);
    const auth_response = await request(auth_app)
        .post('/api/users/signin')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(200);
    expect (auth_response.get('Set-Cookie')).toBeDefined
});
```

Next, we move on to the current user route handler.

Tests for current user route handler

Create a file called **currentuser.test.ts** in the **__test__** folder in the **routes** folder.

Import **request** and **app** as shown in the following code snippet:

```
import request from 'supertest';
import {auth_app} from '../../app';
```

More books: https://t.me/booksforits

Create a positive test to return **200** when **signin** is done using the **global.signin()** that we covered earlier. Here, we make a get request to **/api/users/currentuser** by setting the cookie returned by **global.signin()**. We also expect the response to contain the email that was used during the signup process:

```
it('Responds with the details of current user', async ()=> {


    const auth_cookie = await global.auth_signin();


    const auth_response = await request(auth_app)
        .get('/api/users/currentuser')
        .set('Cookie',auth_cookie)
        .send()
        .expect(200);


        expect(auth_response.body.currentUser.auth_email).toEqual('test@
test.com');
});
```

Create a negative test to return null in the response when we try to get the current user. Here, we make a get request to the **/api/users/current** user without setting the cookie returned by **global.signin()**:

```
it('Response should be null if user is not authenticated', async ()=> {


    const auth_response = await request(auth_app)
        .get('/api/users/currentuser')
        .send()
        .expect(200);
    expect(auth_response.body.currentUser).toEqual(null);
});
```

We now move to the last test for the Signout route handler.

Tests for Signout route handler

Create a file called **signout.test.ts** in the **__test__** folder in the **routes** folder.

More books: https://t.me/booksforits

Import **request** and **app** as follows:

```
import request from 'supertest';
import {auth_app} from '../../app';
```

Create a positive test to return the cookie in the response when we try to do a sign out. Here, we make a post request to **/api/users/signup** followed by a post request to **/api/users/signout** and expect the cookie to be returned in the response:

```
it('Cookie should be cleared after signing out', async ()=> {
    await request(auth_app)
        .post('/api/users/signup')
        .send({
            auth_email: 'test@test.com',
            auth_password: 'password'
        })
        .expect(201);
    const auth_response = await request(auth_app)
        .post('/api/users/signout')
        .send({})
        .expect(200);
    expect (auth_response.get('Set-Cookie')).toBeDefined
});
```

This completes the tests for the auth service. Now is the time to execute the test.

Executing the tests for the Auth Service

Navigate to the auth folder and in the **package.json**, put the following entries. We specify the test command and the JEST configuration to specify the setup file, to be picked up to get the testing configuration:

```
 "scripts": {
    "start": "ts-node-dev src/index.ts",
    "test": "jest --watchAll --no-cache"
  },
  "jest": {
    "preset": "ts-jest",
    "testEnvironment": "node",
```

More books: https://t.me/booksforits

```
    "setupFilesAfterEnv": [
      "./src/test/setup.ts"
    ]
```

Open a command prompt and navigate to the **auth** folder. Execute **npm run test**. This will execute the tests and the final output is shown in the following screenshot:

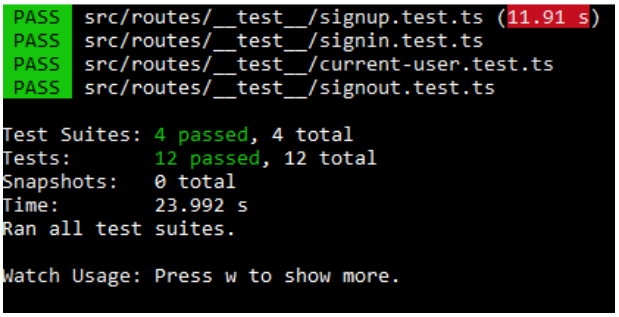

Figure 8.2: *Tests report for Supertest*

Now, we will take a look at the **index.ts** refactor for the posts service.

The index.ts refactor for the POST service

The refactor for the posts service is very much like the auth service. Just that the **index.ts** has been split into two files. The **app** file is imported into the **index.ts** file. The modified **index.ts** for the posts service is shown in the following code:

```
import express from 'express';
import 'express-async-errors';
import {json} from 'body-parser';
import mongoose from 'mongoose';
import {natsWrapper} from './nats-wrapper'
import {post_app} from './app';

const post_startup = async()=>{
    if(!process.env.AUTH_JWT_KEY){
        throw new Error('Jwt key must be defined');
    }

    if(!process.env.POSTS_MONGO_URI){
        throw new Error('MONGO_URI must be defined');
```

More books: https://t.me/booksforits

```
    }
    if(!process.env.POSTS_NATS_CLIENT_ID){
        throw new Error('NATS_CLIENT_ID must be defined');
    }
    if(!process.env.POSTS_NATS_URL){
        throw new Error('NATS_URL must be defined');
    }
    if(!process.env.POSTS_NATS_CLUSTER_ID){
        throw new Error('NATS_CLUSTER_ID must be defined');
    }

    try{
        await natsWrapper.connect(process.env.POSTS_NATS_CLUSTER_
ID,process.env.POSTS_NATS_CLIENT_ID,process.env.POSTS_NATS_URL);
        natsWrapper.client.on('close',()=>{
            console.log('NATS connection closed!');
            process.exit();
        });

        process.on('SIGINT',()=>natsWrapper.client.close());
        process.on('SIGTERM',()=>natsWrapper.client.close());

    await mongoose.connect(process.env.POSTS_MONGO_URI);
    console.log('Connected to Mongo DB');
    } catch(err){
        console.error(err);
    }
    post_app.listen(3100, ()=>{
    console.log('Listening on port 3100');
    });

}
post_startup();
```

More books: https://t.me/booksforits

Now, we look at the **app.ts** file. Create a new file called **app.ts** and export it. The export is needed so that the **app.ts** can be imported into the **index.ts** file. The **app.ts** file will have the code mentioned:

```
import express from 'express';
import 'express-async-errors';
import {json} from 'body-parser';
import cookieSession from 'cookie-session';
import {errorHandler,NotFoundErr,currentUser} from '@pcblog/common';

import {createPostRouter} from './routes/createPost';
import {indexPostRouter} from './routes/index';
import {showPostsRouter} from './routes/showPosts';
import {updatePostRouter} from './routes/update';

const post_app=express();
post_app.set('trust proxy', true);

post_app.use(json());
post_app.use(cookieSession({
    signed: false,
    secure: process.env.NODE_ENV !== 'test'
})
);

post_app.use(currentUser);
post_app.use(createPostRouter);
post_app.use(indexPostRouter);
post_app.use(showPostsRouter);
post_app.use(updatePostRouter);

post_app.all('*',async(req,res)=>{
    throw new NotFoundErr();
});
post_app.use(errorHandler);
export {post_app};
```

More books: https://t.me/booksforits

After the refactor, we will create a new file called **setup.ts** and the individual tests for the posts service.

Setup for the POST service

We will now work on the setup file for the posts service. Let us split the code and understand each piece.

Import **request**, **MongoMemoryServer**, **mongoose**, and **app** as shown in the following code:

```
import request from 'supertest';
import { MongoMemoryServer } from 'mongodb-memory-server';
import mongoose from 'mongoose';
import { post_app } from '../app';
import jwt from 'jsonwebtoken';
Declare a global variable that will include a sign in function that
returns a promise:
declare global {
    namespace NodeJS {
        interface Global {
            signintoapp(): Promise<string[]>;
        }
    }
};
```

Declare a **beforeAll** function which runs before all tests. This function creates an instance of **MongoMemoryServer**. **MongoMemoryServer** is an in-memory MongoDB database used for automated testing. Get the Mongo URI and connect using the URI:

