**Batch: A1 Roll No.: 1811015,1811016,1811018**

**Experiment / assignment / tutorial No.\_\_5\_\_**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| Title: Demonstrate the use of RESTful web services. |

**AIM:** To Create a RESTful API server using Express, Node.js. and Mongodb and test it on Postman application.

**Problem Definition:**

Demonstrate the use of RESTful API and perform CRUD operations according to the Mini-project topics.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Expected OUTCOME of Experiment:

**CO 2:** Architect MEAN stack applications from scratch. **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. Express .js Deep API reference, by Azat Marden, Apress, 2nd edition, 2015.
2. <https://codeburst.io/building-a-rest-api-using-mongo-db-75cac3403fab>
3. <https://www.edureka.co/blog/rest-api-with-node-js/>
4. https://bezkoder.com/node-express-mongodb-crud-rest-api/

**Pre-Lab/ Prior Concepts:**

**RESTful** Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

Following four HTTP methods are commonly used in REST based architecture.

* **GET** − Provides a read only access to a resource.
* **POST** − Used to create a new resource.
* **DELETE** − Used to remove a resource.
* **PUT** − Used to update a existing resource or create a new resource.

**Methodology:**

* Creation of the project
* Creation of database
* Connection of database with the project
* Use of middleware to process json objects
* Creation of methods to handle requests
* Implementation of logic in thode methods
* Testing API with the postman
* Terminating the connection link.

**Implementation Details:**

**Mini Project Topic: E-commerce application**

**To perform a user authentication system, which includes registration of new users, updating the user data, displaying user profiles and deleting an account.**

* **POST** − Used to create a new resource.

code:

//auth.js

router.post("/register", async (req, res) => {

  const user = new User({

    name: req.body.name,

    email: req.body.email,

    phone: req.body.phone,

    password: req.body.password,

  });

  try {

    const savedUser = await user.save();

    console.log(savedUser)

    res.send(savedUser);

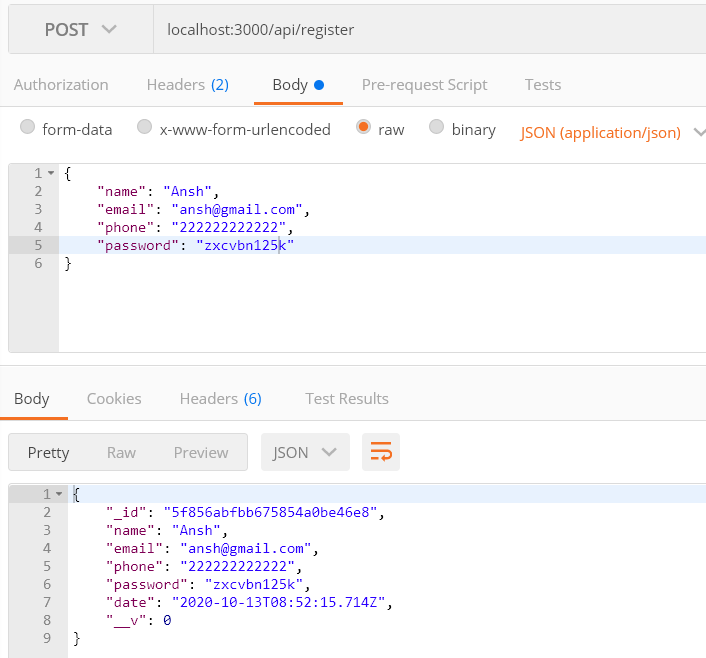
  } catch (err) {

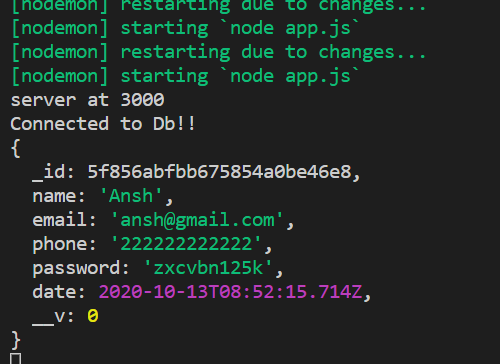
    res.status(400).send(err);

