|  |
| --- |
| Review |
| Install npm # 1. npm install package\_name –save  # 2. use package :  const express = require('express'); var app=require(package\_name);  # 3. call it :-  var app=package\_name();  #4. Set up app to listen for connections  app.listen(3011,function(){  console.log("Server started..");  });  #5. root route  app.get('/',function(req,res){  res.send("This is a home page");  });  app.get('/content',function(req,res){  res.send("This is the CONTENT page");  });  app.get('/content/:myParam',function(req,res){  var myVar=req.params.myParam;  res.send("You sent " + myVar + " to the server ");  });  Type null>file.js to create file On; browser type: localhost:3000/content |

|  |
| --- |
| REST |
| -REpresentational State Transfer  -An architecture for building web apps  Characteristics:  -can return data w/o exposing methods  -can be used on almost any protocol ….HTTP  -almost any content JSON  -works on http using standard http methods.  State is data in some form  State transfer means data transfer  **REST Constraints:**  -stateless- api calls are independent  -client server separation :-separate entities that comm. But are not otherwise connected  -cachedable :- encourage caching .Server can include info like version info  -uniform interface:-all api requests will look the same regardless of origin  -layered-layers allow add additional functions…security,load-balancing,etc.  CREATE:-  *New* /dataitem/new get show a create form (front end )  *CREATE* /dataitem post create a new dataitem in DB (backend)  READ: *show all*  /dataitem get list all data items  *Show item* /dataitem/:id get list details for one item  Update :  *Edit* /dataitem/:id/edit get show an edit form (front-end)  *Update* /dataitem/:id put update item info in DB  Delete:-  *Delete* /dataitem/:id delete delete item from database |

|  |
| --- |
| REST IN BRIGHTSPACE:  REST stands for REpresentational State Transfer and is currently the most popular architecture for web application development. Some things that make REST appealing for web development include:   1. It can return data without exposing methods 2. Can be used over almost any protocol but HTTP for web 3. Supports any content type - mainly JSON 4. Works in HTTP with just regular HTTP methods   One notable thing that is not supported by REST is the ability to maintain state as REST is considered stateless.  There are some important constraints for REST:   1. Client-Server Separation 2. Stateless 3. Cacheable 4. Uniform Interface 5. Layered System   **Client Server Separation**  The client and server are separate from one another and one is not dependent on the other. Ie. You should be able to change the front end of your application without affecting the data structure in the database.  **Stateless**  As we mentioned above, REST APIs are stateless. This means that any call to an API can be made independently of any other. It also implies that any API call contains all of the information it needs to execute and complete. It should never have to rely on any contextual information or information from the server for this.  **Cacheable**  APIs should be designed to encourage data caching. When a server returns data, it can include information similar to version information so a client doesn't need to re-request the same data.  **Uniform Interface**  By designing a uniform interface to the API, all requests to the API look the same regardless of whether they come from Firefox or any other browser. This principle helps with #1.  **Layered System**  There may be numerous layers between the Client and the Server to perform various functions such as security or load-balancing or other functions. These layers should be invisible to the client and have no affect on how the client communicates with the server. **RESTful Routes** A RESTful API will be implemented to provide a set of RESTful routes which encapsulate so-called CRUD functionality. With web applications, these are set up using HTTP methods GET, POST, PUT and DELETE. The routes look like the following:  7 REST routes:  new /dataitem/new GET show a new dataitem creation form  create /dataitem POST create a new dataitem and then redirect  index /dataitem GET list all dataitems  show /dataitem/:id GET show a single dataitem  edit /dataitem/:id/edit GET show an edit form for a single dataitem  update /dataitem/:id PUT update a particular dataitem and then redirect  destroy /dataitem/:id DELETE delete a particular dataitem and then redirect  Note that while there are seven routes, some can be implemented in the front-end while some must be implemented as the back-end. The individual routes using PUT, POST and DELETE are implemented in the backend while the front-end provides an initial form or view(GET) which in turn triggers these routes.  rest constraints: <https://restfulapi.net/rest-architectural-constraints/> |