```
let post_mongo: any;
beforeAll(async () => {
    process.env.JWT_KEY = 'asdfadfas';

    post_mongo = new MongoMemoryServer();
    const post_mongoURI = await post_mongo.getUri();

    await mongoose.connect(post_mongoURI);
});
```

More books: https://t.me/booksforits

Declare a **`beforeEach`** function which runs before each test where we pull all MongoDB collections and assign them to collections. We then iterate through each and delete the data:

```
beforeEach(async () => {
    const post_collections = await mongoose.connection.db.collections();

    for (let collection of post_collections) {
        await collection.deleteMany({});
    }
});
```

Declare an **`afterAll`** function which runs after all tests where we close the MongoDB collection:

```
afterAll(async () => {
    await post_mongo.stop();
    await mongoose.connection.close();
});
```

The **`global.signin`** function helps to sign in to the application using the email and password that is provided here. The explanations are in the form of comments as follows:

```
global.signintoapp = () => {

//Build a JWT payload. {id,email}
const payload = {
    id: new mongoose.Types.ObjectId().toHexString(),
    auth_email: 'pinakinc@yahoo.com'
};

//Create the JWT
const token = jwt.sign(payload,process.env.JWT_KEY!);

//Build a session object {jwt: MY_JWT}
const session = {jwt: token};

//Turn that session into JSON
```

More books: https://t.me/booksforits

```
const sessionJSON = JSON.stringify(session);

//Take JSON and encode it as base-64
const base64 = Buffer.from(sessionJSON).toString('base64');

//Return a string that's a cookie with the encoded data
return [`express:sess=${base64}`];
};
```

We will next create tests in the **__test__** folder placed in the **routes** folder.

Designing the tests for the Posts Service

The next step is to design the tests for the **createPost**, **index**, **showPosts**, and update route handlers. We start with the **createPost** route handler test.

Tests for createPost route handler

Create a file called **createpost.test.ts** in the **__test__** folder in the **routes** folder.

Import **request**, **app**, and **Post** model as shown here:

```
import request from 'supertest';
import {post_app} from '../../app';
import {Post} from '../../models/posts';
```

Create a negative test to return **404** when a body is not sent with a post request. Here, we make a post request to **/api/posts** and send it a blank body. We expect the return code to be **404**:

```
it('has a valid route handler listening to /api/posts for post
requests', async () => {
    const response = await request(post_app)
                        .post('/api/posts')
                        .send({});
    expect (response.status).not.toEqual(404);
});
```

Create a negative test to return **401** when the user tries to create a post without signing in. Here, we make a post request to **/api/posts** and send it a blank body. We expect the return code to be **401**:

More books: https://t.me/booksforits

```
it('can be accessed only if the user is signed in', async () => {
    const response = await request(post_app)
                        .post('/api/posts')
                        .send({})
                        .expect(401);
});
```

Create a test to return a status other than **401** (**Invalid Request**) if the user is signed in. Here, we make a post request to **/api/posts** and send in a cookie generated by **global.signin()**. We expect the return code to be different from **401**:

```
it('if the user is signed in, returns a status other than 401', async ()
=> {
    const auth_response = await request(post_app)
                        .post('/api/posts')
                        .set('Cookie',global.signintoapp())
                        .send({
                            title: 'title',
                            content: 'content'
                        });
    expect(auth_response.status).not.toEqual(401);
});
```

Create a negative test to return **400** (**Invalid Request**) when a blank or missing title is provided. Here we make a post request to **/api/posts** and send it a blank title first followed by a missing title in the body. We expect the return code to be **400** in both cases:

```
it('if an invalid title is provided it returns an error', async () => {
    await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: '',
            content: 'some'
        })
        .expect(400);
```

More books: https://t.me/booksforits

```
    await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            content: 'some'
        })
        .expect(400);
});
```

Create a negative test to return **400** (**Invalid Request**) when we provide blank or missing content:

```
it('if an empty content is provided returns an error', async () => {
    await request(post_app)
    .post('/api/posts')
    .set('Cookie',global.signintoapp())
    .send({
        title: 'my title',
        content: ''
    })
    .expect(400);

await request(post_app)
    .post('/api/posts')
    .set('Cookie',global.signintoapp())
    .send({
        title: 'my title'
    })
    .expect(400);

});
```

Create a positive test to return **201** when valid inputs are provided. Here, we make a post request to **/API/posts** and send it a valid title and content in the body. We expect the return code to be **201**. In the end, we expect to find one post in the MongoDB database:

More books: https://t.me/booksforits

```
it('when valid inputs are provided, create a post', async () => {
    let posts = await Post.find({});
    expect (posts.length).toEqual(0);
    await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: 'my title',
            content: 'my content'
        })
        .expect(201);

    posts = await Post.find({});
    expect (posts.length).toEqual(1);

});
```

These all were the tests for the **`createPost`** route handler. We will now move to the **`updatePost`** route handler.

Tests for the updatePost route handler

Create a file called **`updatepost.test.ts`** in the **`__test__`** folder in the **`routes`** folder.

Import **`request`** and **`app`** as follows:

```
import request from 'supertest';
import {post_app} from '../../app';
import mongoose from 'mongoose';
```

Create a negative test to return **`404`** when a non-existent post ID is provided. Here, we make a post request to **`/api/posts`** after giving a randomly generated post ID that does not exist:

```
it('if the provided id does not exist returns a 404 ', async () => {
    const id = new mongoose.Types.ObjectId().toHexString();
    const response = await request(post_app)
        .put(`/api/posts/${id}`)
```

More books: https://t.me/booksforits

```
        .set('Cookie',global.signintoapp())
        .send({
            title:'My_Title',
            content: 'My_content'
        })
        .expect(404);


});
```

Create a negative test to return **401** when the user is not authenticated. Here, we make a post request to **/api/posts** without invoking **global.signintoapp()**:

```
it('if the user is not authenticated, return a 401', async () => {
    const id = new mongoose.Types.ObjectId().toHexString();
    const response = await request(post_app)
        .put(`/api/posts/${id}`)
        .send({
            title:'My Title',
            content: 'My content'
        })
        .expect(401);
});
```

Create a negative test to generate **401** if the user that updates the post is different from the one who created it:

```
it('if the user does not own the post, return a 401', async () => {
    const response = await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: 'My Title',
            content: 'My Content'
        });
        await request(post_app)
        .put(`/api/posts/${response.body.id}`)
        .set('Cookie',global.signintoapp())
```

More books: https://t.me/booksforits

```
        .send({
            title: 'My new Title',
            content: 'My new Content'
        })
        .expect(401);
});
```

Create a negative test to generate 400 if the user provides an invalid title or content:

```
it('if the user provides an invalid title or content, return a 400',
async () => {
    const cookie = global.signintoapp();
    const response = await request(post_app)
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content'
    });

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: '',
        content: 'My Content'
    })
    .expect(400);

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'My title',
        content: ''
```

More books: https://t.me/booksforits

```
    })
    .expect(400);


});
```

Create a positive test to return 200 when valid inputs are provided. Here, we also validate values coming from the response:

```
it('updates the post when valid inputs are provided', async () => {
    const cookie = global.signintoapp();
    const response = await request(post_app)
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content'
    });

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'New Title',
        content: 'New Content'
    })
    .expect(200);

    const ticketResponse = await request(post_app)
                           .get(`/api/posts/${response.body.id}`)
                           .send();
    expect(ticketResponse.body.title).toEqual('New Title');
    expect(ticketResponse.body.content).toEqual('New Content');
});
```

Next, we move on to the tests for the index route handler.

More books: https://t.me/booksforits

Tests for the indexPost route handler

Create a file called **indexpost.test.ts** in the **__test__** folder in the **routes** folder.

Import **request** and **app** as follows:

```
import request from 'supertest';
import {post_app} from '../../app';
Create a sample post as follows:
const createPost = () => {
    return request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: 'My Title',
            content: 'My Content'
        });

};
```

Create a positive test to return the number of posts created. Here, we created 3 posts before pulling the list of posts:

```
it('fetchs a list of posts', async () => {
    await createPost();
    await createPost();
    await createPost();
    const auth_response = await request(post_app)
                        .get('/api/posts')
                        .send()
                        .expect(200);
    expect(auth_response.body.length).toEqual(3);
});
```

We now move to the last test for the **showPosts** route handler.

More books: https://t.me/booksforits

Tests for the showPosts route handler

Create a file called **showpost.test.ts** in the **__test__** folder in the **routes** folder.

Import **request**, **app**, and **mongoose** as follows:

```
import request from 'supertest';
import {post_app} from '../../app';
import mongoose from 'mongoose';
```

Create a negative test to return **404** if the post is not found. Here, we make a get request to **/api/posts/[randomly generated id]**. Since the post ID is randomly generated, the post is not found:

```
it('if the post is not found, return 404', async () => {
    const id = new mongoose.Types.ObjectId().toHexString;
    const response = await request(post_app)
        .get(`/api/posts/${id}`)
        .send();

});
```

Create a positive test to return the post if the post is found. Here, we make a post request to **/api/posts/** to generate the post and save the response. Later, the post ID is picked from the response and the corresponding post is pulled. The values are inspected to be present in the response:

```
it('returns the post if found', async () => {
    const title = 'My blog';
    const content = 'some content';
    const response = await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title,content
        })
        .expect(201);
    const postResponse = await request(post_app)
                            .get(`/api/posts/${response.body.id}`)
```

More books: https://t.me/booksforits

```
                                .send()
                                .expect(200);
    expect(postResponse.body.title).toEqual(title);
    expect(postResponse.body.content).toEqual(content);

});
```

This completes the tests for the posts service. Now, it is the time to execute the test.

Executing the tests for the Posts Service

Navigate to the **`posts`** folder and in the **`package.json`**, put the following entries. We specify the test command and the jest configuration to specify the setup file to be picked up to get the testing configuration:

```
 "scripts": {
        "start": "ts-node-dev src/index.ts",
        "test": "jest --watchAll --no-cache"
    },
    "jest": {
        "preset": "ts-jest",
        "testEnvironment": "node",
        "setupFilesAfterEnv": [
            "./src/test/setup.ts"
        ]
    }
```

Open a command prompt and navigate to the **`posts`** folder. Execute **`npm run test`**. This will execute the tests and the final output is shown in the following screenshot:

```
PASS  src/routes/__test__/updatePost.test.ts (26.534 s)
PASS  src/routes/__test__/createpost.test.ts (10.789 s)
PASS  src/routes/__test__/showpost.test.ts
PASS  src/routes/__test__/indexpost.test.ts

Test Suites: 4 passed, 4 total
Tests:       14 passed, 14 total
Snapshots:   0 total
Time:        43.046 s
Ran all test suites.
```

Figure 8.3: *Tests report for Supertest*

More books: https://t.me/booksforits

This completes the chapter on automated testing. We incorporated automated testing for the posts and auth services.

Conclusion

In this chapter, we looked at creating automated tests for the auth and posts services. It was a straightforward process to set up SuperTest with JEST. The automated tests speed up development by reducing the time that is spent in bringing up a Kubernetes cluster.

In the next chapter, we will look at incorporating the Comments service to add comments to a post.

Questions

1. Explain what an automated test is.
2. Explain how to set up a simple test suite using **`Supertest`** and **`Jest`**.
3. Explain the **`global.signin()`** function.
4. Explain the process of cookie generation.
5. What is a MongoDB Memory server?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Chapter 9
Integrating the Comments Service

Introduction

In the previous chapter, we saw how to write automated tests for the posts service using SuperTest and JEST. By automating our tests, we could test the functionality of the posts service without bringing up a Kubernetes cluster. Now, we have the posts service completed.

In this chapter, we will look at the concepts behind building and integrating the comments service. Concepts such as sub-documents and references about MongoDB will be introduced. We will see the pros and cons of using sub-documents and learn the advantages of references.

Structure

This chapter will predominantly focus on:

- Comments service
- Nesting comments inside posts
    - Pros and cons of nesting
- What are sub-documents?

More books: https://t.me/booksforits

    - Pros and cons of sub-documents
- What are references?
    - Advantages of references

Objectives

After reading this chapter, we will understand the concepts of integrating two services together, and what sub-documents and references are. We will also understand the pros and cons of using sub-documents and the advantages of references.

Comments service

The comments service will consist of either fetching all comments for a particular post or a particular comment for a post ID. The comments service will be a child of the posts service. We will see the approaches of using sub-documents and references and compare the differences between the two.

The comments schema will have a comment ID which will be auto-generated. In addition, it will have the actual comment, the user who added the comment, and the date on which the comment was added.

A point to note is that the comments service will have a schema but no model. The model created will be the posts model.

Let us see the various approaches to integrating comments and posts.

Nesting comments inside posts

The first approach to integrating posts and comments is directly embedding the entire comments schema inside the posts schema as follows:

```
{
"_id":ObjectId("53402597d852426020000002"),
"title": "My Blog",
"content": "Blog content",
"comments": [{
"_id": ObjectId("534009e4d852427820000002"),
"comment_text":"First Comment",
"createdDT": 18-OCT-2022,
"createdBy": "XYZ"
```

More books: https://t.me/booksforits

```
}]
}
```

Pros and cons of nesting

We will look at the pros and cons of nesting in this section.

Pro

The pros of nesting are as follows:

- Query construction is simple since it is a single schema.

Cons

The cons of nesting are as follows:

- If the columns in the nested documents increase, it can hamper performance.

The disadvantage of using this option is that if the nested schema is huge, we might face performance issues.

Let us look at another method of integrating comments with posts which is called **sub-documents**. We will first understand what sub-documents are.

What are sub-documents?

We can have the posts and the comments schemas as separate entities. The posts schema will look like the one as follows:

post schema

```
{
"_id":ObjectId("53402597d852426020000002"),
"title": "My Blog",
"content": "Blog content",
"comments": [comment]
}
```

comment schema

Here, we see that the comments are an array of the comments schema as follows:

```
"comment": {
"_id": ObjectId("534009e4d852427820000002"),
```

More books: https://t.me/booksforits

```
"comment_text":"First Comment",
"createdDT": 18-OCT-2022,
"createdBy": "XYZ"
}
```

Pros and cons of sub-documents

We will look at the pros and cons of sub-documents in this section.

Pros

The pros of sub-documents are as follows:

- Query construction is simple

Cons

The cons of sub-documents are as follows:

- It can hamper performance if the size of the child document is large.

Let us look at what references are.

What are references?

References are simply pointers referring to child schemas. The child schemas are referenced using the MongoDB object ID as follows:

post schema

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "title": "My Blog",
   "content": "This is my blog",
   "comments": [{
       "$ref": "comment",
       "$id": ObjectId("534009e4d852427820000002")
   }]
}
```

More books: https://t.me/booksforits

comment schema

```
{
"_id": ObjectId("534009e4d852427820000002"),
"comment_text":"First Comment",
"createdDT": 18-OCT-2022,
"createdBy": XYZ
}
```

Advantages of references

The advantages of references are as follows:

- Performance-wise, references are recommended
- Queries can be effectively constructed using populate
- Easier for debugging

This completes the chapter on integrating the comments service. We got an idea of effectively integrating the comments schema with the posts schema.

Conclusion

In this chapter, we looked at three different approaches for integrating the comments service with the posts service. We saw the pros and cons of nesting and sub-documents. We eventually looked at the advantages of references in MongoDB.

In the next chapter, we will look at the actual implementation of the integration of posts and comments service.

Questions

1. Explain the three approaches for integrating two schemas.
2. Explain how to design a schema with the nesting approach.
3. Explain how to design a schema with the sub-document approach.
4. Explain how to design a schema with the reference approach.
5. Explain the differences between nesting, sub-document, and reference approaches.

More books: https://t.me/booksforits

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Chapter 10
Creating the Comments Service

In the previous chapter, we saw how to use MongoDB references to incorporate the comments service.