  }

});

Output:







* **GET** − Provides a read only access to a resource.

code:

router.get("/profile", async (req, res) => {

  const user = await User.find();

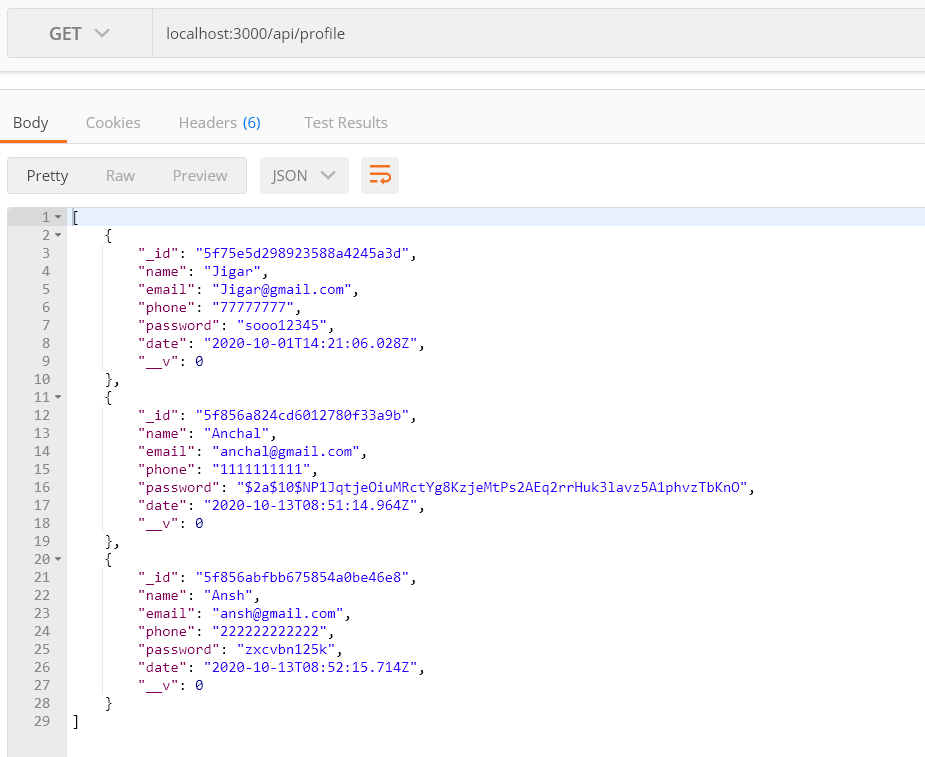
  if(!user) return res.status(400).send('User not found');

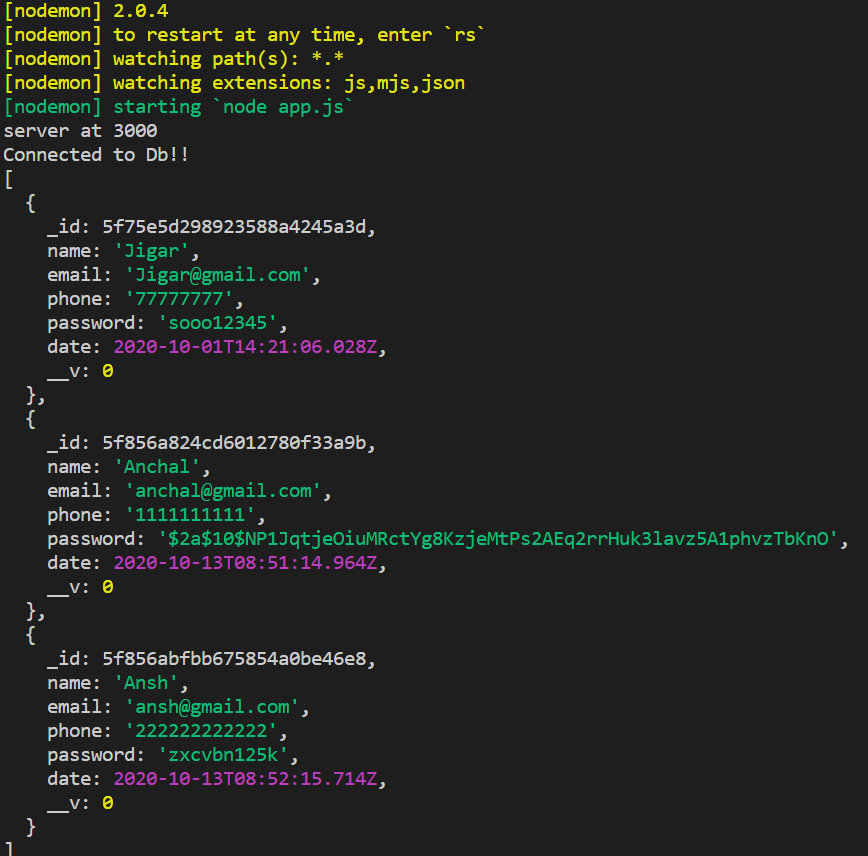
  res.send(user);

