Course Transcript

Securing, Testing, and Deploying MEAN Applications

Securing a MEAN Application

Testing MEAN Applications

Testing Server-side Components

Testing Client-side Components

Deploying a MEAN App on Heroku

1. Introduction to Deployment

2. Installing the Heroku Toolbelt

3. Preparing an App for Deployment

4. Deploying an App to Heroku

5. Provisioning a Database

Scaffolding Using Yeoman Generators

1. Introduction to Yeoman Generators

2. Installing the Yeoman MEAN Generator

3. Using the Yeoman MEAN Generator

Practice: Deploying a MEAN Application

1. Exercise: Deploy a 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 Securing, Testing, and Deploying MEAN applications. When developing any application, it is important to ensure that the application is secure from unauthorized and malicious access. It is also important to test the application to ensure that it works properly in all situations. Finally, once the application is developed, secured, and tested, it needs to be deployed on a server. This course explains how to secure, test, and deploy MEAN applications.

Introduction to Authentication and Authorization

Learning Objective

After completing this topic, you should be able to

- *identify the fundamentals of authentication and authorization*

1.

When developing any application, it is important to ensure that the application is secure from unauthorized and malicious access. In this topic, we'll learn about two key concepts of application security that are – authentication and authorization. Authentication confirms that the users are who they say they are. For example, when a user logs on to a web site by providing a username and password, the username and password are verified against the credentials stored in a database to confirm that the user is who he or she claims to be. This is called authentication. Authorization, on the other hand, confirms whether a user must be granted access to a particular resource or not. For example, once the users are logged on to a web site, they must be able to access only the information for which they are authorized. Let us take the example of a blogging web site. When users log on to a blogging web site, they can view the blogs of all the users. But they should be able to edit or delete their own blogs only. This is called authorization. In this topic, we learned about the difference between authentication and authorization.

Setting Up Views for Sign Up and Sign In

Learning Objective

After completing this topic, you should be able to

- *set up views for sign up and sign in*

1.

In this topic, we'll learn how to set up the views for sign up and sign in. I've already created a web application that allows users to submit and view Articles. Now we want to add authentication to this web application. We'll start by setting up the views for sign up and sign in. For this I'll go into the public folder of my application and add two files named signup.html and signin.html. I'll first open signup.html and add some code to create the signup form. I'll now Save the file and close it. I'll now open signin.html and add some code to create the signin form. Again I'll Save the file and close it. I'll now open index.html and add links to the signup page and the signin page. Again I'll Save the file and close it. Now I'll open articleApp.js file and add some front-end routing code to it. Now all requests to the path /signin will be directed to signin.html, and all requests to the path /signup will be directed to signup.html. A controller named authController will be associated with both these types of requests.

[The "localhost:3000/#/" web address is open in the Google chrome web browser. The web page has the header as Articles. There are two links above the header, Articles and About. There are three text fields below the header for name, article title, and article text. Below the text fields is the Post button. The previously posted articles are displayed below the Post button. The presenter navigates to the public folder. To create a new file, she clicks on empty document in the new document option.]

I'll now add a controller named authController to this file. This controller declares two scope variables – one to store the details of a user and the other to store a message. It also creates two functions that are bound to the $scope. The first function handles signin requests and responds with a message saying that a signin request has been received. The second function handles signup requests and responds with a message saying that a sign up request has been received. Let us Save this file and close it. I'll now go to a Terminal window, terminate the running application, and start it again. I'll now go to the browser and reload the page. You can see that the Sign Up and Sign In links are now visible. Let me click the Sign Up link and provide a Username and Password to Sign Up. We get a message saying, "Sign up request received for user : sam." Now let me click the Sign In button. And again provide a Username and Password. Again we get a message saying that a Sign in request has been received for user : sam. In this topic, we learned how to set up the views for Sign Up and Sign In.

[She navigates to the web browser and reloads the address to verify the creation. The Articles page now has four links: Articles, About, Sign Up, and Sign In. She clicks the Sign Up link. A new web page opens. This page has the header as Register. Below the header, there are two text fields, username and password. Below the text fields is the Sign Up button. She then clicks Sign In link. A new web page opens. This page has the header as Sign In. Below the header, there are two text fields, username and password. Below the text fields is the Sign In button. She fills in Sam's details to verify the creation.]

Fundamentals of Passport Authentication

Learning Objective

After completing this topic, you should be able to

- *identify the fundamentals of passport authentication*

1.

In this topic, we'll learn about Passport authentication. Passport is a Node.js middleware that helps authenticate requests sent to an express application. Passport is designed to serve a single purpose that is authentication. And all the other functionality is handled by other parts of the application. This separation of concerns keeps the code clean and maintainable. It also makes Passport extremely easy to integrate into any application. As each application has unique authentication requirements, Passport uses an extensible set of plugins known as strategies for various types of authentication. The most widely used strategy is called local. This strategy involves authenticating users through a username and password. This strategy is supported by the passport-local module. Another strategy is called OpenID, which is an open standard for federated authentication. In this strategy, users present an open ID to sign in to a web site after choosing an open ID provider. The provider issues an assertion to confirm the user's identity and the web site verifies this assertion to sign the user in. This strategy is supported by the passport-openid module. Another strategy is Oauth that enables authentication through Oauth authentication providers, such as Facebook, Twitter, and Google. This strategy is supported by the passport-oauth module. In this topic, we learned about Passport authentication and some strategies that Passport uses for authentication.

Creating the User Model

Learning Objective

After completing this topic, you should be able to

- *create the user model*

1.

In this topic, we'll learn how to create the user model that we'll use to store data into the database for the purpose of authentication. To create the user model, I'll go into the models folder and create a new file by the name of users.js. I'll now open this file and add some code to create the user model. Now I've created a model by the name of User that is based on UserSchema, which in turn contains two properties – username and password. Let us also add some code here to store some user details in the database. We need this so that we can check whether the model is being able to save the data to the database. For this I'll add these two User.create statements here. Let me now Save this file and now I'll Open the app.js file. Now, in this file, I'll add a require statement for the users.js file that I've just created. I'll now Save this file and close it.

*[The "*users.js (~/articles/models) – gedit" window is open. The presenter adds two user statements "User.create ({username: 'sam', password: 'sam'})" and "User.create ({username: 'tom', password: 'tom'})". The "*app.js (~/articles/models) – gedit" window is open. The presenter adds "require ('. /models/users')" in the file.]*

I'll now open a Terminal window and start the mongod process. Let me open another Terminal window, moving to the articles folder and start the application. Now the application has started and it would have created the users in the database. So let us open another Terminal window and run the mongo process to open a MongoDB client. I'll switch to the articlesDB – database – and use the db.users.find().pretty() command. You can see that the details of the users have been added to the database. Now as we actually want the passwords to be hashed and secured before they are stored in the database. Right now, I'll just remove the user details that I have added. For this I'll use the db.users.remove({}) command. Also let us go back to the users.js file and remove that code that we added to create the users. I'll now Save the file. In this topic, we learned how to create the user model that can be used to store user details in the database.

[The presenter types "use articlesDB" in the terminal to switch to the articles DB database.]

Writing the Server-side Code for Authentication

Learning Objective

After completing this topic, you should be able to

- *write server-side code for authentication*

1.

In this topic, we'll learn how to write the server-side code for authentication. I've already created an app that allows users to submit and view articles. For authenticating users of this application, we'll be using the local strategy of Passport. This means that we'll be storing the user login details in a database. To store the login details in the database, I've also created a user model based on the Mongoose schema. So here's the model that I have created. Let me close this file. Now, to implement authentication using Passport, we need to first install the Passport module. So let us open a Terminal window and use the `npm install passport --save` command to install Passport. Now, as we'll be using the local strategy of Passport, we also need to install the passport-local module. For this, I'll use `npm install passport-local --save` command. We'll also need to use another module named express session for handling sessions. To install this, I'll use `npm install express-session --save` command. Finally, we need to use a module that can help us save our passwords as hashes. This is because it is never a good idea to store passwords in text form in the database. So I'll use the `npm install bcrypt-nodejs --save` command to install bcrypt nodejs.

*[The presenter navigates to the "*users.js (~/articles/models) – gedit" window to view the following model: var mongoose = require ('mongoose'); var UserSchema = new mongoose. Schema ({ username: String, password: String }); var User = mongoose.model ('User', UserSchema); She types the "npm install passport --save" command in the terminal window to install the passport module. She types the "npm install passport-local --save" command to install the passport local module. Then she types the "npm install express-session --save" to install the express session module. Finally she types the "npm install bcrypt-nodejs –save" command to install the bcrypt node js.]*

Now that all the required modules are installed, I'll open app.js. And first of all, I'll add some require statements for the modules that I just installed. Now I'll change the middleware section to use the session module. And then I'll add Passport as an application-level middleware. I'll now Save this file. We'll now create a new file by the name of passport-intialize.js. I'll now open this file and add some boiler plate code for initializing Passport. Now in this code, first of all, we define a Mongoose model by the name of user. Then we require the passport-local and the bCrypt nodejs modules. And, after this, we have the code to initialize Passport.

