

Pragmatic APIs with NodeJS and Express

Richard Krasso

First Edition

Table of Contents

<i>Chapter 1. Getting Started with Express</i>	3
Chapter Overview	3
Learning Objectives	3
What is Express?	3
Understanding Express in Web Development	5
Core Components of an Express Application	7
The Express Ecosystem and Middleware	13
Hands-On 1.1: Express.js Reflection	15
<i>Chapter 2. Building a Web Server</i>	16
Chapter Overview	16
Learning Objectives	16
Introduction to the Express Generator CLI	16
Structure of an Express Application	18
Serving HTML and Static Files in Express	22
Basics of Routing in Express	25
Handling Errors in Express	29
Hands-On 2.1: Building your Own Web Server	35
<i>Chapter 3. Developing a JSON Web Service</i>	36
Chapter Overview	36
Learning Objectives	36
Creating a GET Service with Express	37
Exploring findAll Endpoint with TDD	38
Exploring findById Endpoint with TDD	44
Hands-On 3.1: Developing a JSON Web Service	50
<i>Chapter 4. Manipulating Data in a JSON Web Service – Part I</i>	51

Chapter Overview	51
Learning Objectives	51
Understanding CRUD Operations	52
POST Endpoint: The TDD Way for Data Creation	52
DELETE Endpoint: The TDD Way for Data Removal	60
Hands-On 4.1: Manipulating Data in your Web Service	62
<i>Chapter 5. Manipulating Data in a JSON Web Service – Part II</i>	63
Chapter Overview	63
Learning Objectives	63
PUT Endpoint: The TDD Way for Data Updating	64
Hands-On 5.1: Manipulating Data in your Web Service	72
<i>Chapter 6. Securing Your API – Part I</i>	73
Chapter Overview	73
Learning Objectives	73
Understanding Authentication and Authorization	74
Exploring Password Hashing with bcryptjs	75
TDD and Express: Implementing Registration Functionality	77
Hands-On 6.1: Implementing User Authentication	87
<i>Chapter 7. Securing Your API – Part II</i>	88
Chapter Overview	88
Learning Objectives	88
Understanding Password Reset Functionality	88
Password Reset: A Secure Approach with TDD	91
Hands-On 7.1: Enhancing API Security	100
<i>Chapter 8: Deploying Your API</i>	102
Chapter Overview	102
Learning Objectives	102
Understanding Deployment	102
Preparing for Deployment	105
Deploying with Render	107
Post-Deployment Monitoring and Maintenance	110
Hands-On 8.1: Deploying Your API on Render	111

Chapter 1. Getting Started with Express

Chapter Overview

In this chapter, we will embark on an exciting journey into the world of Express.js, a fast, unopinionated, and minimalist web framework for Node.js. We will begin by exploring what Express is and why it has become such a popular choice for web development. Next, we will explore the core components of an Express application, discussing server setup, request and response handling, and routing. We then touch on the Express ecosystem, highlighting its extensibility through various modules and explore Express's concept of middleware, which forms the building blocks for any Express application.

All examples and programs in this book have been created using the most recent version of Node.js at the time of writing, which is version 20. Node.js typically rolls out a new stable version approximately once a year. Despite our best efforts to keep pace with the latest's version of Node.js, it's not always achievable. Therefore, to maximize the benefits of this book, it's strongly advised that you use the same version as used in this book.

Learning Objectives

By the end of this chapter, you should be able to:

- Define what Express is and explain its role in web development.
- Analyze the core components of an Express application.
- Interpret the role of middleware in an Express application.
- Examine the Express ecosystem and its significance in enhancing Express applications.

What is Express?

Express is a third-party npm package, built on top of Node.js that uses Node.js's built-in http module to simplify the development of a web server. Express makes it easier to organize an application and provides built-in support for middleware interactions and routing. At its core, Express is nothing more than a library that provides utility functions that extend the functionality of the http module. Technically, you do not need Express or any other Web framework to build web server applications using Node.js, but with Express, the development process is simplified.

To fully understand the advantages of Express, consider the following: imagine you are creating a superhero team like the Justice League. You could create every superhero from scratch, designing their powers, their costumes, their backstories, and rival villains. That would be like using Node.js's built-in http module to build a website. But what if you could start with a team of superheroes already? Like having Superman, who can fly; Batman, who's a master detective with really cool tech; and Wonder Woman, who's a skilled warrior with a magical lasso. Each of these superheroes have their own special abilities, strengths and weaknesses. And, each of these superheroes can handle their own types of problems.

Express is like that team of superheroes. It gives you a bunch of tools that you can use to build websites, without having to write everything from scratch. For example, Express has tools for handling user requests, routing, and sending responses.

A web framework (like Express) is a software solution that has been built to support the development of web applications, like websites, web services, web resources, and web APIs. Web frameworks standardize the process of building and deploying web applications using industry adopted best practices. The automation of commonly performed activities like URL routing (navigating between web resources and pages), handling HTTP request and response, and interacting with a database are the standard utilities provided within a web framework. In other words, to build a web application, you either need to use a pre-existing web framework or write your own web framework.

Express plays a significant role in the Node.js landscape, because it is one of the most popular and widely used web frameworks among Node.js developers and within the JavaScript full-stack marketplace. It's hard to know the exact percentage of Node.js developers using Express, but if I had to guess, I would estimate the percentage to be anywhere between 60% and 80% of the market is using Express. Especially, if you are considering the applications that have been around for at least the last 5-10 years. Here are some of the key features and benefits of Express:

1. **Simplification of HTTP methods and middleware:** Node.js, at its core, is a low-level platform, and working directly with the http module can be complex and time consuming. Express simplifies this process by providing ready to use interfaces that override and often times improve the functionality of core features within the Node.js platform.
2. **Middleware Support:** Express supports middleware, which are functions that have access to the incoming and outgoing HTTP communications. Express's middleware separates HTTP communications into a request object, response object, and a next function (callback function). This separation allows developers to take a complicated process, like handling HTTP communications, and break it down into smaller, more manageable and reusable components.
3. **Routing:** Express provides a routing mechanism that is robust and easy to use. This provides developers with the ability to design complex routing schemas to support a wide-range of use cases.
4. **Community and Ecosystem:** Express has a large following and is backed by some very large organizations. This means there is a lot of resources, tools, and support available to developers. Thus, making it easier for developers to find solutions to commonly encountered problems and strategies for extending applications with existing toolsets.

5. **Foundation for other frameworks:** Many web frameworks that are on the market today were built based on the foundational concepts implemented in Express. For instances, Sails.js, LoopBack and NestJS are all built on top of Express, extending its functionality for their own features.

In the next section, we will take a look at how Express extends Node.js's http module and some examples of what we can build with Express.js.

Understanding Express in Web Development

In this section we will take a look at several code examples that illustrate how Express extends the core functionality of the Node.js http module. None of the examples in this section are complete (e.g., the code will not compile), instead they are intended to show you differences in implementation details and give you an opportunity to decide for yourself whether Express adds any additional value to the development of a web application.

Now, let's compare the core http module in Node.js with Express using the examples from the previous section:

1. Simplifying HTTP methods:

- a. Node.js http module:

```
const http = require("http");

const server = http.createServer((req, res) => {
  if (req.url === "/" && req.method === "GET") {
    res.write("Hello, World!");
    res.end();
  }
});

server.listen(3000);
```

- b. Express:

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

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

```
app.listen(3000);
```

2. Middleware Support:

- a. Node.js http module:

```
const server = http.createServer((req, res) => {  
  console.log(` Request method: ${req.method}, Request URL:  
  ${req.url} `);  
  // Rest of the code...  
});
```

- b. Express:

```
app.use((req, res, next) => {  
  console.log(` Request method: ${req.method}, Request URL:  
  ${req.url} `);  
  next();  
});
```

3. Routing:

- a. Node.js http module:

```
const server = http.createServer((req, res) => {  
  if (req.url === '/about' && req.method === 'GET') {  
    res.write('About page');  
    res.end();  
  } else if (req.url === '/login' && req.method === 'POST') {  
    // Handle login...  
  }  
});
```

- b. Express:

```
app.get('/about', (req, res) => {  
  res.send('About page');  
});  
  
app.post('/login', (req, res) => {  
  // Handle login...  
});
```

Let's take a look at some use cases for using Express to build web applications:

1. **Building a REST API:** Express is often used to build RESTful APIs, because of its out-of-the-box support for HTTP communications. Through Express you can easily define routes for different types of resources and to handle standard CRUD (Create – Read – Update – Delete) operations.
2. **Serving Static Files:** Express can be used to serve static files like HTML, CSS, and JavaScript, making it an excellent choice for building server-side websites and single-page, server-side rendered applications. In fact, many websites built using the MEAN Stack (Mongo, Express, Angular, and Node.js), MERN (Mongo, Express, React.js, and Node.js), and MEVN (MongoDB, Express, Vue.js, and Node.js) serve static content using this feature.
3. **Building a Web Application:** Express can be used to build full-fledged web applications. It can handle rendering views through a template engine like EJS, Pug, or Handlebars (covered in a later chapter), deal with authorization and authentication, and deal with propagated errors in a graceful and effective manner.
4. **Handling Form Data:** Express can handle form data and file uploads its middleware utility functions (covered in the next section).
5. **Building a Real-Time Application:** Express can be combined with WebSocket libraries to build real-time applications like chat rooms, video conferencing platforms, and project management tools.
6. **Microservices:** Express is also commonly used in microservice architectures, because it is lightweight, easy to use, and efficient. There are several large streaming platforms that have built their entire microservices architecture using Node.js with Express.

In the next section, we will take a deeper look at the core components of an Express application.

Core Components of an Express Application

The building blocks of an Express application are middleware functions, routing, request and response objects, and error handling. In this section we will take a deep dive into how these components work. To start, let's take a look at the most commonly used built-in middleware functions in an express application:

1. **express.json():** This is a built-in middleware function that allows and parses incoming requests using JSON payloads. A payload in API development refers to the data sent in an HTTP request or response body. It is the data that is being sent to or from an API. For example, let's say you are making a POST request to create a new record in the database. The “payload” is the data you capture from the HTML form

that gets sent in the body of the HTTP POST request. For HTTP response, the “payload” is the data that the server sends back to you in response to some type of request. Let’s say for example, you click on a button in your favorite comic book collection website to see a list of comic books. The “payload” is the response data that the sever sends back to the website in response to your click on the comic book listing.

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

2. **express.urlencoded()**: This is a built-in middleware function that parses incoming requests with URL-encoded payloads. URL encoding basically means, the payload was sent in the form of a URL parameter. This typically occurs when you are working with HTML form data that has been encoded. For example, imagine a user filled out a log in form, which send the following payload to our login API:

```
username=krasso&password=s3cret
```

`express.urlencoded()` also includes various configuration options. For example, if we set `extended: true` it allows for objects and arrays to be encoded into a URL-encoded format similar to the structure of a JSON object. Using the above payload as an example, if we set `extended: true`, the payload is parsed into the following object structure:

```
{  
  username: “krasso”,  
  password: “s3cret”  
}
```

3. **express.static()**: This is a built-in middleware function that is used to serve static files, like images, CSS files, and JavaScript.

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

In this code example, a folder named `public` has been identified as the location where static files are being stored/accessible. In other words, if you had a folder named `public` at the root of your Express project, you could place static files in it (CSS, JavaScript, images, etc.,) and whenever a request is made to your server for one of those resources, Express would automatically look for the files in this folder. For example, let’s you are building a webpage and in this webpage, you have a hero image that spans the top of the `<header>` element in your webpage. When a user visits your webpage, the browser will make a request to your web server for the image. Assuming the image was place in the `public` folder, it will be returned by Express’s middleware.

4. **express.Router():** This is a built-in middleware function that is used as a route handler. That is, it creates an instance of the Router object and assigns it to a variable. The advantage of this approach is you make the router portable and exportable to other files in your application. That is you can define routes in separate files, export them using `module.exports`, and import then using require statements. This portability, makes it easier for developers to manage larger applications with many routes.

```
const router = express.Router();
```

5. **express.raw():** This is a built-in middleware function in Express that parses incoming request with raw bodies. What this means is the payload was sent as raw data. An example of when you might need this is when you are building an application that allows for file uploads. In this scenario, the client sends an image as a buffer in the body of the POST request. Using `express.raw()` will allow you to read the buffer from the request body and then save it to a database or a folder on your server.
6. **express.text():** This is a built-in middleware function in Express that parses incoming requests with text payloads.

In Express, routing refers to how an applications endpoint (URIs) respond to client requests. But, before we dive into routing, it's important to level set on what routing is. There are two main types of routing: server-side routing and client-side routing.

1. **Server-side routing:** Server-side routing is the traditional approach to handling navigation in a website. When a user clicks on a link in a webpage (Contact Us link), the browser sends a request to the web server for the resource that was selected (contact.html), the server locates the resource, and sends it back to the user's browser. The result is a full-page refresh and the Contact Us page is displayed. In Express, this type of routing happens when you use methods like `app.get()` and `app.post()`.
2. **Client-side routing:** Client-side routing is a more modern approach to handling navigation in a website. It is based on the principle of single-page applications (SPA), where routing is handled internally by the JavaScript that runs on the page. When a user clicks on a link in a webpage (Contact Us link), the request for a new page does not go to the server, instead, the JavaScript code you wrote catches the request and displays the new content. The fetched data could either be from an API call to a web server or another HTML component within your JavaScript code (think Angular, React.js, and Vue.js). The result is a partial page refresh (only the content is updated). Usually, this type of strategy provides a smoother user experience because there are no full-page refreshes (no round trips to the server).

Here is a simple analogy:

- Server-side routing is like Batman having to return to the Batcave each time he needs a new piece of tech gear. He makes a request (clicks a link), travels to the Batcave (server), picks up a new piece of tech gear (loads the page), and returns to his mission (the user's browser).
- Client-side routing is like Batman carrying a utility belt full of tech gear. He still gets all of his tech gear from the Batcave (server), but they are all delivered at once when the mission starts. Then as Batman encounters different dangers (clicks links), he can quickly pull an item out of his utility belt (load new content) without having to make a trip to the Batcave. This makes his mission go smoother and faster, because he is not making repeated trips to the Batcave to grab new tech gear items. The disadvantage to this approach is the time it takes to load all of the items in his utility belt and the weight it adds (initial page load times). However, once the utility belt is full (page loads), it's much faster and smoother.

With a clearer picture of the differences between client-side and server-side routing, we can explore how Express handles HTTP server-side routing. Here is a breakdown of the most commonly used routing methods:

1. `app.get(path, callback)`: This method is used to handle HTTP GET requests at a specified path location. GET is the default method used when a URL is entered into a web browser. That is, a request is made to the server using the HTTP GET protocol.

```
app.get('/about', (req, res) => {  
  res.send('About page');  
});
```

In this code example, the first actual parameter is the path to the route `/about` and the second actual parameter is an anonymous callback object. If you were to run this application from your local computer the full URL would be `http://localhost:3000/about`. If it were attached to a known domain, like `bellevue.edu`, the full path would be, `https://bellevue.edu/about`.

2. `app.post(path, callback)`: This method is used to handle HTTP POST requests at a specified path location. POST requests are typically used to send data to a server for a new resource (like form data). For example, adding a new comic book to your wishlist.

```
app.post('/comics', (req, res) => {  
  // Add comic book to wishlist...  
});
```

The data sent in a post request is accessible through the `req.body` object (covered in the next section).

3. `app.put(path, callback)`: This method is used to handle HTTP PUT requests at a specified path location. PUT requests are typically used to update existing data on the server.

```
app.put('/comics/:id', (req, res) => {  
  // Update comic book data...  
});
```

The data sent in PUT request is accessible through the `req.body` object (covered in the next section) and the data for the id is accessible through the `req.params` object (covered in the next section).

4. `app.patch(path, callback)`: This method is used to handle HTTP PATCH requests at the specified path location. PATCH requests are typically used to update existing data on the server, but only for the fields that were included in the request.

```
app.patch('/user/:id', (req, res) => {  
  // Update part of comic book...  
});
```

5. `app.delete(path, callback)`: This method is used to handle HTTP DELETE requests at the specified path location. DELETE requests are typically used to delete existing data on the server.

```
app.delete('/user/:id', (req, res) => {  
  // Delete comic book...  
});
```

There is subtle difference between PUT and PATCH, but those differences should be clearly understood. Both PUT and PATCH are HTTP methods used for updating existing data on a server. However, they are used in different contexts.

1. **PUT**: This is an idempotent method. No matter how many times you repeat the request, the result will be the same. You should use PUT when you want the full data entity updated. That is, all fields in the record are updated. If you omit a field in the PUT request, it will be set to null or its default value.
2. **PATCH**: This is not an idempotent method. That is, the result will depend on how many times you repeat the request. You should use PATCH when you only want to update certain fields in a record. This can be more efficient, especially when you are working with large data sets and records.

In summary, **PUT** is used when you want to update all of the fields in a record and **PATCH** is used when you only want to update certain fields in a record.

Every route in express is made up of a path and callback object. This callback object is referred to as the **route handler**. Every route handler includes three objects: req, res, and next. The route handler is where you define what happens when a route is requested from your server.

1. **req (Request)**: This is an object that contains information about the incoming request. This includes the HTTP headers, query strings, data format, and incoming data. You use this object to retrieve the data that was sent from the client to your server.
2. **res (Response)**: This is an object that contains methods for sending a response back to the client. You use this object to send data back to the client in response to some type of request. This includes the HTTP response headers, status codes, requested data, etc.,
3. **next**: This function tells Express to move to the next middleware function in the request-response call cycle. That is, this function passes control back to the next middleware function. If you do not call this function, the request will be left “hanging.”

The request and response objects in Express come with several useful properties and methods. Below outlines some of the most commonly used properties and methods:

Request Object (req):

- **req.body**: Contains a key-value pair of data submitted in the request body. By default, this property is left undefined, until it is populated with data from a client request (submitting a form in an HTML page).
- **req.params**: This is an object that contains properties mapped to the named route parameters. For example, a route to `/comicbook/:id`, would have a **req.params** property for “id”, which would be accessible from **req.params.id**.
- **req.query**: This property is an object containing a property for each of the query strings parameters in the route. Consider the following route: `http://localhost:3000/comicbooks?publisher=dc`. The query parameter ‘publisher=dc’ would be accessible through the **req.query** object. In this case, a call to **req.query.publisher** would result in ‘dc’.

- `req.method`: This property is a string represents the HTTP request method used. For example, GET, PUT, PATCH, etc.,
- `req.url`: This property is a string that represents the URL of the incoming request.

Response Object (res):

- `res.send()`: This method sends an HTTP response. The body can be a Buffer object, string, array, etc.,
- `res.status()`: This method sets the HTTP status code for the response. It is a chainable method and can be used in conjunction with `res.send()`. For example,

```
res.status(204).send({ id: 123 });
```

- `res.json()`: This method sends a JSON response. Behind the scenes it is using `JSON.stringify()` to convert the data to a JSON string.

```
res.json( { name: 'Professor Krasso', title: 'Associate Professor' });
```

- `res.set()`: This method sets the HTTP header field to the specified value. If you want to set multiple fields at once, you will need to pass in an object literal (no arrays).

In the next section we will take a look at the Express ecosystem and some third-party middleware packages that can enhance the functionality of an express application.

The Express Ecosystem and Middleware

The Express ecosystem is vast and feature rich. There are many third-party libraries that can be used to enhance the functionality or extend the functionality of an Express application. As of this writing, here are some of the most popular ones:

1. **cookie-parser**: This middleware package helps developers parse cookies that are sent through the HTTP headers of a request.
2. **cors**: This middleware module is used for enabling Cross Origin Resource Sharing (CORS). With this module, you can manually configure which domains are allowed to share resources with your server.
3. **helmet**: This middleware module helps secure an Express application by setting various HTTP header properties. This is not a “one-size-fits-all” solution. But it can be used to enhance the security of your Express application.
4. **morgan**: This middleware module is used for logging requests in your application.

5. **multer:** This middleware module is used for handling `multipart/form-data`, which is primarily used for uploading files to your server.
6. **passport:** This middleware module is used for authentication. You can use this package to enforce industry best practices for user authentication like, username and password authentication and third-party services like Facebook, Instagram and X.
7. **express-session:** This middleware module allows you to manage user session in an Express application (how to store and share data between HTTP requests).
8. **express-validator:** This is a middleware module that provides a set of middleware utility functions for validating and sanitizing requests.

In Express, the flow of execution order for a request is determined by the order of route handlers in your code. When a request is invoked, Express starts at the top of your middleware stack and executes each middleware function until it reaches the request URL and HTTP method. A routing mechanism is used that compares each route in your application against what is being invoked. When a match is found, it executes the corresponding route handler. If middleware functions are registered before the route, they will be executed before the route handlers. That is, everything is executed sequentially. And, if a middleware function does not end the request-response cycle, it must call the `next()` function to pass control back to the next middleware function in the route handler.

