Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is an open source framework developed and maintained by the Node.js foundation.

Step1: npm init

Step2 npm install --save express

Step3npm install -g nodemon

Create index.js

var express = require('express');

var app = express();

app.get('/', function(req, res){

res.send("Hello world!");

});

app.listen(6000);

nodemon index.js

app.get(route, callback)

### res.send()

This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.

app.listen(port, [host], [backlog], [callback]])

This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

|  |  |
| --- | --- |
| **S.No.** | **Argument & Description** |
| 1 | **port**  A port number on which the server should accept incoming requests. |
| 2 | **host**  Name of the domain. You need to set it when you deploy your apps to the cloud. |
| 3 | **backlog**  The maximum number of queued pending connections. The default is 511. |
| 4 | **callback**  An asynchronous function that is called when the server starts listening for requests. |

A special method, ***all***, is provided by Express to handle all types of http methods at a particular route using the same function. To use this method, try the following.

app.all('/test', function(req, res){

res.send("HTTP method doesn't have any effect on this route!");

});

## **Third Party Middleware**

### body-parser

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using **npm install** --save body-parser and to mount it, include the following lines in your index.js −

var bodyParser = require('body-parser');

//To parse URL encoded data

app.use(bodyParser.urlencoded({ extended: false }))

//To parse json data

app.use(bodyParser.json())

To view all available options for body-parser, visit its github page.

### cookie-parser

It parses *Cookie* header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using npm install --save cookie-parser and to mount it, include the following lines in your index.js −

var cookieParser = require('cookie-parser');

app.use(cookieParser())

### express-session

It creates a session middleware with the given options. We will discuss its usage in the Sessions section.

We have many other third party middleware in ExpressJS. However, we have discussed only a few important ones here

Forms are an integral part of the web. Almost every website we visit offers us forms that submit or fetch some information for us. To get started with forms, we will first install the *body-parser*(for parsing JSON and url-encoded data) and multer(for parsing multipart/form data) middleware.

## **Setting up Mongoose**

npm install --save mongoose

Install mongodb locally:

Terminal1 -> C:\Program Files\MongoDB\Server\4.0\bin>mongod.exe --dbpath "C:\db\data"

Termimal2 -> C:\Program Files\MongoDB\Server\4.0\bin>mongo.exe

use my\_db

var personSchema = mongoose.Schema({

name: String,

age: Number,

nationality: String

});

var Person = mongoose.model("Person", personSchema);

app.get('/person', function(req, res){

res.send('person');

});

## **Retrieving Documents**

Mongoose provides a lot of functions for retrieving documents, we will focus on 3 of those. All these functions also take a callback as the last parameter, and just like the save function, their arguments are error and response. The three functions are as follows −

### Model.find(conditions, callback)

This function finds all the documents matching the fields in conditions object. Same operators used in Mongo also work in mongoose. For example,

Person.find(function(err, response){

console.log(response);

});

This will fetch all the documents from the person's collection.

Person.find({name: "Ayush", age: 20},

function(err, response){

console.log(response);

});

This will fetch all documents where field name is "Ayush" and age is 20.

We can also provide projection we need, i.e., the fields we need. For example, if we want only the **names** of people whose **nationality** is *"Indian"*, we use −

Person.find({nationality: "Indian"}, "name", function(err, response){

console.log(response);

});

### Model.findOne(conditions, callback)

This function always fetches a single, most relevant document. It has the same exact arguments as **Model.find()**.

### Model.findById(id, callback)

This function takes in the **\_id**(defined by mongo) as the first argument, an optional projection string and a callback to handle the response. For example,

Person.findById("507f1f77bcf86cd799439011", function(err, response){

console.log(response);

});

Let us now create a route to view all people records −

var express = require('express');

var app = express();

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my\_db');

var personSchema = mongoose.Schema({

name: String,

age: Number,

nationality: String

});

var Person = mongoose.model("Person", personSchema);

app.get('/people', function(req, res){

Person.find(function(err, response){

res.json(response);

});

});

app.listen(3000);

## **Updating Documents**

Mongoose provides 3 functions to update documents. The functions are described below −