*[She navigates to the "*passport-initialize.js (~/articles) – gedit" window and adds the code for initializing the passport.]*

Now in this code, first of all, we have two functions to serialize and deserialize users. This is required to support persistent log in sessions. The first function accepts a user object and returns the ID of the object. And the second function accepts an ID and returns the associated user object after looking it up in the database. After this we have implemented the signup functionality. Setting passReqTocallback to true helps us to pass the entire request to the callback. Now, in this callback, we first try to check whether the user already exists. If there is an error in fetching the user details, we invoke the done function with an error. If there is no error, we check whether the user already exists. If the user already exists, we invoke the done function with a null and with the value false. Now, if there is no error and the

user does not already exist, we create a new user object and initialize the username and password properties of the object.

Before storing the password, we hash the password by using the createHash function, which we have defined here. After this, we save the user details to the database and then invoke the done function with a null value and with the user object. After this, we have implemented the signin functionality. Here again, we first try to find out whether the user already exists in the database or not. If there is an error in fetching the details, we invoke the done function with an error. If there is no error and the user does not exist in the database, we invoke the done function with a null value and the value false. This will redirect the user to the sign in page. Now, if there is no error and the user exists in the database, we check whether the password is valid or not. Again, this function – isPasswordValid – is defined here. Now, if the password is not valid, we invoke the done function with a null and the value false. But, if everything is fine, we sign in successfully and invoke the done function with a null value and with the user object. I'll now Save the file and close it.

Now I'm back on app.js. Now here I'll add the code to require the passportinitialize.js file that I just created. Again, I'll Save the file. And now I'll go into the routes folder. Now here, I'll create a new file by the name of authenticate.js. I'll open this file and add the authentication API. Now here we have defined our middleware APIs. Let us first see the handler for the signup route, which specifies two redirect URLs – one for success and the other for failure. This means that if signup is successful, the request will be redirected to /auth/success. Otherwise, it will be redirected to /auth/failure. The handlers for success and failure are also defined here, which send appropriate responses back to the client. Now similarly, we also have a handler for the signin route. Again, it specifies a success and a failure handler. Finally, we have a handler for the /signout route. This will lock the user out of the session and redirect the user to the application route. I'll now Save the file and close it.

*[She returns to the "*app.js (./articles) – gedit" window and types the following code: var initPassport = require ('. /passport-initialize'); initPassport (passport); Then she navigates to the "*authenticate.js (./articles/routes) – gedit" file and adds the authentication api. She returns to the "*app.js (./articles) – gedit" file and adds the following code: var authenticate = require ('. /routes/authenticate') (passport); She also adds the following code for the authenticate.js file: app.use (' /auth', authenticate);]*

Again I'm back on app.js. Now here I'll add a reference to the authenticate.js file that I just created. Next I'll add an app.use statement for the authenticate.js file that I had created. I'll now Save the file and close it. The authentication API is now ready. So let me open a Terminal window, `clear` the screen, and start the application. Before I start the application, I need to move into the application folder. I'll now open Postman to send a few requests to the authentication API.

[She navigates to the terminal window and types "cd articles". Then she types "npm start" to start the application.]

Let us first send a POST request to local host:3000/auth/signup. I also need to add some data here. So I'll specify a username and password and send the request. So the user details have been successfully stored in the database. Let us now send a POST request to localhost:3000/auth/signin. I'll send the same data along. So the user details have been fetched, compared, and a success status has been returned. Now, if I change the name to something else and then send the request, now since I don't have a user by this name in the database, the state failure has been returned. In this topic, we learned how to write the server-side code for implementing authentication.

Writing the Client-side Code for Authentication

Learning Objective

After completing this topic, you should be able to

- *write client-side code for authentication*

1.

In this topic, we'll learn how to write client-side code for authentication. I have already created an app that allows a user to submit and view articles. I have also created views for sign up and sign in, and also created the server-side code for authentication. I have also confirmed that the authentication APIs are working properly by using PostMan. Now we need to write the client-side code for authentication. For this, I'll open my articleApp.js file and replace the code within the $scope.signin = function with some other code. So whenever the user provides credentials and clicks the Sign In button, an HTTP post request is sent to /auth/singin. If the request is processed successfully, then the user is redirected to the application root. Otherwise, a relevant message is displayed. Now similarly, I'll replace the code written within $scope.signup = function with some other code. So whenever the user provides credentials and clicks the Sign Up button, an HTTP post request is sent to /auth/singup. If the request is processed successfully, the user is redirected to the application root. Otherwise, a relevant message is displayed.

[The presenter navigates to the "articleApp.js (~/articles/public/javascripts) – gedit" file. This file includes the following set of codes: $scope.signin = function ( ) { $http.post (' /auth/signin', $scope.user). success (function (response) { if (response.state == 'success') { $location.path ('/'); } else { $scope.msg = response.message; }]

Now here we are using the $http and $location services. So we need to add these services as dependencies for the authController. So I'll add $http and $location here. Let me now Save the file and open a Terminal window. I'll move into the application folder and start the application. I'll now open the browser and go to localhost:3000. Now I'll click the Sign Up button and provide a Username and Password. Now, as I provided a valid username and password, I've been redirected to the home page. Let me now click the Sign In button and provide a valid Username and Password. Again, I've been redirected to the home page. I'll click the Sign In button again and this time I'll provide an invalid Username and Password. This time I get a message saying "Invalid username or password."

[She opens the terminal window and types "cd articles" to move into the applications folder and then starts the application. She navigates to the "localhost: 3000" web address on the web browser. She verifies code by using the Sign Up and the Sign In links.]

Now the code that we wrote is working fine. But we would want that when the user is signed in, the Sign In and Sign Up links should not be visible. And instead, we should get a Sign Out link and a welcome message for the current user. For this, we need some mechanism to get the name of the current user. Since we need to access this name throughout the application, I'll use a $routeScope variable to store this value. This is because a variable declared with $routeScope is accessible throughout the application. For this, I'll go back to my articleApp.js file and I'll add some code here. Now here we have added some initialization code that sets the $rootScope.authenticated variable to a value of false, indicating that initially the user is not authenticated. Also, the value of the $rootScope.current_user variable has been set to 'Guest' indicating that the user is a guest user and is not authenticated. Further, we have defined a function named signout that is bound to $rootScope, indicating that it can be accessed from any part of the application. The signout function uses the $http service to send a get

request to 'auth/signout'. It then resets $rootScope.authenticated to false and $rootScope.current_user to 'Guest'.

[She returns to the "articleApp.js (~/articles/public/javascripts) – gedit" file. This file includes the following code: .run (function ($http, $rootScope) { $rootScope. authenticated = false; $rootScope. current_user = 'Guest'; $rootScope.signout = function ( ) { $http.get ('auth/signout'); $rootScope. authenticated = false; $rootScope. current_user = 'Guest'; };]

Now, once the user is signed in, we need to change the values of these $rootScope variables. So when the user is successfully signed in, $rootScope.authenticated needs to be set to true and username of the authenticated user needs to be stored in the $rootScope.current_user variable. So I'll locate the code where I'm checking whether the user has been successfully logged in or not. And here I'll add these two statements to set the value of the $rootScope.authenticated variable to true and $rootScope.current_user variable to the username of the logged in user. I'll add the same two lines here where I'm checking whether the user has successfully signed up or not. Now, as we are using $rootScope here, we need to declare it as a dependency for the authController. Let us now Save this file and close it.

Now I need to conditionally display the Sign Up, Sign In, and Sign Out links based on the authentication status of the user. For this, I'll open my index.html file and I'll replace the mark up for the Sign Up and Sign In links with some other mark up. Now here we've added two span elements. One of these has an ng-hide attribute whose value is set to authenticated. This means that this span element will be hidden if the value of the $rootScope.authenticated variable is true. The other span element has an ng-show attribute with the value authenticated. This means that this span element will be shown if the value of the $rootScope.authenticated variable is true.

[She navigates to the "index.html (~/articles/public) – gedit" file and replaces the markup for the Sign Up and Sign In links.]

Let me now Save this file and close it. I'll now go to the Terminal window, terminate the running application, and start it again. I'll now open the browser and go to localhost:3000. I'll click the Sign In link and provide a valid Username and Password. You can see that the Sign Up and Sign In links are not visible now. And instead, we have a Sign Out link and a welcome message for the currently logged in user. If I click the Sign Out link, the Sign Up and Sign In links are again visible. In this topic, we learned how to write the front-end code for authentication.

[She returns to the terminal window and restarts the application. She navigates to the "localhost: 3000" web address on the web browser. She verifies the codes for Sign Out and Welcome messages.]