In this chapter, we will look at the creation of the comments service and see how we can use SuperTest to run automated tests to test our comments service.

Structure

This chapter will predominantly focus on:

- Comments service
- Comments and comments mongo yamls
- Comments model
- Duplicating the comments model inside the post
- Referencing the comments model inside the post model
- Updating the routes
- Updating the tests
- Executing the tests

More books: https://t.me/booksforits

Objectives

After reading this chapter, the reader will understand how to incorporate the comments service into the framework. We will integrate the comments service using references in MongoDB.

Comments service

The comments service will be referenced from the posts service using MongoDB references. The comments schema will have **`content`** and **`createdDt`** as its columns:

Comments and comments Mongo YAMLs

In the **`infra/k8s`** folder, create two files very similar to the posts service by the names **`comments-depl.yaml`** and **`comments-mongo-depl.yaml`**.

The contents of **`comments-depl.yaml`** are as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: comments-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: comments
  template:
    metadata:
      labels:
        app: comments
    spec:
      containers:
        - name: comments
          image: pinakinc/comments:latest
          env:
            - name: POSTS_NATS_CLIENT_ID
              valueFrom:
```

More books: https://t.me/booksforits

```
              fieldRef:
                fieldPath: metadata.name
            - name: POSTS_NATS_URL
              value: 'http://nats-srv:4222'
            - name: POSTS_NATS_CLUSTER_ID
              value: 'blog'
            - name: POSTS_MONGO_URI
              value: 'mongodb://comments-mongo-srv:27017/comments'
            - name: AUTH_JWT_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: AUTH_JWT_KEY
---
apiVersion: v1
kind: Service
metadata:
  name: comments-srv
spec:
  selector:
    app: comments
  ports:
    - name: comments
      protocol: TCP
      port: 3100
      targetPort: 3100
```

The contents of **`comments-mongo-depl.yaml`** are as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: comments-mongo-depl
spec:
```

More books: https://t.me/booksforits

```
  replicas: 1
  selector:
    matchLabels:
      app: comments-mongo
  template:
    metadata:
      labels:
        app: comments-mongo
    spec:
      containers:
        - name: comments-mongo
          image: mongo
---
apiVersion: v1
kind: Service
metadata:
  name: comments-mongo-srv
spec:
  selector:
    app: comments-mongo
  ports:
    - name: db
      protocol: TCP
      port: 27017
      targetPort: 27017
```

We will next look at changes to the ingress yaml.

Changes to ingress yaml

We will have similar changes to comments as the posts service:

```
          - path: /api/comments/?(.*)
            pathType: Prefix
            backend:
```

More books: https://t.me/booksforits

```
      service:
        name: comments-srv
        port:
          number: 3100
```

Make sure to add this before the catch all block as follows:

```
    - path: /?(.*)
      pathType: Prefix
      backend:
        service:
          name: client-srv
          port:
            number: 80
```

To make the comments service work with the posts service, we duplicate the comments service in the posts service.

We will see how to do this next.

Duplicating the comments model inside the post

To make the posts model work, we need to duplicate the comments model inside the folder corresponding to the posts service. The duplicated service is shown in the following screenshot:

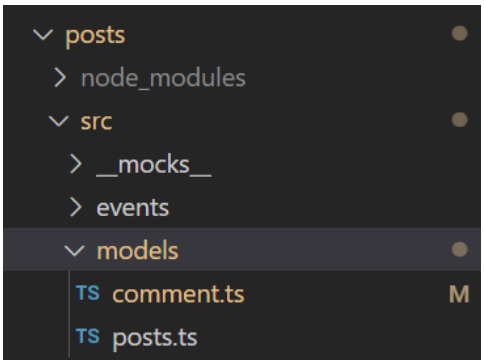

***Figure 10.1**: Folder structure of the Posts service*

More books: https://t.me/booksforits

Referencing the comments model inside the post model

We will make use of MongoDB references to use the comments model inside the posts model. The comments model is given as follows:

```
import mongoose, { Mongoose } from 'mongoose';

interface CommentAttribs {
    content: string;
    createdDt: Date;
}

export interface CommentDoc extends mongoose.Document {
    content: string;
    createdDt: Date;
 }

interface CommentModel extends mongoose.Model<CommentDoc> {
    build(attribs:CommentAttribs): CommentDoc

}

const commentSchema = new mongoose.Schema ({
    content: {
        type: String,
        required: true

    },

    createdDt: {
        type: Date,
        required: true

    },

}, {
    toJSON:{
```

More books: https://t.me/booksforits

```
        transform(doc, ret){
            ret.id = ret._id;
            delete ret._id;
        }
    }
});

commentSchema.statics.build = (attribs: CommentAttribs)=>{
    return new Comment(attribs);
};

const Comment = mongoose.
model<CommentDoc,CommentModel>('Comment',commentSchema);

export {Comment};
```

As seen in the preceding code, the comments model has the field **content** and **createdDt** `columns` which are mandatory. The comments model is a normal model without any references. We will be referencing this model inside the posts model.

We will next see the posts model. There are **title**, **content**, and **comments** fields out of which title and **content** are strings, whereas comments refer to **CommentDoc** which is a reference to the **comments** model. Comments is an array of objects. The configuration object has a type of **mongoose.Schema.Types.objectId** and the ref has a value of **Comment** which refers to the **comments** model:

```
        type:mongoose.Schema.Types.ObjectId,
        ref: 'Comment',

import mongoose, { Mongoose } from 'mongoose';
import {CommentDoc} from './comment'

interface PostAttribs {
    title: string;
    content: string;
    userId: string;
    comments: CommentDoc;

}
```

More books: https://t.me/booksforits

```
interface PostDoc extends mongoose.Document {
    title: string;
    content: string;
    userId: string;
    comments: CommentDoc;
 }

interface PostModel extends mongoose.Model<PostDoc> {
    build(attribs:PostAttribs): PostDoc

}

const postSchema = new mongoose.Schema ({
    title: {
        type: String,
        required: true

    },

    content: {
        type: String,
        required: true

    },
    userId: {
        type: String,
        required: true

    },
    comments: [{
        type:mongoose.Schema.Types.ObjectId,
        ref: 'Comment',
        required: false
    }]

}, {
```

More books: https://t.me/booksforits

```
    toJSON:{
        transform(doc, ret){
            ret.id = ret._id;
            delete ret._id;
        }
    }
});

postSchema.statics.build = (attribs: PostAttribs)=>{
    return new Post(attribs);
};

const Post = mongoose.model<PostDoc,PostModel>('Post',postSchema);

export {Post};
```

We will now update the routes for the posts service.

Editing the routes

We have two routes that must be updated. We will start with the route to create posts.

The createPost route

We pull the comment from the request and then create an object of comment. The built object is given as a reference to the comments field. The comment is saved followed by the post.

Here is the entire **`createPost`** route:

```
import express from 'express';
import {Request, Response} from "express";
import {body} from 'express-validator';
import {requireAuth, validateRequest, NotFoundErr} from '@pcblog/
common';
import {Post} from '../models/posts';
import {Comment} from '../models/comment';
import {PostCreatedPublisher} from '../events/publishers/post-created-
publisher';
```

More books: https://t.me/booksforits

```
import {natsWrapper} from '../nats-wrapper';

const createpost_router = express.Router();

createpost_router.post('/api/posts',requireAuth,[
    body('title').not().isEmpty().withMessage('Title is required'),
    body('content').not().isEmpty().withMessage('Content is required')

],validateRequest,async (req : Request,res : Response) =>
    {
        const {title,content,comments}=req.body;

        const commentB = Comment.build({
            content: req.body.comments[0].content,
            createdDt: req.body.comments[0].createdDt
        });

        const post = Post.build({
            title,
            content,
            userId: req.currentUser!.id,
            comments: commentB
        });
        await commentB.save(function(err,data){
            console.log('Saved data',data)
        });
        await post.save(function(err,data){
            console.log('Saved data',data)
        });
        new PostCreatedPublisher(natsWrapper.client).publish({
            id: post.id,
            title: post.title,
            content: post.content,
            userId: post.userId
        });
```

More books: https://t.me/booksforits

```
        res.status(201).send(post);

    });

export {createpost_router as createPostRouter};
```

The updatePost route

Here is the **`updatePost`** route. The route for the update post has the: ID passed as a parameter in the URL as **`/api/posts/:id`**. We extract the post that was saved in the database. We then build the comment using the comment passed in the request body and give a reference of the comment to the post. The comment is saved followed by the post.