  console.log(user);

});

**Output:**

****

****

* **PUT** − Used to update an existing resource or create a new resource.

**code:**

**router.put("/update", async (req, res) => {**

**var user = await User.findOneAndUpdate({name: req.body.name},{$set: {phone: req.body.phone}});**

**if(!user) return res.status(400).send('User not found');**

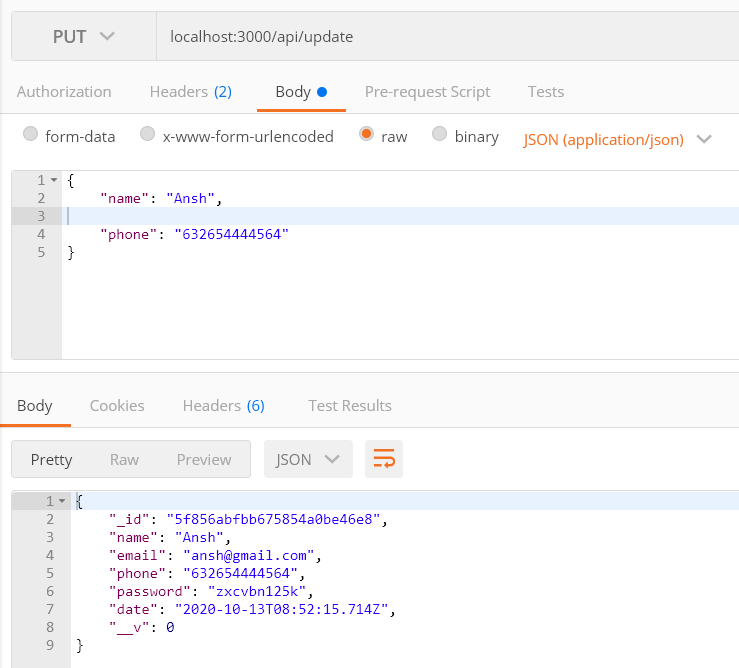
**user = await User.findOne({name: req.body.name});**

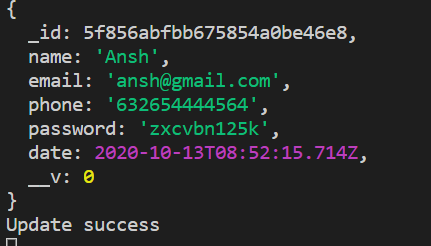
**res.send(user);**

**console.log(user,'\nUpdate success');**

**});**

**Output:**

****

****

* **DELETE** − Used to remove a resource.

