Course Transcript

Developing and Deploying Web Applications with MEAN.js

Introduction to the MEAN Stack

Working with AngularJS

Working with Node.js and Express

Working with MongoDB

Integrating AngularJS with Node.js Projects

1. Using an AngularJS Application as a View

2. Connecting to the Server

3. Creating and Using Services

4. Using the ngResource Module

Practice: Creating a MEAN Application

1. Exercise: Create a Simple MEAN Application

Course Introduction

Learning Objective

After completing this topic, you should be able to

- *start the course*

1.

Hello and welcome to this course on developing Web applications with MEAN.js. While developing a Web application can be difficult to determine the right combination of technologies or a front-end framework, a back-end server and a database. MEAN.js provides a great combination of technologies for developing end-to-end Web applications with MongoDB, Express, AngularJS, and Node.js. This course provides an introduction to developing Web applications by using the MEAN stack.

The Architecture of the MEAN Stack

Learning Objective

After completing this topic, you should be able to

- *identify the components of the MEAN stack*

1.

In this topic, we'll learn about the architecture of the MEAN stack. The MEAN stack is a collection of tools that can be used for building robust web applications. It consists of four main components – M for MongoDB, which is the database; E for Express, which is a server-side framework; A for AngularJS, which is a front-end framework; and N for Node.js, which is a server-side JavaScript environment. Let us now look at each of these components in a little more detail. First of all, let us talk about MongoDB. MongoDB is a NoSQL database system. This means that it does not store data in the form of tabular relations that are used in relational databases. Instead, it stores data in the form of JSON-like documents with a dynamic schema. The term dynamic schema means that there is no well-defined model according to which data is stored in the database. The documents stored in the database can have varying sets of fields and different data types for the fields. The data in a MongoDB database is stored in the form of key-value pairs. MongoDB is open source and provides support for wide range of operating systems. It provides high performance and high availability. It is also highly scalable. Node.js is a runtime environment for JavaScript web applications. It is built on top of Google Chrome's V8 JavaScript engine. It is extremely lightweight and allows running server-side JavaScript code on a wide variety of platforms such as Windows, OS X, and Linux. Node.js is open source and also provides high performance.

Express is the most common server-side framework for developing web applications on Node. It provides a set of robust web application features to efficiently manage requests, routes, and views. It also helps generate boilerplate code for your applications. Express is lightweight and fast. It helps abstract the complexities of Node.js and helps organize server-side JavaScript code into testable and maintainable units. AngularJS is a front-end JavaScript framework. It is designed to build single-page applications by using the MVC architecture. AngularJS extends HTML by using special attributes to bind JavaScript logic with HTML elements. The ability of AngularJS to extend HTML allows for a cleaner way to manipulate DOM and implement two-way data binding. Finally, AngularJS is an open-source technology and is very flexible. In this topic, we learned about the four main components of the MEAN stack and tried to understand the role of each of the four components in developing a MEAN application.

The Benefits of the MEAN Stack

Learning Objective

After completing this topic, you should be able to

- *identify the benefits of the MEAN stack*

1.

In this topic, we will talk about the benefits of the MEAN stack. The main benefit that the MEAN stack provides is that it enables full-stack JavaScript development. This means that it uses JavaScript at both the front end and the back end. The four technologies that it includes – MongoDB, Express, AngularJS, and Node.js – are all JavaScript based. For example, MongoDB is a JavaScript-based database, Express is a JavaScript-based server, AngularJS is a front-end JavaScript framework, and Node.js is a server-side JavaScript platform. This makes the MEAN stack an attractive option for web professionals who have been using JavaScript for client-side coding. The MEAN stack provides support for the MVC design pattern or the Model View Controller design pattern. MVC is a strong design pattern that helps guide the architecture of a MEAN application. It separates the application data, the presentation, and the flow of application into three separate components – the model, the view, and the controller. The model manages the application data and the business rules. The view manages the display of information to the user. And the controller handles user input, collaborates with the model to store and retrieve data, and provides data to the relevant view, which in turn displays the information to the user. This separation of model, view, and controller within an application allows you to change one component without affecting the others.

Node.js has a huge module library that provides several components that you can use. For example, you can use the passport module to implement authentication in your web application. Similarly, there are several other components available in this library that can be used out of the box. All components of the MEAN stack are open source. This means that the stack is free to use, gets updated regularly, and can be customized as per requirement. In this topic, we learned about the benefits of the MEAN stack. We learned that the MEAN stack provides a full-stack JavaScript development option. It supports the MVC design pattern, provides a huge module library, and is open source. All these benefits make MEAN a very popular web development option to work with.

Installing the Prerequisites

Learning Objective

After completing this topic, you should be able to

- *install the prerequisites for the MEAN stack*

1.

In this topic, we will learn about installing the prerequisites of the MEAN stack. To get started using the MEAN stack, you need to install Node.js and MongoDB. Let us first see how to install Node.js. To install Node.js, you need to download it from the URL nodejs.org. However, instead of downloading it from this web page, you can also download it from the command shell by specifying the download URL. The instructions for installing Node.js on different platforms may be different. You may have to search for the instructions to install Node.js on your chosen platform. Detailed instructions on installing Node.js on various platforms are easily available over the Web. To download Node.js on my Linux system, I'll simply use the wget command. After downloading Node.js, I'll use the tar command to extract the binary package into my system's local package hierarchy. You can see that I have used the strip-components option with the tar command. I have done this to strip off the version directory in which the Node.js archive is packaged. Let me clear the screen. Now Node.js is installed. You can check the version of the installed Node.js by using the node --version command.

[The presenter opens the terminal window and executes the "wget https://nodejs.org/dist/v0.10.30/node-v0.10.30-linux-x64.tar.gz" command to download the node.js. Then she executes the "sudo tar --strip-components 1 -xzvf node-v -C /usr/local" command.]*

Let us now see how to install MongoDB. To install MongoDB, you need to first download it. You can download MongoDB by going to www.mongodb.org/downloads. On this web page, you will find the options to download MongoDB for different platforms such as Windows, Linux, Mac OS X, and Solaris. You can download the appropriate build as per the operating system that you are using. As I am using the Linux operating system, I'll click the Linux tab over here. The installation instructions for each build may be different. You can access the installation instructions for the selected build by clicking the Installation Instructions link over here. Instead of downloading MongoDB from this page, you can also download it from the Linux shell. For this, you will need the link to the MongoDB archive. You can click the Copy Link button to copy the link to the MongoDB archive. Let me show you how to download and install MongoDB on my Linux box. I will download MongoDB directly from the Linux shell. For this, I'll use the curl command.

Now I'll extract the files from the downloaded archive by using the tar command. The file is now extracted. Let me just clear the screen. Now I need to create a new directory named mongodb. Once the mongodb directory is created, I need to copy the extracted archive to the mongodb directory. For this, I'll use the cp command. Once this is done, I need to ensure that the MongoDB binaries are included in the path variable. For this, I need to make an entry in the .bashrc file. So I'll open the .bashrc file in gedit. And now I'll add an export statement to include the path of the mongodb directory in the path variable and Save the file and then exit. As I have made some changes to the .bashrc file, I need to open a new instance of the shell so that the changes take effect.

[The presenter executes the "mkdir -p mongodb", "cp -R -n mongodb-linux-x86_64-3.0.7/. mongodb", and "gedit .bashrc" commands in the terminal window.]

Now MongoDB is installed. However, before we can run MongoDB, we need to create the data directory for MongoDB. The data directory is a directory to which MongoDB will write data. The default path for the data directory is /data/db. Let me create the data directory. I have used sudo as I need elevated permissions to create a data directory in the root folder. Now that I have created the data directory, I need to ensure that the user account running MongoDB has read/write permissions on the data directory. Let me check who I am logged in as. Because I am logged in as user1, let me change the ownership of the data directory so that instead of root, it is owned by user1. For this, I'll use the chown command. Now that the current user has read/write permissions on the data directory, we can run MongoDB. For this, we'll need to run the mongod process. Now you can see that MongoDB is waiting for connections on port 27017. You can send requests to this instance of MongoDB from a client or from your application.

[The presenter executes the "sudo mkdir -p /data/db", "sudo chown user1.user1 /data -R", and "mongod" command in the terminal window.]

The Key Features of AngularJS

Learning Objective

After completing this topic, you should be able to

- *identify the key concepts of AngularJS*

1.

In this topic, we will learn about some key concepts of AngularJS. AngularJS is a front-end MVC JavaScript framework. A framework is a standard set of concepts and procedures used for dealing with a common type of problem. This standard set of concepts and procedures can be used as a reference for solving new problems of a similar kind. A front-end JavaScript framework is a package consisting of a collection of files and folders that contain standardized code, which can be used as a starting point for building web sites. The objective is to provide a common structure so that developers don't have to reinvent the wheel and can reuse the code included in the framework. In this way, JavaScript frameworks – such as AngularJS – help developers save a lot of time and effort. AngularJS is an MVC framework. An MVC framework is a package that helps guide the architecture of a web application and helps divide an application into three parts – the model, the view, and the controller where the model represents the data of the application, the view represents the user interface of the application, and the controller manages the application flow by handling user input, collaborating with the model to store or retrieve data, and providing data to the relevant view, which in turn displays the information to the user.