|  |
| --- |
| Database Connectivity :  -various packages available for DBs  -generally 2 types  -direct/native connection  Eg:- mysql2,MongoDB  -ORM/ODM (Object Document Mapping/Object Relational Mapping)  Eg:- Mongoose,Sequelize |
| **Mongoose :**  -an ODM package  -install as usual but will need an additional package: ***body-parser***  -set variables w/ require(‘ ‘ );  -call connect method on mongoose to connect to DB  -use string similar to Mongo connect string |
| **1.create a Schema Object**  **2. Use schema to create a model object**  **3.Use model to issue DB calls → npm install mongoose body-parser** |

|  |
| --- |
| Implementation: |
| Step1:  *// setup node*  npm init  *// setup our express package manager →* npm install express - -save  const express = require('express');  const server = express();   * at the end of our code setup LISTENER to run or server express package manager:   *//set up server to listen for connection*  server.listen(3010,function(){  console.log("Server started....")  }); |
| Step 2:  // setup mongo db connection ( ***npm install mongoose body-parser - -save***)  const bodyParser =require('body-parser');  const mongoose =require('mongoose');  Step 3:  // CREATE A CONNECTION TO THE MONGODB  //At home localhost:27017/movie  mongoose.connect( "mongodb://192.168.75.128:27017/movie",  {useNewUrlParser:***true***} )  server.***use***(bodyParser.json());  Step 4:  // DEFINE SCHEMA OBJECT CORRESPONDING TO DB COLLECTION  in our table(collections) of movie we have 4 fields: title, year, imdb, type  we make a connection to DB with schema  var movieSchema= new mongoose.Schema({  title: {type: String, required:false},  year: {type: Number, required:false},  imdb:{type: String, required:false},  type: {type: String, required:false}  });  Step5:  // CREATE MODEL OBJECT BASED ON THE ABOVE SCHEMA OBJECT  var Movie= mongoose.**model**("Movie" , movieSchema);  Step6:  //CREATING ROUTE  //ROUTE FOR OUR BACKEND  server.get( '/', function(req,res){  res.redirect("/movie");  })  //GET LIST ALL ROUTE  server.get( '/movie', function(req,res){  Movie.find({} , function( err , movies ){  if(err) {  return res  .status(400)  .json( { success : false , error : err } );  }  if( !movies.length){  return res  .status(404)  .json({ success : false , error : "No movies found"});  }  return res  .status(200)  .json({ success : true , data : movies});  });  }) |

today’s code

//#region step1 // setup node -> npm init // setup our express package manager → npm install express - -save

const express = require("express");

const server = express();

//#endregion

//#region step2 // setup mongo db connection ( npm install mongoose body-parser - -save)

const bodyParser = require("body-parser");

const mongoose = require("mongoose");

//#endregion

//#region step3 // CREATE A CONNECTION TO THE MONGODB

//At-home localhost:27017/movie At-class:"mongodb://192.168.75.128:27017/movie"

mongoose.connect("mongodb://localhost:27017/movie", { useNewUrlParser: true });

server.use(bodyParser.json());

//#endregion

//#region step4 // DEFINE SCHEMA OBJECT CORRESPONDING TO DB COLLECTION

//in our table(collections) of movie we have 4 fields: title, year, imdb, type // we make a connection to DB with schema \*/

var movieSchema = new mongoose.Schema({

title: { type: String, required: false },

year: { type: Number, required: false },

imdb: { type: String, required: false },

type: { type: String, required: false },

});

//#endregion

//#region step5 // CREATE MODEL OBJECT BASED ON THE ABOVE SCHEMA OBJECT

var Movie = mongoose.model("Movie", movieSchema);

//#endregion

//#region step6: //CREATING ROUTE //ROUTE FOR OUR BACKEND

////////////ROUTE FOR OUR BACKEND

server.get("/", function (req, res) {

res.redirect("/movie");

});

////////////GET LIST ALL ROUTE

server.get("/movie", function (req, res) {

Movie.find({}, function (err, movies) {

if (err) {

return res.status(400).json({ success: false, error: err });

}

if (!movies.length) {

return res.status(404).json({ success: false, error: "No moviesfound" });

}

return res.status(200).json({ success: true, data: movies });

});

});

//#endregion

//#region step1

server.listen(3010, function () {

console.log("Server is starting...");

});

//#endregion