**//auth.js**

**router.delete("/delete", async (req, res) => {**

**var user = await User.deleteOne({name: req.body.name});**

**if(!user) return res.status(400).send('User not found');**

**res.send(user);**

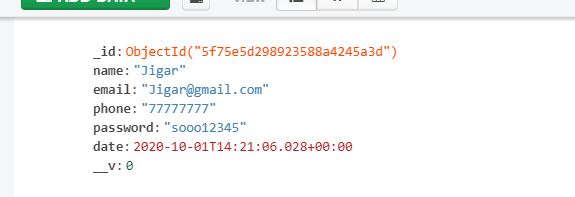
**console.log(user,'\nDelete success');**

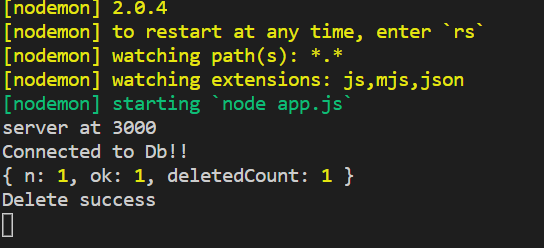
**});**

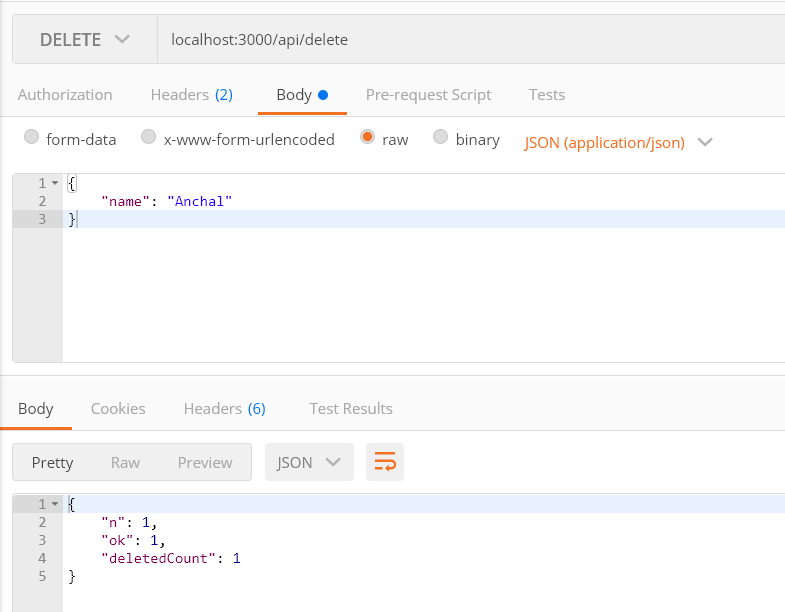
**Output:**

****

**After deleting user:**

****

****

****

**Steps for execution:**

* Creation of the project
  1. npm init
  2. We are going to create a new project to implement the above problem statement.
* Installation of all the required modules:

 We are going to need the modules of mongodb, mongoose and mongoose to interact with the mongodb database and do CRUD operations.

* 1. npm install mongoose
* Installing mongodb:
  1. We go to the official website of mongodb and install the community server as per our operating system.
  2. Mongo db compass also gets installed with this installation of community server.
* Creating database:
  1. Either by manually opening mongodb compass and creating a cluster or by using the “use” command to create a database.
  2. We created a database named ‘Market’ and a collection named ‘user’.
* Connecting database:
  1. we connect to the database created locally by using the mongodb module. This was an open database so no authentication was required. On connection successful the program would go further but in case some errors occurs than we return the function with an error
* Creation of schema:
  1. We can create the schema of the user. We will create a document that can accept the name, email address, contact number, password, date of joining, etc.
* Insertion of data:
  1. We will use the rest api to get the request containing the details about the user. In the post request the new user will be given the value, passed in the request and the new user will be stored in the database.
  2. Success message will be returned in the response.
* Retrieval of data:
  1. We pass the name or id of the user in the request and in return we will get the details of the particular user.
  2. If no particular id or name is passed then we will get the details of all the users.
  3. We retrieve the data in the get request.
  4. Response is filled with the details of the user.
* Update of data:
  1. We are using findByIdandUpdate to find the particular element and than update it
  2. We send the id in the request and find the element in the database with the help of the id.
  3. We pass the updated value in the request. Once we find the object we update it’s certain characteristics with the help of the put request.
* Delete of data:
  1. We are using findOneandDelete() to delete a certain document with a particular id. We will be using a delete request for this operation.
  2. if retrieval succeeds, we display delete successful message
  3. In case of errors we return the error message.
  4. We send the id of the element in the request which is going to be deleted.
  5. We find that element in the database and delete it.
  6. In response we send the successful message or error code we achieved.