Let's say you have an Express application with middleware parsing for URL-encoding and JSON bodies, and two routes: `/about` and `/contact`. Here is how the process works:

1. **Middleware:** When a request comes in, Express first runs the middleware functions in the order they are defined. In our example, it would first call the middleware function to parse URL-encoded bodies then the one for parsing JSON bodies.

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

// Middleware for parsing URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// Middleware for parsing JSON bodies
app.use(express.json());
```

2. **Routing:** Next, Express looks for a route handler that matches the request URL and HTTP method. That is, if a request is made to `/about`, then it looks for a route

handler of /about with a GET request. If it finds a match, it executes the corresponding route handler:

```
// Route for '/about'
app.get('/about', (req, res) => {
  res.send('About page');
});

// Route for '/contact'
app.get('/contact', (req, res) => {
  res.send('Contact page');
});
```

3. **What happens if `next` is not called:** If a middleware function does not call `next`, the request-response cycle will hang unless the response is ended within the current middleware function.

```
app.use((req, res, next) => {
  console.log('This middleware does not call next');
  // If next is not called here, the request-response cycle will hang
});
```

In this code example, the middleware function does not call `next`, so the response is never sent (ended), so the request-response cycle will hang and the client will be left waiting for a response that never comes. The `next` function can also be used to pass errors to a middleware error handling function (covered in the next section). This is possible, because the `next` function gives control back to the next middleware function in the stack.

Hands-On 1.1: Express.js Reflection

1. **What is Express?** Write a short paragraph explaining what Express.js is, what it's used for, and why it's important in web development.
2. **Understanding Express in Web Development:** List three reasons why Express.js is commonly used in web development. For each reason, provide an example of a problem that Express.js helps to solve.
3. **Core Components of an Express Application:** Identify and briefly describe the four core components of an Express application.

4. **The Express Ecosystem:** Research and write a short paragraph about other tools or libraries that are commonly used with Express.js in the Node.js ecosystem. Explain what they do and how they complement Express.js.
5. **Middleware:** Explain what middleware is in the context of Express.js. Provide an example of a common use case for middleware in an Express application.
6. **Reflection:** Reflect on what you've learned in this chapter. What was the most interesting thing you learned? What do you want to learn more about? Write a short paragraph about your thoughts.

Chapter 2. Building a Web Server

Chapter Overview

In this chapter, we will dive into the practical aspects of building a web application using Express.js. Having been introduced to Express.js in the previous chapter, we now turn our attention to the actual development of an Express application. We begin with the Express Generator CLI, a tool that accelerates the setup process for a new Express.js application. This command-line interface tool helps us scaffold new applications quickly, by providing us with the basic structure and dependencies for an Express.js application.

Next, we will explore the structure of the generated application, discussing its role, purpose, and various files and directories. This will give us a roadmap for working with the architecture of an Express project. We will then move on to exploring how to serve HTML files and static content in an Express application, how to setup our first route to direct traffic, and how to use middleware functions to handle errors and send appropriate responses to the client.

Learning Objectives

By the end of this chapter, you should be able to:

- Analyze the use of the Express Generator CLI
- Examine the structure of an Express application.
- Evaluate methods for serving HTML and static files in Express.
- Assess routing and error handling techniques in Express.

Introduction to the Express Generator CLI

This section introduces the Express Generator CLI, a powerful tool that helps quickly generate a new skeleton application. It is a command-line interface tool that saves us time by scaffolding a new application with all of the required dependencies and structure needed to quickly start building a new Express web server. You can install this tool globally by entering the following command: `npm install -g express-generator@version`. As of this writing, the latest stable version of Express is version 4. Express 5, is in the works, but there

is no known ETA. Understanding version 4 is essential because many of the applications on the market today that use Express were either built with version 3 or 4. The differences between 3 and 4 are subtle and will be pointed out as necessary.

Let's go ahead and install the express generator, globally:

npm install -g express-generator@4.

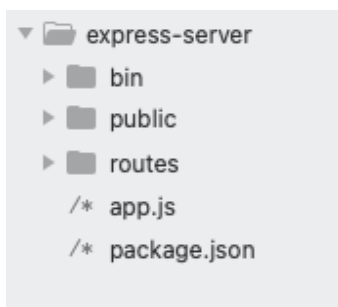
To test the installation of the express generator, enter `express -h` from the CLI window. You should see an output with instructions on how to use the tool. This CLI tool is simple, but powerful and includes the following configuration options:

1. **--version:** This flag outputs the version number of the CLI generator.

express --version

2. **-e, --ejs:** This flag sets the view engine to support EJS views.
3. **--no-view:** This flag instructs the generator to use static HTML instead of a view engine.
4. **-c, --css:** This flag allows you to add a stylesheet to the project and optionally set a CSS engine.
5. **--git:** This flag adds a .gitignore file to the Express project with default configuration settings.
6. **-h, --help:** This flag shows the help menu.

To generate a new application, you use the following command: `express [options] [dir]`. Where `options` are the optional arguments to include in the project generation and `dir` is the location and name of the project to create. For example, entering the following command `express --no-view express-server` will generate a new Express application without a view engine in the current working directory with a project name of `express-server`.



The last step is to install the dependencies by running `npm install`. In the next section, we will take a look at the structure of the generated application and learn how to start it.

Structure of an Express Application

In this section, we will dive into the structure of an Express application, explaining the organization of files and folders in a typical Express.js project. We will cover the purpose and role of each file and directory, including the entry point file (usually the `app.js` or `server.js` file), routes, middleware functions, and public directories.

In the previous section, we created a new Express project using the CLI tool: `express-generator`. The scaffold of this project included the following directories: `bin`, `public`, and `routes`. And, the following files: `app.js` and `package.json`.

The generated `package.json` file includes the following dependencies:

```
{
  "name": "express-server",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "morgan": "~1.9.1"
  }
}
```

In addition to the above dependencies, we will add `nodemon` as a development dependency, so we can enable live reloading. Enabling live reloading greatly improves the development experience, because you no longer need to stop and restart the server each time you want to test a new feature or change in the source code. To install `nodemon`, enter the following command from the CLI window:

`npm install --save-dev nodemon`

Next, we will update the `package.json` file by adding a new script for development deployments, leveraging the `nodemon` npm package.

```
"scripts": {
  "start": "node ./bin/www",
```

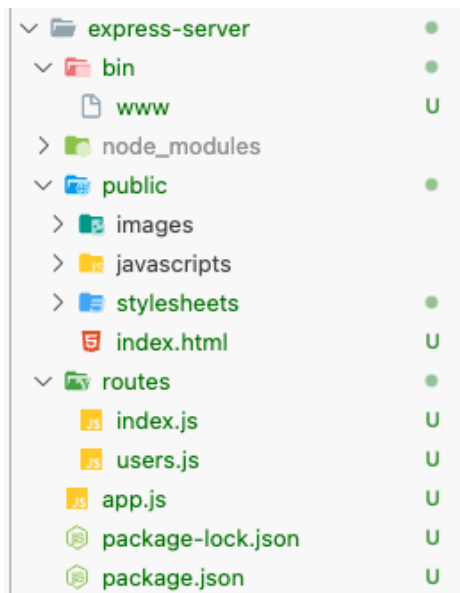
```
"dev": "nodemon ./bin/www"
},
```

By adding this script, we can now start our application using `nodemon`. To do this, run the command: `npm run dev` from the CLI window. This should start the server and you should see the following output (or something similar) in the CLI window:

```
> express-server@0.0.0 dev
> nodemon ./bin/www

[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node ./bin/www`
█
```

To stop the server, enter `Ctrl + C` from the open CLI window. The generated express application comes with several pre-built files and folders. Each of these files and folders become the skeleton of our Express web server.



Inside of the `bin` folder, you will notice a file named `www`. This is the entry point of the application and it is responsible for setting up the web server. It is separate from the `app.js` or `server.js` file, which is primarily focused on setting up the application itself. The `www` file does the following:

1. Imports the necessary modules, including the Express application, the debug module, and the http module for creating our web server.
2. Sets the port number for the web server to listen on, either from an environment variable set by the CLI/hosting platform or a default value of 3000. Using a default value of 3000 is a standard port number for all Node.js servers. And, something we will be following throughout this textbook.
3. It creates the HTTP server using the Express library.
4. It starts the server listening on the port specified in the `port` variable.

Express recommends this approach, because it separates the server setup from the application logic. In doing so, the code is easier to manage and understand. It also allows the Express application to be imported and exported anywhere in the codebase, without having to actually start the server. This greatly improves the ability to unit test your application. To add context to this, let's create a simple Express server without the `express-generator` tool. Create a new folder on your computer and name it `simple-express-server`. Open a new CLI window, navigate to the project and run the following commands:

- **npm init:** This command generates a new `package.json` file with the default settings.
- **npm install express@4:** This command installs Express version 4.
- **npm install --save-dev nodemon:** This command installs nodemon as a development dependency.

Next, update the `package.json` file with two new scripts: `start` and `dev`. The `start` script will start the application by calling `node app.js` and the `dev` script will start the application by calling `nodemon app.js`.

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

Finally, add a new file to the project and name it `app.js`. This file will serve as the entry point and configuration point for the Express server. Using the following code to build your `app.js` file.

```
// Require statements  
const express = require("express");  
  
// Create an Express application  
const app = express();
```

```

// Define the middleware configurations
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Define the port number
const port = process.env.PORT || 3000;

// Define the routes
app.get("/", (req, res, next) => {
  res.send("Hello World!");
});

// Start the server
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});

```

Comparing this code to the generated code from the `app.js` and `www` files in the `express-server` application you will notice everything is now in a single file. While this approach might seem “better” and “easier” to understand, the disadvantage is evident when you start building unit tests. Consider the following:

Scenario:

If you wanted to write a unit test for the GET route in the above code example, you might have something similar to:

```

// test.spec.js
const request = require('supertest');
const app = require('./app');

describe('GET /', () => {
  it('responds with Hello World!', done => {
    request(app)
      .get('/')
      .expect('Hello World!', done);
  });
});

```

The issue with this unit test is, the server would start because of the `app.listen()` call in the `app.js` file, which is not ideal for unit testing. However, if you were to run this unit test against the generated `express-server` application, the server would not start, because the Express application (`app`) is being exported as a module in the `app.js` file. This makes

writing unit tests easier, because you can send HTTP requests and check responses, all without having to start the server.

Starting a server during testing is not ideal for the following reasons:

1. **Unnecessary overhead:** Starting a server takes time and uses resources, which can slow down the speed of your unit tests. This is especially true if you have a test suite with a large volume of unit tests.
2. **Port conflicts:** The server listens on a specific port (in our case, port 3000). If you run the tests while the server is running, you will get an error, because the port will already be in use.
3. **Isolation:** All unit tests should be run in isolation and they should not depend on the state of an external server. Starting the server to run your unit tests introduces state that makes your unit tests less predictable.
4. **Control:** When testing, you want control over how and when your unit tests are run. Starting the server automatically when you import the app.js file, takes some of that control away.

In the next section, we will take a look at how you can use Express to serve HTML and static files.

Serving HTML and Static Files in Express

In this section we will discuss how Express.js servers HTML and other static files like images, CSS, and JavaScript files. We will cover the use of the `express.static()` middleware function, which is built into Express.js and provides an easy way to serve static files from a specified directory. This section is not intended to teach you how to build websites using Express, rather its purpose is to explore some of the built-in features of Express, which should open the door for future discoveries.

Using the `express-generator` CLI tool, we will create a new Express web server with view engine support. Open a new CLI window and enter the following command:

```
express --hbs express-web-server
```

Next, you will need to install the dependencies by running, `npm install` in the directory where the project was created. The `package.json` file from the generated project, should look similar to the following:

```
{  
  "name": "express-web-server",
```

```

"version": "0.0.0",
"private": true,
"scripts": {
  "start": "node ./bin/www"
},
"dependencies": {
  "cookie-parser": "~1.4.4",
  "debug": "~2.6.9",
  "express": "~4.16.1",
  "hbs": "~4.0.4",
  "http-errors": "~1.6.3",
  "morgan": "~1.9.1"
}
}

```

Let's take a look at the top portion of the app.js file, where the server setup is located:

```

var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);

```

Starting from the top, the first 5 lines of code are require statements for npm packages. The next two lines of code are require statements that import the route modules: one from the index.js file and one from the users.js file. This is a great example of portability, where

routes are defined in separate files and require statements are used to import them into the app.js file. Under the code comment 'view engine set' you will notice

1. **app.set("views", path.join(__dirname, "views"));** This line of code sets the directory where Express will look for view files. That is, the webpages you will build for the website. `__dirname` is a Node.js global variable that gives the directory name of the current module. In our example, `path.join(__dirname, "views")` is creating a path to the views folder in the project. The path is relative, which means it will change based on the location where the project is being stored.
2. **app.set("view engine", "hbs");** This line of code sets the view engine for the Express application to `hbs`. This means that Express will use the Handlebars view engine to render views. When you call `res.render("viewname")` in your routes, Express will use Handlebars and the "views" folder to render the view. I selected Handlebars for this example, because it is similar to the syntax used in Angular.

The next few lines of code, configure the middleware with various configuration options. The final line in the `app.use` section

app.use(express.static(path.join(__dirname, "public")));

Tells Express to serve static files (like images, CSS files, and JavaScript files) from the `public` directory in your project. This means, if you have a file at `public/images/photo.jpg`, you can access it at `http://localhost:3000/images/photo.jpg` in your web browser. This allows you to easily reference images and other static content from your webpages (views).

Let's start the server to see what happens.

npm start

Open a new browser window and navigate to `http://localhost:3000`. You should see an empty webpage with the text 'Express' and 'Welcome to Express'.

Let's make a couple modifications so you can see how changes from static files are applied to the views. Stop the server (Ctrl + C) and open the `style.css` file from the `public/stylesheets` folder. Change the body background color to `#1c1e21` and font color to `#fff`. Change the anchor text color to `#fff`, set the font-color for `h1` to `#ccc` and the font-color for `h3` to `#0074a6`. Next, open the `index.hbs` file and replace the paragraph element with an `h3` element. Finally, restart the server (`npm start`), open a new browser window (`http://localhost:3000`), and observe the changes.

Express

Welcome to Express

In the next section, we will take a look at the basics of routing in an Express application.

Basics of Routing in Express

In this section we introduce routing, a crucial aspect of any Express application. We will cover how to define different routes, how to navigate between routes in an Express application, and how to use route parameters to pass data between routes. While this textbook does not cover the specifics on how to build full-fledged websites, it is beneficial to learn how routing works in an Express application, because it will set the foundation for how we build RESTful endpoints in later chapters.

We already have two route files in our `express-web-server` project: `index.js` and `users.js`. And, in the `app.js` file (lines 7 and 8) we have `require` statements that import the router module, which is where we have our routes registered. These middleware functions are responsible for handling HTTP requests that come to those paths.

In the `app.js` file, the lines:

```
app.use('/', indexRouter);  
app.use('/users', usersRouter);
```

are registering the routes. Any requests to the application with a path that starts with `'/'` will be handled by `indexRouter`, and any request with a path that starts with `'/users'` will be handled by the `usersRouter`. These routers define how the application will respond to different types of requests at the paths specified in the route files. When you define routes in Express, you are giving directions to your application on how to respond to different types of requests. The line of code that has `app.use("/users", usersRouter);` is like a main road

sign that says, “For anything going to `/users` follow the directions provided in `usersRouter`.” So, let’s pretend inside `usersRouter` we have more specific instructions:

- **`router.get("/:id/address", ...)`**: This is like a sign that says, “If the path after `/users` is something like `/1007/address`, then do what’s defined in the function that follows.
- **`router.get("/books/:publd/preorder", ...)`**: This is like a sign that says, “If the path after `/users` is something like `/books/99/preorder`, then do what’s defined in the function that follows.

In other words, if you were to navigate in a browser to `localhost:3000/users/1007/address`, you are following these signs:

1. Go to `/users` because that is what you have defined in the `app.js` file:
`app.use("/users", usersRouter);`
2. Then go to `/1007/address/` because that is what you have defined in the `users.js` file
`(router.get("/:id/address", ...);`

In either case, you start with `/users` and then follow more specific instructions defined in `usersRouter`.

Continuing from where we left off in the previous project (`express-web-server`), let’s make some modifications. First, create a new file named `views/superhero.hbs` in the `views` directory.

In the `views/index.hbs` file, add the following code:

```
<a href="/superhero">Superhero</a>
<a href="/superhero?name=Batman">Batman</a>
```

In the new `views/superhero.hbs` file, add:

```
<h1>{{ name }} to the rescue!</h1>
```

Now, let’s adjust our routes. Rename `routes/users.js` to `routes/superhero.js`. The `routes/index.js` file should be updated to:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
```

```
res.render('index', { title: 'Superhero App' });
});
```

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

The routes/superhero.js file should be updated to:

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
  const name = 'name' in req.query ? req.query.name : 'Superhero';
  res.render('superhero', {name: name});
});

module.exports = router;
```

Next, in app.js, we need to update the application to use routes/superhero.js and remove the usage of routes/users.js, which we've renamed. Find the routing setup below the registration of express.static middleware:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

Replace /users with /superhero and change usersRouter to superheroRouter. The updated lines should be:

```
app.use('/', indexRouter);
app.use('/superhero', superheroRouter);
```

Change them to load routes/superhero.js and update the variable name:

```
var indexRouter = require('./routes/index');
var superheroRouter = require('./routes/superhero');
```

If all steps were followed correctly, you should be able to start your Express server with:

npm start

Now, when you visit /superhero in your browser, you will see "Superhero to the rescue!" and when you visit /superhero?name=Batman, you will see "Batman to the rescue!".
Homepage ('/):

Superhero App

Welcome to Superhero App

Superhero Batman

Superhero link:

Superhero to the rescue!

Batman link:

Batman to the rescue!

One item of note, in the above code examples, I am using single quotes for string values. This is not a personal preference, rather it is the default behavior from the `express-generator` CLI tool. It is common best practice to stay consistent with the tools you are using. In other words, because the `express-generator` uses single quotes for string values, these examples are using single quotes. The decision on whether to use single or double quotes is entirely up to you and the team you are working on. The main point is to remain consistent and follow a single coding standard (i.e., do not use single quotes for half of the files and double quotes for the other half).

In the final section of this chapter, we will explore middleware error handling.

Handling Errors in Express

In this final section of chapter 2, we will focus on error handling in Express.js. To do this, we will discuss the importance of error handling and how Express.js provides built-in mechanism for it. We will cover the use of middleware for error handling, how to define error handling functions, and how to send appropriate error responses to the client.

A popular package for handling errors in an Express server is `http-errors`. This package simplifies the creation of HTTP errors in Node.js by providing a set of constructors for commonly used HTTP errors. The most commonly used and encountered errors in an HTTP server are:

- **BadRequest():** This constructor creates a 400 error. This is primarily used when the server could not understand the request. Some examples include: syntax errors and malformed JSON.
- **Unauthorized():** This constructor creates a 401 error. This is primarily used when a request requires authentication. For example, entering an invalid username and password or accessing a webpage without an authentication token.
- **Forbidden():** This constructor creates a 403 error. This is primarily used when the server understood the request, but could not authorize it. For example, a route requires a JSON Web Token, but its either expired or invalid.
- **NotFound():** This constructor creates a 404 error. This is primarily used when the server cannot find the resource that is being requested. For example, navigating to a webpage that does not exists in your website.
- **MethodNotAllowed():** This constructor creates a 405 error. This is primarily used when the HTTP request is not supported by the resource. Making a GET request to a route that only accepts POST requests.
- **InternalServerError():** This constructor creates a 500 error. This is primarily used when an unexpected error occurs and it does not match any of the previously mentioned error types.

Each error object that is created using the `http-errors` constructor has the following properties:

```
{  
  name: 'BadRequestError',  
  message: 'Invalid input',  
  statusCode: 400,  
  status: 400,
```

```
  expose: true
}
```

To achieve a similar structure in Express, you would need to manually create the error object yourself. For example,

```
const err = new Error('Invalid input');
err.status = 400;
err.statusCode = 400;
err.expose = true;
err.name = 'BadRequestError';

console.log(err);
```

The other disadvantage of creating the error objects manually, is it becomes difficult to manage in large applications. This is because of the following reasons:

1. **Consistency:** When creating error object manually, it is easier to introduce inconsistencies in how errors are created, the types of messages being used, and the status codes being used. Each developer that works on the application might represent the object just a little different. While with a few routes this may not seem like a big deal. But, what happens when there are hundreds of developers working on thousands of routes? This could potentially make error debugging, troubleshooting, and writing unit tests more difficult.
2. **Reusability:** When creating error object manually, you will likely be writing the same code multiple times, which will make the code harder to maintain.
3. **Specificity:** The built-in Error class in JavaScript is very general. If you want to create more granular error objects, you will have to create them yourself. This can be very time consuming and is more error-prone.
4. **Error Handling:** If you are creating and throwing errors manually, your code will need to be updated to add manual checks to determine the type of errors being thrown and how to handle them. This can lead to complex and hard to maintain code segments.

Whereas, in the case of the `http-errors` module, most of these concerns are mitigated through a consistent and reliable interface that follows industry standard best practices. Thus, the code is easier to maintain and makes it easier to handle specific types of errors in your error handling middleware functions.

By default, Express comes with a built-in error handler, which takes care of any errors that might occur in your app. This default error handler is added to the stack trace when you

pass an error to `next()`. The error will be written to the client in the form of a stack trace, which is not included in production environments. The problem with this approach is, the error handler does not know the difference between operational and programming errors. For example, a 'Page Not Found' error will be treated the same as a 'Null Reference' error in your code. Ideally, operational errors ('Page Not Found') should be handled separately from programmer errors. This will make your code easier to maintain and test.

We already have a middleware error handler in our **express-web-server** application, which was added when we used the `express-generator` tool. The problem with this error handler is, it is trying to return a view to the client. This assumes we have a Handlebars view named `error.hbs`. In most cases, it's better to return a JSON object that represents the error being thrown.

Let's update our middleware error handler to return a JSON object instead of an error page.

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  res.status(err.status || 500);

  res.json({
    type: 'error',
    status: err.status,
    message: err.message,
    stack: req.app.get('env') === 'development' ? err.stack : undefined
  });
});
```

The first middleware function is used to catch 404 errors, which occurs when the client tries making a request to a resource that does not exist or one the server cannot find. This function should be placed in the `app.js` after all other routes and router imports so that it's called when all other routes have been executed. When a 404 error occurs, an error object is created using the `http-errors` module and passed to the next middleware function in the stack.

```
next(createError(404));
```

The second middleware function is a generic error handler. It is defined with 4 formal parameters: `err`, `req`, `res`, `next`. This tells Express that this is an error handler middleware function and it should be called whenever `next()` is called with an error. In this function, we

first set the HTTP status of the response to the status of the error or 500, if no status code was provided in the error. Setting the default to 500 is a standard best practice (if all else fails, set the status to 500). It then sends a JSON response to the client with the details of the generated error, which includes a stack trace for development environments. As a standard best practice, you never want to send a full stack trace of an error to a production environment.