The code for the **`updatePost`** route is given as follows:

```
import express, { Request, Response } from 'express';
import {body} from 'express-validator';
import {
    validateRequest,
    NotFoundErr,
    requireAuth,
    NotAuthErr
} from '@pcblog/common';
import {Post} from '../models/posts';
import {Comment} from '../models/comment';
import {PostUpdatedPublisher} from '../events/publishers/post-updated-
publisher';
import {natsWrapper} from '../nats-wrapper';

const updatepost_router = express.Router();
updatepost_router.put('/api/posts/:id',requireAuth,
[
    body('title')
    .not()
    .isEmpty()
    .withMessage('Title is required'),
    body('content')
```

More books: https://t.me/booksforits

```
    .not()
    .isEmpty()
    .withMessage('Content is required')
],
validateRequest,
async (req: Request,res: Response)=>{
    const post_entry = await Post.findById(req.params.id);
    if (!post_entry) {
        throw new NotFoundErr();
    }
    if (post_entry.userId !== req.currentUser!.id) {
        throw new NotAuthErr();
    }
    const commentB = Comment.build({
        content: req.body.comments[0].content,
        createdDt: req.body.comments[0].createdDt
    });
    post_entry.set({
        title: req.body.title,
        content: req.body.content,
        comments: commentB
            });

    await commentB.save();
    await post_entry.save();
    new PostUpdatedPublisher(natsWrapper.client).publish({
        id:post_entry.id,
        title:post_entry.title,
        content: post_entry.content,
        userId: post_entry.userId

    });
    res.send(post_entry);
});
export {updatepost_router as updatePostRouter};
```

More books: https://t.me/booksforits

We will update the tests next.

Updates to the tests

We will now look at the changes to the tests. We will update the **`createPost`** test followed by the **`updatePost`** test. Let's start with the **`createPost`** test.

Changes to the createPost test

In the **`createPost`** test, wherever we have a configuration object for the post, we should add a comments configuration object. Eventually, we will publish it to NATS.

The changes in the code for the **`createPost`** test are shown as follows:

```
it('if the user is signed in, returns a status other than 401', async ()
=> {
    const auth_response = await request(post_app)
                        .post('/api/posts')
                        .set('Cookie',global.signintoapp())
                        .send({
                            title: 'title',
                            content: 'content',
                            comments:
[{content:'hi',createdDt:20230101}]
                        });
    expect(auth_response.status).not.toEqual(401);
});

it('when valid inputs are provided, create a post', async () => {
        const post_response=await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: 'my title',
            content: 'my content',
            comments: [
                {
                    content:'hi',
                    createdDt:20230101
```

More books: https://t.me/booksforits

```
                }
            ]
        })
        .expect(201);

    let posts = await Post.find({});
    expect (posts.length).toEqual(1);
    let comments = await Comment.find({});
    const extracted_Post = await
         Post.findById(post_response.body.id).populate('comments');
});

it('publishes an event',async ()=>{
    await request(post_app)
    .post('/api/posts')
    .set('Cookie',global.signintoapp())
    .send({
        title: 'my title',
        content: 'my content',
        comments: [
            {
                content:'publish',
                createdDt:20233101
            }
        ]

    })
    .expect(201);
    expect(natsWrapper.client.publish).toHaveBeenCalled();
});
```

Changes to the updatePost test

The code for the **`updatePost`** test follows the same changes as have been implemented for the **`createPost`** test. Whenever we would want to update a post, we first extract the Post and the comment from the database and populate the comment in the post.

More books: https://t.me/booksforits

The post is fetched by giving the id as a request param, the changed configuration object for the post and the comments are specified and the post is updated.

The **`updatePost`** test is shown in the following code which tests for a non-existent id:

```
it('if the provided id does not exist returns a 404 ', async () => {
    const id = new mongoose.Types.ObjectId().toHexString();
    const response = await request(post_app)
        .put(`/api/posts/${id}`)
        .set('Cookie',global.signintoapp())
        .send({
            title:'My_Title',
            content: 'My_content',
            comments: []
        })

        .expect(404);

});
```

The following code tests for a user who is not authenticated:

```
it('returns a 401 if the user is not authenticated', async () => {
    const id = new mongoose.Types.ObjectId().toHexString();
    const response = await request(post_app)
        .put(`/api/posts/${id}`)
        .send({
            title:'My Title',
            content: 'My content',
            comments: []
        })
        .expect(401);
});
```

The following code tests whether a return code of **`401`** is thrown when a user does not own a particular post:

More books: https://t.me/booksforits

```
it('returns a 401 if the user does not own the post', async () => {
    const response = await request(post_app)
        .post('/api/posts')
        .set('Cookie',global.signintoapp())
        .send({
            title: 'My Title',
            content: 'My Content',
            comments: [            {
                content:'hi',
                createdDt:20230101
            }
]
        });
        await request(post_app)
        .put(`/api/posts/${response.body.id}`)
        .set('Cookie',global.signintoapp())
        .send({
            title: 'My new Title',
            content: 'My new Content',
            comments: [            {
                content:'hi',
                createdDt:20230101
            }
]
        })
        .expect(401);
});
```

The following code gives a return code of **400** if the user provides an invalid title or content:

```
it('returns a 400 if the user provides an invalid title or content',
async () => {
    const cookie = global.signintoapp();
    const response = await request(post_app)
```

More books: https://t.me/booksforits

```
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content',
        comments: []
    });

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: '',
        content: 'My Content',
        comments: []
    })
    .expect(400);

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'My title',
        content: '',
        comments: []
    })
    .expect(400);

});
```

The following code tests for a successful updation when valid inputs are provided:

```
it('updates the post when valid inputs are provided', async () => {
    const cookie = global.signintoapp();
    const response = await request(post_app)
```

More books: https://t.me/booksforits

```
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content',
        comments: [
            {
            content: 'content',
            createdDt: 20230131
            }
        ]
    });

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'New Title',
        content: 'New Content',
        comments: [    {
            content: 'content',
            createdDt: 20230131
            }]
    })
    .expect(200);

    const postResponse = await request(post_app)
                            .get(`/api/posts/${response.body.id}`)
                            .send();
});
```

The following code tests for a successful update of post by adding a comment:

```
it('updates the post by adding a comment', async () => {
    const cookie = global.signintoapp();
```

More books: https://t.me/booksforits

```
    const response = await request(post_app)
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content',
        comments: [{}]
    });
    const tempId = response.body.id
    const response1=await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content',
        comments: [{
            content: 'Nice Post',
            createdDt: 20230131
        }]
    })
    .expect(200);
    const tempId1 = response.body.id
    const postResponse = await request(post_app)
                            .get(`/api/posts/${response1.body.id}`)
                            .send();
    const populated_post = await Post.findById(tempId1).
populate('comments');


});
```

The following code tests for a successful event publish:

```
it('publishes an event', async () => {
    const cookie = global.signintoapp();
    const response = await request(post_app)
```

More books: https://t.me/booksforits

```
    .post('/api/posts')
    .set('Cookie',cookie)
    .send({
        title: 'My Title',
        content: 'My Content',
        comments: [{
            content: 'Nice Post',
            createdDt: 20230131
        }]
    });

    await request(post_app)
    .put(`/api/posts/${response.body.id}`)
    .set('Cookie',cookie)
    .send({
        title: 'New Title',
        content: 'New Content',
        comments: [{
            content: 'Nice Post yes',
            createdDt: 20230131
        }]
    })
    .expect(200);
    expect(natsWrapper.client.publish).toHaveBeenCalled();
})
```

We now move to the last part, that is, the test execution

Executing the tests

We will look at executing the entire suite of tests for the posts service. Navigate to the **`posts`** folder and issue the command as follows:

```
npm test
```

The following output shows that all the tests have passed:

More books: https://t.me/booksforits

```
Test Suites: 4 passed, 4 total
Tests:       17 passed, 17 total
Snapshots:   0 total
Time:        24.769 s
Ran all test suites.
```

***Figure 10.2**: Test execution results*

This concludes the chapter on integrating the comments service.

