

Agenda

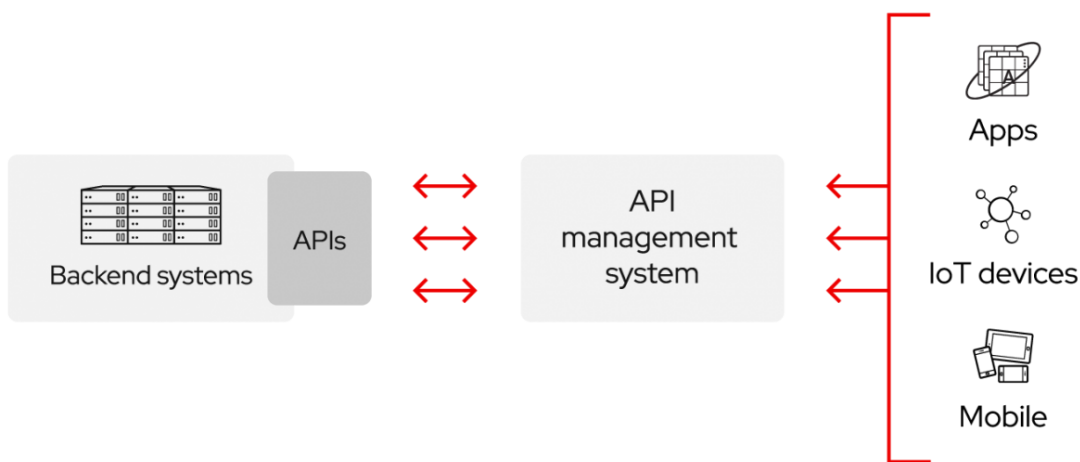
- Introduction to APIs and Rest APIs
- What is Express JS?
- How to create a server using express?
- Express Routing
- Postman and demo

What's an API?

An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response). For example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.

You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It's also a way for an organization to share resources and information while maintaining security, control, and authentication—determining who gets access to what.



REST

REST (Representational State Transfer) is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information or representation, is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XML, Python, PHP, or plain text. JSON is the most generally popular file format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

Something else to keep in mind: Headers and parameters are also important in the HTTP methods of a RESTful API HTTP request, as they contain important identifier information as to the request's metadata, authorization, uniform resource identifier (URI), caching, cookies, and more. There are request headers and response headers, each with their own HTTP connection information and status codes.

In order for an API to be considered RESTful, it has to conform to these criteria:

- **Client-server architecture:** REST architecture is composed of clients, servers, and resources, and it handles requests through HTTP.
- **Statelessness:** No client content is stored on the server between requests. Information about the session state is, instead, held with the client.
- **Cacheability:** Caching can eliminate the need for some client-server interactions.
- **Layered system:** Client-server interactions can be mediated by additional layers. These layers could offer additional features like load balancing, shared caches, or security.
- **Code on demand (optional):** Servers can extend the functionality of a client by transferring executable code.
- **Uniform interface:** This constraint is core to the design of RESTful APIs and includes 4 facets:
 - **Resource identification in requests:** Resources are identified in requests and are separate from the representations returned to the client.

- **Resource manipulation through representations:** Clients receive files that represent resources. These representations must have enough information to allow modification or deletion.
- **Self-descriptive messages:** Each message returned to a client contains enough information to describe how the client should process the information.
- **Hypermedia as the engine of application state:** After accessing a resource, the REST client should be able to discover through hyperlinks all other actions that are currently available.

Intro to Express JS

What is Express JS?

Express is a node js web application framework that provides broad features for building web and mobile applications. It is used to build a single pag multipage, and hybrid web application.

Express.js provides an easy way to create web server and render HTML pages for different HTTP requests by configuring routes for your application.

It's a layer built on the top of the Node js that helps manage servers and routes.

Why Express JS?

- Express was created to make APIs and web applications with ease,
- It saves a lot of coding time almost by half and still makes web and
- mobile applications are efficient.
- Another reason for using express is that it is written in javascript as javascript is an easy language even if you don't have a previous
- knowledge of any language. Express lets so many new developers enter the field of web development.

The reason behind creating an express framework for node js is:

- Time-efficient
- Fast
- Economical
- Easy to learn
- Asynchronous

Features of Express JS

Middleware

Middleware is a request handler that has access to the application's request-response cycle.

Routing

It refers to how an application's endpoint's URLs respond to client requests.

Templating

It provides templating engines to build dynamic content on the web pages by creating HTML templates on the server.

Debugging

Express makes it easier as it identifies the exact part where bugs are.

Companies That Are Using Express JS

- Netflix
- IBM
- ebay
- Uber

How to create a server using express?

Previously we saw how we can create a server using http module in nodejs. Now we will se how we can create a server using express js.

First, create a new folder and initialize it with a blank **package.json** file using the command below.

```
npm init -y
```



The `-y` flag when passed to NPM commands tells the generator to use the defaults instead of asking questions. Example:

```
npm init -y
```

Will simply generate an empty npm project without going through an interactive process.