Implementing Authorization

Learning Objective

After completing this topic, you should be able to

- *implement authorization*

1.

In this topic, we'll learn how to implement authorization in a MEAN application. I've already developed a MEAN application that allows users to submit and view articles. I've also implemented authentication in the application. Now I want to implement a couple of things. First of all, I want that only signed in users should be able to post articles though all users may view the articles posted by others. Also as the user needs to log in to post articles, I want that the user should not be required to enter his or her username while posting an article. Finally, I want that the users should be able to edit and delete their own articles, but they should not be able to edit or delete the articles posted by other users. So let us first make some changes so that only signed in users are able to post articles. For this, I'll open my api.js file and add a middleware function before all the middleware functions that are already defined in this file. Now this function intercepts all requests to /api/articles endpoint. If the request is a GET request, it calls the next middleware function in sequence. Otherwise, it checks whether the request is authenticated or not. If the request is authenticated, it calls the next middleware function in sequence. Otherwise, it sends a response to the client with a status specifying authentication failure. So, in case the request is a POST request and the user is not authenticated, the request won't get pass this middleware function and the middleware function for the POST request will never get executed. Let me now Save this file and open the articleApp.js file.

*[The presenter navigates to the "*api.js (~/articles/routes) – gedit" window and explains the following code: router.use (' /articles', function (req, res, next) { if (req.method == 'GET') return next ( ); else if (req. is Authenticated ( )) return next ( ) ; else res.send ({status: 'Authentication failure'}); });]*

Now here I'll look for the code that sends a POST request to the article API. So here it is. Now here we are using a service to send a POST request to save an article. The API will send a response, which I'll accept in a variable, say, response. And now I'll add some code here. Now this code will check the status of the response. And, if there is an Authentication Failure, the user will be redirected to the Sign In page. But for this, I'll need to specify $location as a dependency here. Let me now Save this file. Now that we have taken care that only a signed in user can post articles, let us now remove the textbox that we had added to accept the username. For this, I'll open the main.html file and I'll remove this textbox from here. Now again, I'll Save this file and go back to my articleApp.js file. Now, before invoking the articleService to store the article in the database, I'll set the value of newArticle.username to $rootScope.current_user. And then I'll have to declare $rootScope as a dependency here. I'll now Save this file. Now as I want to allow the users to edit and delete articles, let us open the api.js file and add some middleware functions to handle the edit and delete requests. Now here we have defined a middleware function that handles all delete requests sent to the /api/articles/:id endpoint. The function invokes the Article.remove method to delete the article with a specified id from the database.

*[She then navigates to the "*articleApp.js (~/articles/public/javascripts) – gedit" window. The window includes the following set of codes: articleService.save ($scope.newArticle, function (response) { if (response.status == 'Authentication Failure') $location.path (' /signin'); She opens the "*main.html (~/articles/public) – gedit" window and removes the following textbox: <input required type="text" placeholder="Your name" ng-model="newArticle.username" />

 She returns to the "*api.js (~/articles/routes) – gedit" window and explains the following code: router.delete(' /articles/:id'*

function(req, res, next) { Article. remove ({_id: req.params.id}, function(err, article) { if (err) { return res.send (err) ; } res.json ({message: 'Successfully deleted'}); }); The put function in the file includes the following code: router.put(' /articles/:id' function(req, res, next) { Article.findOne ({_id: req.body._id}, function (err, article) { if (err) { return res.send (err); }]

Next we have defined a middleware function that handles all PUT requests sent to /api/articles. It first finds a record in the database whose id matches the id of the article object sent along with the request. It then updates the properties of the fetched object as per the updated details sent by the client. And finally, it saves the updated article object in the database. Let me now Save this file and go back to my articleApp.js file. Now as I'll now be using my article service to send edit and delete requests to the back end, I also need to make some changes to the service. So, first of all, I'll specify an articleId here. And then I'll say that the articleId comes from the id field and the service will use an update method to send an HTTP PUT request to the server. Again I'll Save this file. And Now I'll make some changes to the front end. So let me open the main.html file. Now here I'll add some code to display the Edit and Delete buttons against each of the submitted articles. Now these buttons have ng-click attributes that are bound to appropriate JavaScript functions that get invoked when these buttons are clicked. Also, as I have added additional submit buttons, I'll remove the ng-Submit attribute from here. And instead I'll add an ng-click attribute over here and I'll bind it to the post() JavaScript function. I'll also move the closing form tag from here and place it here.

*[She navigates to the "*articleApp.js (~/articles/public/javascripts) – gedit" window and discusses the edit, update, and delete functions. To ensure that the user is able to edit or delete only his/her functions, she executes the following command: $scope.isUserOwner = function (article) { return (article.username == $rootScope.current_user); }]*

Now, when I click the Edit button, the corresponding article should be displayed in the form over here. At that time, the Post button on the form should disappear and instead another button with the caption Update should be displayed. So let me add another button over here and I'll bind it to the update() JavaScript method. Now I want that only one of these buttons should be visible at a time. So let me use an ng-show attribute over here and bind it to a Boolean variable named editMode. I'll copy the same thing over here but I'll change ng-show to ng-hide. Now this means that the Update button will be shown when the value of the editMode variable is true. And the Post button will be shown when the value of the editMode variable is false. Let me now Save this file and open the articleApp.js file. Now here I'll first define a variable named $scope.editMode and set its value to false. I'll now define the functions to handle the clicking of Edit, Update, and Delete buttons that we just added to our form. The edit function sets the value of editMode to true and then uses the article service to fetch the details of the selected article from the database. It then populates the details in the form that appears on the screen.

The update function updates the timestamp of the edited article and uses the article service to update the articles in the database. If the back-end API returns a status of Authentication Failure, it redirects the user to the Sign In page. Otherwise, it fetches the updated details from the database and stores them in the articles array. Finally, it sets the editMode variable to false. The delete function shows a confirm dialog box to confirm whether the user actually wants to delete the article or not. If the user confirms deletion, it uses the article service to delete the selected article from the database. If the back-end API returns the status as Authentication Failure, the user is redirected to the Sign In page. Otherwise, the updated record is retrieved from the database and stored in the articles array. Now there is one last requirement that I need to implement. A user should be able to edit or delete only his or her own articles. For this, I'll add a function named isUserOwner that is bound to $scope. This function will return a Boolean value indicating whether the currently logged in user is the owner of the selected article or not. I'll now Save this file and open the main.html file.

Now I want to display the Edit and Delete buttons against an article only if the current user is the owner of the article. For this, I'll add ng-show attributes and assign the return value of the isUserOwner function to this attribute. I'll copy the same thing to the Delete button also. Again I'll Save this file. I'll now go to the Terminal window and start the application. Let me open the browser and reload this application. You

can see that now I don't need to provide my username here. Now, if I try to post an article, it redirects me to the Sign In page, because I'm not signed in yet. So let me Sign In and then try to submit an article again. This time I have been able to submit an article. You can also see that the Edit and Delete buttons are displayed only against the articles that have been submitted by the currently logged in user. Let me now see if I am able to Edit an article or not. So the article has appeared here I can make changes and store the updates. You can see that the article has been updated. Let me see if I can Delete an article or not. So it asks me "Are you sure you want to delete this article" – I say OK. And the article has been deleted. In this topic, we learned about implementing authorization in a MEAN application.

[The presenter navigates to the terminal to start the application. She then navigates to the "localhost:3000/#/" web address in the Google chrome web browser to reload the application. She types "Hi" in the article title field and "Hello everyone!!!" in the article text field after signing in. The article is successfully displayed. She then presses the Edit button and changes the article text to "Hello everyone!!! Welcome to MEAN.js." The article gets edited successfully. She then presses the delete button. The article gets deleted successfully.]

Fundamentals of Testing a MEAN Application

Learning Objective

After completing this topic, you should be able to

- *identify the fundamentals of testing a MEAN application*

1.

In this topic, we'll learn about the fundamentals of testing a MEAN application. JavaScript has evolved significantly over the last few years. It is now the pillar of complex server-side and client-side architectures. Due to lack of JavaScript testing tools, developers were faced with the challenge of managing a large code base manually, which is a very difficult task. However, lately some new tools and frameworks for JavaScript testing have emerged. Some of these include Mocha and Jasmine. These frameworks allow developers to write tests in a structured manner. In the traditional application development process, testing was performed towards the end of the application development life cycle. Test-driven development or TDD is an application development approach that works on the test-first concept. This means that tests that define the expected requirement of an isolated junk of code are created first and then the code is written to make the tests pass. In TDD, a cycle of activities is repeated to implement each requirement. First of all, a test is written after analyzing the requirements. Till this point, no code has been written to implement the functionality being tested. Therefore, the test fails. After this, minimal code is written to make the test pass. This code may be a quick and dirty implication of the requirement, containing hard coded values and nondescriptive variable names. After this, the code is cleaned up and improved. And, after making these changes, the test is run again to ensure that it still passes.