In the upcoming chapters of this book, we will embark on an exciting journey of building a Cookbook App – An API-based application with a landing page. This application will serve as a platform for food enthusiasts to explore, create, and manage a variety of recipes. In the initial phase, we will lay the foundation by setting up the base project and crafting an engaging landing page. As we progress through the chapters, we will incrementally enhance the application by adding various API endpoints and unit tests to validate their functionality. This will allow us to view a list of recipes, retrieve a single recipe by its unique identifier, create new recipes, delete existing recipes, and update recipes based on their IDs.

Furthermore, we will also incorporate user registration functionality, enabling users to create their own accounts and become part of the Cookbook App community. To ensure a smooth user experience, we will also provide users with the ability to reset their password by answering their selected security questions. While this application will be primarily an API-driven project, we will add a landing page to serve as the face of our Cookbook App.

The starter project for the Cookbook application is located in the data files folder.

1. Use your code editor to go to the cookbook folder of your data files.
2. Open the app.js file. Enter your name, date, file name, and description as code comments at the top of the file. Save the file.
3. In the app.js file, set up an Express application by adding the following require statements:

```
const express = require("express");  
const bcrypt = require("bcryptjs");  
const createError = require("http-errors");
```

4. Create an express application by defining a variable and assigning it the Express module.

```
const app = express(); // Creates an Express application
```

5. Add two app.use statements, one to tell Express to parse incoming requests as JSON payloads and one for parsing incoming urlencoded payloads.


```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

6. Add middleware functions to handle 404 and 500 errors. The 404 middleware should be added after all other routes, and the 500 middleware should be added last.
7. For the 500-error middleware, return a JSON response with the error details, include the error stack only if the application is running in the development environment.
8. Add a GET route for the root URL ("/"). This route should return an HTML response with a fully designed landing page that represents the Cookbook project.

```
app.get("/", async (req, res, next) => {

  // HTML content for the landing page
  const html = `
<html>
<head>
  <title>Cookbook App</title>
  <style>
    body, h1, h2, h3 { margin: 0; padding: 0; border: 0;}
    body {
      background: #424242;
      color: #fff;
      margin: 1.25rem;
      font-size: 1.25rem;
    }
    h1, h2, h3 { color: #EF5350; font-family: 'Emblema One', cursive;}
    h1, h2 { text-align: center }
    h3 { color: #fff; }
    .container { width: 50%; margin: 0 auto; font-family: 'Lora', serif; }
    .recipe { border: 1px solid #EF5350; padding: 1rem; margin: 1rem 0; }
    .recipe h3 { margin-top: 0; }

    main a { color: #fff; text-decoration: none; }
    main a:hover { color: #EF5350; text-decoration: underline;}
  </style>
</head>
<body>
  <div class="container">
    <header>
      <h1>Cookbook App</h1>
```

```

    <h2>Discover and Share Amazing Recipes</h2>
  </header>

  <br />

  <main>
    <div class="recipe">
      <h3>Classic Beef Tacos</h3>
      <p>1. Brown the ground beef in a skillet.<br>2. Warm the taco shells in the oven.<br>3. Fill the taco shells with beef, lettuce, and cheese.</p>
    </div>
    <div class="recipe">
      <h3>Vegetarian Lasagna</h3>
      <p>1. Layer lasagna noodles, marinara sauce, and cheese in a baking dish.<br>2. Bake at 375 degrees for 45 minutes.<br>3. Let cool before serving.</p>
    </div>
  </main>
</div>
</body>
</html>
` ; // end HTML content for the landing page

res.send(html); // Sends the HTML content to the client
});

```

9. Test the server by running, `npm run dev` or `npm start`.

Assuming you followed the instructions correctly, you should see a landing page with a title, sub title, and two divs with recipes. While the server is still running, try entering `http://localhost:3000/about` in the browser. You should see a JSON message similar to the following:

```

{"type":"error","status":404,"message":"Not Found","stack":"NotFoundError: Not Found"}

```

This is being generated from the middleware error handler for 404 errors. Go ahead and stop the server, so we can test if the 500-middleware error handler is working correctly too. Open the `app.js` file and inside the GET route add the following line of code (this is only for testing purposes).

```

next(createError(501)); // Creates a 501 error

```

Restart the server, open a new browser window, and enter `http://localhost:3000`. You should see the following JSON message:

```
{"type": "error", "status": 501, "message": "Not Implemented",  
/Users/username/projects/webdev/web-420-solutions/week-2/}
```

This is the result of our second middleware error handler picking up the 501 error. Now that we can confirm that both of our middleware error handlers are working correctly, we should remove this line of code from our project (this was only added to test error handling).

Hands-On 2.1: Building your Own Web Server

In this hands-on project, you will be embarking on an exciting journey of building your own API-driven application called “In-N-Out-Books.” This application will serve as a platform for managing collections of books.

The “In-N-Out-Books” project is designed as a build-per-chapter solution, which each chapter you will add new features and functionalities to your application. This approach allows you to incrementally build and improve your application, while also providing ample opportunities to practice and reinforce the concepts being covered in this book.

The idea of the “In-N-Out-Books” was inspired by the love of books and the desire to create a platform where users can manage their own collection of books. Whether you are an avid reader who wants to keep track of the books you’ve read, or a book club organizer who needs to manage a shared collection, “In-N-Out-Books” is designed to cater to your needs.

For the first hands-on assignment, you will be setting up the initial project structure and creating the server for your application.

Instructions:

1. Use your code editor to go to the `in-n-out-books` folder of your data files.
2. Open the `app.js` file. Enter your name, date, file name, and description as code comments at the top of the file. Save the file.
3. In the `app.js` file, set up an Express application.
4. Add a GET route for the root URL (`"/`). This route should return an HTML response with a fully designed landing page that represents the “in-n-out-books” project.
5. Add middleware functions to handle 404 and 500 errors.
6. In the 500-error middleware, return a JSON response with the error details. Include the error stack only if the application is running in development mode.

7. Export the Express application from the app.js file.

Grading: You will earn 20 points for each of the following

- Correct setup of the Express application.
- Correct implementation and design of the GET route and the landing page.
- Correct implementation of error handling middleware.

This assignment is worth a total of 60 points. If two tasks are completed correctly, you will earn 40 points.

Hints:

- For setting up the Express application, remember to require the Express module and call it as a function.
- When creating the GET route, remember that the route handler methods (like GET) take a path and a callback function as arguments.
- For the middleware functions, you can use `app.use()`. The middleware function for handling 404 errors should be added after all other routes.
- In the 500 error middleware, you can set the status code using `req.status()`. To return a JSON response, you can use `res.json()`.
- Don't forget to export the Express application at the end of the app.js file.

Chapter 3. Developing a JSON Web Service

Chapter Overview

In this chapter, we will enhance the cookbook project by introducing GET services using Express. We will focus on the `findAll` and `findById` endpoints, exploring their implementation through the lens of Test-Driven Development (TDD). Along with making these services, we will also learn how to create unit tests. This helps us make sure our APIs are stable and reliable. By the end of this chapter, you'll have a practical understanding of creating and testing GET services, reinforcing your web development skills.

Learning Objectives

By the end of this chapter, you should be able to:

- Analyze the creation of a GET service in Express
- Examine the database folder and its files in the cookbook application
- Evaluate the implementation of `findAll` and `findById` functions using TDD
- Assess the effectiveness of unit tests for APIs

Creating a GET Service with Express

In this section, we will extend the express-web-server project by introducing a GET service. We will walk through the process of creating a GET request that returns a JSON payload, providing a practical example of how to use Express to build web services.

In the context of API development, a GET service is a type of HTTP request that retrieves (or “gets”) data from a server. Let’s imagine we have a digital library of DC superheroes. Each superhero has their own page with information about them, like their superpowers, weaknesses, enemies, and allies. A GET service in this context would be like asking the librarian (the server) for information on Batman (GET request). The librarian would retrieve this page and provide you with the information.

Starting with where we left off from the express-web-server project in Chapter 2, open the superhero.js file. In this file we will be adding a JSON web service that retrieves information about a superhero. Add the following code to your file:

```
router.get('/batman', async (req, res, next) => {
  try {
    const batman = {
      name: 'Batman',
      realName: 'Bruce Wayne',
      city: 'Gotham City',
      superpower: 'Rich',
      allies: ['Robin', 'Batgirl', 'Superman'],
      enemies: ['Joker', 'Two-Face', 'Bane'],
    }

    res.send(batman);
  } catch (err) {
    console.error(err);
    next(err);
  }
});
```

Here is a breakdown of what the function is doing:

1. **router.get('/batman', async (req, res, next) => {});** This is the functions signature, which creates an HTTP GET endpoint at the path '/batman'. When a user makes a request to <http://localhost:3000/superhero/batman>, the asynchronous function is executed.

2. **try {...} catch (err) { ... }** This is a try-catch block. The code inside the try block is executed first. If an error occurs while executing the code, the catch block takes over and the code inside of the catch block is executed.
3. **const batman = {...};** This code creates a JavaScript object called batman with properties for name, realName, city, superpower, allies, and enemies.
4. **res.send(batman);** This line of code sends the batman object back to the client as the response to the GET request. The res.send() method will automatically set the content-type and parse the object into a JSON string.
5. **console.error(err);** This line logs the error to stderr.
6. **next(err);** This line passes the error to the next middleware function in the stack. In our case, this means the middleware error handlers from the app.js file will be called.

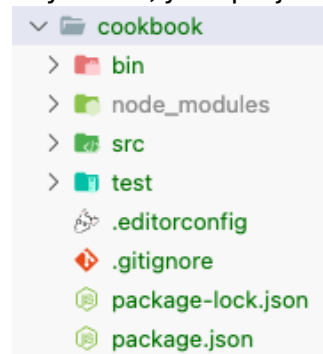
To test this route, start the server by running `npm start`. Next, open a new browser window and enter the URL: `http://localhost:3000/superhero/batman`. Assuming there are no errors, you should see a JSON response with Batman's details.

In the next section, you will learn how to use TDD principles to create a `findAll` endpoint.

Exploring `findAll` Endpoint with TDD

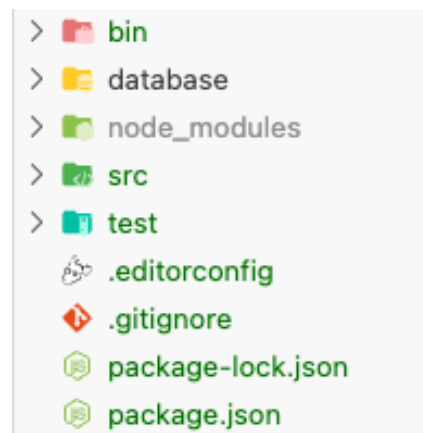
This section delves into the `findAll` endpoint. We will explore the database folder and its files, and demonstrate how to create the `findAll` function for our cookbook application using TDD principles. This will provide a practical example of how to build and test an API endpoint. But, before we begin, a bit of a disclaimer. This book assumes you have some previous knowledge with unit testing and TDD principles. If you are not familiar with either of these concepts, you may find the remaining chapters in this textbook difficult to follow.

In chapter 2, the cookbook application was introduced and we built the base structure and landing page for the project. Assuming you were able to get through the chapter without any errors, your project should resemble the following:



The `bin` folder contains a single file named `www`, which is what we use to initialize and start the Express server. The `src` folder contains our `app.js` file and any other files we might need to support the development of our APIs. Finally, the `test` folder will be used to build our unit tests using TDD principles and Jest.

To simulate database interactions, I've included four files in the `data-files` folder. These files are: `books.js`, `collection.js`, `recipies.js`, and `users.js`. Each of these files' serves a different purpose, but they will be used throughout the remaining chapters in this book. For now, open the cookbook project in your editor and create a new folder named `database` and place it at the root of the cookbook project:



Add the `collection.js` and `recipies.js` files to this folder from the `data-files` directory. The `collection.js` file should contain the following code:

```
class Collection {
  constructor(data) {
    this.data = data;
  }

  find(query = {}) {
    const items = this.data.filter(item =>
      Object.keys(query).every((key) => item[key] === query[key])
    );
    return Promise.resolve(items);
  }

  findOne(query) {
    const item = this.data.find(item =>
      Object.keys(query).every((key) => item[key] === query[key])
    );

    if (!item) {
```

```

    return Promise.reject(new Error('No matching item found'));
  }

  return Promise.resolve(item);
}

insertOne(item) {
  this.data.push(item);

  return Promise.resolve({ result: { ok: 1, n: 1 }, ops: [item] });
}

updateOne(query, update) {
  const item = this.data.find(item =>
    Object.keys(query).every((key) => item[key] === query[key])
  );

  if (!item) {
    return Promise.reject(new Error('No matching item found'));
  }

  const index = this.data.indexOf(item);

  this.data[index] = { ...this.data[index], ...update };

  return Promise.resolve({ result: { ok: 1, nModified: 1 }, matchedCount: 1,
modifiedCount: 1 });
}

deleteOne(query) {
  const index = this.data.findIndex(item => item.id === query.id);

  if (index === -1) {
    return Promise.reject(new Error('No matching item found'));
  }

  this.data.splice(index, 1);

  return Promise.resolve({ result: { ok: 1, n: 1 }, deletedCount: 1 });
}
}

module.exports = Collection;

```


And, the `recipes.js` file should contain the following code:

```
const Collection = require("../collection");  
  
const recipes = new Collection([  
  { id: 1, name: "Pancakes", ingredients: ["flour", "milk", "eggs"] },  
  { id: 2, name: "Spaghetti", ingredients: ["pasta", "tomato sauce", "ground beef"] },  
  { id: 3, name: "Chicken Salad", ingredients: ["chicken", "lettuce", "tomatoes",  
    "cucumber"] },  
  { id: 4, name: "Beef Stew", ingredients: ["beef", "potatoes", "carrots", "peas"] },  
  { id: 5, name: "Fish Tacos", ingredients: ["fish", "tortillas", "avocado", "salsa"] },  
]);  
  
module.exports = recipes;
```

The details of these two files are outside the scope of this book. However, it is important to understand how they are used in the context of the APIs we will be building. The `recipes.js` file contains an array of recipes with properties for `id`, `name`, and `ingredients`. And, the `collection.js` file contains a class with several methods that simulate the commonly used operations invoked on a MongoDB database. To use these files, you will need to import the `recipes.js` file in the `app.js` file using a `require` statement. Then, you can access the methods from the `collection.js` file by using dot notation. For example,

```
recipes.find();  
recipes.findOne();
```

One of the software development methodologies that is widely used is Test-Driven Development (TDD). It involves an iterative process of building programs. The developer creates a unit test that is meant to fail, then updates the code to pass the test, and finally refactors and implements the code in their program. This cycle is commonly referred to as “Red, Green, Refactor,” and it can be repeated multiple times during the development of an application or until the designed outcome is achieved. As previously mentioned, all of the code examples and hands-on projects in this book will use this methodology to demonstrate how to build APIs using Node.js and Express.

As a matter of convenience, the npm package `supertest` has already been added to the `cookbook` starter project. `supertest` is a high-level abstraction npm package for testing HTTP communications. It allows us to test HTTP endpoints in a streamline and intuitive way. Typically, testing packages like `supertest`, are added to a project as a development dependency. Meaning they are not necessary for running the application in production. As we work through the examples in this book, the advantages of `supertest` will become more apparent.

1. Open the `app.spec.js` file in your editor.

2. Add require statements for the app.js file and supertest.

```
const app = require("../src/app");  
const request = require("supertest");
```

3. Create a new test suite using Jest's describe method:

```
describe("Chapter 3: API Tests", () => { ... })
```

4. In the body of the describe method, add a unit test to check if an array of recipes is returned when a client calls the endpoint: `/api/recipes`.

This unit test should have the following checks:

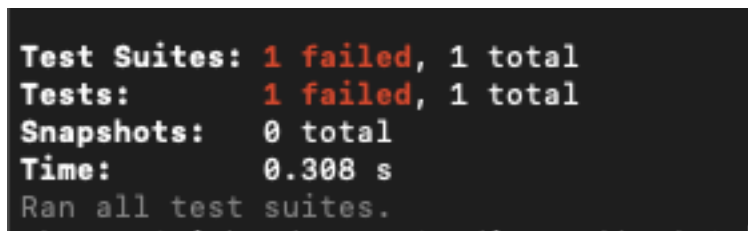
- a. 200 status code
- b. The response body should be an instance of an array
- c. Each item in the array should have the following properties: id, name, and ingredients.

```
const app = require("../src/app");  
const request = require("supertest");  
  
describe("Chapter 3: API Tests", () => {  
  it("it should return an array of recipes", async () => {  
    const res = await request(app).get("/api/recipes");  
    expect(res.statusCode).toEqual(200);  
    expect(res.body).toBeInstanceOf(Array);  
  
    res.body.forEach((recipe) => {  
      expect(recipe).toHaveProperty("id");  
      expect(recipe).toHaveProperty("name");  
      expect(recipe).toHaveProperty("ingredients");  
    });  
  });  
});
```

In this code example, we define a test suite for our API endpoint. This suite contains a single test that verifies the functionality of the `/api/recipes` endpoint. Let's dive into each part of this code to understand how it works.

- **describe("Chapter 3: API Tests", () => { ... });** This sets up the test suite named "Chapter 3: API Tests".
- **It("it should return an array of recipes", async() => { ... });** This defines a unit test that checks if the `/api/recipes` endpoint returns an array of recipes.
- **const res = await request(app).get("/api/recipes");** This sends a GET request to the `/api/recipes` endpoint and waits for a response, using the `supertest` npm package.
- **expect(res.statusCode).toEqual(200);** This checks if the response status code is 200, indicating the request was successful.
- The `forEach` loop, checks every recipe in the response body to ensure it has properties for id, name, and ingredients.

To ensure our test fails, run `npm test` from a new CLI windows.



```

Test Suites: 1 failed, 1 total
Tests:      1 failed, 1 total
Snapshots:  0 total
Time:       0.308 s
Ran all test suites.

```

Passing Test (Green):

1. Open the `app.js` file.
2. Add a require statement for the `recipes.js` file.