To install the latest and stable version Express in your project, run the following command.

```
npm i express
```

Upon execution of this command, you will have the express framework installed in your project. Let's create a sample code to test out Express framework.

Web Server

First of all, import the Express.js module and create the web server as shown below.

- `app.js`

```
const express = require('express');
const app = express();
```

```
// define routes here..
```

```
const server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

In the above example, we imported Express.js module using `require()` function. The `express` module returns a function. This function returns an object which can be used to configure Express application (app in the above example).

The app object includes methods for routing HTTP requests, configuring middleware, rendering HTML views and registering a template engine.

The `app.listen()` function creates the Node.js web server at the specified host and port. It is identical to Node's `http.Server.listen()` method.

Run the above example using `node app.js` command and point your browser to <http://localhost:5000>. It will display **Cannot GET /** because we have not configured any routes yet.

Configuring Routes

Use app object to define different routes of your application. The app object includes `get()`, `post()`, `put()` and `delete()` methods to define routes for HTTP GET, POST, PUT and DELETE requests respectively.

The following example demonstrates configuring routes for HTTP requests.

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello World</h1></body></html>');
});
```

```
app.post('/submit-data', function (req, res) {
  res.send('POST Request');
});
```

```
app.put('/update-data', function (req, res) {
  res.send('PUT Request');
});
```

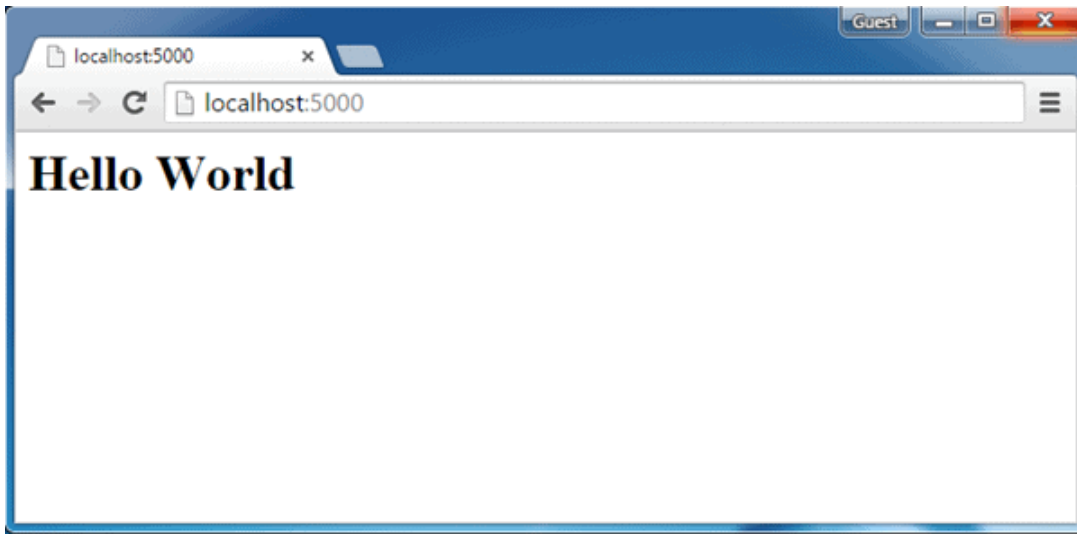
```
app.delete('/delete-data', function (req, res) {
  res.send('DELETE Request');
});
```

```
var server = app.listen(5000, function () {
```

```
    console.log('Node server is running..');  
  });
```

In the above example, `app.get()`, `app.post()`, `app.put()` and `app.delete()` methods define routes for HTTP GET, POST, PUT, DELETE respectively. The first parameter is a path of a route which will start after base URL. The callback function includes `request` and `response` object which will be executed on each request.

Run the above example using `node server.js` command, and point your browser to <http://localhost:5000> and you will see the following result.



Handle POST Request

Here, you will learn how to handle HTTP POST request and get data from the submitted form.

First, create `Index.html` file in the root folder of your application and write the following HTML code in it.

```
<!DOCTYPE html>  
  
<html>  
<head>  
  <meta charset="utf-8" />  
  <title></title>  
</head>  
<body>  
  <form action="/submit-student-data" method="post">  
    First Name: <input name="firstName" type="text" /> <br />  
    Last Name: <input name="lastName" type="text" /> <br />  
    <input type="submit" />  
  </form>  
</body>  
</html>
```



Body Parser

To handle HTTP POST request in Express.js version 4 and above, you need to install middleware module called `body-parser`. The middleware was a part of Express.js earlier but now you have to install it separately.

This `body-parser` module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request. Install `body-parser` using NPM as shown below

```
npm install body-parser
```



Earlier versions of Express used to have a lot of middleware bundled with it. `bodyParser` was one of the middleware that came with it. When Express 4.0 was released they decided to remove the bundled middleware from Express and make them separate packages instead. The syntax then changed from `app.use(express.json())` to `app.use(bodyParser.json())` after installing the `bodyParser` module.