Another development approach that emerged from TDD is behavior-driven development. In BDD, the focus is on testing the behavior rather than the implementation. A BDD test framework provides developers with a set of self-explanatory methods to describe the test process. Test frameworks help developers to create and organize tests. However, they often lack the ability to test a Boolean expression that represents the test results. For this, the community has developed several assertion libraries. Assertion libraries allow developers to use assertion expressions to examine a certain predicate. When the test is run, the assertion is evaluated. If the assertion evaluates to false, the test fails. Besides frameworks, there are utilities called test runners that enable the developer to easily run and evaluate tests. A test runner uses a testing framework with a set of preconfigured properties to evaluate test result in different contexts. For example, a test runner may be configured to run the same test on different browsers or with different environment variables. In this topic, we learned about the fundamentals of testing a MEAN application. We also learned about test-driven development, behavior-driven development, test frameworks, assertion libraries, and test runners.

Introduction to Client-side and Server-side Testing

Learning Objective

After completing this topic, you should be able to

- *identify the fundamentals of client-side and server-side testing*

1.

In this topic, we'll learn about testing different parts of a MEAN application. All the code in a MEAN application is written in JavaScript. However, the code is distributed across different platforms. Some part of the code runs on the client-side while the remaining runs on the server-side. Client-side testing involves testing the front-end code while service-side testing involves testing the back-end code. In a MEAN application, client-side testing involves testing the AngularJS code while server-side testing involves testing the Express part of the application. In this topic, we learned about client-side and server-side testing.

Introduction to Testing Tools

Learning Objective

After completing this topic, you should be able to

- *identify the tools used for testing MEAN applications*

1.

In this topic, we'll learn about some tools used for testing MEAN applications. To test the server-side code of a MEAN application, we'll use the Mocha test framework. Mocha is a versatile test framework that supports both test-driven and behavior-driven development. It is also very easy to set up. Mocha is minimal by structure and does not include a built-in assertion library. However, popular assertion libraries such as should.js can be integrated with Mocha. Mocha also supports testing of asynchronous code. The Mocha API includes various methods to help you write test cases. Related test cases are organized under a test suite. You can see a test suite on your screen. This test suite contains a single test case but a real test suite will contain multiple related test cases.

Let us now see some of the methods that the Mocha API provides us to help us with writing the test cases. The describe method is used to wrap each test suite with a description. The it method is used to provide a description for each test case. It also specifies a callback function that includes the test logic. Then there are four hook functions named – before, beforeEach, after, and afterEach. The before function is executed before all the tests in a test suite. The beforeEach function is executed before each test specification in a test suite. The after function is executed after all the tests in a test suite. And the afterEach function is executed after each test specification in a test suite. To test the AngularJS part of a MEAN application, we'll use the Jasmine test framework and the Karma test runner. Jasmine is easy to set up and has a BDD framework whose API is quite similar to that of Mocha. Unlike Mocha, Jasmine comes prebundled with assertion capabilities. However, a test runner such as Karma needs to be used along with Jasmine. In this topic, we learned about some testing tools that can be used to test client-side and server-side code of a MEAN application.

Installing the Tools for Server-side Testing

Learning Objective

After completing this topic, you should be able to

- *install the tools required for server-side testing*

1.

In this topic, we'll learn how to install the tools for server-side testing. For server-side testing, we'll be using Mocha along with chai.js and request.js modules. Let us first see how to install Mocha. To install Mocha, we'll use the `sudo npm install -g mocha` command. Once Mocha is installed, I'll install chai by using the `sudo npm install chai` command. Now, in order to test the asynchronous part of my application, I'll need a module that can help me send HTTP requests to the server. For this, I'll install the request module by using the `sudo npm install request` command. Now all the tools that I need for server-side testing of my MEAN application have been installed. In this topic, we learned how to install the tools for server-side testing of a MEAN application.

[The terminal window is open. The presenter types the commands to install mocha, chai, and request module in this window. These three applications are installed and the output is displayed.]

Configuring the Test Environment

Learning Objective

After completing this topic, you should be able to

- *configure the test environment*

1.

In this topic, we'll learn how to configure the test environment before testing an application. The environment in which we test an application is usually different from the production environment of the application. For example, the database server that you connect to during testing may be different from the database server that you used in the production environment. So it is a good idea to configure the test environment such that the app always connects with the URL of the appropriate server. To do this, at the time of running our application, we'll make use of an environment variable NODE_ENV. And we'll set its value to either production or development depending upon whether we want to run that application in the production environment or the testing environment. And after this, we'll use the npm start command to start our application. But before we start our application we need to make some changes to our application. So I'll remove this and go into my application folder where I'll create a new file by the name of config.js. Now let me open this file and I'll add some code to this file. Now, in this code, I am first requiring the config.js file that I just created. And then I am creating a new instance of the config object. After this, I am displaying the value of the conf.DB_URI property returned by the config module that I just created. Now this statement will display the address of the appropriate server depending on the value of the NODE_ENV variable.

*[The presenter navigates to the "*app.js (~/myproject) – gedit" window and adds the following code: var config = require (". /config.js"); car conf = new config ( ); console.log (conf.DB_URI);]*

So let me Save this file and go to the Terminal window. Now here I'll write `NODE_ENV=development npm start`. Now you can see that it returns the message – URL of development server. Now let me terminate the application and run it again. But this time, I'll set the value of the NODE_ENV variable to production. You can see that this time it has returned the message – URL of production server. In this topic, we learned how to configure the test environment before testing an application.

[She returns to the terminal window and runs the application.]

Writing and Running a Mocha Test

Learning Objective

After completing this topic, you should be able to

- *write and run a mocha test*

1.

In this topic, we'll learn how to write and run a Mocha test. For this demo, we'll be following the test-driven development approach. So I'll start with a blank project and I'll be developing a simple calculator application by using the TDD approach. So let me create that new folder by the name of calculator. Let me cd into the calculator folder. Now I'll create the boiler plate code for an express application by using the `express --ejs .` command. Let me `clear` the screen. I'll now install all the dependencies by using the npm install command. All the dependencies are now installed. Let me `clear` the screen. I'll now install all the tools required for server-side testing. So, first of all, let me install mocha and now I'll install chai. So the tools are now installed.

[The terminal window is open. The presenter types "mkdir calculator" to create a new folder by the name of calculator. She then types "cd calculator". After installing the dependencies, she types "sudo npm install –g mocha" to install mocha. The output is displayed. After this, she types "sudo npm install chai" to install chai.]

Let me now go into my folder structure and create a New Folder by the name of test. This is the folder in which I'll store all my test suites. So let me open this folder and here I'll create a new file by the name of calculatorTest.js. Now let me open this file and add some code to it. Now, in this code, first of all, we require the expect assertion style of chai. We then require the calculator.js file that contains the actual code that needs to be tested. As we are following the test-driven development approach, right now, we have not written the code that we are going to test. We'll first create the test cases. And then write the code to make the tests pass. Now, in this code here, we have used the describe function to describe a test suite. The test suite has two other nested test suites – one for checking the Addition functionality and the other for checking the Subtraction functionality.

*[She navigates to the "*calculatorTest.js (~/calculator/test) – gedit" window and adds the following code: var expect = require ("chai") .expect; var calculator = require ("../calculator"); describe ("Calculator", function ( ) { describe ("Addition", function ( ) { it ("produces correct result", function ( ) { var sum = calculator.add(6,6); expect (sum).to.equal (12); }); }); describe ("Subtraction", function ( ) { it ("produces correct result", function ( ) { var diff = calculator.subtract(15,6); expect (diff).to.equal (9); }); }); });]*

Now, within the Addition test suite, we have defined a test case by using the it method. This test case checks whether the addition of two numbers produces the correct result or not. For this, we first invoke the add method with two values and then use the expect method to assert whether the result returned by the add function is equal to the expected result or not. If the assertion returns true, the test succeeds otherwise the test fails. Similarly, within the Subtraction test suite, we have a test case that checks whether the subtraction of two numbers produces the correct result or not. If the assertion returns true, the test succeeds otherwise the test fails. Let us now Save this file and go to the Terminal window. I'll now test the application. Now to test the application, I need to use the mocha command. Now, with the mocha command, we also need to specify the path of the folder that contains a test suite. And we also need to specify the reporter that we want to use with the test.

Here I am using the spec reporter that displays the test results in a hierarchical view. There are various other reporters available that report test results in different ways. You can learn more about reporters by going to https://mochajs.org/#reporters. Now coming back to the command that I need to use to run the test suite. Now instead of typing this entire command to execute my tests, I would want to simply use the command npm test. Now to do this, I need to open my package.json file and I need to add an entry in this file which says, "test": "mocha ./test/ --reporter spec". Let me now Save this file and close it.