### Model.update(condition, updates, callback)

This function takes a conditions and updates an object as input and applies the changes to all the documents matching the conditions in the collection. For example, following code will update the nationality "American" in all Person documents −

Person.update({age: 25}, {nationality: "American"}, function(err, response){

console.log(response);

});

### Model.findOneAndUpdate(condition, updates, callback)

It finds one document based on the query and updates that according to the second argument. It also takes a callback as last argument. Let us perform the following example to understand the function

Person.findOneAndUpdate({name: "Ayush"}, {age: 40}, function(err, response) {

console.log(response);

});

### Model.findByIdAndUpdate(id, updates, callback)

This function updates a single document identified by its id. For example,

Person.findByIdAndUpdate("507f1f77bcf86cd799439011", {name: "James"},

function(err, response){

console.log(response);

});

Let us now create a route to update people. This will be a **PUT** route with the id as a parameter and details in the payload.

var express = require('express');

var app = express();

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my\_db');

var personSchema = mongoose.Schema({

name: String,

age: Number,

nationality: String

});

var Person = mongoose.model("Person", personSchema);

app.put('/people/:id', function(req, res){

Person.findByIdAndUpdate(req.params.id, req.body, function(err, response){

if(err) res.json({message: "Error in updating person with id " + req.params.id});

res.json(response);

});

});

app.listen(3000);

To test this route, enter the following in your terminal (replace the id with an id from your created **people**) −

curl -X PUT --data "name = James&age = 20&nationality = American

"http://localhost:3000/people/507f1f77bcf86cd799439011

This will update the document associated with the id provided in the route with the above details.

## **Deleting Documents**

We have covered **Create, Read** and **Update**, now we will see how Mongoose can be used to **Delete** documents. We have 3 functions here, exactly like update.

### Model.remove(condition, [callback])

This function takes a condition object as input and removes all documents matching the conditions. For example, if we need to remove all people aged 20, use the following syntax −

Person.remove({age:20});

### Model.findOneAndRemove(condition, [callback])

This functions removes a **single**, most relevant document according to conditions object. Let us execute the following code to understand the same.

Person.findOneAndRemove({name: "Ayush"});

### Model.findByIdAndRemove(id, [callback])

This function removes a single document identified by its id. For example,

Person.findByIdAndRemove("507f1f77bcf86cd799439011");

Let us now create a route to delete people from our database.

var express = require('express');

var app = express();

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my\_db');

var personSchema = mongoose.Schema({

name: String,

age: Number,

nationality: String

});

var Person = mongoose.model("Person", personSchema);

app.delete('/people/:id', function(req, res){

Person.findByIdAndRemove(req.params.id, function(err, response){

if(err) res.json({message: "Error in deleting record id " + req.params.id});

else res.json({message: "Person with id " + req.params.id + " removed."});

});

});

app.listen(3000);

To check the output, use the following curl command −

curl -X DELETE http://localhost:3000/people/507f1f77bcf86cd799439011

This will remove the person with given id producing the following message −

{message: "Person with id 507f1f77bcf86cd799439011 removed."}

This wraps up how we can create simple CRUD applications using MongoDB, Mongoose and Express. To explore Mongoose further, read the [API docs.](http://mongoosejs.com/)

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user’s actions.

The following are the numerous uses of the HTTP Cookies −

* Session management
* Personalization(Recommendation systems)
* User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code −

npm install --save cookie-parser

Now to use cookies with Express, we will require the **cookie-parser**. cookie-parser is a middleware which *parses cookies attached to the client request object*. To use it, we will require it in our **index.js** file; this can be used the same way as we use other middleware. Here, we will use the following code.

var cookieParser = require('cookie-parser');

app.use(cookieParser());

cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like −

var express = require('express');

var app = express();

app.get('/', function(req, res){

res.cookie('name', 'express').send('cookie set'); //Sets name = express

});

app.listen(3000);

To check if your cookie is set or not, just go to your browser, fire up the console, and enter −

console.log(document.cookie);

You will get the output like (you may have more cookies set maybe due to extensions in your browser) −

"name = express"

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

console.log('Cookies: ', req.cookies);

Next time you send a request to this route, you will receive the following output.