* Drop Collection:
  1. We use dropCollection to drop the collection created.
  2. if retrieval succeeds, we display delete successful message
  3. In case of errors we return the error message.

**Conclusion:**

**Hence in this experiment we learn and implement the various CRUD operations in REST Api, using postman.**

**Post lab questions:**

1. **What are the different principles of REST API?**

#### Uniform interface

As the constraint name itself applies, you MUST decide APIs interface for resources inside the system which are exposed to API consumers and follow religiously. A resource in the system should have only one logical URI, and that should provide a way to fetch related or additional data. It’s always better to **synonymize a resource with a web page**.

Any single resource should not be too large and contain each and everything in its representation. Whenever relevant, a resource should contain **links (HATEOAS) pointing to relative URIs** to fetch related information.

Also, the resource representations across the system should follow specific guidelines such as naming conventions, link formats, or data format (XML or/and JSON).

All resources should be accessible through a common approach such as HTTP GET and similarly modified using a consistent approach.

*Once a developer becomes familiar with one of your APIs, he should be able to follow a similar approach for other APIs.*

#### Client–server

This constraint essentially means that client application and server application MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs, and that’s all. Today, this is standard practice in web development, so nothing fancy is required from your side. Keep it simple.

*Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.*

#### Stateless

Roy fielding got inspiration from HTTP, so it reflects in this constraint. Make all client-server interactions stateless. The server will not store anything about the latest HTTP request the client made. It will treat every request as new. No session, no history.

If the client application needs to be a stateful application for the end-user, where user logs in once and do other authorized operations after that, then each request from the client should contain all the information necessary to service the request – including authentication and authorization details.

*No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.*

#### Cacheable

In today’s world, the caching of data and responses is of utmost importance wherever they are applicable/possible. The webpage you are reading here is also a cached version of the HTML page. Caching brings performance improvement for the client-side and better scope for scalability for a server because the load has reduced.

In REST, caching shall be applied to resources when applicable, and then these resources MUST declare themselves cacheable. Caching can be implemented on the server or client-side.

*Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.*

#### Layered system

REST allows you to use a layered system architecture where you deploy the APIs on server A, and store data on server B and authenticate requests in Server C, for example. A client cannot ordinarily tell whether it is connected directly to the end server or an intermediary along the way.

#### Code on demand (optional)

Well, this constraint is optional. Most of the time, you will be sending the static representations of resources in the form of XML or JSON. But when you need to, you are free to return executable code to support a part of your application, e.g., clients may call your API to get a UI widget rendering code. It is permitted.

*All the above constraints help you build a truly RESTful API, and you should follow them. Still, at times, you may find yourself violating one or two constraints. Do not worry; you are still making a RESTful API – but not “truly RESTful.”*

1. **Explain the advantages of RESTful web services?**

1.     **Separation between the client and the server**: the REST protocol totally separates the user interface from the server and the data storage. This has some advantages when making developments. For example, it improves the portability of the interface to other types of platforms, it increases the scalability of the projects, and allows the different components of the developments to be evolved independently.

2.     **Visibility, reliability and scalability**. The separation between client and server has one evident advantage, and that is that each development team can scale the product without too much problem. They can migrate to other servers or make all kinds of changes in the database, provided the data from each request is sent correctly. The separation makes it easier to have the front and the back on different servers, and this makes the apps more flexible to work with.

3.     **The REST API is always independent of the type of platform or languages**: the REST API always adapts to the type of syntax or platforms being used, which gives considerable freedom when changing or testing new environments within the development. With a REST API you can have PHP, Java, Python or Node.js servers. The only thing is that it is indispensable that the responses to the requests should always take place in the language used for the information exchange, normally XML or JSON.