*[She navigates to the "*package.json (~/calculator) – gedit" window and adds the following entry: "test": "mocha ./test/ --reporter spec".]*

Let me go back to the Terminal window. So now I'll be able to run my test by using the `npm test` command. Now you can see that my tests have resulted in failure. This was expected, because I haven't yet written the code that I need to test. I've simply defined the behavior that my code needs to show. Let me now go back into my folder structure and create a new file by the name calculator.js. Now this is the file in which I'll add the code that needs to be tested. Now here is the code that needs to be tested. Here I've simply defined two functions that accept two numbers and return their sum and difference respectively. Let me Save this file and close it. I'll now go to the Terminal window again and use the `npm test` command to run my tests. You can see that this time my tests have passed. In this topic, we learned how to write and run a Mocha test.

*[Thereafter, she returns to the terminal window to test the application. The output displays a failed test. To rectify the error, the presenter navigates to the "*calculator.js (~/calculator) – gedit" window and adds the following code: module.exports.add = function (x, y) { return (x+y); }; module.exports.subtract = function (x, y) { return (x-y); }; She navigates to the terminal window to run the test again. The output displays that the test has passed.]*

Testing Asynchronous Code

Learning Objective

After completing this topic, you should be able to

- *test asynchronous code*

1.

In this topic, we'll learn how to test an asynchronous application by using Mocha. I've already created a calculator application in which I first created a test suite to check whether the addition and subtraction functionality of the calculator is working as expected or not. I then wrote the code for implementing the addition and subtraction functionality. And finally, executed the tests to ensure that all my test cases passed the test. Now I want to make this functionality available over a server. So my test cases will need to send an HTTP request to the server in order to check whether the addition and subtraction functionality works as expected. To enable my test cases to send the HTTP requests to the server, I will install the request module. For this, I'll use the `sudo npm install request` command. Now that the request module has been installed. Let me create a test suite to check whether the HTTP requests sent to the server for addition and subtraction produce the correct result. For this, I'll go into the test folder and create a file by the name of calculatorHttpTest.js. You can see that I already have a test suite here. So, when I execute my test, both these test suites will get executed.

[The presenter installs the request module in the terminal window.]

I'll now open calculatorHttpTest.js file and add some code to it. In this code, first of all, we require the expect assertion style of chai. Then we require the request module. And after this, we have a test suite for the Calculator API. Now within this test suite, we have two nested test suites – one to check the Addition functionality and the other to check the Subtraction functionality of the calculator. Now in the first test suite, we start with a before function which is a hook function that gets executed before all the test cases in a test suite. In this function, we have initialized the URL to which we need to send an HTTP request for performing Addition. After this, we have a test case that sends a request for addition to the server and checks whether the request returns a statusCode of 200 or not. If it does, the test succeeds otherwise the test fails. Next we have a test case that sends a request for Addition to the server and checks whether the request returns the correct sum. Again, we have defined similar test cases for Subtraction. Let us now Save this file and close it. I'll now go to the Terminal window, `clear` my screen, and start my application. I'll now open another Terminal window and use the `npm test` command to test my application.

*[Then she navigates to the "*calculatorHttpTest.js (~/calculator/test) – gedit" window. The window includes the following set of codes: Describe ("Addition", function ( ) { before (function ( ) { url = "http: //localhost:3000/add?x=10&y=5"; }); She returns to the terminal window and types "npm test" command to test the application. The output displays a failed test.]*

Now, at this stage, my tests have resulted in failure. This was expected because I haven't yet written the code that I need to test. I've simply defined the behavior that my code needs to show. So let me open my app.js file. Now, in this file, I first need to add a require statement for the calculator.js file that contains the code for addition and subtraction. Also I can remove these two require statements as I am not using them yet. I'll now add some middleware functions to handle the requests for addition and subtraction. Again I can remove these app.use statements from here as I don't need them as of now. Now the first function here responds to all requests to the /add endpoint, it retrieves the numbers passed along as query parameters and returns their sum. The second function responds to all requests to the

/subtract endpoint, retrieves the numbers passed along as query parameters and returns their difference. Let us now Save this file and close it. I'll now go back to the Terminal window, terminate the running application, and start it again. I'll now switch to the other Terminal window. Let me `clear` the screen and then use the `npm test` command to run the tests. You can see that this time my tests have succeeded. In this topic, we learned how to write test cases for asynchronous code.

*[To rectify the error, she navigates to the "*app.js (./calculator) – gedit" window and adds the following code: var calculator = require (". /calculator"); She adds some more functions in the window. The functions include the following code: app.get ("/add", function (req, res) { var x = parseInt (req.query.x); var y = parseInt (req.query.y); var sum = calculator.add (x, y); res.send ({"sum": sum}); }); Then she navigates to the terminal window and restarts the application. The output displays that the test has passed.]*

Installing the Tools for Client-side Testing

Learning Objective

After completing this topic, you should be able to

- *install the tools required for client-side testing*

1.

In this topic, we'll learn how to install the tools for client-side testing. First of all, let me create a folder that will contain the application that needs to be tested. Let me now cd into this folder. Now, for client-side testing, we'll be using Karma and Jasmine. First of all, let's install the Karma command line interface by using the `sudo npm install -g karma-cli` command. Let us now install Karma by using the `sudo npm install karma` command. Now Karma is installed. Let me `clear` the screen. Now we'll install the Karma plugin adapter for the Jasmine framework. For this, we'll use the `npm install karma-jasmine` command. Now you also need to install angular.js and angular-mocks.js libraries. To install angular.js, you can use the `npm install angular` command. Similarly, to install angular-mocks.js, you can use the `npm install angular-mocks` command. Now angular-mocks allows you to inject and mock angular components to help you test your application. Now all the tools that you need for the client-side testing of a MEAN application have been installed. In this topic, we learned how to install the tools for client-side testing of a MEAN application.

[The terminal window is open. The presenter types "mkdir myapp" to create a new folder by the name of calculator. She then types "cd myapp". To install the karma command line interface, she types "sudo npm install –g karma-cli". She further types "sudo npm install karma" to install karma. The output is displayed and karma is installed. To install the karma plug-in adapter for the jasmine framework, she types "npm install karma-jasmine" command. She types "npm install angular" and "npm install angular-mocks" commands to install the angular.js and angular-mocks.js libraries.]

Configuring the Karma Test Runner

Learning Objective

After completing this topic, you should be able to

- *configure the Karma test runner*

1.

In this topic, we'll learn how to configure the Karma test runner. I have already created a blank application folder and installed the tools for client-side testing. I now need to configure the Karma test runner. To control Karma's test execution, we need to configure Karma using a special configuration file placed in the root folder of our application. When executed, Karma will automatically look for the default configuration file named karma.conf.js in the application's root folder. You can either create this file manually or use the `karma init` command to create it automatically by answering a set of questions. Now it first asks me which testing framework I want to use. I'll accept the default value that is jasmine. It then asks me whether I want to use Require.js. I don't need it right now. So I'll accept the default value that is no. Next it asks me against which browser I want to test my application. I'll accept the default value right now that is Chrome. I can also specify additional browsers by hitting the Tab key to select an appropriate browser and then pressing the Enter key to select it. Right now I'll just hit Enter to specify that I don't need any additional browser. It then asks me to provide the location of my source and test files.

[The presenter types "karma init" in the terminal window to configure the karma test runner. She explains the output.]

Here first of all, I need to provide the path for the angular.js file. So I'll provide the path as node_modules/angular/angular.js. Similarly, I need to provide the path for angular-mocks.js file. So I'll provide the path as node_modules/angular-mocks/angular-mocks.js. Now all my source files will be there in the app folder. So I specify the path app/*.js. Right now it complains that there is no file matching this pattern. This is because we have not created any source file till now. Now all my test files will be there in the test folder. So I'll specify the path for the test file as test/*.js. Again it complains that there is no file matching this pattern. This is because we have not created any test files till now. I'll now hit Enter to specify that I don't want to include any other files. It then asks me whether any of the files included in the previous patterns should be excluded. I don't want to specify any files here. So I'll just hit Enter. Finally, it asks me if I want to watch all the files and run the tests on change. I'll select the default option that is yes here. Now the karma.conf.js file has been created. The file contains the configuration settings for the Karma test runner. You can open the karma.conf.js file and see that all the options we selected have been included in this file. In this topic, we learned how to configure the Karma test runner.

Testing AngularJS Modules

Learning Objective

After completing this topic, you should be able to

- *test AngularJS modules*

1.