Cookies: { name: 'express' }

## **Adding Cookies with Expiration Time**

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

//Expires after 360000 ms from the time it is set.

res.cookie(name, 'value', {expire: 360000 + Date.now()});

Another way to set expiration time is using **'maxAge'** property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

//This cookie also expires after 360000 ms from the time it is set.

res.cookie(name, 'value', {maxAge: 360000});

## **Deleting Existing Cookies**

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named **foo**, use the following code.

var express = require('express');

var app = express();

app.get('/clear\_cookie\_foo', function(req, res){

res.clearCookie('foo');

res.send('cookie foo cleared');

});

app.listen(3000);

In the next chapter, we will see how to use cookies to manage sessions.

HTTP is stateless; in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side. Sessions solve exactly this problem. You assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID.

We will need the *Express-session*, so install it using the following code.

npm install --save express-session

We will put the **session** and **cookie-parser** middleware in place. In this example, we will use the default store for storing sessions, i.e., MemoryStore. Never use this in production environments. The session middleware handles all things for us, i.e., creating the session, setting the session cookie and creating the session object in **req** object.

Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted). We can add more properties to the session object. In the following example, we will create a view counter for a client.

var express = require('express');

var cookieParser = require('cookie-parser');

var session = require('express-session');

var app = express();

app.use(cookieParser());

app.use(session({secret: "Shh, its a secret!"}));

app.get('/', function(req, res){

if(req.session.page\_views){

req.session.page\_views++;

res.send("You visited this page " + req.session.page\_views + " times");

} else {

req.session.page\_views = 1;

res.send("Welcome to this page for the first time!");

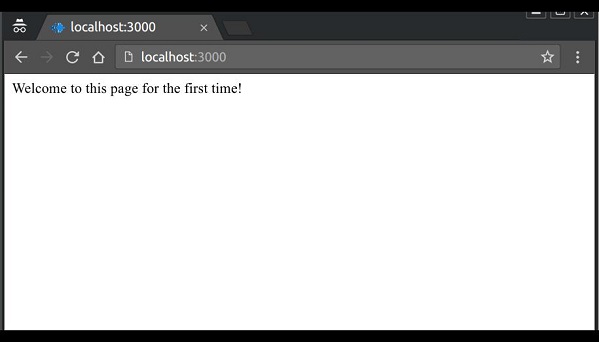
}

});

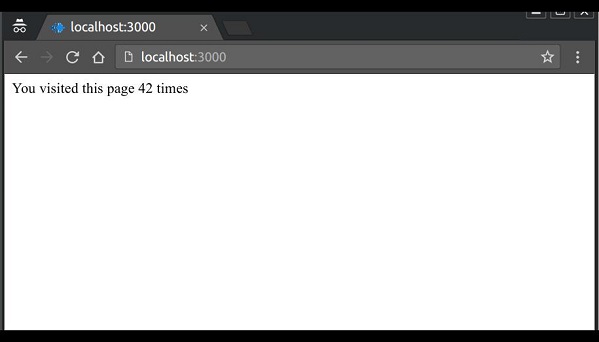
app.listen(3000);

What the above code does is, when a user visits the site, it creates a new session for the user and assigns them a cookie. Next time the user comes, the cookie is checked and the **page\_view** session variable is updated accordingly.

Now if you run the app and go to **localhost:3000**, the following output will be displayed.



If you revisit the page, the page counter will increase. The page in the following screenshot was refreshed 42 times.



Authentication is a process in which the credentials provided are compared to those on file in a database of authorized users' information on a local operating system or within an authentication server. If the credentials match, the process is completed and the user is granted authorization for access.

For us to create an authentication system, we will need to create a sign up page and a user-password store. The following code creates an account for us and stores it in memory. This is just for the purpose of demo; it is recommended that a persistent storage (database or files) is always used to store user information.