Conclusion

In this chapter, we looked at integrating the comments service with the MEAN framework and we understood how to implement references in MongoDB.

In the next chapter, we will look at the actual implementation of the integration of posts and comments service.

Questions

1. Explain what ref is in terms of MongoDB.
2. Design a sample framework for products and categories where categories are referenced in Products.
3. Explain how to display the actual reference object instead of the **`objectId`**.
4. Explain how to push a child object into a parent object.
5. Try to set up a similar ref for a sample document.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Chapter 11
Implementing the Frontend

Introduction

In the previous chapter, we saw how to implement the comments service. We looked at sub-documents and references. We chose to use references. We also tested the application using SuperTest.

This chapter will wrap up things by implementing the front-end. We will look at how to go about creating the frontend with Angular. We will also learn about the various pieces of an Angular application like components and services. We will see what the app component is and how to plug in the various components in the app component.

Structure

This chapter will predominantly focus on:

- What is the App component?
- Nesting other components inside the App component
- What are components and services?
- Looking at a sample component

More books: https://t.me/booksforits

- Auth Service
- Running the app

Objectives

After reading this chapter, the reader will understand how to implement the frontend with Angular. We start from the App component and then move to the child component. We will take a look at how to do error handling in Angular as well.

What is the App component?

The **App** component is the root component and acts as a parent component of all other components. This component replaces the **app-root** tag as follows:

```
<app-root>Loading...</app-root>
```

If we take a look at the app component's **.ts** file, the contents are shown in the following code snippet:

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular-Mean';
}
```

As seen in the preceding code, we can see that the selector has **app-root** mentioned. Whatever is mentioned in the template URL, will be loaded in the **index.html**. Let us take a quick look at the **app.component.html**:

```
<nav-bar></nav-bar>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

As seen in the preceding code, there is not much stuff in the app.component.html. We see a peculiar tag **<router-outlet>** though. The **router-outlet** acts as a

More books: https://t.me/booksforits

placeholder that Angular dynamically fills based on the current router state. All child components will be loaded in this tag.

Let us see how we can nest other components in the **app** component.

Nesting other components inside the app component

As a part of nesting other components, we shall take the example of the **Register** component. This will correspond to the **Register** page. We plug the **Register** component in the **app.module.ts**. To achieve this, we place it in an **import** statement as follows:

```
import { RegisterComponent } from './Components/register/register.
component';
```

In the **declarations** section, we place in the register component as follows:

```
declarations: [
    AppComponent,
    NavBarComponent,
    ProfileComponent,
    RegisterComponent,
    HomeComponent,
    DashboardComponent],
```

That is all we need for nesting child components inside the app component. We will also include a sample service that will be used to fetch data from the underlying API.

You must be wondering what components and services are. We will take a look at that next.

What are components and services?

We mentioned components and services in the earlier section. Let us understand what these are in this section. We will understand the components first.

Components

Angular components are a subset of directives and are always associated with a template. Unlike other directives, only one component is allowed for a given element in a template. A component must belong to **NgModule** for it to be available to another component or application.

More books: https://t.me/booksforits

Services

Service is nothing but a broad category encompassing any feature, value, or function that an application requires. A service is a class with a small, well-defined purpose. It should do something specific and execute it well.

Angular separates components and services to enhance reusability and modularity.

Ideally, a component's job is only to enable the user experience. A component should have properties and methods for data binding to communicate between the view and the application logic. The view is what the template renders and the application logic is what includes a model.

A component should use services for tasks that do not involve the view or application logic. Services are good for tasks such as fetching data from the server or logging to the console. By defining such processing tasks in an injectable service class, you can make those tasks available to any component. You can also make your application more adaptable by injecting different providers which provide the same kind of service, as appropriate in different situations.

Let us construct a sample component. We will take the example of the Register component.

Looking at the register component

Navigate to the **`root`** folder and create a folder called **`components`** by issuing the following command :

```
ng generate component Register
```

The preceding command will generate 4 files as shown in the following screenshot:

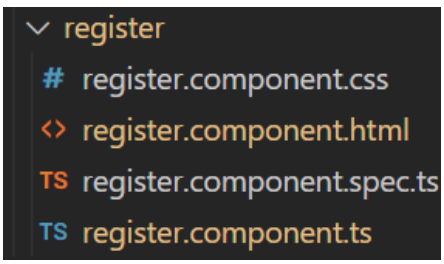

***Figure 11.1**: Building blocks of the register component*

The **`.css`** file is used to style the **`register`** component. The **`register.component.html`** contains code for displaying the view. The **`register.component.spec.ts`** is concerned with testing and the **`register.component.ts`** file is responsible for the business logic for the register component.

More books: https://t.me/booksforits

Register.component.html file

This file holds the HTML code for our register component. Let us understand the code given below. The first two divs are for displaying the **`success`** and **`error`** messages. On clicking the submit button, the **`onRegisterSubmit`** function is called. This function is in the **`register.component.ts`** file. Each of the individual fields has conditions which are evaluated as soon as the user types something in. This is possible since we have used reactive forms. The **`FormsModule`** must be imported to make use of the Angular Forms feature. The rest of the code is simple HTML:

```
<!-- Registration Form -->
    <div class="success" *ngIf="successMsg">Saved successfully</div>
    <div class="error" *ngIf="serverErrors">{{errorMessage}}</div>
    <form [formGroup]="form" (ngSubmit)="onRegisterSubmit()">
    <!-- Username Input -->
          <div class="form-group">
           <label for="username">Username</label>
           <div [ngClass]="{'has-error': (form.controls.username.errors &&
                           form.controls.username.dirty),'has-success':
                           !form.controls.username.errors}">
        <input type="text" name="username" class="form-control"
autocomplete="off"
                           placeholder="*Username"
formControlName="username" />
        <ul class="help-block">
          <li *ngIf="form.controls.username.errors?.required &&
                  form.controls.username.dirty">This field is required</li>
         <li *ngIf="form.controls.username.errors?.minlength || form.control

                s.username.errors?.maxlength">Minimum allowed characters 3
                   Maximum allowed
                  characters 15</li>
          <li *ngIf="form.controls.username.errors?.validateUsername &&
                  form.controls.username.dirty">Username should not have any
                  special characters</li>
```

More books: https://t.me/booksforits

```
        </ul>
      </div>
    </div>

    <!-- Email Input -->
    <div class="form-group">
      <label for="email">Email</label>
      <div [ngClass]="{'has-error': (form.controls.email.errors && form.
controls.email.dirty),'has-success': !form.controls.email.errors}">
        <input type="text" name="email" class="form-control"
autocomplete="off" placeholder="*Email" formControlName="email"/>
        <ul class="help-block">
          <li *ngIf="form.controls.email.errors?.required && form.
controls.email.dirty">This field is required</li>
          <li *ngIf="form.controls.email.errors?.minlength || form.
controls.email.errors?.maxlength && form.controls.email.dirty">Minimum
allowed characters 5 Maximum allowed characters 30</li>
          <li *ngIf="form.controls.email.errors?.validateEmail && form.
controls.email.dirty">This must be valid email</li>
        </ul>
      </div>

    </div>

    <!-- Password Input -->
    <div class="form-group">
      <label for="password">Password</label>
      <div [ngClass]="{'has-error': (form.controls.password.errors &&
form.controls.password.dirty), 'has-success': !form.controls.password.
errors}">
        <input type="password" name="password" class="form-control"
autocomplete="off" placeholder="*Password" formControlName="password"/>
        <!-- Validation -->
        <ul class="help-block">
          <li *ngIf="form.controls.password.errors?.required && form.
```