In this topic, we'll learn how to test an AngularJS module. Now I am working on one of the pages of an online shopping app. And I'll be following the test-driven development approach to develop this application. This means that before writing any code to implement the functionality, I'll define the behavior of the code using test cases and then develop the code to make the tests pass. I've already installed the tools that I need for client-side testing. I've also configured the Karma test runner and created a file named karma.conf.js that contains all the configuration settings. Let us now write a test for an AngularJS module. Testing an AngularJS module is very simple. It simply involves testing that the module is properly defined and exists in the test context. So let me create a folder named test. Now this is the folder in which I'll include all my test suites. Let me open this folder and create a file named test.js in this. Now I'll open this file and add the code to create a test suite. Now, in this code, first of all we use the describe method to define a test suite. In this test suite, we're going to include test cases for the Main module. Now, within this test suite, first of all we've declared a variable named mainModule. Then we have defined a beforeEach function. And, within this function, we have created a mock instance of the mainModule. After this, we've used the it method to define a test case. This test case is designed to check whether the module in question is registered and exists in the test context.

*[The presenter navigates to the "*test.js (~/shoppingApp/test) – gedit" window and adds the following code: describe ('Main module', function ( ) { var mainModule beforeEach (function ( ) { }); it ('Should be registered', function ( ) { expect (mainModule). toBeDefined ( ); }); });]*

In this test case, we have used an assertion to check whether the mainModule is actually defined or not. If the assertion returns true, the test passes. Otherwise, it fails. So let us Save this file and go to the Terminal window where we'll execute the test. For this, we'll use the `karma start` command. You can see that right now the test fails. This is because we've not yet written the code that needs to be tested. So let us write the code now. So let me first go back to the root folder of my application and create a new folder by the name app. Now, in this folder, I'll create a new file named shoppingApp.js. And let me open this file. And now I'll add some code to this file. Now this code simply defines a module by the name of mainModule. Let me Save this file and go back to the Terminal window. You can see that the test has automatically executed again. And, this time, the test has passed. In this topic, we learned how to test AngularJs modules.

[She types "karma start" in the terminal window to execute the test. The test fails. She navigates to the "shoppingApp.js (~/shoppingApp/app) – gedit" window and specifies the following code that needs to be tested: var mainModule = angular.module ('mainModule', ['ngMock']); She returns to the terminal window to execute the test again. The test runs successfully.]

Testing AngularJS Controllers

Learning Objective

After completing this topic, you should be able to

- *test AngularJS controllers*

1.

In this topic, we'll learn how to test an AsngularJS controller. In this demo, I'll be working on one of the pages of an online shopping app. This page allows users to register with the app by providing their e-mail ID and password. I'll be following the test-driven development approach to develop this application. This means that before writing any code to implement the functionality, I'll define the behavior of the code using test cases and then develop the code to make the test cases pass. I've already installed the tools that I need for client-side testing. I've also configured the Karma test runner and created a file named karma.conf.js that contains all the configuration settings. I've also created a simple test case that checks whether the module is registered with the app or not. And I've also created the associated module and ensured that the test case passes.

Let us now see how to write a test for an AngularJS controller. I'll write the test in my test.js file. Now the controller that I am going to develop needs to grade the password provided by the user as strong, weak, or medium depending upon its length. So, if the password is more than nine characters long, it will be graded as strong. If it is more than four characters long, it will be graded as medium. And otherwise, it will be graded as weak. For this I'll create a function named passwordStrength in my controller. But, before I create the function, let me create the test suite for the controller. Now here I have a test suite in which I have a nested test suite to define the test cases for checking the password strength functionality. Before each of these test cases, the beforeEach function has been used to create a mock module and a mock controller. Here the inject function has been used to access the $controller service that is responsible for instantiating controllers. The underscores around the service name are ignored by the injector when resolving the reference name. Now, in this test suite, we have three test cases. The first one checks whether or not the value of $scope.strength is set to strong if the password is longer than nine characters.

Here we first define a blank $scope object, then create an instance of the UserRegistrationController. Then we assign a string longer than nine characters to the $scope.password variable. And then we invoke the $scope.passwordStrength function. After this, we have an assertion that checks whether the value of $scope.strength has been set to strong or not. If the assertion returns true, the test passes. Otherwise, the test fails. Similarly, we have a test case to check whether the value of $scope.strength is set to weak if the password is less than four characters. Also, we have another test case to check whether the value of $scope.strength is set to medium if the password is longer than four characters and less than ten characters. You can see that there is a lot of duplication in the three test cases as we are initializing a $scope object and the controller object in all the three test cases. So let us create a beforeEach function and copy the duplicate code within this function. I'll have to declare these variables outside the beforeEach function. I'll now remove these two statements from all the test cases.

Let us now Save this file and go to the Terminal window and use the `karma start` command to execute our test cases. You can see that all the three test cases that I just created have failed. This was expected because we haven't yet written the code that we need to test. So let us open our shoppingApp.js file and add the code for the controller that needs to be tested. Now this code sets the value of the $scope.strength variable according to the length of the password. Let us now Save this file

and go back to the Terminal window. You can see that the test has automatically executed again. And, this time, all my tests have passed. In this topic, we learned how to test AngularJS controllers.

Testing AngularJS Services

Learning Objective

After completing this topic, you should be able to

- *test AngularJS services*

1.

In this topic, we'll learn how to test an AngularJS service. In this demo, I'll be working on one of the pages of an online shopping app. Now this page simply lists all the available product categories. I'll be following the test-driven development approach to develop this application. This means that before writing any code to implement the functionality, I'll define the behavior of the code using test cases and then develop the code to make the test cases pass. I've already installed the tools that I need for client-side testing. I've also configured the Karma test runner and created a file named karma.conf.js that contains all the configuration settings. I've also created a simple test case that checks whether the module is registered with the app or not. And I've also created the associated module that needs to be tested. Let us now see how to write a test for an AngularJS service. I'll write this test in my test.js file. Now the service that I am going to develop needs to fetch the details of all the product categories from the endpoint /categories/categories.json. To access this endpoint, the service uses the $resource service. Now, as my application uses the $resource service, first of all I need to install angular-resource.js module. For this, I'll go to the Terminal window and use the `npm install angular-resource` command.

[The presenter types "npm install angular-resource" in the terminal window to download the angular resource.js module.]

Now I need to include angular-resource.js file in my karma.conf.js file. So I'll add an entry for this file here. Let me Save this file. And now I'll go back to my test.js file. And here I'll write the code to create the test for my service. Now here I have defined a test suite in which I have a nested test suite that contains a test case for testing whether a GET request for a list of categories returns an appropriate list of categories or not. Now, before the test case, we have a set of beforeEach functions which will be executed to first create a mock module and then a mock controller. Here the inject function has been used to access the $controller service that is responsible for instantiating controllers. Now my service will use the $resource service to interact with the back end. However, during unit testing, we want our unit test to run quickly and have no external dependencies. So we don't want to send request for data to a real server. Instead, we will use a mock implementation of the service to respond with a prebaked response and assert that the result is as per our expectation. To implement the mock service, we've used the $httpBackend service. And we have created an instance of this service here.

[Then she navigates to the "karma.conf.js (~/shoppingApp) – gedit" window and types the following code: 'node_modules/angular-mocks/angular-mocks.js.' After this she opens the "test.js (~/shoppingApp/test) – gedit" window and explains the code. Thereafter she returns to the terminal window and types "karma start" to run the test. The test fails. She navigates to the "shoppingApp.js (~/shoppingApp/app) – gedit" window and specifies the following code that needs to be tested: var mainModule = angular.module ('mainModule', ['ngMock', 'ngResource']); mainModule.factory ('CategoryService', function ($resource) { return $resource ('/categories/categories.json') }); mainModule.controller ('CategoryController', function ($scope, CategoryService) { $scope.categories = CategoryService.query ( ); });]

Next we have used the $httpBackend.when method to specify the data that the service should respond with when it receives a GET request on this endpoint. Next we have created an instance of the CategoryController within another beforeEach function. And finally, we have a test case that uses the $httpBackend.flush method to execute the pending http requests. When this is done, the mock service responds by sending a prebaked response which will be stored in the $scope.categories array. Now we have a few assert statements here. The first statement checks whether the $scope.categories array contains exactly two categories because that is the number of categories included in the prebaked response. The second and the third assert statements check whether the first and the second elements of the $scope.categories array contain the respective categories as sent by the mock service. Let us now Save this file and go back to the Terminal window. And here I'll use the `karma start` command to run the tests. You can see that the test has failed. This was expected because we have not yet written the code that needs to be tested.

[She returns to the terminal window to execute the test again. The test runs successfully.]

Let us now write the code in the shoppingApp.js file. Now let me paste the code here. Now here we have first defined a service named CategoryService that is associated with the /categories/categories.json endpoint. And then we have defined a controller named CategoryController that queries the service to store the results in the $scope.categories array. Now, before I save this file, I need to do one more thing. I need to add the ngResource dependency here. I can now Save my file and go back to the Terminal window. You can see that the test has automatically executed again. And, this time, all my test cases have passed. In this topic, we learned how to test an AngularJS service.