AngularJS is designed to build single-page applications or SPAs. SPAs are web applications that load a single HTML page and keep updating the page dynamically as the user interacts with the web page without sending or receiving a full-page request. This is made possible because of the MVC architecture that separates the data from its presentation. As the page is updated, the URL keeps changing giving the illusion that a new page has been loaded. This is done by using the concept of front-end routing. AngularJS supports two-way data binding. Two-way data binding refers to the automatic synchronization of data between the view and the model. A view is a projection of the data contained in a model. So you can have multiple views of the data in the same model. Two-way data binding means that whenever the data in the view changes, the data in the model changes accordingly. Similarly, whenever the data in the model changes, the data in the view changes accordingly.

AngularJS extends HTML syntax by including its own attributes called directives. It also binds data to HTML by using something called expressions. HTML has traditionally been used to develop documents or static web pages. AngularJS enhances HTML so that it would work like it was designed for developing web applications. By using AngularJS, you can eliminate the need to write large chunks of code that you would otherwise have to write. In this way, it bridges the gap between HTML and JavaScript. In this topic, we learned that AngularJS is a front-end JavaScript framework for developing single-page applications that use the MVC architecture. The framework supports two-way data binding and bridges the gap between HTML and JavaScript by extending the HTML syntax to include additional attributes called directives and by binding data to HTML by using expressions.

[Heading: The Key Features of AngularJS. The following set of code is illustrated: Name: <input type="text" ng-model="name"> {{name}}]

Developing an AngularJS Application

Learning Objective

After completing this topic, you should be able to

- *develop a basic AngularJS application*

1.

In this topic, we'll learn how to develop a basic AngularJS application. To start developing an AngularJS application, you first need to install AngularJS. AngularJS is a front-end JavaScript framework. Therefore, to install it, you simply need to include the frameworks – JavaScript files – in the main page of your application. To do this, you can download the files that you need and store them in a folder within your project. You can then include a reference to these files in the main page of your application by using a script tag. Alternatively, you can use the CDN of these files and load the files directly through the CDN server. The basic functionality of Angular can be accessed by simply including the angular.js file. However, to use other advanced functionality, you may need to include other JavaScript files provided by Angular such as angular-route.js and angular-resource.js.

[Heading: Developing an AngularJS Application. The following script tags are illustrated: <script src="/global/angular.js"></script> <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.js"></script>]

AngularJS allows you to extend HTML by using new attributes called directives. Directives are attributes whose names start with the prefix "ng-". In the given example, ng-app directive is used to declare that this is an Angular app. The ng-init attribute is used to initialize the application's data. It specifies the initial value for the variables in an application. The ng-model attribute is used to bind the value of an HTML control such as an input field with an application variable. The ng-bind attribute is used to replace the text content of the specified HTML element with the value of an expression. When the value of the expression changes, the text content of the HTML element is automatically updated. In the given example, the value of the input element is automatically reflected in the paragraph element. If the value of the input element is changed, the value of the paragraph element also changes automatically. Instead of ng-bind, AngularJS expressions can also be used to bind application data to HTML elements. AngularJS expressions are written inside double braces. The value of the expression is displayed at the position where the expression is inserted in an HTML document. Let us now develop a simple AngularJS application and see it in action. Let me first create a directory by the name of demo to store a new application. I'll now cd into the demo directory. I'll now create an HTML file named demo.html.

[Heading: Developing an AngularJS Application (continued). An illustration of a "Directive" is as follows: Name: <input type="text" ng-model="firstName">   An illustration of an "Expression" is as follows: Name: <input type="text" ng-model="firstName"> {{firstName}}   The presenter executes the "mkdir demo", "cd demo", and "gedit demo.html" commands in the terminal window.]

Let me add some markup to this file. This is a simple HTML file containing the usual HTML markup and some AngularJS. Here we have a script tag that includes a reference to the AngularJS framework JavaScript file. In this case, we are accessing the file from a CDN sever. To use AngularJS in an application, we must include a reference to this JavaScript file. Here we have the ng-app directive, which declares that this is an AngularJS application. The ng-init attribute specifies the initial value for the firstName variable. The ng-model attribute finds the value of the input field with the firstName variable.

And the ng-bind attribute binds the text content of this paragraph element with the value of the firstName variable. So, when the value of the firstName variable changes, the content of this paragraph element will also change automatically. Let us now Save this file and then view this file in a browser. You can see that the initial value Peter of the firstName variable is being displayed in both the input field and the paragraph element. If I change the value in the input field, the new value is automatically reflected in the paragraph element. Let me go back and make some changes in the demo.html file. Now, instead of using ng-bind attribute, I'll use an expression to display the value of the firstName variable. Let us now view the web page in a web browser. You can see that the result is same as was there with the ng-bind attribute. Now, if I change the value in the input field, the new value is automatically reflected in the paragraph element.

Implementing the AngularJS MVC Pattern

Learning Objective

After completing this topic, you should be able to

- *implement two-way data binding in an AngularJS MVC application*

1.

In this topic, we will learn how AngularJS follows the MVC design pattern and encourages loose coupling between data, presentation, and logic components. Let us first see how we declare models in AngularJS. A model in AngularJS is declared as a simple JavaScript object. For example, a model that needs to store the firstName and lastName of a user can be declared as shown here. In AngularJS, we do not declare a model separately. Rather, the model is declared within a controller. Let us now see how to declare a controller in AngularJS. Now here we have a controller named PersonController. Within this controller, we have included the model that we declared earlier. You can also see the $scope object here. $scope ties a view to a controller. Both the controller and the view can access this $scope object. So it can be used for communication between a controller and a view. The $scope object can be used to house both the data and the functions that need to be accessed from the view like in this example, $scope.person represents the application data and $scope.fullName represents a function. Both $scope.person object and $scope.fullName function can be accessed from a view associated with this controller. In AngularJS, a controller is defined within a module. So, before we see how to create views, let us see how to define a controller within a module.

[Heading: Implementing the AngularJS MVC Pattern. An illustration of a "Model" is as follows: var person; person = { firstName: "", lastName: "" } An illustration of a "Controller" is as follows: function PersonController ($scope) { var person; person = { firstName: "", lastName: "" $scope.person = person; $scope.fullName = function () { return $scope.person.firstName + " " + $scope.person.lastName; }]

A module is a container for different parts of an application. A module can be created by using angular.module function. In this code, the angular.module method has been used to create a module named PersonApp. And its reference is stored in a variable named app. Further, the app.controller method has been used to add a controller named PersonController to the module. Now that we have seen how to declare a model and a controller and how to include a controller in a module, let us see how to declare a view that is bound to the model that we just saw. First of all, we have a div element that has an attribute named ng-app. This attribute is assigned a value – PersonApp – which is also the name of the module that we saw on the previous slide. Again, there is an attribute named ng-controller with the value PersonController. Now PersonController is the name of the controller that we created on the previous slide. These two values tell Angular which module and controller this section of HTML document is associated with. The first input element has the ng-model attribute that binds this field with the firstName property of the person object. Similarly, the second input element is bound with the lastName property of the person object. Finally, there is an expression that includes a function called to the fullName method that we saw on the previous slide. The value returned by the function will be displayed at the point where the function call is included.

*[Heading: Implementing the AngularJS MVC Pattern (continued). An illustration of a "View" is as follows: First Name: <input type="text" ng-model="person.firstName"/>
 Last Name: <input type="text" ng-model= "person.lastName"/>

 Full Name: {{fullName ()}}]*

Let us now see this code in action. I have already created two files – demo.html and demo.js. Let us first take a look at demo.js. This file contains a module that in turn includes the controller and the model. This is the exact code that we just discussed. Let us now take a look at demo.html. This file constitutes the view for the application and contains the regular HTML elements and several AngularJS attributes that we just discussed. It also includes a reference to the demo.js file that contains the model and the controller. Let me now view the demo.html file in a browser. Now, on this web page – if I enter a First Name and a Last Name – the Full Name is automatically displayed. Let us now create another AngularJS application that allows users to post and view articles. Let me first give you a demo of the application that we will be building. Now this is the application that we are going to build. Anyone who wants to post an article needs to enter their name, the Article Title, and the Article Text. And then click the Post button.

The article is displayed at the bottom of the web page with the name of the person and a timestamp. Let me post another article. You can see that the latest article is displayed at the top of the articles list. Now, to create this app, let us first create a directory named articleApp in which we will store the application files. Now let us cd into the directory. Let us create another directory to store the JavaScript files. Let us now cd into the javascripts directory. Now here we can create a file named articleApp.js. Now let me add some code to this file. Now here we have a module. And, within this module, we have a controller. Now, within the controller, we have defined an array to store the articles. We have also defined an object named newArticle that can be used to store the name of a user, the title and text of the article submitted by the user, and the time at which the article was submitted. We also have a function named post, which is invoked when the user submits the article. This function adds the current timestamp to the model object, and then pushes the model object into the articles array. Let us now Save and close this file.