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var multer = require('multer');

var upload = multer();

var session = require('express-session');

var cookieParser = require('cookie-parser');

app.set('view engine', 'pug');

app.set('views','./views');

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use(upload.array());

app.use(cookieParser());

app.use(session({secret: "Your secret key"}));

var Users = [];

app.get('/signup', function(req, res){

res.render('signup');

});

app.post('/signup', function(req, res){

if(!req.body.id || !req.body.password){

res.status("400");

res.send("Invalid details!");

} else {

Users.filter(function(user){

if(user.id === req.body.id){

res.render('signup', {

message: "User Already Exists! Login or choose another user id"});

}

});

var newUser = {id: req.body.id, password: req.body.password};

Users.push(newUser);

req.session.user = newUser;

res.redirect('/protected\_page');

}

});

app.listen(3000);

Now for the signup form, create a new view called **signup.jade.**

### SIGNUP.JADE

html

head

title Signup

body

if(message)

h4 #{message}

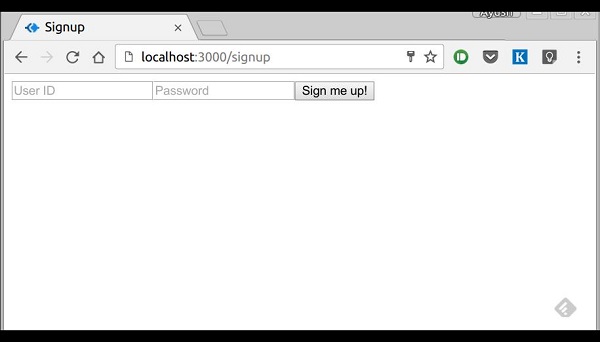
form(action = "/signup" method = "POST")

input(name = "id" type = "text" required placeholder = "User ID")

input(name = "password" type = "password" required placeholder = "Password")

button(type = "Submit") Sign me up!

Check if this page loads by visiting localhost:3000/signup.



We have set the required attribute for both fields, so HTML5 enabled browsers will not let us submit this form until we provide both id and password. If someone tries to register using a curl request without a User ID or Password, an error will be displayed. Create a new file called **protected\_page.pug** in views with the following content −

html

head

title Protected page

body

div Hey #{id}, How are you doing today?

div Want to log out?

div Logout

This page should only be visible if the user has just signed up or logged in. Let us now define its route and also routes to log in and log out −

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var multer = require('multer');

var upload = multer();

var session = require('express-session');

var cookieParser = require('cookie-parser');

app.set('view engine', 'pug');

app.set('views','./views');

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use(upload.array());

app.use(cookieParser());

app.use(session({secret: "Your secret key"}));

var Users = [];

app.get('/signup', function(req, res){

res.render('signup');

});

app.post('/signup', function(req, res){

if(!req.body.id || !req.body.password){

res.status("400");

res.send("Invalid details!");

} else {

Users.filter(function(user){

if(user.id === req.body.id){

res.render('signup', {

message: "User Already Exists! Login or choose another user id"});

}

});

var newUser = {id: req.body.id, password: req.body.password};

Users.push(newUser);

req.session.user = newUser;

res.redirect('/protected\_page');

}

});

function checkSignIn(req, res){

if(req.session.user){

next(); //If session exists, proceed to page

} else {

var err = new Error("Not logged in!");

console.log(req.session.user);

next(err); //Error, trying to access unauthorized page!

}

}

app.get('/protected\_page', checkSignIn, function(req, res){

res.render('protected\_page', {id: req.session.user.id})

});

app.get('/login', function(req, res){

res.render('login');

});

app.post('/login', function(req, res){

console.log(Users);

if(!req.body.id || !req.body.password){

res.render('login', {message: "Please enter both id and password"});

} else {

Users.filter(function(user){

if(user.id === req.body.id && user.password === req.body.password){

req.session.user = user;

res.redirect('/protected\_page');

}

});

res.render('login', {message: "Invalid credentials!"});

}

});

app.get('/logout', function(req, res){

req.session.destroy(function(){

console.log("user logged out.")

});

res.redirect('/login');

});

app.use('/protected\_page', function(err, req, res, next){

console.log(err);

//User should be authenticated! Redirect him to log in.

res.redirect('/login');

});

app.listen(3000);

We have created a middleware function *checkSignIn* to check if the user is signed in. The **protected\_page** uses this function. To log the user out, we destroy the session.