```
const recipes = require("../database/recipes");
```

3. Create a GET endpoint for `/api/recipes`. In the body of this endpoint, add a try-catch block for unexpected errors, and return an array of recipes using our mock databases `find()` method. Place this endpoint underneath the code you wrote for the landing page.

```
app.get("/api/recipes", async (req, res, next) => {
  try {
    const allRecipes = await recipes.find();
    console.log("All Recipes: ", allRecipes); // Logs all recipes
    res.send(allRecipes); // Sends response with all books
  } catch (err) {
```

```

    console.error("Error: ", err.message); // Logs error message
    next(err); // Passes error to the next middleware
  }
});

```

In this code example, we define a GET endpoint `/api/recipes` in our Express.js application. This endpoint retrieves all recipes from the `recipes.js` module and sends them back to the client. Let's take a closer look at the functionality of this API.

- **`app.get("/api/recipes", async (req, res, next) => { ... });`** This sets up an asynchronous GET endpoint at the path `/api/recipes`.
- **`const allRecipes = await recipes.find();`** This line of code retrieves all recipes from the `recipes.js` module. The `await` keyword is used to wait for the promise returned by the `recipes.find()` to resolve.
- **`res.send(allRecipes);`** This sends the retrieved recipes back to the client as the response of the GET request.
- The `catch` block handles any errors that occur during the execution of the `try` block. It logs the error message to `stdout` and passes the error to the next middleware function in the stack.

Let's rerun the test command (`npm test`) to validate that the unit test now passes.

```

PASS test/app.spec.js
Chapter 3: API Tests
  ✓ it should return an array of recipes (20 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.19 s, estimated 1 s
Ran all test suites.

```

In the next section we will take a look at how to build a `findById` endpoint using TDD principles.

Exploring `findById` Endpoint with TDD

In this section, we will focus on the `findById` endpoint. We will guide you through the process of creating unit tests for this API, reinforcing the importance of TDD in ensuring the reliability and robustness of our web services.

1. Open the `app.spec.js` file in your editor.
2. In the body of the test suite, we created for Chapter 3, add a unit test to check if a single recipe is returned when a client calls the endpoint: `/api/recipes/:id`.

The unit test should have the following checks:

- a. 200 status code.
- b. The response body should have properties `id`, `name`, and `ingredients`.
- c. For ID 1, the ingredients should have values: `flour`, `milk`, and `eggs`.

```
it("should return a single recipe", async () => {  
  const res = await request(app).get("/api/recipes/1");  
  
  expect(res.statusCode).toEqual(200);  
  expect(res.body).toHaveProperty("id", 1);  
  expect(res.body).toHaveProperty("name", "Pancakes");  
  expect(res.body).toHaveProperty("ingredients", ["flour", "milk", "eggs"]);  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes/:id` endpoint. Let's dive into each part of the code to understand how it works.

- **It("should return a single recipe", async () => { ... });** This defines a unit test that checks if the `/api/recipes/:id` endpoint returns a single recipe.
- **const res = await request(app).get("/api/recipes/1");** This sends a GET request to `/api/recipes/:id` endpoint and waits for a response, using the `supertest` npm package.
- **expect(res.statusCode).toEqual(200);** This checks if the response status code is 200, indicating the request was successful.
- **expect(res.body).toHaveProperty(...);** This checks the response body to have the property passed in key-value property. These properties include: `id`, `name`, and `ingredients`. The last `expect()` statement checks that the `ingredients` property has elements for `flour`, `milk`, and `eggs`.

To ensure our tests fails, run `npm test` from a new CLI window.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 1 passed, 2 total
Snapshots:  0 total
Time:       0.26 s, estimated 1 s
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Create a new GET endpoint for `/api/recipes/:id`. In the body of this endpoint, add a try-catch block for unexpected errors, and return a single recipe using our mock databases `findOne()` method.

```
app.get("/api/recipes/:id", async (req, res, next) => {
  try {
    const recipe = await recipes.findOne({ id: Number(req.params.id) });
    console.log("Recipe: ", recipe);
    res.send(recipe);
  } catch (err) {
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we define a GET endpoint at `/api/recipes/:id` in our Express application. This endpoint retrieves a specific recipe by its ID from our mock database and sends it back to the client. Let's take a closer look at the functionality of this API.

- **`app.get("/api/recipes/:id", async (req, res, next) => { ... });`** This line sets up the asynchronous GET endpoint at `/api/recipes/:id`.
- **`const recipe = await recipes.findOne({ id: Number(req.params.id) });`** This line of code fetches the recipe with the specified ID from the mock database. The `await` keyword is used to pause execution until the promise returned by the `findOne()` method is resolved. `req.params.id` is used to retrieve the value being passed to the endpoint. In the endpoint `/api/recipes/:id` the `:id` at the end of the endpoint is being used as a "placeholder" for us to pass in an actual parameter value that we can use in our API function. To access this value, we use dot notation. The "placeholder" variable name must match the name of the property being referenced in the `req.params` object. For example, if I have a route param variable named `:coolParamValue` and I try accessing it as `req.param.id`, the system will generate an undefined error, because `id` does not exist in the `req.params` object. Instead, I

would retrieve the value by calling, `req.params.coolParamValue`. Also, note, we are using `Number(req.params.id)` to explicitly cast the actual parameter value into a numerical value, which is what our mock database is expecting.

- **`res.send(recipe);`** This line of code sends the retrieved recipe back to the client as the response to the GET request.

Rerun the unit tests to ensure they are passing (`npm test`).

```
PASS test/app.spec.js
Chapter 3: API Tests
  ✓ it should return an array of recipes (23 ms)
  ✓ should return a single recipe (3 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.253 s, estimated 1 s
Ran all test suites.
```

Let's also try testing this endpoint manually, using different parameter values to see how fault tolerate our endpoint is. Ensure your server is running and open a second CLI window and run the following Node.js command to test a different use case:

```
node -e "require('http').get('http://localhost:3000/api/recipes/abc', (res) => {
res.on('data', (chunk) => { process.stdout.write(chunk); }); });"
```

This command sends a GET request to the `/api/recipes/:id` endpoint. Since `abc` is not a numerical ID, you should see an error.

```
{"type":"error","message":"No matching item found",
```

This highlights a flaw in our API. We are assuming that the client will always send numerical values for the ID. However, as our test demonstrates, this may not always be the case. The solution, as you might have guessed, is to add error handling into our API to account for non-numerical ID values. But, before we dive into modifying our code to support this feature, let's adhere to the principles of TDD and write a unit test that first checks for this specific error handling. Once we have the test in place, we will then write the code to pass the test.

1. Open the `app.spec.js` file.

2. In the body of the describe method, add a unit test that checks a 400 error is returned when the ID is not a number.

The unit test should have the following checks:

- a. 400 status code.
- b. Response body message to equal “Input must be a number”.

```
it("should return a 400 error if the id is not a number", async () => {  
  const res = await request(app).get("/api/recipes/foo");  
  expect(res.statusCode).toEqual(400);  
  expect(res.body.message).toEqual("Input must be a number");  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes/:id` endpoint. Let's dive into each part of this code to understand how it works.

- **it("should return a 400 error if the id is not a number", async () => { ... });** This defines a unit test that checks if the `/api/recipes/:id` endpoint returns a 400 status code when the ID is not a number.
- **const res = await request(app).get("/api/recipes/foo");** This sends a GET request to the `/api/recipes/:id` endpoint and waits for a response using the `supertest` npm package.
- **expect(res.statusCode).toEqual(200);** This checks if the response status code is 400. Indicating a Bad Request.
- **expect(rss.body.message).toEqual("Input must be a number");** This checks if the response message is “Input must be a number”.

To ensure our test fails, run `npm test`.

```
Test Suites: 1 failed, 1 total  
Tests:      1 failed, 2 passed, 3 total  
Snapshots:  0 total  
Time:       0.263 s, estimated 1 s  
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Add validation to the GET endpoint that checks if the req.params.id is a numerical value. If it's not, generate a 400 error and pass it to our middleware error handler.

```
app.get("/api/recipes/:id", async (req, res, next) => {
  try {
    let { id } = req.params;
    id = parseInt(id);

    if (isNaN(id)) {
      return next(createError(400, "Input must be a number"));
    }

    const recipe = await recipes.findOne({ id: id });

    console.log("Recipe: ", recipe);
    res.send(recipe);
  } catch (err) {
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we add error checking to test if the req.params.id value is a number. Let's take a closer look at what this code is doing:

- **let { id } = req.params;** This line of code is destructuring the `id` property from the `req.params` object.
- **id = parseInt(id);** This line of code is using JavaScript's built-in `parseInt()` function to parse the string value into a numerical one.
- **if (isNaN(id)) { ... }** This block of code is using JavaScript's built-in `isNaN()` function to check if the `id` variable is `NaN` (not a number). JavaScript's built-in `parseInt()` function will return `NaN` if the string value cannot be converted to a numerical one.
- **return next(createError(400, "Input must be a number"));** This line of code is calling the `http-errors` npm package, creating a 400 error with the message "Input must be a number".

Let's rerun the test command (`npm test`) to confirm that our unit tests now pass.

```
PASS test/app.spec.js
  Chapter 3: API Tests
    ✓ it should return an array of recipes (22 ms)
    ✓ should return a single recipe (2 ms)
    ✓ should return a 400 error if the id is not a number (2 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.258 s
Ran all test suites.
```

Hands-On 3.1: Developing a JSON Web Service

In this hands-on project, you will be building APIs for the in-n-out-books project using a mock database. You will be applying Test-Driven Development (TDD) principles and using Jest for testing.

Instructions:

1. Continue working in the project from the previous chapter's hands-on project (in-n-out-books).
2. In your project, ensure you have the following file structure:

```
src
  app.js
test
  app.spec.js
package.json
database
  books.js
  collection.js
```

The files for the mock database are located in the data-files folder. Contact your instructor if you need access to this folder.

3. You will be building routes in the app.js file:
 - a. A GET route at `/api/books` that returns an array of books from the mock database. Use a try-catch block to handle any errors.
 - b. A GET route at `/api/books/:id` that returns a single book with the matching `id` from the mock database. Use a try-catch block to handle any errors. Add

error handling to check if the `id` is not a number and throwing a 400 error if it is not with an error message.

4. Write unit tests for each of these routes using Jest in your `app.spec.js` file. You should write the tests before you implement the routes, following TDD principles. Create a test suite named “Chapter [Number]: API Tests”. Use the following test cases:
 - a. Should return an array of books.
 - b. Should return a single book.
 - c. Should return a 400 error if the `id` is not a number.

Grading:

You will earn 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- Use a `require` statement to import the `books.js` file from the database folder.
- Use the `findOne` and `find` methods to retrieve books from the mock database.
- Remember to convert the `id` from the request parameter to a number before using it in the `findOne` method.
- Use the `toBeInstanceOf`, `toHaveProperty`, and `toEqual` methods from Jest to write your assertions.

Chapter 4. Manipulating Data in a JSON Web Service – Part I

Chapter Overview

In this chapter, we will delve deeper into the concept of CRUD (Create, Read, Update, Delete) operations, a fundamental aspect of any web application. We will specifically focus on the creation and deletion of data using POST and DELETE endpoints. Through the use of TDD principles, we will explore how to build these endpoints and ensure their functionality with unit tests. By the end of this chapter, you will have a solid understanding of implementing and testing creation and deletion operations in a web API.

Learning Objectives

By the end of this chapter, you should be able to:

- Interpret the role of CRUD operations in web applications
- Determine the process for data creation using the POST endpoint and TDD
- Compare different methods for data removal, focusing on the DELETE endpoint using TDD
- Justify the use of TDD in ensuring robust CRUD operations

Understanding CRUD Operations

Create, Read, Update, and Delete (CRUD) operations form the backbone of any web application. They establish the primary ways in which we engage with data stored in various data sources, such as databases, files, and other APIs. As the acronym suggests, these four operations encompass the following:

- **Create:** This operation is responsible for generating or inserting new entries into the data source.
- **Read:** This operation involves retrieving a collection of entries from the data source.
- **Update:** This operation modifies existing data within the data source.
- **Delete:** This operation removes existing data from the data source.

These operations represent the essential interactions typically performed on a data source. Understanding them is crucial for effectively managing data within a data source.

CRUD operations are the basic building blocks of any data-driven application. Whether you are working on a social media platform, an online store, a booking system, or any other application that involves storing and manipulating data, you will be using CRUD operations. Understanding CRUD operations is crucial because they allow you to interact with data in meaningful ways. For example, when a new user registers in your application (Create), when you display a user's profile information (Read), when a user updates their profile information (Update), or when a user closes their account (Delete), you are using CRUD operations.

The idea behind CRUD operations is language agnostic. Meaning, it does not depend on a singular programming language or database system. CRUD operations can be used with SQL, NoSQL, Java, Python, C#, JavaScript, or any other language or database. Traces of CRUD operations can be found in virtually every web application. This makes them a fundamental concept in web development and programming in general.

In the next section, we will take a look at how to use TDD principles to build a create endpoint for data creation.

POST Endpoint: The TDD Way for Data Creation

In this section, we will explore the POST endpoint, which is used in CRUD operations. We will guide you through the process of building this endpoint using TDD principals, ensuring that the data creation API is functional, reliable, and robust.

Continuing where we left off from in the previous chapter's cookbook project:

1. Open the app.spec.js file in your editor.
2. Create a new test suite using the description: 'Chapter 4: API Tests'.
3. In the body of the test suite, add an HTTP POST unit test to check if a 201 status code is returned when adding a new recipe. Use the endpoint: `/api/recipes`.

The unit test should have the following checks:

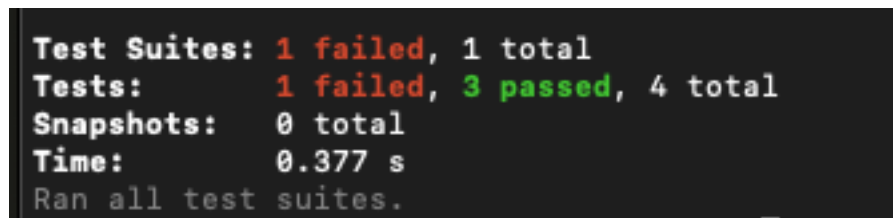
- a. 201 status code.

```
it("should return a 201 status code when adding a new recipe", async () => {  
  const res = await request(app).post("/api/recipes").send({  
    id: 99,  
    name: "Grilled Cheese",  
    ingredients: ["bread", "cheese", "butter"],  
  });  
  
  expect(res.statusCode).toEqual(201);  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes` POST endpoint. Let's dive into each part of the code to understand how it works.

- **it("should return a 201 status code when adding a new recipe", async() => { ... });** defines a unit test that checks if the `/api/recipes` POST endpoint returns a 201 status code for successful creations.
- **const res = await request(app).post("/api/recipes").send({ ... });** This sends a POST request to `/api/recipes` endpoint and waits for a response.
- **Expect(res.statusCode).toEqual(201);** This checks if the response status code is 201, indicating the request was successful.

To ensure our test fails, run `npm test` from a new CLI window.



```
Test Suites: 1 failed, 1 total  
Tests:      1 failed, 3 passed, 4 total  
Snapshots:  0 total  
Time:       0.377 s  
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Create a new POST endpoint for `/api/recipes`. In the body of the endpoint, add a try-catch block for unexpected errors, insert a new recipe into our mock database, and return a 201 status code with the newly inserted recipes ID.

```
app.post("/api/recipes", async (req, res, next) => {  
  try {  
    const newRecipe = req.body;  
  
    const result = await recipes.insertOne(newRecipe);  
    console.log("Result: ", result);  
    res.status(201).send({ id: result.ops[0].id});  
  } catch (err) {  
    console.error("Error: ", err.message);  
    next(err);  
  }  
});
```

In this code example, we define a POST endpoint at `/api/recipes` in our Express application. This endpoint retrieves inserts a new recipe into our mock database and sends back a status code of 201 and the newly inserted recipe ID. Let's take a closer look at the functionality of this API.

- **app.post("/api/recipes", async (req, res, next) => { ... });** This line sets up the asynchronous POST endpoint at `/api/recipes`.
- **const newRecipe = req.body;** This assigns the request body to a variable named `newRecipe`.
- **const result = await recipes.insertOne(newRecipe);** This line of code calls the `insertOne` method to insert a new recipe into the mock database, using the `newRecipe` object. An explanation of the keyword `await` was provided in the previous chapter.
- **res.status(201).send({ id: result.ops[0].id });** This line of code returns a status code of 201 and the ID of the newly created recipe. The `ops` property, is an array of objects that contains the object we just added to the mock database. Had we inserted multiple recipes, the array would contain an array of recipes. The other two properties being returned are `ok` (flag set to 1) and `n` (flag set to 1). The `ok` property represents whether the insertion was successful and the `n` property represents how many objects were affected in the mock database. For the purpose of this API we

are only concerned with the inserted recipes ID. The other properties are there to mimic the behavior of MongoDB.

Rerun the unit test to ensure its passing name (npm test):

```
PASS test/app.spec.js
  Chapter 3: API Tests
    ✓ it should return an array of recipes (23 ms)
    ✓ should return a single recipe (2 ms)
    ✓ should return a 400 error if the id is not a number (1 ms)
  Chapter 4: API Tests
    ✓ should return a 201 status code when adding a new recipe (5 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.28 s, estimated 1 s
Ran all test suites.
```

Let's also try testing this endpoint manually, using different recipe property values to see how our API behaves. Ensure your server is running and open a second CLI window and run the following Node.js command:

```
node -e "http.request('http://localhost:3000/api/recipes', { method: 'POST', headers: {'Content-Type': 'application/json'}}, (res) => { res.setEncoding('utf8').once('data', console.log.bind(console, res.statusCode))}).end(JSON.stringify({invalidField: 'invalidFieldValue'}))"
```

Notice the output:

```
201 {}
```

In the same CLI window, run the following Node.js command:

```
node -e "http.get('http://localhost:3000/api/recipes', (res) => res.setEncoding('utf-8').once('data', console.log))"
```

And, notice the output:

```
[
  {
    "id": 1,
    "name": "Pancakes",
    "ingredients": [
      "flour",
```

```
    "milk",
    "eggs"
  ]
},
{
  "id": 2,
  "name": "Spaghetti",
  "ingredients": [
    "pasta",
    "tomato sauce",
    "ground beef"
  ]
},
{
  "id": 3,
  "name": "Chicken Salad",
  "ingredients": [
    "chicken",
    "lettuce",
    "tomatoes",
    "cucumber"
  ]
},
{
  "id": 4,
  "name": "Beef Stew",
  "ingredients": [
    "beef",
    "potatoes",
    "carrots",
    "peas"
  ]
},
{
  "id": 5,
  "name": "Fish Tacos",
  "ingredients": [
    "fish",
    "tortillas",
    "avocado",
    "salsa"
  ]
},
{
```



```
"invalidField": "invalidFieldValue"
}
]
```

This highlights a flaw in our API. We are assuming the client will always send us the correct data to insert into our mock database. However, as our tests demonstrate, this may not always be the case. We need some way to ensure that the records we add to our mock database meet the data schema requirements of our database. That is, objects include an id, name, and ingredients.

1. Open the app.spec.js file.
2. In the body of the ‘Chapter 4: API Tests’, add a unit test that checks if a 400 error is returned with a message of ‘Bad Request’ when adding a new recipe with missing name.