Now let us move to the articleApp directory. I'll now create the view for the app. For this, I'll create a new file named main.html. Now I'll add some markup to this file. Now this file includes a reference to the AngularJS CDN and a reference to the articleApp.js file that we just created. The ng-app attribute of the body tag specifies the name of the module that we just created. The ng-controller attribute specifies the name of the controller that we just created. Next we have a form for accepting the user name and the article details. The form elements are bound to the respective fields of the model. The ng-Submit attribute of the form element includes a call to the post function, which gets executed when the form is submitted. We also have a div element here. This div element displays the articles that are submitted. Now, in this div element, we have used the ng-repeat directive to create a template. This means that the div element will be repeated for each article in the articles array. And, within each of these divs, the details of the respective article will be displayed as per the format specified within this div element. Notice the use of orderBy: ' timestamp ' : true with the ng-repeat directive. This indicates that the articles should be displayed in the descending order of timestamp. You'll also notice the use of the date filter with the timestamp expression. This has been used to display the date in a proper format. Now let us Save the file and view it in a web browser.

Let me submit a couple of articles. Another one. You can see that the details of the articles are displayed in the descending order of timestamp. You would have also noticed that visually our web page is not very appealing. If you want, you can enhance the visual appeal of the web page by using a UI framework such as bootstrap.

Injecting Dependencies in AngularJS

Learning Objective

After completing this topic, you should be able to

- *inject dependencies in an AngularJS application*

1.

In this topic, we'll learn about injecting dependencies in AngularJS. Dependency injection is a software design pattern that is widely used in AngularJS. Dependency injection deals with the dependencies between software components. In this design pattern, an object is given its dependencies instead of the object creating them itself. It helps remove hard-coded dependencies and enables us to change dependencies whenever required. With dependency injection, the process of creating and consuming dependencies is separated. The consumer only needs to be concerned about how to use the dependency and the process of creating the dependency can be handled by somebody else. Consider this code where we have a module. And, within the module, we have a controller. Now this controller needs to use a predefined service called $http. It also needs to use the $scope object. So we need to inject these dependencies into the controller. The dependencies can be injected by including them as parameters to the controller function as shown here. Once the dependencies are injected, the controller doesn't need to know how the dependencies are created. It only needs to know about the contract provided by these dependencies and how to consume them. An alternate way to inject dependencies would be to use an array containing the names of all dependencies followed by the controller function itself as shown here. In this topic, we learned what dependency injection means. We also learned two ways of injecting dependencies in a controller in AngularJS.

[Heading: Injecting Dependencies in AngularJS. The following set of codes is illustrated: var app = angular.module('myApp', []); app.controller('customerCtrl', function($scope, $http) { // Code that uses $http and $scope });]

The Key Features of Node.js and Express

Learning Objective

After completing this topic, you should be able to

- *describe the purpose and benefits of Node.js and Express*

1.

In this topic, we'll learn about the key features of Node.js and Express. Node.js is a runtime environment for JavaScript web applications that is built on top of Google Chrome's V8 JavaScript engine. It is used for easily building fast and scalable web applications. It allows running server-side JavaScript code on a wide variety of platforms such as Windows, OS X, and Linux. It uses an event-driven, nonblocking I/O model that uses a single thread. Unlike traditional web servers where each connection spawns a new thread, Node.js uses single-threaded, nonblocking I/O model to support a much larger number of requests than a traditional server. This makes Node.js extremely lightweight, efficient, and highly scalable. It also provides a rich module library, which makes the web application development process a lot easier. npm stands for Node Package Manager. As its name indicates, it is the default package manager for Node.js. npm is to Node what Gem is to Ruby and NuGet is to .NET. npm can be used for installing Node modules. It comes bundled with a Node.js installation. This means that when you install Node.js, npm is automatically installed.

Express is a Node.js web application server framework. It provides a robust set of features for developing web and mobile applications. It enables developers to set up the middleware to respond to HTTP requests. It also enables developers to implement routing based on the URL and HTTP method. In this topic, we learned the key features of Node.js and Express. We also learned about the Node Package Manager and its use.

Creating a Barebones Application

Learning Objective

After completing this topic, you should be able to

- *create and explore a barebones application*

1.

In this topic, we'll learn to create a simple barebones Node.js application. But, before we create that, let me verify that Node.js is properly installed on my system. So I type node and press Enter. Now I am inside the Node.js interactive console. Here I can type a JavaScript statement and see if it works. So I type console.log ("Hello World"). You can see that it displays the message "Hello World" on the console. I can exit the interactive console by pressing Ctrl+C twice. Let us now create a file named demo.js. So I'll type gedit demo.js. Now, within this file, I'll add some JavaScript code. Now this is a simple JavaScript code that declares two variables, adds their values, and displays the sum. I'll Save this file and close it. Now, on the shell prompt, I'll type node demo.js. And it displays the expected message that is "Sum is: 30". Now that we have ensured that node is working properly, let us learn more about creating a Node.js application. To create a Node.js application, we need a server that will listen to the client's request and return a response.

Let us see how to create a simple Node.js server. I'll create a file named server.js. I'll add some JavaScript code to this file. Now, to create a server, we need a node module named http. So here we use the require method to specify that we need the http module. Then we use the http.createServer method to create a server that listens on port 8081. Here we specify the type of content that the server will send as response. And here we specify the actual response that will be sent by the server.

Let us now Save the file and close it. Let me clear the screen. And now I'll type node server.js. The server has now started. Now I can go to the browser and type localhost:8081. And you can see the "Hello World" message. Now the server that we created here is a very basic server and doesn't offer much functionality. You can add better functionality to this server by adding more code or, instead, you could use a framework like Express. Express is a minimal and flexible Node.js web application framework that abstracts the complexities of a web server and makes things simple and easier for you. To use the Express framework, you need to install it by using the Node Package Manager or npm. So let us go back to the command shell. Let us press Ctrl+C to end the process that is running. Now, to install Express, you can use the command npm install express. Now, when you install Express, you can start creating an Express application, but you have to create it from scratch. So, instead of using Express, you can use a scaffolding app called express-generator. This app helps you to create a basic folder structure for a Node.js Express application. You can use the -g switch here. This switch specifies that express-generator should be installed globally, which means that its path will be added to the system path. Specify the password, okay. So express-generator is installed.

Now, once express-generator is installed, you can use the express --ejs command to create a barebones application. Here ejs means that we want to use the EJS rendering engine. We also need to specify the name of the directory in which the directory structure of the barebones application should be created. So let me use the demo1 directory. So now the folder structure of the barebones application has been created. So let us see the folder structure. So I'll open the demo1 folder. Here is the folder structure that has been created by express-generator. So we have a bin folder, a public folder, which in turn contains folders to store images, JavaScript files, and stylesheets. Then we have a routes folder, a views folder, an app.js file, and a package.json file. package.json is a JSON file, which contains information about your project. This information may include the name, version, description and may

vary depending upon your project. One of the important things in this file is the dependencies, which is nothing but the tools you need to run your project. By default, some of the typical tools needed for running a Node.js application are included in this file as dependencies. You can add or remove dependencies as per your requirement.

Let me close this file. Now the dependencies that are defined in the package.json file – they need to be installed. So let us see how to install the dependencies. Let me clear the screen. Now, to install the dependencies that are included in the package.json file, let me first go into the demo1 folder. And then I'll use the command npm install. Now this command will install all the dependencies that are declared in the package.json file. So all the dependencies have been installed. Let us now see the contents of the demo1 folder. So you can see that a new directory named node_modules has been added to the folder. This folder contains all the dependencies that were defined in the package.json file. So now let me go back to the command shell. Let me clear the screen again. And now, to start the application, I'll use the command npm start. The application is now started. Now you can go to the web browser and type localhost:3000 to access the application. Now you get a web page that displays the message "Welcome to Express". This is as per the default functionality included in the boilerplate code that we generated by using the express-generator. In this topic, we learned how to create a barebones Node.js Express application.

Creating and Using Modules

Learning Objective

After completing this topic, you should be able to

- *create and use modules*

1.

In this topic, we'll learn about creating and using modules. Modules are very critical for building applications on Node. Modules allow developers to include other JavaScript files in their applications. As a result, developers can include external libraries, such as libraries for authentication and database access in their applications. Modules help developers organize code into smaller parts where each part has a specific responsibility. Let us take a look at a few methods of creating and using modules. First of all, let us take a very simple example. Here we have a module in the form of a hello.js file that simply displays the message "Hello World." Now we can use this module in another file named app.js by using the require method. Now, when we run app.js, it will display the message "Hello World." Here we have another example of a module with a function. In the hello.js file, we have declared a function named hello that displays the message "Hello World." To invoke this function from the app.js file, we need to require ('./hello.js') and then call the hello function.