Let us now create the login page. Name the view as **login.pug** and enter the contents −

html

head

title Signup

body

if(message)

h4 #{message}

form(action = "/login" method = "POST")

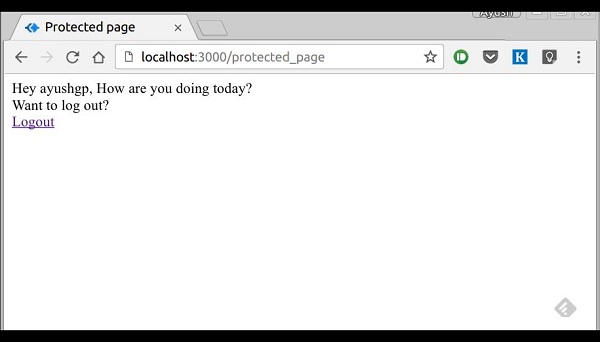
input(name = "id" type = "text" required placeholder = "User ID")

input(name = "password" type = "password" required placeholder = "Password")

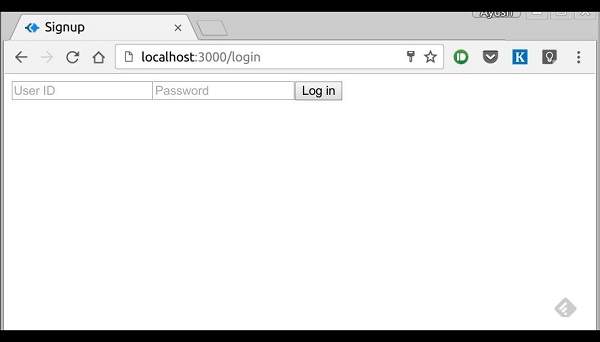
button(type = "Submit") Log in

Our simple authentication application is now complete; let us now test the application. Run the app using *nodemon index.js*, and proceed to localhost:3000/signup.

Enter a Username and a password and click sign up. You will be redirected to the **protected\_page** if details are valid/unique −



Now log out of the app. This will redirect us to the login page −



This route is protected such that if an unauthenticated person tries to visit it, he will be edirected to our login page. This was all about basic user authentication. It is always recommended that we use a persistent session system and use hashes for password transport. There are much better ways to authenticate users now, leveraging JSON tokens.

Scaffolding allows us to easily create a **skeleton for a web application**. We manually create our public directory, add middleware, create separate route files, etc. A scaffolding tool sets up all these things for us so that we can directly get started with building our application.

The scaffolder we will use is called **Yeoman**. It is a scaffolding tool built for **Node.js** but also has generators for several other frameworks (like flask, rails, django, etc.). To install Yeoman, enter the following command in your terminal −

npm install -g yeoman

Yeoman uses generators to scaffold out applications. To check out the generators available on **npm** to use with Yeoman, you can click on this [link](http://yeoman.io/generators/). In this tutorial, we will use the *'generator-Express-simple'*. To install this generator, enter the following command in your terminal −

npm install -g generator-express-simple

To use this generator, enter the following command −

yo express-simple test-app

You will be asked a few simple questions like what things you want to use with your app. Select the following answers, or if you already know about these technologies then go about choosing how you want them to be.

express-simple comes with bootstrap and jquery

[?] Select the express version you want: 4.x

[?] Do you want an mvc express app: Yes

[?] Select the css preprocessor you would like to use: sass

[?] Select view engine you would like to use: jade

[?] Select the build tool you want to use for this project: gulp

[?] Select the build tool you want to use for this project: gulp

[?] Select the language you want to use for the build tool: javascript

create public/sass/styles.scss

create public/js/main.js

create views/layout.jade

create views/index.jade

create views/404.jade

create app.js

create config.js

create routes/index.js

create package.json

create bower.json

identical .bowerrc

identical .editorconfig

identical .gitignore

identical .jshintrc

create gulpfile.js

I'm all done. Running bower install & npm install for you to install the

required dependencies. If this fails, try running the command yourself.

It will then create a new application for you, install all the dependencies, add few pages to your application(home page, 404 not found page, etc.) and give you a directory structure to work on.

This generator creates a very simple structure for us. Explore the many generators available for Express and choose the one that fits you right. Steps to working with all generators is the same. You will need to install a generator, run it using Yeoman; it will ask you some questions and then create a skeleton for your application based on your answers.