The unit test should have the following checks:

- a. 400 status code.
- b. Response body message to equal ‘Bad Request’.

```
it("should return a 400 status code when adding a new recipe with missing
name", async () => {
  const res = await request(app).post("/api/recipes").send({
    id: 100,
    ingredients: ["bread", "cheese", "butter"]
  });

  expect(res.statusCode).toEqual(400);
  expect(res.body.message).toEqual("Bad Request");
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes` POST endpoint. Let’s dive into each part of this code to understand how it works.

- **it(“should return a 400 status code when adding a new recipe with missing name”, async () => { ... });** This defines a unit test that checks if the `/api/recipes` POST endpoint returns a 400 status code with a message of “Bad Request”.
- **const res = await request(app).post("/api/recipes").send({ ... });** This sends a POST request to the `/api/recipes` endpoint and waits for a response using the `supertest` npm package.

- **`expect(res.statusCode).toEqual(400);`** This line of code checks if the response status code is 400. Indicating a bad request.
- **`expect(res.body.message).toEqual("Bad Request");`** This line of code checks if the response body has a property named message with a value of "Bad Request".

To ensure our test fails, run `npm test` again.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 4 passed, 5 total
Snapshots:  0 total
Time:       0.328 s
Ran all test suites.
```

Passing Test (Green):

1. Open the `app.js` file.
2. Add validation to the POST endpoint that checks if the request body only contains properties for id, name, and ingredients.

```
app.post("/api/recipes", async (req, res, next) => {
  try {
    const newRecipe = req.body;

    const expectedKeys = ["id", "name", "ingredients"];
    const receivedKeys = Object.keys(newRecipe);

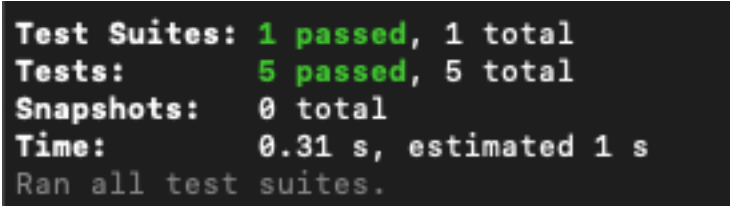
    if (!receivedKeys.every(key => expectedKeys.includes(key)) ||
receivedKeys.length !== expectedKeys.length) {
      console.error("Bad Request: Missing keys or extra keys", receivedKeys);
      return next(createError(400, "Bad Request"));
    }

    const result = await recipes.insertOne(newRecipe);
    console.log("Result: ", result);
    res.status(201).send({ id: result.ops[0].id});
  } catch (err) {
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we add error checking to test if the request body only contains the required fields in our recipe object (id, name, and ingredients). Any other fields or missing files will be rejected with a 400 status code and an error message of “Bad Request.” Let’s take a closer look at how we implemented this code.

- **const expectedKeys = ["id", "name", "ingredients"];** This line of code creates an array containing only the properties that are allowed for a recipe object.
- **const receivedKeys = Object.keys(newRecipe);** This line of code retrieves all of the keys from the request body (property values). We can use this to verify whether the properties being sent in the request body match the allowable fields in our expected keys array.
- **if (!receivedKeys.every(key => expectedKeys.includes(key)) || receivedKeys.length !== expectedKeys.length) { ... }** This if statement checks if the fields from the request body match the allowable fields in the expected keys array and whether the length of the fields being sent in the request body match the number of fields in the expected keys array. Using this information, we can determine if the client is sending more fields than what we are expected in the mock database.

Let’s rerun the test command (**npm test**) to confirm that our unit tests now pass.



```
Test Suites: 1 passed, 1 total
Tests:      5 passed, 5 total
Snapshots:  0 total
Time:       0.31 s, estimated 1 s
Ran all test suites.
```

And, just to be sure, let’s rerun our CLI commands to test the functionality of our API. Ensure your server is running and in another CLI window run the following Node.js command:

```
node -e "http.request('http://localhost:3000/api/recipes', { method: 'POST', headers: {'Content-Type': 'application/json'}}, (res) => { res.setEncoding('utf8').once('data', console.log.bind(console, res.statusCode))}).end(JSON.stringify({invalidField: 'invalidFieldValue'}))"
```

You should see the following output:

```
400 {"type":"error","status":400,"message":"Bad Request"}
```

In the next section, we will take a look at how to use TDD principles to build an API for data removal.

DELETE Endpoint: The TDD Way for Data Removal

In the final section of this chapter we focus on the DELETE endpoint, a crucial part of data removal in CRUD operations. We will demonstrate how to build and test this endpoint using TDD, providing a practical example of how to ensure the reliability of our data removal functionality.

Continuing where we left off from in the cookbook project:

1. Open the `app.spec.js` file in your editor.
2. In the body of the test suite for chapter 4, add a unit test to check if 204 status code is returned when deleting a recipe.

The unit test should have the following checks:

- a. A 204 status code.

```
it("should return a 204 status code when deleting a recipe", async () => {  
  const res = await request(app).delete("/api/recipes/99");  
  
  expect(res.statusCode).toEqual(204);  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes/:id` DELETE endpoint. Let's dive into each part of the code to understand how it works.

- **it("should return a 204 status code when deleting a recipe", async () => { ... });** This defines a unit test that checks if the `/api/recipes/:id` endpoint returns a 204 status code when deleting recipe from our mock database.
- **expect(res.statusCode).toEqual(204);** This line of code checks if the response status code is 204, indicating the request was successful.

You might be wondering, what about data validation? To keep the focus strictly on how to implement a DELETE endpoint, data validation, like numerical versus a string ID are omitted from this API. If this were a “real” production application, you would need to add validation checks to ensure the validity of the ID being passed in the request params object. Strategies for how to add data validation to an API have been covered in other sections of this book.

To ensure our tests fail, run `npm test` from a new CLI window.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 5 passed, 6 total
Snapshots:  0 total
Time:       0.297 s, estimated 1 s
Ran all test suites.
```

Passing Test (Green):

1. Open the `app.js` file.
2. Create a new DELETE endpoint for `/api/recipes/:id`. In the body of this endpoint, add a try-catch block for unexpected errors, delete a recipe from the mock database, and return a 204 status code, indicating a successful deletion.

```
app.delete("/api/recipes/:id", async (req, res, next) => {
  try {
    const { id } = req.params;
    const result = await recipes.deleteOne({ id: parseInt(id) });
    console.log("Result: ", result);
    res.status(204).send();
  } catch (err) {
    if (err.message === "No matching item found") {
      return next(createError(404, "Recipe not found"));
    }

    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we define a DELETE endpoint at `/api/recipes/:id` in our Express application. This endpoint deletes a specific recipe by its ID from our mock database and sends back a 204 status code to the client. Let's take a closer look at the functionality of this API.

- **`app.delete("/api/recipes/:id", async (req, res, next) => { ... });`** This line sets up the asynchronous DELETE endpoint at `/api/recipes/:id`.
- **`const result = await recipes.deleteOne({ id: parseInt(id) });`** This line of code calls the `deleteOne` method from the mock database and removes the recipe.

- **res.status(204).send();** This line of code returns a status code of 204 to the client, indicating the record was removed from our mock database.

Rerun the unit tests to ensure they are passing (**npm test**).

```
Test Suites: 1 passed, 1 total
Tests:      6 passed, 6 total
Snapshots:  0 total
Time:       0.305 s, estimated 1 s
Ran all test suites.
```

Hands-On 4.1: Manipulating Data in your Web Service

In this hands-on project, you will be extending the APIs for the in-n-out-books project using a mock database. You will be applying Test-Driven Development (TDD) principles and using Jest for testing.

Instructions:

1. Continue working in the project folder from the previous chapter's hands-on project.
2. In your project, ensure you have the following file structure:

```
src
  app.js
test
  app.spec.js
package.json
database
  books.js
  collection.js
```

3. You will be building the following routes in your app.js file:
 - a. A POST route at `/api/books` that adds a new book to the mock database and returns a 201-status code. Use a try-catch block to handle any errors, including checking if the book title is missing and throwing a 400 error if it is. Include a message that is appropriate for the 400 error.
 - b. A DELETE route at `/api/books/:id` that deletes a book with the matching id from the mock database and returns a 204-status code. Use a try-catch block to handle any errors.

4. Write unit tests for each of these routes using Jest in your `app.spec.js` file. You should write the tests before you implement the routes, following TDD principles. Create a test suite named “Chapter [Number]: API Tests”. Use the following test cases:
 - a. Should return a 201-status code when adding a new book.
 - b. Should return a 400-status code when adding a new book with missing title.
 - c. Should return a 204-status code when deleting a book.

Grading:

You will earn 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- When writing your tests, remember to use the `request` function from the `supertest` library to simulate HTTP requests to your Express app. You can use the `.post()` and `.delete()` methods to the returned object to simulate POST and DELETE requests, respectively.
- In the POST route test, send a book object with `id`, `title`, and `author` properties using `.send()`.
- In the DELETE route test, include the `id` of the book to delete in the URL.

Chapter 5. Manipulating Data in a JSON Web Service – Part II

Chapter Overview

In this chapter, we will delve into a critical element of web application development: the modification of data through the PUT endpoint. Through the use of TDD principles, we will examine the construction of this endpoint, ensuring the dependability and scalability of our cookbook application. By the end of this chapter, you will have a solid understanding of how to implement and test a PUT endpoint for updating data in a web application.

Learning Objectives

By the end of this chapter, you should be able to:

- Define the role of the PUT endpoint in CRUD operations
- Explain the process of data updating using the PUT endpoint and TDD
- Recommend best practices for implementing data updating in web applications
- Evaluate the effectiveness of TDD in ensuring robust data updating operations

PUT Endpoint: The TDD Way for Data Updating

In this section, we will delve into the PUT endpoint, which is integral to data updating in CRUD operations. We will guide you through the process of building this endpoint using TDD principles, providing a practical example of how to ensure the reliability of data in a JSON web service.

Continuing where we left off from in the cookbook project:

1. Open the `app.spec.js` file.
2. Add a new test suite for this chapter's unit tests. Follow the naming conventions we used in the previous chapters.
3. In the body of the test suite, add a unit test that checks if a 204 status code is returned when a recipe is successfully updated.

The unit test should have the following checks:

- a. 204 status code

```
it("should return a 204 status code when updating a recipe", async () => {  
  const res = await request(app).put("/api/recipes/1").send({  
    name: "Pancakes",  
    ingredients: ["flour", "milk", "eggs", "sugar"]  
  })  
  
  expect(res.statusCode).toEqual(204);  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes/:id` PUT endpoint. Let's dive into each part of the code to understand how it works.

- **it("should return a 204 status code when updating a recipe", async () => { ... });** This defines a unit test that checks if the `/api/recipes/:id` PUT endpoint returns a 204 status code for successful updates.
- **const res = await request(app).put("/api/recipes/1").send({ ... });** This sends a PUT request to `/api/recipes/:id` endpoint and waits for a response, using the `supertest` npm package.
- **expect(res.statusCode).toEqual(204);** This checks if the response status code is 204, indicating the request was successful.

To ensure our test fails, run `npm test` from a new CLI window.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 6 passed, 7 total
Snapshots:  0 total
Time:       0.417 s
Ran all test suites.
```

Passing Test (Green):

1. Open the `app.js` file.
2. Create a new PUT endpoint for `/api/recipes/:id`. In the body of the endpoint, add a try-catch block for unexpected errors, call the mock databases `updateOne()` method using the request params values, and return a 204 status code.

```
app.put("/api/recipes/:id", async (req, res, next) => {
  try {
    let id = Number(req.params.id)
    let recipe = req.body;

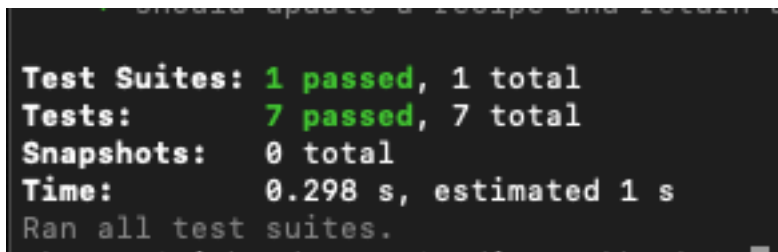
    console.log("ID: ", id);
    console.log("Recipe: ", recipe);

    const result = await recipes.updateOne({ id: id }, recipe);
    console.log("Result: ", result);
    res.status(204).send();
  } catch (err) {
    if (err.message === "No matching item found") {
      console.log("Recipe not found", err.message)
      return next(createError(404, "Recipe not found"));
    }
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we define a PUT endpoint at `/api/recipes/:id` in our Express application. This endpoint updates a specific recipe by its ID from our mock database and sends back a 204 status code to the client. Let's take a look at the functionality of this API.

- **app.put("/api/recipes/:id", async (req, res, next) => { ... });** This line sets up the asynchronous PUT endpoint at `/api/recipes/:id`.
- **const result = await recipes.updateOne({ id: id }, recipe);** This line of code called the mock databases `updateOne()` method to update a recipe by its ID.
- **res.status(204).send();** This line of code sends a 204 status code, indicating the request was successful.

Rerun the unit test to ensure its passing (npm test).



```
Test Suites: 1 passed, 1 total
Tests: 7 passed, 7 total
Snapshots: 0 total
Time: 0.298 s, estimated 1 s
Ran all test suites.
```

Let's also try testing this endpoint manually, using different parameter values to see how fault tolerate our endpoint is. Ensure your server is running and open a second CLI window and run the following Node.js command to test a different use case:

```
node -e "http.request('http://localhost:3000/api/recipes/foo', { method: 'PUT',
headers: {'Content-Type': 'application/json'}}, (res) => { res.setEncoding('utf-
8').once('data', console.log.bind(console,
res.statusCode))}).end(JSON.stringify({name: 'Test Recipe', ingredients: ['test',
'test']}))"
```

This command sends a PUT request to the `/api/recipes/:id` endpoint. Since foo is not a numerical value, you should see an error:

404 {"type":"error","status":404,"message":"Recipe not found"}

This highlights the first flaw in our API. We are assuming that the client will always send numerical values for the ID. However, as our test demonstrates, this may not always be the case.

1. Open the `app.spec.js` file.
2. In the body of this chapter's test suite, add a unit test that checks if a 400 error is return when the ID is not a number.

The unit test should have the following checks:

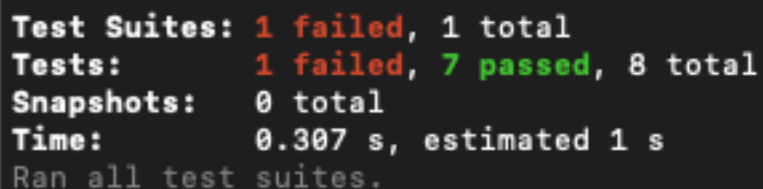
- a. 400 status code.
- b. Response body message to equal "Input must be a number"

```
it("should return a 400 status code when updating a recipe with a non-numeric id", async () => {  
  const res = await request(app).put("/api/recipes/foo").send({  
    name: "Test Recipe",  
    ingredients: ["test", "test"]  
  });  
  
  expect(res.statusCode).toEqual(400);  
  expect(res.body.message).toEqual("Input must be a number");  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/recipes/:id` endpoint. Let's dive into each part of this code to understand how it works.

- **it("should return a 400 status code when updating a recipe with a non-numeric id", async () => { ... });** This defines a unit test that checks if the `/api/recipes/:id` endpoint returns a 400 status code when the ID is not a number.
- **const res = await request(app).put("/api/recipes/foo").send({ ... });** This sends a PUT request to the `/api/recipes/:id` endpoint and waits for a response using the `supertest` npm package.
- **expect(res.statusCode).toEqual(400);** This checks if the response status code is 400. Indicating a Bad Request.
- **expect(res.body.message).toEqual("Input must be a number");** This checks if the response message is "Input must be a number".

To ensure our test tails, run `npm test`.



```
Test Suites: 1 failed, 1 total  
Tests:      1 failed, 7 passed, 8 total  
Snapshots:  0 total  
Time:       0.307 s, estimated 1 s  
Ran all test suites.
```

Passing Test (Green):

1. Open the `app.js` file.

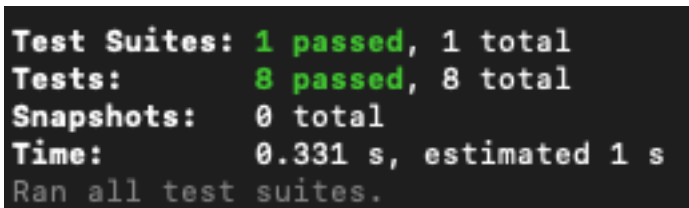
2. Add validation to the PUT endpoint that checks if the request params ID is a numerical value. If it's not, generate a 400 error and pass it to our middleware error handler.

```
app.put("/api/recipes/:id", async (req, res, next) => {
  try {
    let { id } = req.params;
    let recipe = req.body;
    id = parseInt(id);

    if (isNaN(id)) {
      return next(createError(400, "Input must be a number"));
    }

    const result = await recipes.updateOne({ id: id }, recipe);
    console.log("Result: ", result);
    res.status(204).send();
  } catch (err) {
    if (err.message === "No matching item found") {
      console.log("Recipe not found", err.message)
      return next(createError(404, "Recipe not found"));
    }
    console.error("Error: ", err.message);
    next(err);
  }
});
```

Let's rerun the test command (`npm test`) to confirm that our unit test now passes.



```
Test Suites: 1 passed, 1 total
Tests:      8 passed, 8 total
Snapshots:  0 total
Time:       0.331 s, estimated 1 s
Ran all test suites.
```

In this chapter and in previous chapters we've briefly explored data validation, by checking the request params object for numerical values and checking the request body for fields only associated with what our database is expecting. This type of data validation is referred to as "Input Validation." We use this type of validation to prevent "Parameter Pollution."

HTTP Parameter Pollution (HPP) is a technique where someone tries to confuse a website by adding extra information into the website's address. For instance, if a website's address

is `http://dcheroes.com?superhero=batman`, they might change it to `http://dcsuperheroes.com?superhero=batman&superhero=badvalue`. If the website isn't set up to handle this, it can cause unexpected issues. The person doing this might be able to change settings on the website, access information they shouldn't, or even affect the website's database. This technique can also lead to other types of attacks, like causing unexpected pop-ups to appear on the website (Cross-Site Scripting) or manipulating the website's database (SQL Injection). Preventing HPP attacks from occurring, requires awareness and thorough testing. As a general rule of thumb, you should always validate: query string parameters, request parameters, request bodies, and response bodies.

1. Open the `app.spec.js` file.
2. In the body of this chapter's test suite, add a unit test that checks if a 400 error is returned when updating a recipe with missing or extra keys.

The unit test should have the following checks:

- a. 400 status code.
- b. Response body message to equal "Bad Request"

it("should return a 400 status code when updating a recipe with missing keys or extra keys", async () => {

```
  const res = await request(app).put("/api/recipes/1").send({  
    name: "Test Recipe"  
  });
```

```
  expect(res.statusCode).toEqual(400);  
  expect(res.body.message).toEqual("Bad Request");
```

```
  const res2 = await request(app).put("/api/recipes/1").send({  
    name: "Test Recipe",  
    ingredients: ["test", "test"],  
    extraKey: "extra"  
  });
```

```
  expect(res2.statusCode).toEqual(400);  
  expect(res2.body.message).toEqual("Bad Request");  
});
```

To ensure our tests fail, run `npm test`.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 8 passed, 9 total
Snapshots:  0 total
Time:       0.312 s, estimated 1 s
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Add validation to the PUT endpoint that checks the request body for only valid recipe fields. If validation fails, generate a 400 error and pass it to our middleware error handler.

```
app.put("/api/recipes/:id", async (req, res, next) => {
  try {
    let { id } = req.params;
    let recipe = req.body;
    id = parseInt(id);

    if (isNaN(id)) {
      return next(createError(400, "Input must be a number"));
    }

    const expectedKeys = ["id", "name", "ingredients"];
    const receivedKeys = Object.keys(recipe);

    if (!receivedKeys.every(key => expectedKeys.includes(key)) ||
receivedKeys.length !== expectedKeys.length) {
      console.error("Bad Request: Missing keys or extra keys", receivedKeys);
      return next(createError(400, "Bad Request"));
    }

    const result = await recipes.updateOne({ id: id }, recipe);
    console.log("Result: ", result);
    res.status(204).send();
  } catch (err) {
    if (err.message === "No matching item found") {
      console.log("Recipe not found", err.message);
      return next(createError(404, "Recipe not found"));
    }
    console.error("Error: ", err.message);
    next(err);
  }
}
```

});

The validation we are using to handle the expected versus missing keys is identical to the code we used in the HTTP POST request. Let's rerun the test command (`npm test`) to confirm that our tests pass now.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 8 passed, 9 total
Snapshots:  0 total
Time:       0.32 s, estimated 1 s
Ran all test suites.
```

Why didn't our tests pass? Looking over the output, we can see that this unit test passed, but our unit test for "should update a recipe and return a 204 status code" failed:

```
● Chapter 5: API Tests › should update a recipe and return a 204 status code

expect(received).toEqual(expected) // deep equality

Expected: 204
Received: 400

   65 |         ingredients: ["test", "test"]
   66 |       })
>  67 |       expect(res.statusCode).toEqual(204); // Check if the status code
      |                                     ^
   68 |     });
   69 |
```

Well, why is that? The code in this unit test is sending fields for `name` and `ingredients`. But, the code in our API is expecting keys for `id`, `name`, and `ingredients`. In other words, we are missing the `id` field. We can resolve this by simply removing the `id` field from the expected keys array; however, it is important to highlight that this example shows why unit tests are so important to us as software developers. If we did not have unit tests, we would have never known that the code we introduced for HPP broke the update recipe API for successful attempts. Go ahead and remove the `id` key from the expected keys array and rerun the unit tests (`npm test`).