More books: https://t.me/booksforits

```
controls.password.dirty">This field is required</li>
          <li *ngIf="form.controls.password.errors?.minlength && form.
controls.password.dirty || form.controls.password.errors?.maxlength
&& form.controls.password.dirty ">Minimum characters: 8, Maximum
characters: 35</li>
          <li *ngIf="form.controls.password.errors?.validatePassword &&
form.controls.password.dirty">Password must have at least one capital
letter, numbers, symbols and at least one special character</li>
        </ul>
      </div>
    </div>

    <!-- Confirm Password Input -->
    <div class="form-group">
      <label for="confirm">Confirm Password</label>
      <div [ngClass]="{'has-error': (form.controls.confirm.errors &&
form.controls.confirm.dirty) || (form.errors?.matchingPasswords && form.
controls.confirm.dirty),'has-success': !form.controls.confirm.errors &&
!form.errors?.matchingPasswords}">
        <input type="password" name="confirm" class="form-
control" autocomplete="off" placeholder="*Confirm Password"
formControlName="confirm" />
        <ul class="help-block">
          <li *ngIf="form.controls.confirm.errors?.required && form.
controls.confirm.dirty">This field is required</li>
          <li *ngIf="form.errors?.matchingPasswords && form.controls.
confirm.dirty">Passwords do not match</li>
        </ul>

      </div>
    </div>

     <input [disabled]="!form.valid" type="submit" class="btn btn-
primary" value="Submit" />
  </form>
  <!-- Registration Form /-->
```

More books: https://t.me/booksforits

```
  <p>Username: {{form.controls.username.value}}  </p>
  <p>Email: {{form.controls.email.value}}  </p>
  <p>Password: {{form.controls.password.value}}  </p>
  <p>Confirm: {{form.controls.confirm.value}}  </p>
```

Let us have a look at the **.ts** file for **Registration**.

Register.component.ts file

This file will contain the business logic for the register component. In the preceding code, apart from the other validations, we have the **onRegistersubmit** which subscribes to our register API. When a user is created, a success message is shown and when a duplicate is attempted, an error message is displayed. The **success** and **error** messages disappear after some time due to the **setTimeout** function:

```
import { Component, OnInit } from '@angular/core';
import { AbstractControlOptions, FormBuilder, FormGroup,
FormControl,Validators } from '@angular/forms';
import { AuthService } from 'src/app/Services/auth.service';
import User from 'src/app/models/User';


@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  successMsg: boolean;
  errorMsg: any;
  serverErrors: boolean;
  errorMessage: string;
  form: FormGroup;
  formOptions: AbstractControlOptions = {}
  constructor(
    private formBuilder: FormBuilder,private authService: AuthService
  ) {
    this.createForm()
```

More books: https://t.me/booksforits

```
  }


  createForm(){
    this.form = this.formBuilder.group({
      email: ['',Validators.compose([Validators.required,Validators.
minLength(5),Validators.maxLength(30),this.validateEmail])],
      username: ['',Validators.compose([Validators.required,Validators.
minLength(3),Validators.maxLength(15),this.validateUsername])],
      password: ['', Validators.compose([
        Validators.required, // Field is required
        Validators.minLength(8), // Minimum length is 8 characters
        Validators.maxLength(35), // Maximum length is 35 characters
        this.validatePassword // Custom validation
      ])],
      confirm: ['',Validators.required]
    },{validator: this.matchingPasswords('password','confirm')} as
AbstractControlOptions)
  }


  validateEmail(controls:any){
    const regularExpression = new RegExp(/^(([^<>()\[\]\\.,;:\s@"]+(\.
[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/);


    if (regularExpression.test(controls.value)){
      return null;


    } else {
      return {'validateEmail':true}
    }


  }


  validateUsername(controls:any){
    const regExp = new RegExp(/^[a-zA-Z0-9]+$/);
```

More books: https://t.me/booksforits

```
    if (regExp.test(controls.value)){
      return null;

    } else {
      return {'validateUsername':true}
    }

  }

  validatePassword(controls: any) {
    // Create a regular expression
    const regExp = new RegExp(/^(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[\d])
(?=.*?[\W]).{8,35}$/);
    // Test password against regular expression
    if (regExp.test(controls.value)) {
      return null; // Return as valid password
    } else {
      return { 'validatePassword': true } // Return as invalid password
    }
  }

  matchingPasswords(password: any,confirm: any){
    return (group: FormGroup)=>{
      if (group.controls[password].value === group.controls[confirm].
value){
        return null;
      } else {
        return {'matchingPasswords':true}
      }
    }
  }
  onRegisterSubmit() {
    const user = {
```

More books: https://t.me/booksforits

```
    auth_email: this.form.get('email')?.value,
    auth_password: this.form.get('password')?.value
  }
  this.authService.registerUser(user)!.subscribe(data=>{
    this.successMsg=true;
    setTimeout(()=>this.successMsg=false,4000)
    this.errorMsg=data;
    console.log('in authservice',data);
    return this.errorMsg;
  },err=> {
    console.log('error caught');
    this.errorMessage=err;
    this.serverErrors = true;
    setTimeout(()=>this.serverErrors=false,4000)
    console.log('server errors: ',this.errorMessage);
    return this.errorMessage;})
}
  ngOnInit(): void {

  }

}
```

In the preceding code, we are invoking the **`registerUser`** function from the auth service. So, let us see what we have in the auth service.

Auth Service

The Auth Service is responsible for communicating with the API and sending data back. The service is injected into the **`register.component.ts`** file as shown in the following code:

```
constructor(
    private formBuilder: FormBuilder,private authService: AuthService
  ) {
```

More books: https://t.me/booksforits

```
    this.createForm()
  }
```

The following code makes use of the API config service which then deals with API interactions. Following is the code from the auth service:

```
import { Injectable } from '@angular/core';
import { ApiconfigService } from './apiconfig.service';
import User  from '../models/User'
import { Observable, throwError } from 'rxjs';
import {catchError} from 'rxjs/operators';
import { HttpErrorResponse } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  authToken: any;
  user: User;
  errorMsg:any;
  constructor(private apiConfigService: ApiconfigService) {

  }

  registerUser(user:User) {
    const data = {
      auth_email: user.auth_email,
      auth_password: user.auth_password
    }

    return this.apiConfigService.post('api/users/signup',data).
pipe(catchError(this.handleError));
    }

  private handleError(errorResponse: HttpErrorResponse){
    if (errorResponse.status===0){
```

More books: https://t.me/booksforits

```
      console.error('Client side error: ',errorResponse.error.message);
    } else {
      console.error('server side error: ',errorResponse.error.errors[0].
message);
      this.errorMsg=errorResponse.error.errors[0].message;
    }

    return throwError(this.errorMsg);
    //return new Observable();
  }

  storeUserData(token:any, user:User) {
    localStorage.setItem('token', token); // Set token in local storage
    localStorage.setItem('user', JSON.stringify(user)); // Set user in
local storage as string
    this.authToken = token; // Assign token to be used elsewhere
    this.user = user; // Set user to be used elsewhere
  }
  signinUser(user: User) {
    let data = {
      auth_email: user.auth_email,
      auth_password: user.auth_password
    }
    return this.apiConfigService.post('/api/users/signin',user);
    }

  signoutUser(user: User) {
     let data = {
       auth_email: user.auth_email,
       auth_password: user.auth_password
     }
     return this.apiConfigService.post('/api/users/signout',user);
     }
  }
```