Error handling in Express is done using middleware. But this middleware has special properties. The error handling middleware are defined in the same way as other middleware functions, except that error-handling functions **MUST have four arguments** instead of three – **err, req, res, next**. For example, to send a response on any error, we can use −

app.use(function(err, req, res, next) {

console.error(err.stack);

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

});

Till now we were handling errors in the routes itself. The error handling middleware allows us to separate our error logic and send responses accordingly. The next() method we discussed in middleware takes us to next **middleware/route handler**.

For error handling, we have the **next(err)** function. A call to this function skips all middleware and matches us to the next error handler for that route. Let us understand this through an example.

var express = require('express');

var app = express();

app.get('/', function(req, res){

//Create an error and pass it to the next function

var err = new Error("Something went wrong");

next(err);

});

/\*

\* other route handlers and middleware here

\* ....

\*/

//An error handling middleware

app.use(function(err, req, res, next) {

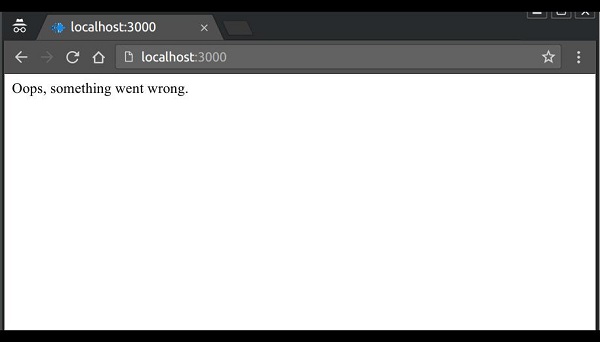
res.status(500);

res.send("Oops, something went wrong.")

});

app.listen(3000);

This error handling middleware can be strategically placed after routes or contain conditions to detect error types and respond to the clients accordingly. The above program will display the following output.



Unlike Django and Rails which have a defined way of doing things, file structure, etc., Express does not follow a defined way. This means you can structure the application the way you like. But as your application grows in size, it is very difficult to maintain it if it doesn't have a well-defined structure. In this chapter, we will look at the generally used directory structures and separation of concerns to build our applications.

First, we will discuss the best practices for creating node and Express applications.

* Always begin a node project using **npm init**.
* Always install dependencies with a **--save** or **--save-dev**. This will ensure that if you move to a different platform, you can just run *npm install* to install all dependencies.
* Stick with lowercase file names and camelCase variables. If you look at any npm module, its named in lowercase and separated with dashes. Whenever you require these modules, use camelCase.
* Don’t push node\_modules to your repositories. Instead npm installs everything on development machines.
* Use a **config** file to store variables
* Group and isolate routes to their own file. For example, take the CRUD operations in the movies example we saw in the REST API page.

## **Directory Structure**

Let us now discuss the Express’ Directory Structure.

### Websites

Express does not have a community defined structure for creating applications. The following is a majorly used project structure for a website.

test-project/

node\_modules/

config/

db.js //Database connection and configuration

credentials.js //Passwords/API keys for external services used by your app

config.js //Other environment variables

models/ //For mongoose schemas

users.js

things.js

routes/ //All routes for different entities in different files

users.js

things.js

views/

index.pug

404.pug

...

public/ //All static content being served

images/

css/

javascript/

app.js

routes.js //Require all routes in this and then require this file in

app.js

package.json

There are other approaches to build websites with Express as well. You can build a website using the MVC design pattern. For more information, you can visit the following links.

<https://code.tutsplus.com/tutorials/build-a-complete-mvc-website-with-expressjs--net-34168>

and,

<https://www.terlici.com/2014/08/25/best-practices-express-structure.html>.

### RESTful APIs

APIs are simpler to design; they don't need a public or a views directory. Use the following structure to build APIs −

test-project/

node\_modules/

config/

db.js //Database connection and configuration

credentials.js //Passwords/API keys for external services used by your app

models/ //For mongoose schemas

users.js

things.js

routes/ //All routes for different entities in different files

users.js

things.js

app.js

routes.js //Require all routes in this and then require this file in

app.js

package.json

You can also use a [yeoman generator](https://github.com/trwalker/generator-express-rest-api) to get a similar structure.