```
Test Suites: 1 passed, 1 total
Tests:      9 passed, 9 total
Snapshots:  0 total
Time:       0.332 s, estimated 1 s
Ran all test suites.
```

As we conclude this chapter, there is an important point to highlight about the structure of our last unit test. In the test case we are checking two different scenarios: updating a recipe with missing keys and updating a recipe with extra keys. While this test case does its job, it is generally a better practice to separate these into two distinct tests.

Why is this important? Each unit test should ideally focus on a single behavior of the system. This way, if a test fails, we can immediately pinpoint what behavior isn't working as expected. This greatly simplifies the debugging process and makes our unit tests easier to maintain. So, in our case, we should have one unit test that checks the response when updating a recipe with missing keys, and another test that checks the response when updating a recipe with extra keys. This would make the test purpose clearer and our test suite easier to maintain. Well-structured tests are just as important as well-structured code.

Hands-On 5.1: Manipulating Data in your Web Service

In this hands-on project, you will be extending the APIs for the in-n-out-books project using a mock database. You will be applying Test-Driven Development (TDD) principles and using Jest for testing.

Instructions:

1. Continue working in the project from the previous chapter's hand-on project.
2. In your project, ensure you have the following file structure:

```
src
  app.js
test
  app.spec.js
package.json
database
  books.js
  collection.js
```

3. You will be building the following routes in your app.js file:
 - a. A PUT route at /api/books/:id that updates a book with the matching id in the mock database and returns a 204-status code. Use a try-catch block to handle any errors, including checking if the book title is missing and throwing a 400 error if it is with an applicable error message.
 - b. Add error handling to check if the id is not a number and throw a 400 error if it is not with an applicable error message.

4. Write unit tests for this route using Jest in your `app.spec.js` file. You should write the tests before you implement the routes, following TDD principles. Create a test suite named “Chapter [Number]: API Tests”, using the following test cases:
 - a. Should update a book and return a 204-status code.
 - b. Should return a 400-status code when using a non-numeric id.
 - c. Should return a 400-status code when updating a book with a missing title.

Grading:

You will earn 20 points for each passing test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- Use Jest’s `toEqual` method to check if the status code returned by the PUT request is as expected (204 for successful update, 400 for errors).
- In the test for non-numeric id, send a PUT request to `/api/books/foo`. Jest should expect a 400-status code and the error message “Input must be a number”.
- In the test for updating a book with a missing title, send a PUT request to `/api/books/1` with a body that only includes an author. Jest should expect a 400-status code and the error message “Bad Request”.

Chapter 6. Securing Your API – Part I

Chapter Overview

In this chapter, we will take a look at the critical aspects of authentication and authorization in web applications. We will explore password hashing using `bcryptjs`, a crucial component for secure authentication. Additionally, we will implement login functionality using Express and TDD principles. By the end of this chapter, you should have a solid understanding how to implement and test authentication in a web application.

Learning Objectives

By the end of this chapter, you should be able to:

- Interpret the concepts of Authentication and Authorization in web applications
- Analyze the process of password hashing using `bcryptjs`
- Decide on the best practices for implementing login functionality using TDD and Express
- Assess the security and reliability of the implemented login functionality

Understanding Authentication and Authorization

In this section, we will be exploring the concepts of authentication and authorization in the context of web development. We will examine specific scenarios where these concepts are utilized and highlight the distinctions between them. This will enhance your understanding of these concepts and prepare you for the upcoming sections, where we will focus on practical implementation of these features through coding exercises.

When it comes to securing user accounts in a web application, there are two fundamental types of security: Authentication and Authorization. Authentication involves verifying the user's identity, just like checking if the person at your door is who they say they are, such as Batman, or an imposter like Joker trying to gain unauthorized access to the Batcave. Here are some scenarios where this concept is utilized:

1. **Login Systems:** Authentication is commonly used in login systems or web applications that require users to sign in to access the systems functionalities. Upon entering their login credentials, such as their username and password, the server verifies their credentials by comparing the entered credentials to the stored database. If the entered credentials match the stored ones, the user is authenticated and granted access. However, if the entered credentials do not match, the system usually displays an error message informing the user that they lack sufficient privileges to access the web application.
2. **Two-Factor Authentication (2FA):** The modern approach (more secure), is to use two-factor authentication when verifying users. After entering their password, the user is required to provide another form of authentication, such as an SMS notification or email with pin code. This “second layer of authentication” adds an additional layer of security, ensuring the user is who they claim to be.

Authorization, on the other hand, controls what authenticated users can do in your application. For instance, even though Robin is a trusted sidekick, they may not have the same level of access to all areas of the Batcave as Batman. Authorization is the process that checks if Robin, despite being authenticated, is allowed to access the Batmobile or Batman's personal arsenal. Here are some scenarios where this concept is utilized:

1. **Access Control:** Once a user is authenticated, they may not have access to the entire system or resources. Authorization is used to determine what resources or sections in the web application that the authenticated user can access. For example, in a file system, a user may be authorized to read and write their own files, but only read other user files.
2. **Role-Based Access Control (RBAC):** In many systems, users are assigned roles that determine what actions they are allowed to perform within a system. It's like being part of a club where not everyone in the club is allowed to do the same thing.

Take for example a website for stargazers. If you are a “Moderator,” you’ve got the power to make, change, or remove any observation reports or star charts on the site. But, if you’re just a “Member,” you can only make and tweak your own reports and charts. This what we call “authorization.” It’s all about what you’re allowed to do in the site, based on the role you’ve been given.

There are several popular npm packages that can help with authentication and authorization in Node.js applications:

1. **passport.js:** This npm package is very popular for user authentication in Node.js applications. It supports a variety of strategies like, username and password (locally saved to a database), OAuth, OpenID, and many others.
2. **bcryptjs (covered in the next section):** This npm package is very useful for hashing and checking locally saved passwords, which is a crucial part of user authentication. This helps to ensure the passwords being saved to a database are not in plain text (readable).
3. **jsonwebtoken:** This npm package allows you to work with JSON Web Tokens (JWT), which is used heavily in API-driven web applications. JWT is typically used for stateless and sessionless authentication. Once a user is signed into a web application, a JWT is generated and sent with every request to the web server.
4. **express-jwt:** This npm package is a middleware package that protects your routes with JWT, which ensures that a valid JWT is present in the request header before the route handler is executed.
5. **casl:** This is a versatile library for handling authorization in Node.js applications. It allows you to define what actions a user can or cannot perform in a web application.

A word of caution. While these npm packages can assist with user authentication and authorization, they are not a “one size fits all” solution. And they do not absolve you from understanding the underlying principles of how to properly authenticate and authorize users in a web application. Always follow industry standard best practices, when it comes to handling user data and permissions. In the next section, we will delve into password hashing and the bcryptjs npm package.

Exploring Password Hashing with bcryptjs

In this section, we will explore password hashing, a critical aspect of secure authentication. We will use bcryptjs, a powerful library for hashing passwords, and guide you through its usage, ensuring you understand how to securely store user passwords.

To secure passwords, a security technique called hashing is employed. In this procedure, a password is converted into a distinct, fixed-length string of characters (the hash) using an algorithm (the hash function). Since this is a one-way transformation, the hash cannot be used to determine the original password. The system generates and saves a hash of the password whenever the user opens an account or modifies their password. The password is hashed once more during the login process, and it's compared to the hash that was previously stored in the system. The password is correct if the hashes match.

Imagine you are a character in the DC universe, and you have a secret identity (your password) that you want to keep safe. Instead of risking exposure to your secret identity, you decide to transform into a unique superhero (the hash) using a special transformation process. This transformation process is unique because it can turn your secret identity into a superhero, but it cannot turn the superhero back into your secret identity. So even if a villain learns this process, they can't figure out who you really are. When you need to prove your identity to the Justice League, you don't reveal your secret identity. Instead, you transform into your unique superhero form. The Justice League has seen this superhero form before and recognizes it, so they know it's really you, without you ever having to reveal your secret identity. That is what password hashing is doing. It keeps your password safe by transforming it into a unique value and only storing and checking that value, not the actual password itself. This way, if some malicious person gets your hashed password, they cannot figure out what your real password is.

`bcryptjs` is a popular npm package used for hashing passwords in Node.js applications. It's a pure JavaScript implementation of the BCrypt password hashing algorithm. Some of the key features for the `bcryptjs` package are:

1. **`bcrypt.hash(password, saltOrRounds)`**: This function takes two actual parameters: a password to hash and either a salt or number of rounds to use when generating a salt. A salt is a random value that is used to ensure the hash that is generated is unique. This way, if two users had the same password, their hashes would be unique (won't match).
2. **`bcrypt.compare(password, hash)`**: This function takes two actual parameters: the user's actual password and the stored hashed value. It hashes the password and compares it to the hashed value. If they match, the password is correct. If they don't match, then it means the password does not match the originally stored hashed value.
3. **`bcrypt.genSalt(saltRounds)`**: This function is used to generate a salt. It takes the number of salt rounds as an actual parameter and returns the salt. The number of rounds determines how complex the salt will be. The higher the number of rounds, the more secure the hash will be. As a general rule of thumb, 10 rounds is a good starting point. However, it should be pointed out that, this number could change over time as computers get faster and new research becomes available.

4. **bcrypt.getRounds(hash):** This function is used to get the number of rounds used to generate the hashed value. If 10 rounds were used during the hashing process, the number 10 would be returned from this function.

In the realm of cryptography and security, I recommend sticking with well-established libraries rather than crafting your own. Using a library like bcryptjs is preferable to creating your own BCrypt algorithm, because your custom version could be susceptible to human or programming errors. When you employ a library like this, you are leveraging a tired-and-tested tool that has already undergone extensive vetting. In matters of security, it's advisable to lean towards using libraries that have been thoroughly examined and approved, rather than developing your own implementation. Here are some things to consider:

1. **Security Assurance:** Cryptography concepts are extremely complex and require a great deal of understanding in many different disciplines. Even a small mistake can introduce vulnerabilities that could be exploited by experienced attackers.
2. **Performance Optimization:** Libraries like bcryptjs have been fine-tuned with performance in mind, through many years of development efforts. Building a custom solution would lose this benefit, leading to potential problems.
3. **Simplified Maintenance:** Maintaining a custom cryptographic algorithm can be a significant burden. It requires staying up-to-date with the latest security vulnerabilities, researching the latest trends in cryptography, and constantly testing and adjusting the algorithm to ensure it remains secure and efficient. With a custom library, like bcryptjs, there is a dedicated team of developers that handle this.
4. **Peer Verification:** Libraries like bcryptjs are peer reviewed. Meaning, experts in the field of cryptography have reviewed the libraries. This level of scrutiny helps to ensure the library is secure, resilient, and working as expected. Do not underestimate the power of peer verification. Of the items listed here, it is probably one of the most important aspects of why using a third-party library like bcryptjs is beneficial.

In the next section, we will take a look at how to implement account registration functionality in an Express application using TDD principles.

TDD and Express: Implementing Registration Functionality

In the final section of this chapter, we will focus on the implementation of account registration functionality using Express and TDD principles. We will guide you through the process of building and testing this crucial feature, providing a practical example of how to ensure the reliability and security of your login API.

Continuing with where we left off in the cookbook project:

1. Open the `app.spec.js` file in your editor.
2. Add a new test suite for this chapter's unit tests. Follow the naming conventions we used in the previous chapters.
3. In the body of the test suite, add a unit test to check if a 200 status code is returned with a message of "Registration successful" when registering a new user.

The unit test should have the following checks:

- a. 200 status code.
- b. The response body should have a message of "Registration successful".

```
it("should return a 200 status code with a message of 'Registration successful'
when registering a new user", async () => {
  const res = await request(app).post("/api/register").send({
    email: "cedric@hogwarts.edu",
    password: "diggory"
  });

  expect(res.statusCode).toEqual(200);
  expect(res.body.message).toEqual("Registration successful");
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/register` endpoint. Let's dive into each part of the code to understand how it works.

- **it("should return a 200 status code with a message of 'Registration successful' when registering a new user", async () => { ... });** This defines a unit test that checks if the `/api/register` endpoint registers a new user and returns a status code of 200 with a message of "Registration successful".
- **const res = await request(app).post("/api/register").send({ ... });** This sends a POST request to `/api/register` endpoint and waits for a response, using the `supertest` npm package.
- **expect(res.statusCode).toEqual(200);** This checks if the response status code is 200, indicating the request was successful.
- **expect(res.body.message).toEqual("Registration successful");** This checks if the response body has a message of "Registration successful".

To ensure our test fails, run `npm test` from a new CLI window.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 9 passed, 10 total
Snapshots:  0 total
Time:       0.411 s
Ran all test suites.
```

Passing Test (Green):

1. Add the `users.js` file from the `data-files` folder to your project's `database` folder. If you do not have access to the `data-files` folder, you will need to contact your instructor.
2. Open the `app.js` file.
3. Add a `require` statement for the `bcryptjs` file to the top of the file. We will use this library to hash user passwords using 10 salt rounds.
4. Create a new POST endpoint for `/api/register`. In the body of this endpoint, add a `try-catch` block for unexpected errors. Next, hash the user's password using the `bcryptjs` package, and then add them to the mock database.

```
app.post("/api/register", async (req, res, next) => {
  try {
    const { email, password } = req.body;

    const hashedPassword = bcrypt.hashSync(password, 10);

    const user = await users.insertOne({
      email: email,
      password: hashedPassword
    });

    res.status(200).send({ user: user, message: "Registration successful" });

  } catch (err) {
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we define a POST endpoint at `/api/register` in our Express application. This endpoint, hashes the user's password using 10 salt rounds, adds them to our mock database, and sends a 200 status code back to the client with a message of "Registration successful". Let's take a closer look at the functionality of this API.

- **`app.post("/api/register", async (req, res, next) => { ... });`** This line sets up the asynchronous POST endpoint at `/api/register`.
- **`const hashedPassword = bcrypt.hashSync(password, 10);`** This line of code calls the `hashSync()` function from the `bcryptjs` library to hash the users password using 10 salt rounds.
- **`res.status(200).send({ user: user, message: "Registration successful"});`** This line of code sends the inserted user object back to the client with a message of "Registration successful" and a status code of 200.

Rerun the unit test to ensure they are passing (`npm test`).

```
Test Suites: 1 passed, 1 total
Tests:      10 passed, 10 total
Snapshots:  0 total
Time:       0.547 s
Ran all test suites.
```

We should also manually check this endpoint, using different values to see how well our endpoint handles problems. We will check the API with these tests:

1. Using email addresses that are already in our mock database.
2. Using too many or too few parameter values.

Ensure your server is running and open a second CLI window and run the following Node.js command:

```
node -e "
  http.request('http://localhost:3000/api/register',
    { method: 'POST', headers: {'Content-Type': 'application/json'}},
    (res) => { res.setEncoding('utf-8').once('data', console.log.bind(console,
res.statusCode))})
  .end(JSON.stringify({email: 'harry@hogwarts.edu', password: 'potter'}))
"
```

You should see something similar to the following message:

200

```
{"user":{"result":{"ok":1,"n":1},"ops":[{"email":"harry@hogwarts.edu","password":"$2a$10$TveDcqTdEA2luIXr/a9MP.GjU4LDRXzcV26FctxTy2LagxrOVWoMW"]}]}
```

The issue with this response is, we are adding another user to the mock database with an email address of harry@hogwarts.edu. While this is not a syntax error or system error, it is a logical error. Our API should not allow multiple users to register an account with the same email address. Let's also try testing for too many and too few parameter values. Enter the following Node.js commands:

Too many parameter values:

```
node -e "
  http.request('http://localhost:3000/api/register',
    { method: 'POST', headers: {'Content-Type': 'application/json'}},
    (res) => { res.setEncoding('utf-8').once('data', console.log.bind(console,
res.statusCode))})
    .end(JSON.stringify({email: 'cedric@hogwarts.edu', password: 'diggory', extra:
'extraValue'}))
"
```

You should see something similar to the following message:

200

```
{"user":{"result":{"ok":1,"n":1},"ops":[{"email":"cedric@hogwarts.edu","password":"$2a$10$VVvdTg/hNmJcXk6sESsr8uCbFaMcbR4ga47/PT19sKBi4IKOXEGPC"]}]}
```

Now, let's try too few parameter values:

```
node -e "
  http.request('http://localhost:3000/api/register',
    { method: 'POST', headers: {'Content-Type': 'application/json'}},
    (res) => { res.setEncoding('utf-8').once('data', console.log.bind(console,
res.statusCode))})
    .end(JSON.stringify({email: 'cedric@hogwarts.edu'}))
"
```

You should see something similar to the following message:

```
500 {"type":"error","message":"Illegal arguments: undefined, string"}
```

We will resolve each of the identified issues using TDD principles.

1. Open the app.spec.js file.

2. Add a new unit test to this chapter's test suite to check if a 409 status code is returned with the message "Conflict" when registering a user with a duplicate email address.

The unit test should have the following checks:

- a. 409 status code
- b. The response body message should be "Conflict"

```
it("should return a 409 status code with a message of 'Conflict' when registering
a user with a duplicate email", async () => {
  const res = await request(app).post("/api/register").send({
    email: "harry@hogwarts.edu",
    password: "potter"
  });

  expect(res.statusCode).toEqual(409);
  expect(res.body.message).toEqual("Conflict");
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/register` endpoint. Let's dive into each part of the code to understand how it works.

- **it("should return a 409 status code with a message of 'Conflict' when registering a user with a duplicate email", async () => { ... });** This defines a unit test that checks if the `/api/register` endpoint returns a 409 status code when registering a user with a duplicate email.
- **const res = await request(app).post("/api/register").send({ ... });** This sends a POST request to `/api/register` endpoint and waits for a response, using the `supertest` npm package.
- **expect(res.statusCode).toEqual(409);** This checks if the response status code is 409, indicating a conflict.
- **expect(res.body.message).toEqual("Conflict");** This checks if the response body has a message of "Conflict"

To ensure our test fails, run `npm test` from a new CLI window.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 10 passed, 11 total
Snapshots:  0 total
Time:       0.65 s, estimated 1 s
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Add validation to the POST endpoint that checks if the email address is already in use. If it is, generate a 409 error and pass it to our middleware error handler.

```
app.post("/api/register", async (req, res, next) => {
  console.log("Request body: ", req.body);
  try {
    const { email, password } = req.body;

    let duplicateUser;
    try {
      duplicateUser = await users.findOne({ email: email });
    } catch (err) {
      duplicateUser = null;
    }

    if (duplicateUser) {
      console.error("Conflict: User already exists");
      return next(createError(409, "Conflict"));
    }

    const hashedPassword = bcrypt.hashSync(password, 10);

    console.log("email: ", email);
    console.log("password: ", hashedPassword);

    const user = await users.insertOne({
      email: email,
      password: hashedPassword
    });

    res.status(200).send({ user: user, message: "Registration successful"});

  } catch (err) {
```

```
    console.error("Error: ", err);
    console.error("Error: ", err.message);
    next(err);
  }
});
```

In this code example, we add error handling to test if the email address is already in use. Let's take a closer look at what this code is doing:

- **let duplicateUser;** This line of code defines a variable for a potential duplicate user.
- **duplicateUser = await users.findOne({ email: email });** This line of code is wrapped in a try-catch block, because the `findOne` method from our mock database will return a rejection when "Not matching item found".
- **duplicateUser = null;** This line of code sets the `duplicateUser` variable to null for use cases when the email address is not in use.
- **if (duplicateUser) { ... }** This line of code checks the truthy of the `duplicateUser` variable. If it's not null, it means the email address is already in use and we should generate a 409 error to pass to our middleware error handler.

Let's rerun the test command (`npm test`) to confirm that our unit tests now pass.

```
Test Suites: 1 passed, 1 total
Tests:      11 passed, 11 total
Snapshots:  0 total
Time:       0.567 s, estimated 1 s
Ran all test suites.
```

Continuing with where we left off:

1. Open the `app.spec.js` file.
2. Add a unit test to this chapter's test suite that checks if a status code of 400 is returned with a message of "Bad Request" when using too many or too few parameter values.

The unit test should have the following checks:

- a. 400 status code
- b. Response body message of "Bad Request"

```

it("should return a 400 status code when registering a new user with too many
or too few parameter values", async () => {
  const res = await request(app).post("/api/register").send({
    email: "cedric@hogwarts.edu",
    password: "diggory",
    extraKey: "extra"
  });

  expect(res.statusCode).toEqual(400);
  expect(res.body.message).toEqual("Bad Request");

  const res2 = await request(app).post("/api/register").send({
    email: "cedric@hogwarts.edu"
  });

  expect(res2.statusCode).toEqual(400);
  expect(res2.body.message).toEqual("Bad Request");
});

```

To ensure our tests fail, run `npm test`.