Introduction to Deployment

Learning Objective

After completing this topic, you should be able to

- *identify the fundamentals of deployment*

1.

In this topic, we'll learn about deployment. Once an application has been developed, it needs to be deployed on a centralized server from where the various users of the application can access it. You can host a MEAN application on a variety of platforms such as Windows Azure, Amazon, Google App Engine, and Heroku. We'll learn how to deploy an app to Heroku. To deploy a MEAN application to Heroku, you need to first create an account on Heroku, then install the Heroku toolbelt, then prepare the app for deployment. Next provision a database for the app. And finally, push the app to Heroku. In this topic, we learned about the process of deploying a MEAN app on Heroku.

Installing the Heroku Toolbelt

Learning Objective

After completing this topic, you should be able to

- *install the Heroku toolbelt*

1.

In this topic, we'll learn how to install the Heroku toolbelt. Heroku toolbelt is a command line tool for creating and managing Heroku apps. In order to use the Heroku toolbelt, you need to first sign up for an account with Heroku. You can sign up for a free account by going to https://www.heroku.com. Once you create an account and log on to Heroku, this is the screen that will appear. You can click the Get Started link under Node.js to go to a tutorial on how to deploy a Node.js project on Heroku. Now, to deploy a MEAN app on Heroku, first of all you need to install the Heroku toolbelt. To install the Heroku toolbelt, you can go to https://toolbelt.heroku.com. Now, from this web page, you can download the appropriate build as per the operating system that you are using.

[The https://dashboard.heroku.com/apps web address is open in the web browser. The home page is divided into two vertical sections. The section on the left contains the tabs for dashboard and personal apps. The section on the right has a search field on the top. Below the search field is the header "Getting started with Heroku." Below the header is the "Get started" button for various softwares including Ruby, Node.js, Python, and Java.]

To download the Heroku toolbelt on my Ubuntu system, I'll copy this command and paste it on my Terminal. So now the toolbelt has been downloaded. And now we can access the heroku command from the shell. But, before that, let me clear the screen. I'll now use the heroku command to log on to my Heroku account. Now, as this is the first time I've used the heroku command, it will first install the toolbelt and then ask me for my credentials. It will take some time to install. So now I'll provide my e-mail and password. So now I'm logged in and ready to start working with Heroku. In this topic, we learned how to install the Heroku toolbelt and how to log on to Heroku from the command shell.

[The presenter navigates to the "https://toolbelt.heroku.com" web address in the web browser to install the heroku toolbelt. The heroku toolbelt homepage opens. This page has several tabs on the top right corner including How It Works, Pricing, and Help. The homepage has "heroku toolbelt" as the header. Below the header there are four buttons: Mac OS X, Windows, Debian/Ubuntu, and Standalone. She clicks the Debin/Ubuntu button and copies the command that is displayed. She navigates to the terminal window and pastes the command there to install the heroku toolbelt. She types "Heroku login" to log on to the heroku account.]

Preparing an App for Deployment

Learning Objective

After completing this topic, you should be able to

- *prepare an app for deployment*

1.

In this topic, we'll learn how to prepare an app for deployment on Heroku. For the purpose of deployment, I've taken a simple app that just displays a message. To prepare this app for deployment, first of all you need to ensure that all the dependencies of the application are specified in the package.json file. Missing dependencies will cause problems when you try to deploy the application to Heroku. Once you've ensured that all the dependencies are defined in the package.json file, you need to also specify the version of node that should be used to run your application on Heroku. You should specify a node version matching the development and testing environment to ensure that your app works properly on the production server. To check the version of node that you're using, you can use the `node --version` command. Now you can specify this version in your package.json file. So I'll open the package.json file and add an entry here which says "engines": { "node": "0.10.30" }. I'll now Save the file and close it.

[The presenter types "node -- version" in the terminal window to check the version of the node being used. Then she navigates to the "package.json (~/helloworld) – gedit" window and types the following code: "engines": { "node": "0.10.30" } }]

Now I need to add a Procfile to my application. A Procfile is a simple text file that Heroku looks for to determine how to start your application. So I'll create a new file by the name of Procfile in the root folder of my application. I'll open this file and I'll write web: npm start in this file. I'll then Save it and close it. Now my app is ready for deployment. But, before this, let me run it locally to see if everything is working fine. Now, before I run the app, I need to ensure that all dependencies are installed. So I'll use the `npm install` command to install the dependencies. The dependencies have been installed. Let me `clear` the screen. I'll now use the `heroku local web` command to run the app locally. The app is now running on https://localhost:5000. I'll open a web browser and go to localhost:5000. So I can the see the app here. This means that my app is now ready for deployment. In this topic, we learned how to prepare an app for deployment.

*[Thereafter she navigates to the "*Procfile (~/helloworld) – gedit" window and types the following code: web: npm start She returns to the terminal window and types "npm install" to install the dependencies. Then she types "heroku local web" in the terminal window to run the app locally. She opens "localhost: 5000" web address in the web browser. The web page displays the app successfully.]*

Deploying an App to Heroku

Learning Objective

After completing this topic, you should be able to

- *deploy an app to Heroku*

1.

In this topic, we'll learn how to deploy an app to Heroku. Before deploying an app to Heroku, you need to initialize a local Git repository and commit your files to it. Git is a powerful version control system and is the means for deploying apps to Heroku. To initialize a local Git repository, first of all I'll move into the application folder and then use the `git init` command to initialize the Git repository. Now you can add your apps to the Git repository by using the `git add .` command, where dot refers to the current directory. Now you can commit your files to Git by using the `git commit -m` command. Now every time we commit something to a Git repository, we need to provide a comment. So I'll provide my comment as "First commit". Now the files have been committed. Once you've committed the changes to Git, you can deploy your app to Heroku, but before that let me clear my screen. Now, before deploying your app to Heroku, you need to ensure that you're logged on to Heroku. So let me use the `heroku login` command to log on to Heroku. I'll provide my Email ID and Password.

[The presenter types "cd helloworld" in the terminal window to move into the application folder. Then she types "git init" to initialize the git repository. Thereafter she types "git add." to add the apps to the git repository. Finally, she types "git commit-m" to commit the files to git and enters "First commit" as the comment. She types "heroku login" in the terminal window to log on to heroku.]

I'm now logged on to Heroku. Now I can use the `heroku create` command to create a new application on Heroku. So this command has created a new application on Heroku and it has also created a Git remote, which is nothing but a reference to the remote repository created on Heroku. You can verify the remote in your Git configuration by using the `git remote -v` command. Now you can use the `git push heroku master` command to push your app to Heroku. Now my app has been pushed to Heroku. Let me `clear` the screen. Now I can open my app in a browser by using the `heroku open` command. Now it says – Created new window in existing browser session. So let me open a browser and here I can see my app, which has been deployed on Heroku. In this topic, we learned how to deploy an app to Heroku.

[Then she types "heroku create" to create a new application on heroku. She verifies the remote by typing the "git remote –v" command. After this she uses the "git push heroku master" command to push the app to heroku. She uses the "heroku open" command to open the app in the browser. She navigates to the "fast-savannah-9927.herokuapp.com" web address in the web browser and views the app that has been deployed on heroku.]

Provisioning a Database

Learning Objective

After completing this topic, you should be able to

- *provision a database on MongoLab*

1.

In this topic, we'll learn how to provision a MongoDB database on Heroku. I have developed an application that allows users to submit and view articles. The application stores the articles in a MongoDB database. Now I want to deploy this application on Heroku. But, for this, I need to provision a MongoDB database on Heroku. For this, I'll use a Heroku addon called MongoLab which is a fully managed cloud-based database service. MongoLab is a paid service and offers multiple plans and prices. It also offers a free Sandbox plan with a storage limit of 496 MB. However, in order to use it, you need to verify your account on Heroku by providing your credit card details. I've already created a free account on MongoLab. Let me show you how to deploy my articles app on Heroku. Now I've already ensured that my package.json file includes the dependencies for the application and specifies an appropriate version of node. I've also added a Procfile in the root folder of my application.

[The heroku homepage is open. The page contains three tabs on the top left corner: Add-ons, Buttons, and Buildpacks. The Add-ons tab is open by default. The tab contains several data stores including Instaclustr, RedisGreen, and MongoLab.]

Now I need to do one more thing before I can push my application to Heroku. Let me open my app.js file. Here I have hardcoded the URL for accessing my local MongoDB database. I need to change this with the URL of the MongoDB database that I'll provision on Heroku. But, instead of hardcoding the URL here, I'll use an environment variable to specify the URL. So I'll specify the URL as process.env.MONGOLAB_URI. Here MONGOLAB_URI is the name of the environment variable that contains the actual URL of the MongoDB database on MongoLab. Let me now Save this file and close it. I'll now push this app to Heroku. For this, first of all I'll use the `git init` command to initialize a Git repository. I'll then add my app to Git by using `git add .` command. And then I'll commit my files to Git by using the `git commit -m` command and I'll provide the comment as "app prepared for deployment." Now the changes have been committed. Let me `clear` the screen.