Now we have an example of a module with an anonymous function. This function can be made available to other modules by assigning it to module.exports. Now, to invoke this function from the app.js file, we need to require ('./hello.js') and store its reference in a variable, say, hello. Then we can invoke the function by making a call to the hello function. Some default variables are available in the scope of each module. These include __filename, which is the filename of the code being executed; __dirname, which is the name of the directory that contains the code being executed; process, which is an object containing a reference to the currently running process; process.argv, which is an array containing the command line arguments. Further we have process.stdin, process.stdout, and process.stderr, which refer to the standard input, output, and error streams of the current process.

Let us now see demos for the three methods of creating modules that we just discussed. First of all, let me create a directory named method1. In this directory, I'll save the files for the first method of creating modules. So let me cd into this directory. Now I'll create a file named hello.js. In this file, I'll simply add a console.log statement that says, "Hello World." I'll Save the file and close it. Now I'll create another file named app.js. In this file, I'll add a require statement and give the path of the hello.js file. I'll Save the file and close it. Now, at the Terminal, I'll type node app.js. You can see that a "Hello World" message is displayed as a result of the console.log statement that was included in the hello.js module.

Let us go back to the home directory. Let me clear the screen. Now I'll create another directory named method2 to demonstrate the second method. I'll cd into this directory. Now I'll create a file named hello.js. Now, in this file, I'll add a named function that simply displays the message, "Hello World." I'll Save the file and close it. Now I'll create another file by the name of app.js. Now, in this file, I'll add a require statement providing the path of the hello.js file. And then I'll include a call to the hello function. Again, I'll Save the file and close it. Now, on the Terminal, I'll type node app.js. Again a "Hello World" message is displayed as a result of the console.log statement that was included in the hello.js module. Let us go back to the home directory and let me clear the screen. Now I'll create another directory named method3 to demonstrate the third method. I'll cd into this directory. Now I'll create a file named hello.js.

Now, in this file, I'll add an anonymous function that simply displays the message "Hello World." Now I'll assign this function to module.exports so that I can access it from outside this file. Let me Save this file and close it. Now I'll create another file named app.js. Now, in this file, again, I'll require ('./hello.js') and I'll store this reference in a var hello. Now I'll make a call to the hello function. Let me Save this file too and then close it. Now, on the Terminal, I'll type node app.js. You can see that again a "Hello World" message has been displayed as a result of the console.log statement that was included in the hello.js module. In this topic, we learned various methods of defining and using modules.

Managing Dependencies with Package.json

Learning Objective

After completing this topic, you should be able to

- *manage dependencies with package.json*

1.

In this topic, we'll learn about managing dependencies by using package.json file. When you create the boilerplate code for an Express application by using Express generator, the folder structure that gets created includes a package.json file. This file contains various metadata about the application including the various packages that the application depends on to run. Against each package, the version number of the package is mentioned. The version numbers are prefixed by a tilde sign, which specifies that the latest patch version of the specified version should be used by the application. For example, ~1.3.5 would mean a version greater than or equal to 1.3.5 and less than 1.4.0. Now these are the packages that my application depends on. So, before I execute the application, the packages that the application depends on need to be installed. Let me first close this file and then open the Terminal window. Now I'll first move into the application folder, and then use the npm install command to install all the packages or dependencies defined in the package.json file.

Now all the dependencies have been installed. Now, at a later stage, it is possible that you want to enhance your application for which you need an additional dependency, such as passport or mongoose. In this case, you can either add the additional dependency to the package.json file and run the npm install command again or you can install the new package from the terminal by using the npm install package name command. Let me first clear the screen. Now suppose you need the mongoose package, you can use the npm install mongoose command to install the mongoose package. Though this command with install the dependency, but the information about the dependency is not present in the package.json file. So, when the application is deployed, this additional dependency will not be known. To avoid this problem, you can add the --save option with the npm install command like this. This option will add the dependency to the package.json file. Let me run this command. Now information about the mongoose package has been added to the package.json file. You can verify this by opening the package.json file. So here you can see that a new dependency – mongoose – has been added to the package.json file. In this topic, we learned how to manage dependencies by using the package.json file.

Defining and Registering Middleware

Learning Objective

After completing this topic, you should be able to

- *define and register middleware*

1.

In this topic, we'll learn about defining and registering middleware. Express allows you to define middleware for your application. Middleware refers to callback functions that handle the requests made to REST endpoints. Middleware help you associate application logic to different HTTP requests. The middleware process the request, return the response, or call the next registered middleware in the sequence. A middleware function has access to the request object, the response object, and the next middleware in sequence. A middleware must either end the request-response cycle by using the res.end() method, or call the next() function to invoke the next middleware. Besides the middleware that you create, some built-in middleware and third-party middleware are also available that you can use out of the box.

Let us now see a demo to understand how to define and register middleware. First of all, let us create a barebones Express application. For this I'll use the express --ejs command and specify the folder name as app1. Now, when I execute this command, a new folder by the name of app1 gets created and the entire folder structure of a new Express application gets created within this folder. Let me first clear the screen and then move into the app1 folder. Now this folder contains a file named app.js. Let me open this file. Now let us try to understand the code in this file. First of all, we create a few JavaScript variables and tie them to certain packages and dependencies. Next we create a few variables and tie them to some routes. Routes direct the traffic and also contain some programming logic. Right now we don't need to use them, so I'll just comment them out. Now this statement instantiates express and stores its reference in the app variable. Now we'll use this variable in the next few statements to configure Express.

This statement tells Express where to find its views. And the next statement specifies that the ejs rendering engine should be used to render the views. Then we have a few app.use statements that have been used to register some third-party middleware that the app uses. Now this app.use statement registers a built-in middleware called static that tells Express to serve static objects from the public directory. As a result, though all the images are stored in the images directory within the public directory, they are accessed at localhost:3000/images. After this we have a few app.use statements that we don't need right now. So I'll just comment them out. After this, we have some error handlers for the development and production environments. And finally, we export the app object as a module so that it can be accessed from other parts of the application if required. So we have seen how to register third-party and built-in middleware. Let us now see how to define and register middleware of our own.

So let me move here. And I'll paste the code to create some middleware functions here. Let us now take a look at the first function. This function is designed to respond to every request that is sent to the server by displaying the message, "Request received" on the console. Thereafter, it calls the next function to pass on the control to the next function in sequence. The next function is designed to respond to every get request on the application root. And it responds by simply displaying the message, "Get request received at application root." The next function is designed to respond to every post request on the application root. And it responds with the message, "Post request received at application root." The next function is designed to respond to every get request to the path that starts with /user. And it responds with the message, "Get details of all the users." Of course, a real application will need to respond with

real data, but for this demo we'll simply return a message. And the last function here is designed to respond to every get request to the path that starts with /user and that passes the /:id parameter. This function responds with the message, get details of the user with the given ID. Now that we have defined and registered our middleware, let us save the file and close it.

Now, before we start the application, let us install all the dependencies defined in the package.json file by using npm install command. Now all the dependencies have been installed. Let us now start the application by saying npm start. Now we have not yet created any front end for our application, and we have simply created some middleware to handle requests to HTTP endpoints. So we can test the app by using a REST client, such as Postman. For this, we'll need to open Google Chrome, and then go to chrome://apps. Here we need to select Postman.

Now we can use this tool to send requests to our REST application. First of all, let us send a request to the application root. I'll select the GET request from here and click the Send button. You can see that the message, "Get request received at application root" is displayed. Now let us make a POST request to the same URL and click the Send button. Now we get the message, "Post request received at application root." Now let us send a request to localhost:3000/user. And let us make it a GET request. Now the message that is displayed is "Get details of all the users." Let us now add an ID parameter here and click send. The message that we get now is "Get details of the user with id :1000." In this topic, we learned how to define and register middleware.

Implementing Routing

Learning Objective

After completing this topic, you should be able to

- *implement routing*

1.

In this topic, we'll learn about implementing routing. Routing determines the behavior of an application in response to a client's request to a particular endpoint. By endpoint, we mean the URI and the HTTP method such as get or post associated with a request. Routing can be implemented by using either application-level middleware or router-level middleware. Application-level middleware are bound to an instance of the app object, which conventionally denotes an instance of an Express application. You can define a route by using the app.<METHOD> function, where METHOD is an HTTP request method such as get or post, PATH is a path on the server, and HANDLER is the function that needs to be executed when the requested route matches the PATH that is specified in this function. In the given code, application-level middleware has been used to define two routes and their associated route handlers. The first handler handles all the requests made to the application root. And the second handler handles requests to paths that start with /about.

[The "app.js (~/app1) – gedit" window is open. Running along the top of the window is the toolbar including the "Save" button. The "app.js" tabbed page is open in this window. This tabbed page includes the following set of codes: var routes = require('./routes/index'); var users = require('./routes/users');]

Router-level middleware are bound to an instance of express.Router(). A router object is an isolated instance of middleware and routes. A router acts as a small application that can only perform middleware and routing functions. A router acts just like middleware. So you can set it as an argument to the app.use() method. You can also use it as an argument to another router's use() method. In the given example, let us first take a look at the code included in the app.js file. Here, we first require express and then create the app object. Next we require the api.js file and store its reference in a variable named routes. Now we specify routes as a handler for requests to the path /api. Now, in api.js, we define handlers for the get and post requests to the /posts subpath. This means that the handlers defined in the api.js file will get invoked whenever there is a get or post request to the path /api/posts.