```

Test Suites: 1 failed, 1 total
Tests:       1 failed, 11 passed, 12 total
Snapshots:   0 total
Time:        0.577 s, estimated 1 s
Ran all test suites.

```

Passing Test (Green):

1. Open the app.js file.
2. Add validation to the POST endpoint that checks if the request parameter values are too many or too few.

```

app.post("/api/register", async (req, res, next) => {
  console.log("Request body: ", req.body);
  try {
    const user = req.body;

    const expectedKeys = ["email", "password"];
    const receivedKeys = Object.keys(user);

```

```

    if (!receivedKeys.every(key => expectedKeys.includes(key)) ||
receivedKeys.length !== expectedKeys.length) {
      console.error("Bad Request: Missing keys or extra keys", receivedKeys);
      return next(createError(400, "Bad Request"));
    }

    let duplicateUser;
    try {
      duplicateUser = await users.findOne({ email: user.email });
    } catch (err) {
      duplicateUser = null;
    }

    if (duplicateUser) {
      console.error("Conflict: User already exists");
      return next(createError(409, "Conflict"));
    }

    const hashedPassword = bcrypt.hashSync(user.password, 10);

    const newUser = await users.insertOne({
      email: user.email,
      password: hashedPassword
    });

    res.status(200).send({ user: newUser, message: "Registration successful" });

  } catch (err) {
    console.error("Error: ", err);
    console.error("Error: ", err.message);
    next(err);
  }
});

```

Let's rerun the test command (`npm test`) to confirm that our unit tests now pass.

```

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        0.523 s, estimated 1 s
Ran all test suites.

```

Hands-On 6.1: Implementing User Authentication

In this hands-on project, you will be extending the APIs for the in-n-out-books project using a mock database. You will be applying Test-Driven Development (TDD) principles and using Jest for testing.

Instructions:

1. Continue working in the project folder from the previous chapter's hands-on project.
2. In your project, ensure you have the following file structure:

```
src
  app.js
test
  app.spec.js
package.json
database
  users.js
  books.js
  collection.js
```

The users.js file is located in the data-files folder. If you do not have access to this folder, contact your instructor.

3. You will be building the following route in your app.js file:
 - a. A POST route at /api/login that logs a user in and returns a 200-status code with 'Authentication successful' message. Use a try-catch block to handle any errors, including checking if the email or password is missing and throwing a 400 error if it is with an applicable message. Also, use the compareSync() method from the bcryptjs npm package to check if the password is valid. If it is not valid, throw a 401 error and a message of 'Unauthorized'.
4. Write unit tests for this route using Jest in your app.spec.js file. You should write the tests before you implement the routes, following TDD principles. Create a test suite named "Chapter [Number]: API Tests". Use the following test cases:
 - a. It should log a user in and return a 200-status with 'Authentication successful' message.
 - b. It should return a 401-status code with 'Unauthorized' message when logging in with incorrect credentials.

- c. It should return a 400-status code with 'Bad Request' when missing email or password.

Grading:

You will earn 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- Remember to use the `compareSync()` method from the `bcryptjs` npm package to check if the password is valid.
- Use the `toEqual` method from Jest to write your assertions.

Chapter 7. Securing Your API – Part II

Chapter Overview

In this chapter, we will continue our journey into securing your API by focusing on crucial aspect of user management – password reset functionality. We will discuss its importance, and look at a secure approach for its implementation, and emphasizes the role of TDD principles in building a secure and reliable password reset web API. By the end of this chapter, you should be able to explain the concept and importance of password reset functionality in web applications and build a JSON web API implementing password reset functionality through the use of TDD principles.

Learning Objectives

By the end of this chapter, you should be able to:

- Explain the concept and importance of password reset functionality in web applications
- Analyze a secure approach to implementing password reset functionality
- Justify the use of TDD in developing secure password reset functionality
- Evaluate the robustness and security of the implemented password reset functionality

Understanding Password Reset Functionality

In this section, we will discuss the importance of password reset functionality in web applications. We will explore why it's a critical feature for user management and how it impacts user experience and security. We will also look at common vulnerabilities that can arise in password reset functionality and how they can be exploited, emphasizing the need for a secure implementation.

The ability to reset passwords is a crucial feature in web applications, enabling users to modify their password if they've forgotten it or wish to enhance their accounts security. This feature is integral to many security-focused web applications, as users frequently forget their passwords. Without a system in place for password resetting, users would be unable to regain access to their accounts. Although the specific implementation may differ, the password reset process generally includes:

1. **Email-Based Reset:** This approach is among the most prevalent methods for password resetting. When a user initiates a password reset, an email is sent to them. This email contains a link that leads to a webpage where they can securely establish a new password. This method is widely adopted due to its simplicity and directness, ensuring a user-friendly experience.
2. **Security Questions:** In this approach, users are asked to answer security questions during the account registration process. If they answer the security questions correct, they are allowed to reset their password. In recent years, this method has become less popular because of the potential security risks it imposes.
3. **Two-Factor Authentication (2FA) Reset:** This method utilizes two different authentication steps to reset a password. The process is started on a device, like a computer, and a special validation code is then sent to another authentication medium, often a mobile phone. The user who has requested the password change must then enter this received code from their phone back into the computer. This approach improves security by forcing the user to confirm their identity via two separate authentication steps, thus adding an extra layer of security to their account. Examples of 2FA mediums includes: computer and mobile phone, website and email, mobile app and SMS, and website and authenticator app. The goal of 2FA is to use two different types of authentication from categories of something you know (like a password), something you have (like a computer or mobile device), and something you are (like a fingerprint or facial recognition).
4. **SMS-Based Reset:** In this approach, the system sends a unique verification code to the user's registered mobile number. The user then enters this code in the website where the reset password request was made. This is different from a 2FA reset, where the user has previously set up a second factor of authentication. This second factor could be a mobile phone, hardware token, or another method. In both cases, a unique code is used to verify the user's identity, but the method of delivery and the type of factor varies.

When selecting or designing a password reset process, it is important to consider the user experience. Here are some key points to consider:

1. **Simplicity:** The process for resetting a password should be easy and straightforward. Users should be able to initiate and complete the password reset

process with minimal effort and steps. In other words, keep the number of steps to reset a password short and efficient.

2. **Guidance:** Always provide clear instructions and guidance on each step of the password reset process. If a user needs to check their email or phone for a code, then you should let them know. Furthermore, password requirements (number of characters, numeric, special characters, etc.,) should be clearly communicated.
3. **Error Handling:** If there is an issue during the password reset process, display a message that provides a clear and helpful explanation of why the error occurred. Just make sure you do not disclose sensitive information in the error messages.
4. **Accessibility:** Make certain that the password reset process is designed to be accessible to all users, including those who utilize assistive technologies.
5. **Security:** Always make it a point to communicate to users about the measures taken to protect their information during the password reset process.
6. **Feedback:** Provide immediate feedback to users at every stage of the reset password process. It's essential to include a verification message once the password reset has been successfully completed. This confirmation message should be straightforward, concise, and easy to find. Furthermore, consider offering helpful suggestions during the process to assist users in creating robust and secure passwords. This approach not only improves the user experience but also strengthens the security of user accounts.

When implementing a password reset process, there are several security considerations to keep in mind:

1. **Secure Communications:** Ensure that HTTPS is utilized for all interactions during the password reset process. This provides an extra security measure and safeguards the information being sent to the server, preventing data from being captured during the transmission process.
2. **Token Generation:** When following JSON Web token (JWT) principles, you should always use tokens that have been randomly generated and are long enough that they cannot be guessed or comprised during a brute-force attack. Another option is hash and store the token in a database.
3. **Token Expiration:** When following JSON Web Token (JWT) principles, tokens should always be set to expire after a certain amount of time. This will limit how long a malicious attacker will have access to the compromised system, application, and/or web server.

4. **Traffic Control:** Within the realm of web security, traffic control, also known as rate limiting, is a method used to regulate the amount of network traffic permitted from a user. This strategy is commonly employed during the password reset process to mitigate various security risks, including Denial of Service (DoS) attacks, Brute-Force attempts, Credential Stuffing Attacks, Web Scraping/Bots, and API misuse.
5. **No Leakage of Information:** “No leakage of Information” is a key concept in cybersecurity. It means that the data in your application should not be exposed either intentionally or unintentionally. To prevent this, ensure data transmissions are using HTTPS, data is stored in a secure location, error messages are generic and does not provide compromising information, all input fields are validated, access control is clearly defined, logging and monitoring are in place to monitor unusual activity, and regular audits are conducted to check for system vulnerabilities.

In the next section we will take a look at how to build an API to reset passwords using the security questions approach.

Password Reset: A Secure Approach with TDD

In the last section of this chapter, we will walk through a secure approach to implementing password reset functionality, focusing on the design of a secure password reset flow. We will also highlight the role of TDD principles in building secure features, demonstrating how to write tests for password reset functionality. This section will provide practical insights into developing and testing a secure password reset feature.

Continuing with where we left off in the cookbook project:

1. Open the `app.spec.js` file in your editor.
2. Add a new test suite for this chapter’s unit tests. Follow the naming conventions we used in the previous chapters.
3. In the body of the test suite, add a unit test to check if a 200 status code is returned with a message of “Password reset successful” when resetting a password.

The unit test should have the following checks:

- a. 200 status code.
- b. The respond body should have a message of “Password reset successful”.

it('should return a 200 status code with a message of 'Password reset successful' when resetting a user's password', async() => {

```

    const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-
password").send({
  securityQuestions: [
    {answer: "Hedwig"},
    {answer: "Quidditch Through the Ages"},
    {answer: "Evans"}
  ],
  newPassword: "password"
});

expect(res.statusCode).toEqual(200);
expect(res.body.message).toEqual("Password reset successful");
});

```

In code example, we define a unit test that verifies the functionality of the `/api/users/:email/reset-password` endpoint. Let's dive into each part of the code to understand how it work.

- **it("should return a 200 status code with a message of 'Password reset successful' when resetting a user's password", async() => { ... }):** This defines a unit test that checks the `/api/:email/reset-password` endpoint returns a 200 status code with a message of "Password reset successful".
- **const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-password").send({ ... }):** This send a POST request to `/api/users/:email/password-reset` endpoint and waits for a response, using the supertest npm package.
- **expect(res.statusCode).toEqual(200);** This checks if the response status code is 200, indicating the request was successful.
- **expect(res.body.message).toEqual("Password reset successful");** This checks if the response body has a message of "Password reset successful".

To ensure our test fails, run `npm test` from a new CLI window.

```

Test Suites: 1 failed, 1 total
Tests:       1 failed, 12 passed, 13 total
Snapshots:   0 total
Time:        0.67 s
Ran all test suites.

```

Passing Test (Green):

1. Open the app.js file.
2. Create a new POST endpoint for `/api/users/:email/password-reset`. In the body of this endpoint, add a try-catch block for unexpected errors, update the user's password, and return 200 status code with a message of "Password reset successful" message.

```
app.post("/api/users/:email/reset-password", async (req, res, next) => {  
  try {  
    const { newPassword, securityQuestions } = req.body;  
    const { email } = req.params;  
  
    const user = await users.findOne({ email: email });  
    const hashedPassword = bcrypt.hashSync(newPassword, 10);  
    user.password = hashedPassword;  
  
    const result = await users.updateOne({ email: email }, {user});  
  
    console.log("Result: ", result);  
    res.status(200).send({ message: "Password reset successful", user: user});  
  } catch (err) {  
    console.error("Error: ", err.message);  
    next(err);  
  }  
});
```

In this code example, we define a POST endpoint at `/api/users/:email/reset-password` in our Express application. This endpoint retrieves updates a user's password and returns a status code of 200 with a message of "Password reset successful". Let's take a closer look at the functionality to this API.

- **app.post("/api/users/:email/reset-password", async (req, res, next) => { ... });** This line sets up the asynchronous POST endpoint at `/api/users/:email/reset-password`.
- **const { newPassword, securityQuestions } = req.body;** This line of code retrieves the newPassword and securityQuestions fields from the request body.
- **const { email } = req.params;** This line of code retrieves the email from the request params object.
- **const user = await users.findOne({ email: email });** This line of code retrieves a user from the mock database using the passed-in email address.

- **const hashedPassword = bcrypt.hashSync(newPassword, 10);** This line of code hashes the password from the request params object using the bcrypt npm package with 10 rounds of salt.
- **user.password = hashedPassword;** This line of code updates the user password with the hashed password.
- **const result = await users.updateOne({ email: email }, {user});** This line of code calls the mock databases updateOne method, updating the users password by email address.

Rerun the unit tests to ensure they are passing (npm test).

```
Test Suites: 1 passed, 1 total
Tests:      13 passed, 13 total
Snapshots:  0 total
Time:       0.679 s, estimated 1 s
Ran all test suites.
```

As you may already know, we haven't yet implemented data validation or checks to verify if the answers to the security questions matches those stored in our mock database. Until now, we've been manually validating input using plain JavaScript code throughout this textbook. However, we're going to explore an alternative approach to data validation in this example. We will employ the Ajv JSON schema validator, also known as "Another JavaScript Validator", to validate the data sent in the request body. This method offers the benefits of reusability and wide acceptance in industry standards. But, before we delve into that, let's first establish a new unit test.

1. Open the app.spec.js file.
2. In the body of the test suite, add a unit test that checks if a 400 status code with a message of "Bad Request" is returned when the request body fails ajv validation.

The unit test should have the following checks:

- a. 400 status code.
- b. The response body should have a message of "Bad Request".

```
it("should return a 400 status code with a message of 'Bad Request' when the
request body fails ajv validation", async() => {
  const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-
password").send({
```

```

    securityQuestions: [
      { answer: "Hedwig", question: "What is your pet's name?" },
      { answer: "Quidditch Through the Ages", myName: "Harry Potter" }
    ],
    newPassword: "password"
  });

  expect(res.statusCode).toEqual(400);
  expect(res.body.message).toEqual("Bad Request");
});

```

In this code example we define a unit test that verifies the functionality of the `/api/users/:email/reset-password` endpoint. Let's dive into each part of this code to understand how it works.

- **it("should return a 400 status code with a message of 'Bad Request' when the request body fails ajv validation", async() => { ... });** This defines a unit test that checks if the `/api/users/:email/reset-password` endpoint returns a 400 status code with a message of "Bad Request".
- **const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-password").send({ ...});** This sends a POST request to the `/api/users/:email/reset-password` endpoint and waits for a response using the supertest npm package.
- **expect(res.statusCode).toEqual(400);** This checks if the response status code is 400. Indicating a Bad Request.
- **expect(res.body.message).toEqual("Bad Request");** This checks if the response message is "Bad Request".

To ensure our tests fail, run `npm test` from a new CLI window.

```

Test Suites: 1 failed, 1 total
Tests:       1 failed, 13 passed, 14 total
Snapshots:   0 total
Time:        0.757 s, estimated 1 s
Ran all test suites.

```

Passing Test (Green):

1. Open the app.js file.

2. Add a require statement for the ajv npm package, create a new instance of the Ajv class, and build an Ajv JSON Schema object to validate the request body against.

```
const Ajv = require("ajv");  
const ajv = new Ajv();  
  
const securityQuestionsSchema = {  
  type: "object",  
  properties: {  
    newPassword: { type: "string" },  
    securityQuestions: {  
      type: "array",  
      item: {  
        type: "object",  
        properties: {  
          answer: { type: "string" }  
        },  
        required: ["answer"],  
        additionalProperties: false  
      }  
    },  
    required: ["newPassword", "securityQuestions"],  
    additionalProperties: false  
  }  
};
```

3. Update the POST method by leveraging the Ajv JSON Schema to validate the request body.

```
app.post("/api/users/:email/reset-password", async (req, res, next) => {  
  try {  
    const { email } = req.params;  
    const { newPassword, securityQuestions } = req.body;  
  
    const validate = ajv.compile(securityQuestionsSchema);  
    const valid = validate(req.body);  
  
    if (!valid) {  
      console.error("Bad Request: Invalid request body", validate.errors);  
      return next(createError(400, "Bad Request"));  
    }  
  
    const user = await users.findOne({ email: email });  
    const hashedPassword = bcrypt.hashSync(newPassword, 10);
```



```

    user.password = hashedPassword;

    const result = await users.updateOne({ email: email }, {user});

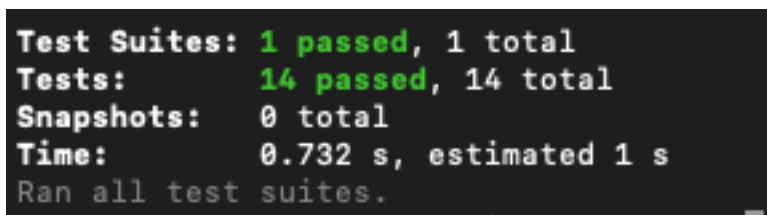
    console.log("Result: ", result);
    res.status(200).send({ message: "Password reset successful", user: user});
  } catch (err) {
    console.error("Error: ", err.message);
    next(err);
  }
});

```

In this code example, we update the POST request by introducing Ajv validation in our Express application. Let's take a closer look at what this code is doing.

- **const Ajv = require("ajv");** This line of code adds a require statement for the Ajv npm package.
- **const ajv = new Ajv();** This line of code creates a new instance of the Ajv class.
- **const securityQuestionsSchema = { ... };** This block of code creates an Ajv JSON Schema validator, which will be used to validate the request body of our POST request.
- **const validate = ajv.compile(securityQuestionsSchema);** This line of code compiles the Ajv JSON Schema and prepares it for validation.
- **const valid = validate(req.body);** This line of code uses the Ajv JSON Schema to validate it against the request body.
- **if (!valid) { ... }** This block of code checks if the validation passed, and if not, it generates a 400 error object and forwards it to our middleware error handler.

Let's rerun the test command (npm test) to confirm that our unit tests now pass.



```

Test Suites: 1 passed, 1 total
Tests:      14 passed, 14 total
Snapshots:  0 total
Time:       0.732 s, estimated 1 s
Ran all test suites.

```

Our last test will be to check to see if the answers supplied in the request body match the answers in our mock database.

1. Open the app.spec.js file.
2. Add a new unit test to this chapter's test suite that checks if a 401 error is returned with a message of "Unauthorized" when the security answers are incorrect.

The unit test should have the following checks:

- a. 401 status code
- b. Response message of "Unauthorized".

```
it("should return 401 status code with a message of 'Unauthorized' when the security answers are incorrect", async() => {  
  const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-password").send({  
    securityQuestions: [  
      { answer: "Fluffy"},  
      { answer: "Quidditch Through the Ages"},  
      { answer: "Evans" }  
    ],  
    newPassword: "password"  
  });  
  
  expect(res.statusCode).toEqual(401);  
  expect(res.body.message).toEqual("Unauthorized");  
});
```

In this code example, we define a unit test that verifies the functionality of the `/api/users/:email/reset-password` endpoint. Let's dive into each part of this code to understand how it works.

- **it("should return 401 status code with a message of 'Unauthorized' when the security answers are incorrect", async() => { ... });** This defines a unit test that checks the `/api/users/:email/reset-password` endpoint returns a 401 status code with a message of "Unauthorized" when the security answers are incorrect.
- **const res = await request(app).post("/api/users/harry@hogwarts.edu/reset-password").send({ ... });** This sends a POST request to the `/api/users/:email/reset-password` endpoint and waits for a response using the supertest npm package.
- **expect(res.statusCode).toEqual(401);** This checks if the response status code is 401. Indicating a Bad Request.

- **expect(res.body.message).toEqual("Unauthorized");** This checks if the response body message is “Unauthorized”.

To ensure our tests fail, run npm test.

```
Test Suites: 1 failed, 1 total
Tests:      1 failed, 14 passed, 15 total
Snapshots:  0 total
Time:       0.816 s, estimated 1 s
Ran all test suites.
```

Passing Test (Green):

1. Open the app.js file.
2. Add validation to the POST endpoint that checks if the saved answers in our mock database match the request body answers. If not, generate a 401 error and pass it to our middleware error handler.