[The presenter navigates to the "app.js (~/articles) – gedit" file and types the following URL: mongoose.connect (process.env.MONGOLAB_URI); Then she opens the terminal window and types "git init" to initialize the git repository. Thereafter she types "git add." to add the app to the git repository. Finally, she types "git commit-m" to commit the files to git and enters "app prepared for deployment" as the comment.]

I'll now log on to Heroku by using the `heroku login` command. I'll provide my e-mail ID and password. Now I'll create a new application on Heroku by using the `heroku create` command. I'll now push the app to Heroku by using the `git push heroku master` command. My app is now pushed to Heroku. Let me `clear` the screen. Now let me provision a database for my application on Heroku. For this, I'll use the `heroku addons:create mongolab` command. A database has been provisioned for my application. If you want to see the URI of the MongoDB database that has been provisioned, you can use the `heroku config | grep MONGOLAB_URI` command.

[She types "heroku login" in the terminal window to log on to heroku. Then she types "heroku create" to create a new application on heroku. After this she uses the "git push heroku master" command to push

the app to heroku. She uses the "heroku addons: create mongolab" command to provision a database for the application on heroku.]

Now here MONGOLAB_URI is the name of an environment variable on the production server. This environment variable contains the URL of the provisioned MongoDB database. If you remember, we used this environment variable in our app.js file to specify the URL of the MongoDB database. So now our application has been published to Heroku and a database has been provisioned for it. So we can use the `heroku open` command to view our application in the browser. So now you can see the main page of the application in the browser. Let us use this form to submit an article. You can see that the article appears at the bottom of the page. Now, if I reload the page, the submitted article is still there. This means that the article has been successfully stored in the MongoDB database that we provisioned. In this topic, we learned how to provision a MongoDB database for our deployed application.

[She uses the "heroku open" command to view the application in the browser.]

Introduction to Yeoman Generators

Learning Objective

After completing this topic, you should be able to

- *identify the benefits of Yeoman generators*

1.

In this topic, we'll learn about Yeoman generators. To create a MEAN application or any other application, developers usually need to perform a number of tasks such as scaffolding folders, downloading libraries and frameworks, downloading template engines, configuring a test environment, and creating a local web server. Developers usually lose a lot of time in doing all these activities for each new project that they work on. A tool called Yeoman has been developed to help developers with these trivial tasks. Yeoman is a scaffolding tool that helps you kick-start new projects. It prescribes best practices and tools to help you stay productive. Yeoman provides an ecosystem of generators that can be used to scaffold complete projects or useful parts of a project. Yeoman can be used to scaffold a wide variety of apps and not just MEAN apps. In this topic, we learned about Yeoman generators.

Installing the Yeoman MEAN Generator

Learning Objective

After completing this topic, you should be able to

- *install the Yeoman MEAN generator*

1.

In this topic, we'll learn how to install the Yeoman MEAN generator. Before installing the Yeoman MEAN generator, you need to ensure that you have the compatible versions of node.js and npm installed. Also before installing Yeoman MEAN generator, you need to have a package called yo installed in your box. To install yo, you can use the `sudo npm install –g yo` command. It will take some time to install. Once yo is installed, you can run a tool called `yo doctor`. This tool does a sanity check on your system. Now, if you see check marks against each of the points listed over here, it means that you are going right. Otherwise, yo doctor will give you advice regarding what you can do to rectify the problem. After installing yo, you can install the Yeoman MEAN generator. For this, you can use the `sudo npm install -g generator-meanjs` command. So now the Yeoman MEAN generator has been installed successfully. In this topic, we learned how to install the Yeoman MEAN generator.

[The presenter types "sudo npm install –g yo" command in the terminal window to install the package. She runs the "yo doctor" tool to run the sanity checks. The output displays check marks against all listed options. Then she types "sudo npm install –g generator-meanjs" to install the generator.]

Using the Yeoman MEAN Generator

Learning Objective

After completing this topic, you should be able to

- *use the Yeoman MEAN generator*

1.

In this topic, we'll learn how to use the Yeoman MEAN generator. Now, before we see how to use the Yeoman MEAN generator to scaffold a MEAN application, you need to ensure that you have Git installed on your box. If Git is not installed, you can install it by using the `sudo apt-get install git` command. I already have Git installed, so I'll skip this step. And now we can use the Yeoman MEAN generator to scaffold a new MEAN application. For this, I'll use the `sudo yo meanjs` command. Now this command will prompt us to answer a few questions about the project that we want to create. You can either answer the questions or accept the default answers by pressing the Enter key. So I'll specify the name of the project as MeanProject. So it has started cloning the MEAN repository.

[The presenter types "sudo yo meanjs" in the terminal window to use the mean generator.]

Now the repository has been cloned. It is still asking me a few questions, so I would like to call my application MeanProject. Here I'll accept the default values. Then it is asking me whether I would like to generate the article example CRUD module. I'll say, "Yes." Whether I would like to generate the chat example module. I'll say, "No." So now my MeanProject has been created. The command is simply installing the dependencies for me by using the `npm install` command. And this process will take a lot of time. So let us go to the project folder that we have created. So you can see that we have an entire folder structure of a MEAN application within the MeanProject folder. The boilerplate code included in this folder can now be customized as per your requirement. In this topic, we learned how to use the Yeoman MEAN generator to scaffold a new mean.js project.

Exercise: Deploy a MEAN Application

Learning Objective

After completing this topic, you should be able to

- *deploy a MEAN application*

1.

Exercise Overview
In this exercise, you will prepare a MEAN application for deployment. You will then deploy the app to Heroku and provision a database for the app. At this point, you can pause the video and perform the exercise. Once you have finished, resume this video and see how I would do it.

[Heading: Exercise: Deploy a MEAN Application.]

Solution
To deploy my app to Heroku, first of all, I need to open my package.json file and check whether all the dependencies have been correctly defined there. So right now all my dependencies have been defined here. I now need to specify an appropriate version of node. Let me Save this file and close it. Now I need to do one more thing before I can push my app to Heroku. For this, let me open my app.js file. Now here I have hardcoded the URL for accessing my local MongoDB database. I need to change this with the URL of the MongoDB database that I'll provision on Heroku. But, instead of hardcoding the URL here, I'll use an environment variable to specify the URL. So I'll specify the URL as process.env.MONGOLAB_URI. Here MONGOLAB_URI is the name of the environment variable that contains the actual URI of the MongoDB database on MongoLab. Let me now Save this file and close it.

[The presenter navigates to the "package.json (~/events) – gedit" file and types the following: "engines": { "node": "0.10.30" } She opens the "app.js (~/events) – gedit" file and types the following URL: mongoose.connect (process.env.MONGOLAB_URI); She opens the terminal window and types "cd events" to move into the events folder.]

I'll now push this app to Heroku. But, before that, I need to move into my events folder and initialize a Git repository there. Now I need to stage the changes. So I'll say `git add .` I'll now commit the changes by saying `git commit -m`. And I'll specify the comment as "App prepared for deployment." Now the changes have been committed. Let me `clear` the screen. I'll now create an app on Heroku by using the `heroku create` command. I'll now push the app to Heroku by using the `git push heroku master` command. My application has now been pushed to Heroku. Let me `clear` the screen. Now I need to provision a database for my app on Heroku. For this, I'll use the `heroku addons:create mongolab` command.

[Then she types "git init" to initialize a git repository. Thereafter she types "git add." to add the app to the git repository. Finally, she types "git commit-m" to commit the files to git and enters "App prepared for deployment" as the comment. She uses the "heroku create" to create a new application on heroku. After this she uses the "git push heroku master" command to push the app to heroku. She types the "heroku addons: create mongolab" command to provision a database for the application on heroku.]

Now a database has been provisioned for my app on Heroku. If you want to see the URL of the MongoDB database that has been provisioned, you can use the `heroku config | grep MONGOLAB_URI` command. Here MONGOLAB_URI is the name of the environment variable on the production server that contains the URL of the provisioned MongoDB database. If you remember, we

used this environment variable in our app.js file to specify the URL of the MongoDB database. So now our application has been pushed to Heroku and a database has been provisioned for it. So we can use the `heroku open` command to view our application in the browser. Now you can see the main page of the application in the browser. Let us use this form to submit an event. You can see that the event appears at the bottom of the page. If I reload this page, the submitted event is still there. This means that the event has been successfully saved in the MongoDB database that we provisioned and the app has been successfully deployed on Heroku.

[Then she types the "heroku config | grep MONGOLAB_URI" to view the URL of the database. Thereafter she uses the "heroku open" command to view the application in the browser.]