Let us now see a demo to understand the concept of application-level and router-level middleware. Let us first create a new express application by using the express --ejs command. And I'll specify the folder name as app2. Now the folder structure of a new Express application has been created inside the app2 folder. Let us check the contents of the app2 folder. Now, within the app2 folder, we have a folder by the name of routes. Let me open this folder. Right now there are two files in this folder, but we'll be using only the index.js file. So let us delete the other file that is users.js. Now I'll go back to the app2 folder and then open the app.js file. Now, within this file, let us remove the users variable as it refers to the users file that we just deleted. But we'll retain the routes variable that refers to the index.js file. Similarly we go down. We'll remove this app.use statement that refers to the users variable, but we'll retain the app.use statement for the routes variable. We'll also change the path specified in this statement to /api. Now this app.use statement will route all requests sent to a path that starts with /api to the index.js file in the routes folder. So we'll need to define the router-level middleware in the index.js file. But, before that, let us define and register some application-level middleware.

[The presenter explains the following set of codes: app.use (function (req, res, next){ console.log ('Request received'); next(); });]

Let me paste the code to create some application-level middleware here. As discussed, application-level middleware are bound to an instance of the app object. All the middleware functions defined here are bound to an instance of the app object. The first function here is designed to respond to every request that is sent to the server. It responds to each request by displaying the message, "Request received" on the console. After this, it makes a call to the next() method to invoke the next function in sequence. The second function is designed to respond to every get request on the application root, and it responds by simply displaying the message, "Get request received at application root." The third function is designed to respond to every post request on the application root, and it responds with the message, "Post request received at application root." The fourth function is designed to respond to every get request to the path that starts with /user, and it responds with the message, "Get details of all the users." And the last function is designed to respond to every get request to the path that starts with /user and that passes an /:id parameter. This function responds with the message, get details of the user with the given ID. Now that the application-level middleware has been defined, let us see how to define the router-level middleware. As I mentioned earlier, router-level middleware needs to be defined in the index.js file within the routes folder.

Please note that the routes variable points to the index.js file within the routes folder. So this app.use statement will pass the control to the index.js file whenever a request is made to a path starting with /api. So let me Save this file and then open the index.js file. In this file, first of all we have stored the reference of express in the express variable. Then we have stored the reference to express.Router() in a variable named router. Then we have a middleware function here, which we don't need. I'll remove this function. And I'll add some middleware functions of my own. Let me paste a few middleware functions here. The first function is designed to respond to every get request to /api/product, and it responds with the message, "Get product list." The second middleware function in this file is designed to respond to every get request to /product/:id, and it responds with the message, get product with the given ID. Let us now save this file and close both the files. Now let us go back to the Terminal. Let me clear the screen. Now let us first move into the app2 folder. Now let us install the dependencies defined in the package.json file by using the npm install command. Now the dependencies have been installed. Let me clear the screen. Now let us start the application by using the npm start command.

Now let us open Postman to test the RESTful application that we just created. So, for that, I'll first open Google Chrome. And I'll go to chrome://apps, and then open Postman from here. Now let us first test some of our application-level middleware. So let us first send a GET request to localhost:3000. We get a response saying, "Get request received at application root." Now let us send a POST request to localhost:3000. We get a response saying, "Post request received at application root." Now let us send a GET request again to localhost:3000/user. We get a response saying, "Get details of all the users." Now let us test some of our router-level middleware. Let us first send a GET request to localhost:3000/api/product. We get a response saying, "Get product list." Now let us again send a GET request to localhost:3000/api/product/2000. We get a response saying, "Get details of product with id: 2000." In this topic, we learned about implementing routing. We learned how to define and register application-level and router-level middleware.

Creating a RESTful Web App with Node.js and Express

Learning Objective

After completing this topic, you should be able to

- *create and test a RESTful Web app*

1.

In this topic, we'll develop a RESTful web app with Node.js and Express. We want to develop an app that allows users to submit and view articles. We have already developed the front end of this application and it is visible on the screen right now. Now we want to develop the middleware for this application using Node.js and Express. The middleware will interact with the database to store and fetch the application data. However, we've still not learnt how to fetch and retrieve data from a database. So we'll simply create the REST endpoints and return appropriate messages indicating the action that needs to be performed by the REST endpoints. To start developing the app, we first need to open the Terminal window. Now let us first create the boilerplate code by using the express --ejs command. And I'll specify the name of the folder as articles. Now this command will create a folder named articles and create the entire folder structure of an express application within the articles folder. Let us now view the contents of the articles folder.

Let us open the routes folder. In this folder, we have two files – index.js and users.js. We need only one file, so let us remove the users.js file. And we also want to rename the index.js file to api.js. We'll come back to this file in a while. Right now let's go to the app.js file. Now, in this file, we'll add the statement var api = require('./routes/api'). Now, in this statement, we require the api.js file and store its reference in the api variable. Now we can remove the references to the index.js file and the users.js file. Again, we come down and we add the statement app.use('/api', api) and remove the app.use statements for the index.js and users.js modules. Let us now Save the file and Open the api.js file.

Let us remove the middleware function that is already defined in this file and replace it with our own middleware functions. Let me paste the code to create the middleware functions. Now here we have created four middleware functions. The first function is designed to respond to all get requests for paths that start with /api/articles. The second function is designed to respond to all get requests for paths that start with /api/articles/id. The third function is designed to respond to all post requests for paths that start with /api/articles/id. And the last function is designed to respond to all put requests for paths that start with /api/articles/id. Let us Save this file and close it. Let us go back to the Terminal. Let me clear the screen first. And now let me move into the articles folder. No I'll install all the dependencies that are defined in the package.json file by using the npm install command.

Now the dependencies have been installed. Let me clear the screen again. So now we can start the application by using the npm start command. Let us now test the application with a REST client, such as Postman. For this, I'll open Google Chrome. And I'll go to chrome://apps. So there I've installed Postman. So I'll open Postman from here. Now first of all, let us send a GET request to localhost:3000/api/articles. We get a message saying "Get details of all articles." Now let us send a GET request to localhost:3000/api/articles/3000. We get a message saying "Get details of article with id:3000." Let us now send a POST request to the same URL. We get a message saying "Store the details of article with id 3000 in the database." Now let us send a PUT request to the same URL. We get a message saying "Update the details of article with id 3000 in the database." In this topic, we created a RESTful web app with Node.js and Express.

The Key Features of MondoDB

Learning Objective

After completing this topic, you should be able to

- *identify the key concepts of MongoDB*

1.

In this topic, we'll learn about the key features of MongoDB. MongoDB is an open-source document database. It provides high performance, high availability, and easy scaling. A MongoDB deployment can hold a number of databases. A database in MongoDB is a set of collections. A collection is equivalent to a table in a relational database and consists of a set of documents. A document is nothing but a record, which is composed of key-value pairs. MongoDB documents are similar to JSON objects. The fields in a MongoDB document may include arrays, other documents, and arrays of documents. In the given example, a document with three fields – name, age, and hobbies – has been defined where name is a string, age is a number, and hobbies is an array of strings. MongoDB supports a dynamic schema. This means that there is no well-defined model according to which data is stored in the database. The documents stored in the database can have varying sets of fields and different data types for the fields.

Using a document database provides several advantages. Storing data in the form of documents provides better synergy between the database and the application. This is because documents are nothing but objects that map to the objects used in many programming languages. The embedded documents and arrays used in databases reduce the need to create joins and the schemaless nature of MongoDB is an ideal match for the constantly evolving data structures in web applications. Another advantage is that the data types supported by MongoDB are very similar to the native JavaScript data types. In this topic, we learned about some key features and advantages of MongoDB.

Working with Basic MongoDB Commands

Learning Objective

After completing this topic, you should be able to

- *work with basic MongoDB commands by using the MongoDB shell interface*

1.

In this topic, we'll learn to work with some basic MongoDB commands. Once you have installed and configured MongoDB, you should be able to start the MongoDB server by using the mongod command. Now MongoDB is running and is waiting for connections on port 27017. Now let us open a new terminal window and use the mongo command to open a new MongoDB client. So now a MongoDB client is running. You can use the cls command to clear the screen. You can open an existing database or create a new database by using the use <database_name> command. Now suppose, you want to create a database by the name of articledb to store the details of articles submitted by users on a web site. For this, you can use the command use articledb. Now this creates a new database by the name of articledb and also opens it. To check your currently selected database, you can use the db command. You can see which all databases exist on MongoDB by using the show dbs command. You'll notice that the articledb database is not there in this list. This is because, right now, there is no data in this database. For a database to be shown in this list, there must be some data in the database. You can drop a database by using the db.dropDatabase command. This will delete the selected database.