More books: https://t.me/booksforits

The **apiconfigservice** will deal with the API communication as shown in the following code. The **HttpClient** instance is injected in the constructor. When an API call is made, the corresponding API route is invoked:

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

@Injectable({
  providedIn: 'root'
})
export class ApiconfigService {
  BASE_URL='https://blog.dev';
  constructor(private httpClient: HttpClient) {

  }
  get(url: string){
   return this.httpClient.get(`${this.BASE_URL}/${url}`);
  }

  post(url: string, data: Object) {
    const httpHeaders = new HttpHeaders();
    httpHeaders.append('content-type','application/json');

    return this.httpClient.post(`${this.BASE_URL}/${url}`,data);

  }

  put(url: string, data: Object){
    return this.httpClient.put(`${this.BASE_URL}/${url}`,data);
  }
  delete(url: string, data: Object){
    return this.httpClient.delete(`${this.BASE_URL}/${url}`);
  }

}
```

More books: https://t.me/booksforits

Running the app

In the **`root`** folder, run the command **`skaffold dev`** followed by **`minikube tunnel`**. Bring up **`blog.dev`** and submit the data as shown in the following screenshot with a **Saved successfully** message:

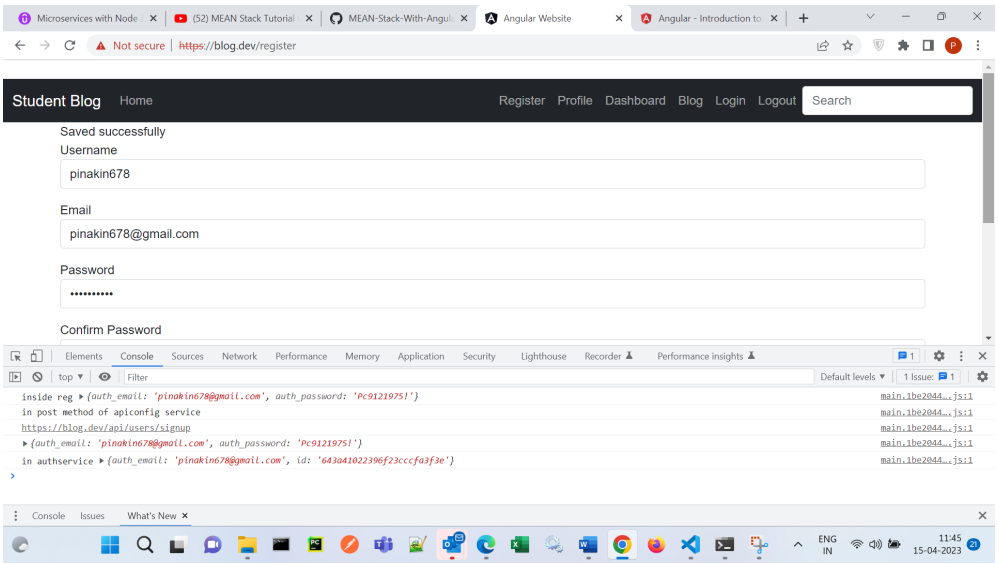

***Figure 11.2**: Registration form in action*

When a retry on the same data is attempted, we get an error as shown in the following screenshot:

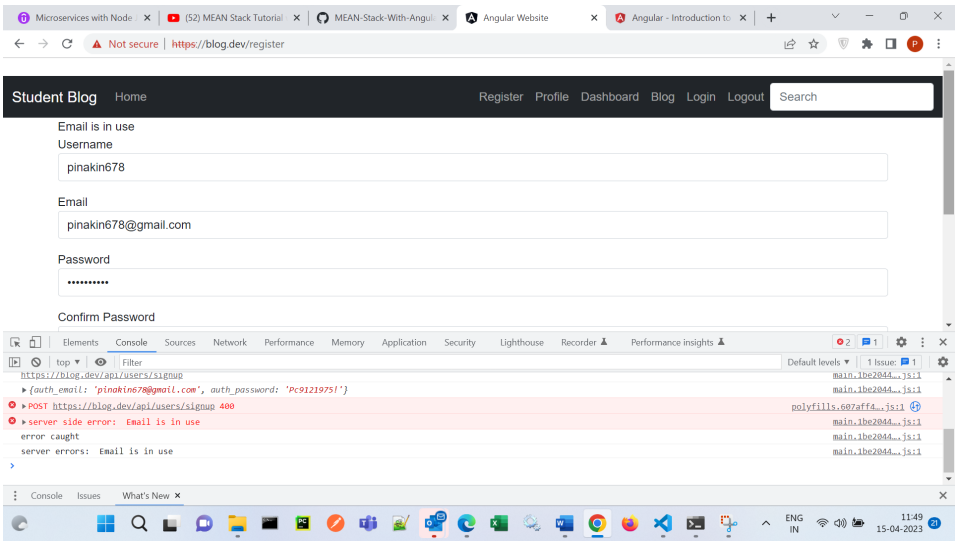

***Figure 11.3**: A Error message for duplicate entry*

More books: https://t.me/booksforits

The following steps must be repeated for all the remaining components [Post and Comment]:

1. Create components for Posts and Comments.
2. Make the changes as shown above for the HTML and typescript files.
3. Plug in the auth service in the individual ts files.
4. In the auth service, include appropriate API calls for the Post and Comment service.

We thus conclude the chapter on frontend integration

Conclusion

In this chapter, we looked at how to integrate the API with the frontend with the use of services. As seen from our Microservices journey, there are many communicating parts of the entire application and understanding how each part communicates with the other is the key to building a solid MEAN stack. We recommend the reader to practice using a sample application, create the individual parts, and then bind all components together to form a good MEAN stack. MEAN microservices is a complicated topic but once you understand the communicating parts, it becomes a lot simpler and fun to learn. This concludes the last chapter of the book. Hope you had an enjoyable journey for the Mean stack

Questions

1. Explain the importance of the app component.
2. Explain how to integrate other components in an app component.
3. What is the use of the spec file?
4. Describe what the function of a service is.
5. Explain the difference between a service and a component.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

More books: https://t.me/booksforits

Index

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

P

R

More books: https://t.me/booksforits

More books: https://t.me/booksforits

More books: https://t.me/booksforits

Mastering MEAN Stack

DESCRIPTION

The MEAN stack, comprising MongoDB, Express.js, Angular, and Node.js, is a widely used and robust web development framework. Acquiring expertise in the MEAN stack will equip you with the necessary skills to strengthen your web development capabilities, enabling you to build efficient and modern web applications.

This book is a comprehensive guide to full stack development using the MEAN stack (MongoDB, Express.js, Angular, and Node.js). It covers all the essential aspects of building robust web applications, from architectural design to implementation. The book introduces the fundamentals of full-stack development and the advantages of using the MEAN stack. It explains the installation and configuration of the MEAN stack components and teaches how to connect them to create powerful full-stack applications seamlessly. The book also covers security mechanisms like authentication and authorization to ensure application security. The book will help you gain proficiency in front-end development with Angular and back-end integration with Node.js. The book also covers real-time data updates using NATS Streaming, automated testing techniques, and the integration of additional services like comments.

By the end of the book, you can confidently build full-stack applications using the MEAN stack.

KEY FEATURES

- Gain a comprehensive understanding of full stack development and the MEAN stack.
- Implement automated testing using Supertest and JEST for reliable and efficient code testing.
- Understand the importance of deployment with containers.

WHAT YOU WILL LEARN

- Install and configure the necessary components for building web applications.
- Master frontend development using Angular, including component creation and data binding.
- Discover the power of Node.js and its integration with Angular for efficient backend development.
- Explore the integration of the Comments service and understand the concepts of subdocuments and references.
- Test the integration of the Posts and Comments service, ensuring smooth communication between the two components.

WHO THIS BOOK IS FOR

Whether you are an absolute beginner or an experienced developer, this book caters to both audiences, providing valuable insights and practical knowledge.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-052-5

More books: https://t.me/booksforits