```
app.post("/api/users/:email/reset-password", async (req, res, next) => {
  try {
    const { email } = req.params;
    const { newPassword, securityQuestions } = req.body;

    const validate = ajv.compile(securityQuestionsSchema);
    const valid = validate(req.body);

    if (!valid) {
      console.error("Bad Request: Invalid request body", validate.errors);
      return next(createError(400, "Bad Request"));
    }

    const user = await users.findOne({ email: email });

    if (securityQuestions[0].answer !== user.securityQuestions[0].answer ||
        securityQuestions[1].answer !== user.securityQuestions[1].answer ||
        securityQuestions[2].answer !== user.securityQuestions[2].answer) {
      console.error("Unauthorized: Security questions do not match");
      return next(createError(401, "Unauthorized"));
    }

    const hashedPassword = bcrypt.hashSync(newPassword, 10);
    user.password = hashedPassword;
```

```

const result = await users.updateOne({ email: email }, {user});

console.log("Result: ", result);
res.status(200).send({ message: "Password reset successful", user: user});
} catch (err) {
  console.error("Error: ", err.message);
  next(err);
}
});

```

In this code example, we add error checking to test if the saved answers from the mock database match the request body. Let's take a closer look at what this code is doing.

- **if (securityQuestions[0].answer !== user.securityQuestions[0].answer || ...)** this code block checks the saved security question answers against what's being passed in the body of the request. If they do not match, a 401 error is generated with a message of "Unauthorized," which is then passed to the middleware's error handler.

Let's rerun the test command (npm test) to confirm that our unit tests now pass.

```

Test Suites: 1 passed, 1 total
Tests:      15 passed, 15 total
Snapshots:  0 total
Time:       0.736 s, estimated 1 s
Ran all test suites.

```

Hands-On 7.1: Enhancing API Security

In this hands-on project, you will be extending the APIs for the in-n-out-books project using a mock database. You will be applying Test-Driven Development (TDD) principles and using Jest for testing.

Instructions:

1. Continue working in the project folder from the previous chapter's hands-on project.
2. In your project, ensure you have the following file structure:

```

src
  app.js
test
  app.spec.js

```

package.json
database
users.js
books.js
collection.js

3. You will be building the following route in your app.js file:
 - a. A POST route at /api/users/:email/verify-security-question that verifies a user's security questions and returns a 200-status with 'Security questions successfully answered' message. To do this, you will need to compare the answers supplied in the request body against what's saved in our mock database. Use a try-catch block to handle any errors, including checking if the request body fails ajv validation and throwing a 400 error if it does with applicable message. If the answers do not match what's saved in the mock database, throw a 401 error with an 'Unauthorized' message.
4. Write unit tests for this route using Jest in your app.spec.js file. You should write the tests before you implement the routes, following TDD principles. Create a test suite named "Chapter [Number]: API Tests". Use the following test cases:
 - a. It should return a 200 status with 'Security questions successfully answered' message.
 - b. It should return a 400 status code with 'Bad Request' message when the request body fails ajv validation.
 - c. It should return a 401 status code with 'Unauthorized' message when the security questions are incorrect.

Grading:

You will earn 20 points for each passing unit test, for a total of 60 points. If two tests pass, you will earn 40 points.

Hints:

- Remember to compare the req.body answers against the saved user's answers from the mock database.
- Use the toEqual method from Jest to write your assertions.
- Use the npm package ajv for JSON schema validation (already included in the starter project). The validation should check if the request is an array of objects with a property for answer that is a string. No additional properties are allowed and answer is required.

Chapter 8: Deploying Your API

Chapter Overview

In the final chapter of this book, we will guide you through the process of deploying your API. We will start by understanding what deployment entails and the necessary preparations. We will then guide you through the process of deploying your API using Render. Finally, we will discuss the importance of post deployment monitoring and maintenance. This chapter will provide you with practical insight into taking your API from a development environment to a live production environment.

Learning Objectives

By the end of this chapter, you should be able to:

- Define the process and importance of deployment in web applications
- Compare deployment stages and preparation steps.
- Analyze the process of deploying with Render
- Assess the importance and methods of post-deployment monitoring and maintenance

Understanding Deployment

In this section, we will discuss what deployment means in the context of an API. We will cover the transition from a development environment to a live production environment and why it's a crucial step in the lifecycle of an API.

Deployment is a pivotal stage in the software development lifecycle. It is the point at which the software, having undergone thorough testing and validation, is finally made accessible to end-users. This stage represents the software's transition from a development setting to a production setting here are some reasons why deployment holds such significance:

1. **Shift from Development to Utilization:** Deployment signifies the transition from the phase of software development to the phase of software usage. Without deployment, the software remains in the development stage and is not available for end-user utilization.
2. **Accessibility for Users:** Deployment guarantees that the software is available for users. This could be on a worldwide scale (like a web application available over the internet) or within a specific network (like an internal business application).
3. **Feedback from Real-World Usage:** Once the software is deployed, it is used in real-world situations. This enables developers to collect valuable feedback and insights about the software's performance, usability, and potential areas for enhancement.

4. **Generation of Revenue:** For commercial software, deployment is the stage where the process for revenue generation begins. Once the software is deployed and accessible to users, it can begin to charge customers and generate revenue.
5. **Updates and Maintenance:** Post-deployment, the software enters what's called the "maintenance phase" of the lifecycle. In this phase, the software is regularly updated to resolve bugs, improve performance, and to introduce new features. This would not be possible without the deployment of the software, because you would not have a customer base to obtain feedback from.

Deployment is not a single-action task; it's a multi-stage process that guarantees the software is accurately installed, set up, and primed for use in a production setting. Understanding these stages is essential for a seamless and successful deployment. Here are the various phases involved in deployment:

1. **Development:** This is the initial stage where the software is built. It's where the code is written and unit tests are built and tested.
2. **Testing:** After development, the software is moved to a test environment where the software is checked for bugs, performance issues, and integration/system testing.
3. **Staging:** In this stage, the software is moved to an environment that closely mirrors the production environment. This allows for user-acceptance testing and final validations before the software is deployed to an actual production environment for end-user consumption.
4. **Production:** This is the final stage of the process and where the software is moved to the production environment and becomes accessible to end-users. This stage typically involves installing the software on the production servers and configuring the servers for end-users.
5. **Maintenance:** After the software is moved to production, it enters the "maintenance phase." During this phase, the software is regularly monitored for bugs, performance issues, and new features are added as requested by end-users.

Deployment strategies are methods used to change or upgrade an application while minimizing downtime, risk, and impacting end-users. There are many different types of deployment strategies to choose from, all with varying levels of advantages and disadvantages. Choosing the right deployment strategy is critical to the survivability of your software. Think of deployment strategies as documented reference guide for how your software will be deployed. Here are three common deployment strategies:

1. **Basic Deployment:** This is the simplest form of deployment. In this strategy, the existing software is taken offline (no longer accessible to end-users) and a new version

of the existing software is brought online. This is the most straightforward way to deploy an application. You simply spin down the server, uninstall the software, and then reinstall the updated version of the software. The disadvantage of this approach is that it leads to downtime, which can affect the end-user's experience.

2. **Blue/Green Deployment:** This strategy is another straightforward approach to deployment. In this strategy, there are two identical production environments, which are known as Blue and Green. One of the production environments is live and the other one is inactive. When you are ready to deploy a new version of your software, you do so in the inactive environment. Then the software is tested to ensure everything is running correctly. If everything works as expected, you switch the router so all incoming requests go to the new version. That is, the previously live environment (old version of the software) becomes the inactive environment and the previously inactive environment (new version of the software) becomes the active environment.
3. **Canary Deployment:** Named after the “canary in a coal mine” principle, this strategy involves deploying a new version of your software to a small set of end-users to test and monitor the software's performance. If everything works as expected, the new version is gradually deployed to the remaining end-users of the software product. The advantage of this approach is, if issues are detected in the new version, only a small subset of end-user's are affected.
4. **Rolling Updates:** In this strategy, a new version of the software is gradually deployed to the end-users. For this strategy to work, there must be multiple instances of the software in production (multiple servers running the same software). Usually, updates are deployed one instance at a time, ensuring there is no downtime during the deployment process. The main disadvantage of this approach is users are likely to experience different versions of the software while updates are being deployed.

For simple deployments, the Blue/Green Deployment strategy is a great choice. It's uncomplicated, easy to understand, and it minimizes the downtime during deployments. When it comes to a strategy that's widely embraced in the industry, the Canary Deployment strategy is frequently used, particularly in large organizations with extensive applications. This specific strategy offers the most flexibility as it allows you to incrementally deploy and test updates with a small group of end-users. If any issues arise, you can quickly revert the changes. Therefore, it's a valuable strategy for reducing the impact of any potential problems while eliminating downtime.

In the next section, we will take a look at how to prepare your software for deployment.

Preparing for Deployment

In this section, we will go over the necessary steps to prepare an application for deployment. This includes finalizing the code, conducting code reviews, end-to-end testing and integration testing, and creating a rollback plan.

Code review and optimization are essential phases in the software development lifecycle as they assure the performance and quality of the code you've developed. Code review is a practice where your peers in the development team examine your code. The aim of code reviews is to detect bugs, ensure consistency in the codebase, and to facilitate knowledge sharing among the development team. Code reviews can lead to an application that is well-structured, efficient, maintainable, and scalable. They also cultivate a collaborative culture and assist developers in their professional growth. Don't view code reviews as a mechanism for management oversight, but rather as an opportunity to improve as a developer. Code optimization, on the other hand, involves restructuring your code to make it more efficient and less resource-intensive. The goal of code optimization is to improve speed, memory allocation and usage, and other aspects of the application. Code optimization should never impact the readability or correctness of your codebase. Instead, code optimization should only be used in the areas where performance is affected and needs improving.

Regardless of the size of your development team, code reviews should be incorporated into the development life cycle. Even if you are a solo developer, reviewing your own code after a break can provide a fresh set of eyes and will likely uncover opportunities for improvement. Many organizations used automated tools for code review and optimization. Tools like SonarQube are used to continually inspect the quality of your code and to provide recommendations on performance roadblocks. Here is a generic workflow for code reviews during sprints using Agile and SCRUM:

1. **Development:** Developers work on their assigned tasks for the sprint. And, once a task has been completed, it is pushed to a feature branch.
2. **Pull Request:** The developer then creates a pull request (PR). This signals to the team that the code is ready for peer review.
3. **Code Review:** Other team members review the code in the PR. They check for bugs, performance concerns, naming conventions, coding standards, and code quality. Feedback is provided and suggestions are made, if necessary.
4. **Revision:** The individual who initiated the PR makes the necessary changes based on the feedback they received during the code review process.
5. **Approval and Merge:** Other team members review the code again, approve the PR, and merge the code into the main branch.

6. **Integration Testing:** After the code has been merged into the main branch, integration tests are run to ensure that the new code works with the existing codebase without errors.

Thorough testing is a critical part of the software development process, because it ensures the quality and reliability of your application. It typically involves different levels of testing, including unit testing, integration testing, and end-to-end testing.

1. **Unit Testing:** This is the process of testing individual components and code blocks in your application. The purpose of unit testing is to verify the code you've written works and behaves as intended. Unit tests are written and maintained by the development team.
2. **Integration Testing:** This involves testing multiple components together to ensure everything works correctly as a group (other API's, third-party tools, etc.). The purpose of integration testing is to catch issues that may not be visible in unit tests or when components were tested individually.
3. **End-to-End Testing:** This involves examining the complete application, from inception to conclusion. The objective of end-to-end testing is to confirm that the system operates as intended and fulfills user requirements. As the final stage in the testing process, end-to-end testing should be conducted using realistic scenarios that mimic actual user interactions. This form of testing focuses on the application's behavior and response in real-world situations.

Creating a rollback plan is a critical part of the development and pre-deployment process. Despite all the testing and code reviews leading up to deployment, things can still go wrong. A rollback plan is intended to provide guidelines on how to quickly revert to a previous version of the software or application state. In other words, if something goes wrong, a rollback plan is there to minimize the impact on users. Here are the steps to crafting a well-structured rollback plan:

1. **Identify Changes:** The first step in a rollback plan is to identify all of the changes that will be made during the deployment process, including code changes, database migrations, server configurations, etc.,
2. **Establish Rollback Protocols:** The next step in a rollback plan is to define what steps need to be taken during the rollback. For code, this could be rolling back to a previous software version. For database migrations, this could mean a reverse migration.

3. **Confirm the Rollback Protocols:** Just like the approach taken with unit tests, its crucial to validate your rollback procedures. Typically, the testing of rollback procedures is carried out in a staging or testing environment.
4. **Automation:** You should always aim to automate rollback procedures. This can be done through automation tools, deployment script, or tools that support rollbacks.
5. **Documentation:** Just like other important processes, a rollback plan should be thoroughly documented and made available to all individuals involved in the deployment process or anyone who might need to utilize it.
6. **Communication:** A rollback plan should be communicated to everyone involved in the deployment process. Everyone must be aware of the plan and be taught how to execute it if necessary.

The goal of the rollback plan is to minimize downtime and to communicate contingencies in case something goes wrong. It's like an insurance policy, you are hoping you will never need it, but when you do need it, you will be glad you had it. In the next section we will take a look at how we can deploy the cookbook application to Render.

Deploying with Render

In this section, we will walk you through the process of deploying your API using Render. We will cover the steps involved, from setting up your Render account to deploying your application and testing it.

Disclaimer: The steps outlined in this section are accurate as the time of writing this book. The deployment process may change due to factors beyond our control. Always refer to the official documentation provided by Render if you encounter any issues or discrepancies.

Now, let's take a look at how to deploy an application with Render:

Applications deployed via Render can be accessed by URL `[websiteName].onrender.com`, where `[websiteName]` should be replaced with the actual name of your application. For instance, if you're deploying an application named "my-app," the URL would be `my-app.onrender.com`. Render requires applications to be hosted on a Git provider and each project should be in its own repository. This means you cannot have multiple projects within a single repository. Therefore, before deploying the cookbook project, we will need to create a new repository.

Getting the Cookbook Project Ready for Deployment:

Before deploying the application to Render, we need to set up the Node and NPM environments for the project.

1. Open the cookbook project's package.json file in your text editor.
2. Ensure that the "engines" section of the package.json file includes entries for Node v20 and NPM v10.

```
{
  "name": "cookbook",
  "version": "1.0.0",
  "description": "Cookbook API",
  "main": "app.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "jest",
    "start": "node ./bin/www",
    "dev": "nodemon ./bin/www"
  },
  "author": "Professor Krasso",
  "license": "MIT",
  "dependencies": {
    "ajv": "^8.12.0",
    "bcryptjs": "^2.4.3",
    "express": "^4.19.2",
    "http-errors": "^2.0.0"
  },
  "devDependencies": {
    "@types/jest": "^29.5.12",
    "jest": "^29.7.0",
    "nodemon": "^3.1.0",
    "supertest": "^6.3.4"
  },
  "engines": {
    "node": "20.9.0",
    "npm": "10.1.0"
  }
}
```

3. Stage your changes, commit them, and then push your work to GitHub.
4. Confirm that the project has been successfully updated on GitHub.

Setting up the GitHub Repository:

1. Log into your Git provider and create a new GitHub repository. Name this repository “cookbook” and set it to public.
2. Clone the newly created repository.
3. Transfer all of the contents inside of the cookbook project to your cloned repository. This includes all nested folders and files. Don’t include the root cookbook project folder.
4. Remove the node_modules folder and the package-lock.json file.
5. Stage your changes, commit them, and then push your work to GitHub.
6. Confirm that the project has been successfully added to the repository you created in the first step of this section.

Deploying on Render: Web Service

1. Register for a free account at <https://dashboard.render.com>
2. Log into render.com and navigate to the Dashboard tab.
3. From the Dashboard tab, select the “New +” button and choose “Web Service.”
4. On the “Create a New Web Service” page, select “Build and deploy from a Git repository” and click Next.
5. When creating a service in the Render dashboard for the first time, you’ll be prompted to connect your GitHub account. Follow Render’s [official documentation](#) to connect your GitHub account, making sure to grant Render the necessary permissions to access the cookbook repository.
6. Link the cookbook repository.
7. On the “You are deploying a web service for [GitHubAccountName]/[RepoName]” page, use the following settings:

Name:	Accept the default
Region:	Accept the default
Branch:	Accept the default

Root Directory:	Accept the default
Runtime:	Node
Build Command:	yarn
Start Command:	node ./bin/www
Instance Type:	Free \$0 / month
Environment Variables:	None

8. Click on “Create Web Service.”
9. Render will automatically generate a URL for your Web Service. If the chosen name is already taken, Render will append random, unique values to the name. To avoid this, consider prefixing the project’s name with your last name, for example, [yourLastName]-cookbook.
10. Wait for the project to complete the build and deployment process. The deployment is finished when you see “Your service is live” in the log console window.
11. Click on the auto-generated link provided by Render for your application. This link should be located under the web service name. Clicking on the link will open a new browser tab displaying the cookbook landing page. If the landing page does not appear, there may have been an error during deployment. To troubleshoot deployment errors, refer to the error messages in the error logs. These logs can be accessed by clicking the Logs link in the left navigation menu.

In the next section we will discuss post-deployment monitoring and maintenance considerations.

Post-Deployment Monitoring and Maintenance

In the final section of this chapter, we will discuss the importance of monitoring and maintaining your API after deployment. We will cover how to monitor your API’s performance, handle updates and changes, and ensure your API remains secure and efficient in a live environment.

Monitoring and maintaining the health of an application is critical to the longevity and survivability of an application it is an important part of the software management process. It involves monitoring the applications performance, uncovering potential issues and bugs, and taking corrective actions once items are identified. Regular health checks can help detect problems early and often before it affects the end-users of an application. Understanding the importance of application health monitoring can lead to better problem-solving skills and improved performance optimization.

1. **Problem Identification:** Regular monitoring identifies issues sooner, before they become a problem or affect the overall performance and usability of an application.
2. **Performance Optimization:** By monitoring an applications health, you can obtain valuable insight into how the application is performing and what steps you can take to optimize the application for better performance.

The key performance metrics to track are response time and error rate. These metrics can provide valuable insight into the applications speed, reliability, and user experience. Response time measures how long your application takes to respond to a user's request. A shorter response time indicates a faster, more performant application. Whereas, a longer response time indicates a slower application with performance bottlenecks. Error rates measure the frequency of errors within a specific timeframe in your application. A lower error rate suggests that the application is dependable and user-centric. Conversely, an elevated error rate signifies subpar application performance and a less than optimal user experience.

Setting up alerts and notifications in your application can help you stay on top of the application's performance and allow you to quickly react to issues. Alerts are generally used to notify you of critical issues that need immediate attention.

1. **Critical Alerts:** These types of alerts notify you of mission critical issues. That is, issues that significantly impact the applications performance, availability, and usability. For example, application crashes and resources being unavailable.
2. **Warning Alerts:** These types of alerts notify you of non-critical issues. Typically, these types of alerts are not immediately critical, but if they are not addressed within a reasonable amount of time, they could lead to mission critical issues.

The remaining essential components of post-deployment activities include routine updates and backups. These are crucial for preserving the security, performance, and reliability of your application. Updates can deliver enhanced performance features, security updates, and bug fixes. Conversely, backups provide a recovery solution for corrupted or lost data and a means to revert to older states and versions of an application. Grasping the importance of regular updates and backups can assist in maintaining an applications performance and protecting its data from server crashes, malicious attacks, and system failures.

Hands-On 8.1: Deploying Your API on Render

Congratulations on making it to the final hands-on project in this book. You've built APIs, applied Test-Driven Development (TDD) principles, written unit tests using Jest, and now it's time to take your in-n-out-books project to the next level – deployment.

In this hands-on project, you will be deploying your application to Render, a fully managed platform that automates the process of deploying web apps. This will allow your application to be accessed from anywhere, turning your local project into a live web application.

Instructions:

1. Continue working in the project from the previous chapter's hands-on project.
2. In your project, ensure you have the following file structure:

```
src  
  app.js  
test  
  app.spec.js  
package.json  
database  
  users.js  
  books.js  
  collection.js
```

3. Deploy your application to Render. Follow the instructions provided in this chapter to deploy your application. Make sure your application is accessible via public URL.
4. Verify that your application is working correctly. You should be able to access your application's landing page and routes via a public URL and receive the expected responses.

Grading:

This hands-on project is graded on an all-or-nothing basis. You will earn 60 points if your application is successfully deployed and accessible via a public URL. If your application is not deployed, you will earn 0 points.

Hints:

- Follow the deployment instructions provided in this chapter. Make sure to configure your application correctly for deployment.
- Test your application thoroughly before submitting your assignment. Make sure that all routes are working as expected.
- If you encounter any issues during deployment, don't hesitate to seek help. Deployment can be a complex process, and it's normal to encounter challenges.

References

- Copilot. (n.d.). OpenAI. *Microsoft Copilot*. computer software. Retrieved December 19, 2023, from <https://www.microsoft.com/en-us/microsoft-copilot>.
- Deploy a node express app on render: Render docs*. Deploy a Node Express App on Render | Render Docs. (n.d.). <https://docs.render.com/deploy-node-express-app>
- Lauret, A. (2019). *The design of web APIs*. Simon and Schuster.
- Node.js. "Index | Node.js V20.10.0 Documentation." *Nodejs.org*, nodejs.org/docs/latest-v20.x/api/index.html. Accessed 9 Jan. 2024.