Right now, I don't want to delete the articledb database. So I'll just erase this command. Now let us create a collection by the name of collection1 within the articledb database. For this, I'll use the db.createCollection command. And, as a parameter, I'll pass the name "collection1". Now please note that you don't need to explicitly create a collection in MongoDB. A collection is automatically created when you insert the first document in a collection. We'll see this in a bit. Right now, let's use the command show collections to see the list of collections that exist in the articledb database. You can see the collection named collection1 that you just created. Let us now create a collection named articles and insert the record in the collection. Actually, we don't need to create the collection explicitly. It will automatically be created when I add a document to the collection by using the insert command. So let me say db.articles.insert({"username":"Tom", "title":"Hello", "text":"Hello World", "timestamp":Date.now()}). Now this creates a collection named articles and adds a document or a record to the collection.

Now you can query data from the articles collection by using the find command – let's say db.articles.find(). You'll notice that the data is displayed in a very unstructured format. You can display it in a better manner by using the command db.articles.find().pretty(). Let me clear the screen. Let us now see how to update a document in a collection. For example, if I need to change the username in the document that I just created, I can use the command db.articles.update ({"username":"Tom"}, {$set:{ "username":"Nancy"}}). Now this command will search for a document where the username is Tom and it will replace the username of this particular document with Nancy. So, now the document has been updated, let us see if the change has taken effect by querying the collection – let's say db.articles.find().pretty(). So you can see that the username has been changed to Nancy. Now you can remove the collection that you just created. To remove a collection, you can use the db.articles.drop() command. This will delete the collection named articles. In this topic, we learned to work with some basic MongoDB commands.

The Key Features of Mongoose

Learning Objective

After completing this topic, you should be able to

- *identify key concepts of mongoose*

1.

In this topic, we will learn about Mongoose and its key features. To connect a Node application to MongoDB, you can use the mongodb module. For this, you can install the mongodb module, by using the npm install mongodb command. If you are familiar with SQL, you will know that SQL enforces schema. However, MongoDB doesn't enforce any schema. This means that it does not require all documents in a collection to have the same structure. Though this provides a lot of flexibility, it may also cause problems because if you don't have a defined schema, you can insert just anything in the database. Documents don't need to have the same structure. When you want to keep your data organized and structured, it is sometimes necessary for documents to be similar. This is where Mongoose can be useful. Mongoose is a Node.js Object Data Mapping or ODM module, which adds MongoDB support to an Express application. Mongoose uses schemas to model entities and to save them as MongoDB documents. The design goal of Mongoose is to bridge the gap between the schemaless approach of MongoDB and the requirements of real-world application development. To use Mongoose in a Node.js Express application, you need to install Mongoose by using the npm install mongoose command. Now, to connect to MongoDB from within an application, you can use the mongoose.connect function. In this topic, we learned about Mongoose and its key features.

Modeling with the Mongoose Schema

Learning Objective

After completing this topic, you should be able to

- *model with the mongoose schema*

1.

In this topic, we'll learn about modeling with the mongoose schema. Mongoose is a robust node.js object relational mapping module that provides MongoDB support to an Express application. This means that it translates data in the database to JavaScript objects that you use in your application. So mongoose provides you the ability to model objects and save them as MongoDB documents. While MongoDB is schemaless, mongoose offers you the opportunity to define schemas for your models. The design goal of mongoose is to bridge the gap between schemaless approach of MongoDB and the requirements of a real-world application. Let us see how to model with the mongoose schema. For this, let us first create the boilerplate code for an Express application by using the express --ejs command. And I'll specify the folder name as mongoose_demo. Now the folder structure for the Express application has been created. Let me first clear the screen and then move into the mongoose_demo folder.

Now I'll install the dependencies by using the npm install command. The dependencies defined in the package.json file have been installed. Let me clear the screen. Now, in addition to the dependencies that are already defined in the package.json file, we need one more dependency, that is the mongoose module. So let us install the mongoose module by using the npm install mongoose command. And, with this, we can use the save option so that the information about the mongoose package is stored in the package.json file. Now all the dependencies including mongoose have been installed. Let me clear the screen. Now I'll open the app.js file and define a mongoose schema and model in it. First of all, we need to require the mongoose module and then we need to connect to a database on a MongoDB instance. So I'll use mongoose.connect(' mongodb://localhost/mydb '). Here mydb is the name of the database on the MongoDB server. Let us now define a schema in this file that defines the structure of the data that needs to be stored in the database.

So I'll define UserSchema = new mongoose.Schema – firstname:, which is a 'string' and lastname:, which is again a 'string'. I'll now define a model based on the schema that I just defined. So var User = mongoose.model, then I specify the name of the model as 'User' and the name of the schema as UserSchema. Now let us create an object based on the user model. So var user1 = new User(). Now I'll assign some values to the firstname and lastname attributes of this object. So user1.firstname = "Sandra" and user1.lastname = "Williams". Now I'll call the save() method on the user1 object to save the user1 document into the database. So user1.save(function (err, user1). Now, if there is an error, respond with an error, otherwise the document is saved. Let us now Save this file and close it.

Now, before we start the application, let us start a new instance of MongoDB. For this I'll use the mongod command. Now, that a MongoDB instance is running, let us go back and start the application. So I'll say npm start. Oh, so I have an error here. I think I must have missed a bracket. So I'll open app.js. Yeah, here. Let me Save the file again and close it. And again let me start the application. Now the code that we wrote in the application to save the user details in the database has executed. Let us see if it has been able to write something into the database. So I'll open a new instance of Terminal and now I'll open a MongoDB client in this window by using the mongo command. I'll now switch to the mydb database by using the use mydb command. Let us see the list of collections in this database. For this I'll use the show collections command. You can see that our collection named users has been created in

the mydb database. Now we can see the documents in the collection by using the db.users.find() command.

So a new document has been created but I see a problem here. It only has the firstname field and the lastname field is not there. Let me go back to the app.js file and see if I made a mistake there. Yes, the spelling of lastname here is incorrect and not as per the defined schema. So that is why it was not stored in the database. So I have corrected the spelling here. Let me Save this file and close it. I'll now start the application again. Now let us go back to the MongoDB client and see if we get the correct data this time. So I'll say db.users.find(). So yes, we have the complete document this time. So, in this topic, we learnt about modeling with the mongoose schema.

Integrating MongoDB with a Node.js Project

Learning Objective

After completing this topic, you should be able to

- *integrate MongoDb with a Node.js project*

1.

In this topic, we'll learn how to integrate MongoDB with the Node.js application. I've used Node.js and Express to develop the middleware for an application that allows users to submit and view articles. Right now, this application simply responds to HTTP requests such as GET and POST with relevant messages. Let us run this app and test it in Postman. To run the app, I'll open a new Terminal window, move into the application folder, and then use the npm start command to start the application. I'll now use the Chrome App Launcher to open Postman. Let me first send a GET request to the URL localhost:3000/api/articles. I get a response saying "Get details of all articles." Let me now send a POST request to the same URL. I get a response saying "Store the details of submitted article in the database."

So, right now, this application simply displays relevant messages. Now we want this application to work with real data that is stored in a MongoDB database. Let us see how to integrate MongoDB with this application by using mongoose. First of all, let us open the folder that contains the application files and subfolders. Let us now define the schema for the data that needs to be stored in the database. For this I'll create a New Folder and call it models. Now, in this folder, I'll create a new file by the name of articles.js. Now, in this file, I'll add some code. Now, first of all, I need to require mongoose. So I'll say var mongoose = require ('mongoose'). I'll now define the schema for the documents that need to be stored in the database. Let me paste the code to create the schema. So we have created a schema by the name of ArticleSchema which has four attributes – username, title, text, and timestamp. Now, after creating the schema, I need to define a mongoose model, which I'll name as Article. So I'll say mongoose.model – 'Article' of the type ArticleSchema. I'll now Save the file and close it. I'll now open the app.js file.

Now, in this file first of all, I need to require mongoose. Then I need to require the mongoose schema module that we just created. So I'll use the require method. And I'll pass the path to the schema module that we just created. And finally, we need to connect to the MongoDB database. For this I'll use the mongoose.connect method and I'll pass in the path to the MongoDB database as a parameter. Let us now Save this file too and close it. I'll now open the api.js file within the routes folder. Now, in this file, we have defined the middleware functions for various HTTP requests to the application. Right now, these functions are simply displaying relevant messages like this one. But we want them to store and retrieve real data. But, before that, we need to create an instance of the Article model that we just created. For this, first of all, I need to require mongoose and then I can create an instance of the Article model by saying var Article = mongoose.model('Article').

Let us now change the way these middleware functions behave. I'll first remove this statement from here and replace it with some other code. The Article.find method will fetch the details of all the documents stored in the database and then send the articles object as a response. The if statement here handles errors, if any, that occur while fetching the data from the database. Let me now remove this statement from here and replace it with some other code. The Article.findById method will fetch the details of the article whose ID matches the ID passed as a parameter with the request. And then it will send the article object as a response. The if statement here handles errors, if any, that occur while fetching the data from the database. Let me now remove the statement from here and replace it with some other code. Now, in

this code first of all, we create an Article object. Then we initialize the username, title, and text properties of this object with the respective values from the request body. Then we save the Article object in the database. The if statement here handles errors, if any, that occur while storing the data to the database. If the save is successful, the Article object is sent as a response to the client. Now let us Save this file and close it.

Since my application now uses mongoose, we need to install a new dependency named mongoose. For this I'll go back to the Terminal window, terminate the running application, and then use the npm install mongoose --save command to install mongoose. So now mongoose is installed. Since my application now uses MongoDB, I need to ensure that an instance of MongoDB is running. For this I'll open another Terminal window and use the mongod command to start a new instance of MongoDB server. Now let us go back to the other command window, clear the screen, and then use the npm start command to start the application.

Again I'll go back to Postman and, first of all, I'll send a POST request to localhost:3000/api/articles. Now, with the POST request, we need to send some data. So let me select the Body tab here and click on the raw option and ensure that JSON is selected from the drop-down list. Let us now enter some data in JSON form here and then click the Send button. Let us scroll down a bit. So the details have been saved in the MongoDB database. Let us post one more article. So I'll just change the name to Nancy and message to "Hello Everyone" and click the Send button. So this record again has been saved to the MongoDB database. Let us now send a GET request to localhost:3000/api/articles. Now you can see that both the documents have been saved to the MongoDB database. In this topic, we learnt how to integrate MongoDB with a Node.js application.

Using an AngularJS Application as a View

Learning Objective

After completing this topic, you should be able to

- *integrate an AngularJS application as a view with a Node.js project*

1.

In this topic, we'll learn how to use an AngularJS application as a view in a Node.js Express application. I've already created an AngularJS application that allows users to submit and view articles. I've also created a REST API with Node.js and Express that stores and retrieves articles in the database. I'll now integrate the two apps into one. First of all, I'll open the Express application and go into the routes folder. Now, in this folder, I already have a file named api.js where I've defined some routing. Now I'll create one more file here named index.js. And, within this file, I'll define some more routing. So let me paste the code to include the routing functionality. Now this code creates a route handler for the application root. This means that if a user sends a request to the application root, the index.html file will be rendered. Now let me Save this file and close it. And now let us open the app.js file. Now, in this file, I need to add a reference to the index.js file that I just created within the routes folder. I'll also add an app.use statement here with the first parameter being the application root and the second parameter being the index variable that I just initialized.

The two statements that we just added basically direct any requests for the application root to the index.js file within the routes folder. Let us now Save this file and close it. Let me now open the AngularJS app that I have created. And I'll Copy the main.html file from this folder, go back to my Express application and into the public folder, and I'll Paste the main.html file here. I'll now rename this file to index.html. Again I'll go back to the AngularJS application and into the javascripts folder. Now I'll Copy this articleApp.js file and then go back to my Express application, into the public folder, and the javascripts folder. And then I'll Paste this file here. Now my application uses MongoDb. So, before I run the application, I need to ensure that an instance of MongoDB is running. So I'll open a Terminal window and use the mongod command to start the MongoDB server. Let me now open a new Terminal window and move into the articles folder where my Express application is stored. I'll now use the npm start command to start the application. So now the application has started. Let me open a web browser and go to localhost:3000.

So the AngularJS web page that I created is now on my web server and I can access it from there. Right now, I only have a single page in my application. Let me create another web page named about.html. So I'll go to the public folder. Here I have my index.html. I'll add a new file here, and I'll name it as about.html. Let me open this file and add some code here. So this is a simple HTML page that displays the message "Welcome to the About page." Let me Save this file and close it. Let me go back to this folder. So now we have two different views. And we can load the appropriate view as per requirement. However, with AngularJS, you don't have to load all your resources again and again. Instead, you can combine these into a single-page application and create partial templates that will load into the view whenever we need them. To create a single-page application, let us first create a copy of our index.html file and save it as main.html. Let us now open the index.html file. Now we'll make a base template layout in this file. And, within this base template, we'll pull in either the main.html view or the about.html view as per requirement.

So let us remove all the code within the body element including this div tag with an ng-controller attribute. And now I'll replace this code with the div element that has an ng-view attribute. So this div basically acts as a placeholder for the partial views that we'll be creating. Let me Save this file and close

it. I'll now open the main.html file. And I'll remove everything except the contents of this div element. So this means I'll be removing this div element with the ng-controller attribute also. I'll now Save the file and close it. Similarly, I'll open the about.html file and remove everything except the contents of the body element. I'll now Save the file and close it. So now we have made our views simpler. As I mentioned, I've also removed the reference of the controller from the HTML files. So how do we know which controller will be used by which partial page? We'll handle this by defining routes. And, for this, we'll need to add the angular-route.js script. We need to add the script in the index.html file. So let me open this file. And I'll paste the script tag to include the angular-route module in this file. We also need to add some links for navigation. So I'll add a simple navigation bar with two links – one to main.html and the other to about.html.

Let us now Save this file and close it. Let us now go into the javascripts folder and open the articleApp.js file. Now, if I have to use routes, I need to include the ngRoute dependency here. Now I'll add some routing code here. Now this code means that when a request is received on the application root, the main.html page should be displayed and the controller that should be used for the request is mainController. Similarly, when a request is received on the path /about, the about.html page should be displayed. And there is no controller that is associated with this request. Let us now Save and close this file. I'll now go back to the Terminal window, terminate the running application, and then start the application again.

So, now that my application is running, I'll go back to the browser and reload this page. You can see that the navigation bar that we added to the index.html file is being displayed here. Also, index.html had a placeholder for a partial view. And, within that placeholder, the content of the main.html file is being displayed. Let us now click the About link. You can see that the URL that is visible here has changed and the about.html partial has got loaded into the placeholder. So effectively, this is a single-page application in which the different partial pages are being loaded. In this topic, we learned about using an AngularJS application as a view. We also learned about developing single-page applications.

Connecting to the Server

Learning Objective

After completing this topic, you should be able to

- *use the $http service to connect to the server*

1.

In this topic, we'll learn how to make API calls to the server from an AngularJS front end. We have already developed the AngularJS front end and the middleware APIs for an application that can be used to submit and view articles. The AngularJS front end has been ported to the server. And, once the server is started by using the npm start command, the AngularJS front end can be accessed from a web browser by going to localhost:3000. Now, if I submit an article using this web page, the article is displayed at the bottom of the web page. But, if I reload the page, the article is lost. This is because we are storing the articles in an array and not in a persistent storage. Now we have also defined the middleware APIs to store and retrieve article details in the database. Let us use Postman to send a few requests to these APIs.

First of all, let us send a POST request to localhost:3000/api/articles. I'll also send some data along with this request. So let me paste the JSON data here and Send the request. Now the request has been sent. Let us check whether the data has been stored in the database or not. I've already opened the articlesDB database in a MongoDB client. So let me switch to the client and use the db.articles.find().pretty() command to check whether the data has been stored in the database. You can see that the data has been properly stored in the database. Now let us go back to Postman and send a GET request to the same URL. You can see that the data in the database has been properly fetched. So now that we have confirmed that the APIs are working properly, let us make our AngularJS app consume these APIs. For this, I need to open the folder structure of my application, go into the public folder and then the javascripts folder, and then open the articleApp.js file.

Now, in order to make API calls to the server, I need to use the $http service. So I need to declare this service as a dependency for my controller. So I'll add $http here as a dependency. Now I'll remove this $scope.post function from here and replace it with some other code. The $http.get method here sends a request to the /api/articles endpoint that retrieves all the documents from the database and sends them as response. If the documents are successfully retrieved, they are assigned to the $scope.articles variable here. The $http.post method sends a post request to the /api/articles endpoint and passes the newly created $scope.newArticle object as a parameter to this method. If the article is successfully saved, it is pushed in the $scope.articles array from where it will be displayed on the user interface through data binding. And finally, we reset the newArticle object to clear the text boxes once the data has been submitted. Let me now Save the file and close it. I'll now go back to the Terminal window and terminate the running application. I'll now start the application again, go back to the web browser, and reload the page.

You can see that the articles that are already stored in the database are appearing at the bottom of the page. I'll now post a new article. Now, when I submit this article, the new article also gets added to the list of articles at the bottom of the page. Let us now open a MongoDB client and check whether the new article has been added to the database. You can see that the new article has got added to the database. In this topic, we learned how to make API calls to the server from an AngularJS front end.

Creating and Using Services

Learning Objective

After completing this topic, you should be able to

- *create and use services*

1.

In this topic, we'll learn about creating and using services. Services are used to organize and share code across an application. For example, if there are two controllers that need to obtain the data about users from a server, then instead of including the logic to fetch user data in each of the two controllers, the logic to fetch data can be included in a service. The two controllers can then use the service to access the user data. AngularJS provides some predefined services such as $http and $route. You can also create custom services to organize and reuse code. Though there are multiple methods for creating a service in AngularJS, right now we'll focus on only one method that is the factory() method. So we'll learn how to create a service by using the factory() method.

In the given code, we create a service by using the factory() method. The service simply creates an object, populates it with an array containing the names of users, and then returns the object. This service doesn't have any dependency. However, we may create services that need dependencies. In such a case, the dependencies can be declared by using the same method as is used for declaring dependencies for controllers. To use the service that we saw on the previous slide, in a controller, you need to first declare the name of the service as a dependency of the controller. You can then call the service from this controller by invoking the getUsers() method of the userService that we saw on the previous slide. Let us now see a demo to understand the concept of services better. I've created an application that can be used to submit and view articles. Let us go into the public folder and then the javascripts folder and then open the articleApp.js file.

In this file, we have defined a controller named mainController. Now, within this controller, we have a $http.get method that fetches the details of all the articles in the database. Now we'll create a service to fetch the details of all the articles in the database. And then, instead of using the $http.get method to fetch the details here, we'll call the service to obtain the article details. So let me remove this code from here. And now I'll add some code here to create the service. Now here we have used the app.factory method to create a service named articleService. Now, in this service, we have defined a method named getArticles that uses the $http.get method to fetch the data from the database. Also, as our service uses the $http service, $http is declared as a dependency for this service. Now we'll call this service from within the mainController to fetch the articles from the database. But, before that, we need to declare the service as a dependency for the mainController.

So I'll add articleService as a dependency here. Let us now add the code to access the service from here. Here we are calling the getArticles() method that we defined in the articleService. The data returned by the service is being assigned to the $scope.articles variable, which will then display the data on the web page through data binding. Let us now Save this file and close it. Let us now go to the Terminal window, move into the application folder, and start the application. Now let us open a web browser and go to localhost:3000. You can see that the articles from the database have been properly fetched and are displayed at the bottom of the web page. So this means our service is working fine. In this topic, we learned about creating and using services.

Using the ngResource Module

Learning Objective

After completing this topic, you should be able to

- *use the ngResource module*

1.

In this topic, we'll learn about using the ngResource module. The ngResource module enables interaction with RESTful web services by using the $resource service. $resource is a higher-level abstraction over the $http service. While $http handles general purpose communication with the HTTP server, $resource is used specifically for interacting with RESTful APIs. To use $resource, you need to include the ngResource module. The ngResource module is a JavaScript file that can be included as a script in an HTML page. You can either download the file or use its CDN. Let us now see how to use the ngResource module. I've created an application that can be used to submit and view articles. Right now, the application uses the $http service to interact with the REST APIs. Let us see how to use the $resource service instead. First of all, we need to include the JavaScript file for ngResource module in our index.html file.

So let me open the index.html file and add a reference to the angular-resource.js file. Let me now Save the file and close it. I'll now go into the javascripts folder and open the articleApp.js file. Now, in this file, we're using the $http service to interact with the REST APIs. Let us see how to use $resource instead. First of all, we need to add ngResource as a dependency for the module. Now let us remove the $http dependency from here and replace it with $resource. I'll now remove all the code in this service and replace it with a single statement that is return $resource('/api/articles').

Now here we have created a resource object that will help us interact with RESTful server-side data sources at the endpoint /api/articles. Now we can use this resource object to fetch and post data to the database. To fetch data from the database, I was earlier using the getArticles() method of the articleService. But now I have removed this method from the service. So let me remove this method call from here. And again, I'll replace it with a single statement that is $scope.articles = articleService.query(). Here we're using the query() method of the resource object returned by the articleService to invoke the httpGet() method associated with the endpoint that the resource is linked to. To save data in the database, I was earlier using the $http.post method. Let me now remove this code and I'll replace it with some other code. Here we're using the save method of the resource object returned by the articleService to invoke the $http.post method associated with the endpoint that the resource object is linked to. This will save the article object in the database and push the newly created article object in the $scope.articles array. The contents of this array will be displayed on the web page through data binding.

Let us now Save this file and close it. I'll now switch to the Terminal window, move into the application folder, and start the application. I'll now go to a web browser and go to localhost:3000. Now the articles already stored in the database are being displayed at the bottom of the web page. You can also submit new articles, which will be added to the list at the bottom of the page and will also be stored in the database. In this topic, we learned how to use the ngResource module.

Exercise: Create a Simple MEAN Application

Learning Objective

After completing this topic, you should be able to

- *create a simple MEAN application*

1.

Exercise Overview
Now that you have learned the basics of developing a MEAN application, it's time to put that knowledge to work. In this exercise, you'll create the AngularJS front end for an application that allows users to post and view details, such as title, venue, and date about events. You'll also create a RESTful API to handle get and post requests for event details using Node.js, Express, and MongoDB. Finally, you'll integrate the front end with the back end. At this point, you can pause the video and perform the exercise. Once you've finished, resume this video and see how I would do it.

Solution
Let us first see how to create the front end for the application. The front end is fairly simple. And I've already written the code to make it working. I'll just quickly explain it to you. Let us first take a look at the eventApp.js file. In this file, we have created a controller within a module. The controller contains the code to create a model and to store the event details submitted by a user in an array. The $scope parameter here helps bind the model data with the UI elements. Let us close this file and open the main.html file. Now, in this file, we have added the code to accept event details from the user and show them on a web page. Let us close this file and open the contactus.html file. This file creates a simple web page that displays a welcome message. Again, let us close this file. And now let us view the main.html file in a web browser.

Now here let us add some event details – MEAN.js training in ABC Training Room, LAMP training in XYZ training room on 12/03/2015. So the details of the event are appearing in the list at the bottom of the web page. So the front end part of the application is done. Now let us take a look at the back end. To create the back end, first of all I'll open a Terminal window and use the express --ejs command with folder name events to create the boilerplate code for the application. Now let me go into the folder structure of the newly created application and go into the routes folder. Now here I'll rename the users.js file as api.js. Now this is the file in which I'll be defining the APIs for storing and retrieving event details. But before that, I need to add a reference to this file in the app.js file.

Now, as I have changed the name of the users file in the routes folder, I need to make the same changes here also. And again, I'll change the references here. I'll Save the file and close it. I'll now create the mongoose schema model according to which data will be stored in the database. For this, I'll create a New Folder named models. And, within this folder, I'll add a New Document named models.js. Now, in this folder, I'll add the code to create a model named Event based on the EventSchema, which in turn has three attributes – title, venue, and date. Let me Save this file and close it. We now need to open the app.js file and add a few statements to connect to the MongoDB database using mongoose. So let me paste the code to connect to the MongoDB database. Save the file and close it. We'll now create the APIs for storing and retrieving event details in the database. So let me open the api.js file. I'll remove the middleware function that is already included in this file. And I'll replace it with APIs of my own. Now here I have defined two middleware functions – one to handle the get requests for the events and the other to handle the post requests for the events. Let me now Save this file and close it. As we're using mongoose and mongoose is not defined as a dependency in the package.json file, we need to add this to the package.json file.

So let me add mongoose as a dependency here. Save the file and close it. I'll now go to the Terminal window, clear the screen, move into the events folder, and install all the dependencies. This may take some time. Now all the dependencies are installed. Let me clear the screen and start the application. I'll now use Postman to send a few requests to the APIs. So let me first send a POST request to localhost:3000/api/events. And I'll send some data along with the request. So here is my data, and I'm sending the request. So the data has been successfully stored to the database. Let me now send a GET request. Data has been fetched from the database. So now the back end is ready. We now need to integrate the front end with the back end. To integrate the front end with the back end, I'll first go into the folder structure of my front end application and copy the eventApp.js file from there. I'll then go back into my Express application and Paste this file in the public/javascripts folder. Again I'll go into the folder structure of my front end application and Copy both the HTML files from there. And then I'll paste them within the public folder in my Express application.

I'll create a Copy of the main.html file and rename it as index.html. I'll now open index.html to define the base template layout for the application. For this, I'll remove everything from within the <body> tag and replace it with a element containing the ng-view directive. I'll also add references to the angular-route.min.js and angular-resource.min.js files here. Further, I'll also add some links for navigation. Let me now Save this file and close it. We'll now convert our main.html and contactus.html files into partial views. Let me first open main.html. I'll remove everything from this file except the contents of this element. I'll now Save this file and close it. I'll now open the contactus.html file and remove everything from this file except the contents of the <body> element. Again I'll Save the file and close it. Let us now open the eventApp.js file. Now, in this file, first of all I'll declare a dependency on ngRoute and ngResource. Then I'll add some front end routing code. And then I'll define a service to interact with our REST APIs. Now I'll invoke the service from within this controller to fetch data from the database. But, for that, I need to declare the service as a dependency for this controller.

I'll now invoke the service from within this post method to save data into the database. Let me now Save this file and close it. I'll now go to the Terminal, terminate the running app, clear the screen, and start the app again. I'll now open a browser and go to localhost:3000. You can see that a form is being displayed. And below the form, the data from the database has been fetched. Let me add one more event. So AngularJS training, ABC Training Room on 11/29/2015. So the data has been displayed below the form. So this means that our MEAN application is working as expected.