`bodyParser` was added back to Express in release 4.16.0, because people wanted it bundled with Express like before. That means you don't have to use `bodyParser.json()` anymore if you are on the latest release. You can use `express.json()` instead.

Now, import body-parser and get the POST request data as shown below.

```
var express = require('express');
var app = express();

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', function (req, res) {
  res.sendFile('index.html');
});

app.post('/submit-student-data', function (req, res) {
  var name = req.body.firstName + ' ' + req.body.lastName;

  res.send(name + ' Submitted Successfully!');
});

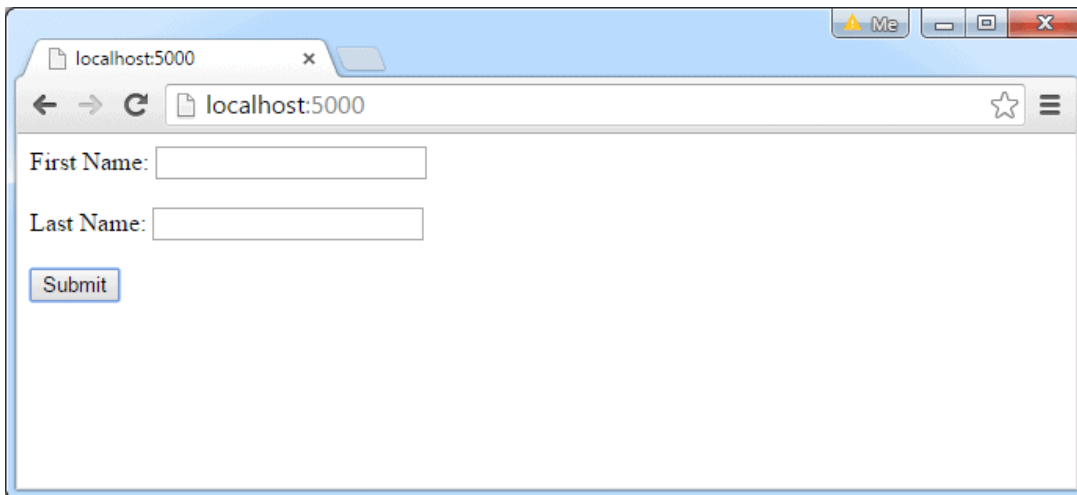
var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

The **`**res.sendFile()`** function basically transfers the file at the given path and it sets the Content-Type response HTTP header field based on the filename extension.

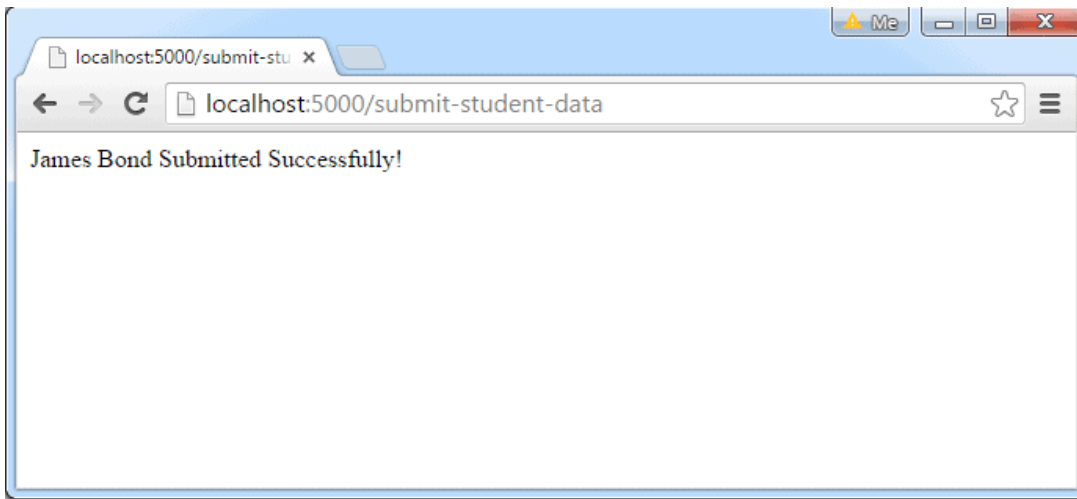
The **`res.send()`** function basically sends the HTTP response. The body parameter can be a String or a Buffer object or an object or an Array.

In the above example, POST data can be accessed using req.body. The req.body is an object that includes properties for each submitted form. Index.html contains firstName and lastName input types, so you can access it using req.body.firstName and req.body.lastName.

Now, run the above example using **`node server.js`** command, point your browser to <http://localhost:5000> and see the following result.

A screenshot of a web browser window. The address bar shows 'localhost:5000'. The page content includes two text input fields labeled 'First Name:' and 'Last Name:'. Below these fields is a button labeled 'Submit'. The browser window has a standard title bar with 'Me' and window control buttons.

Fill the First Name and Last Name in the above example and click on submit. For example, enter "James" in First Name textbox and "Bond" in Last Name textbox and click the submit button. The following result is displayed.



This is how you can handle HTTP requests using Express.js. We will learn rest of the request types in upcoming modules

Learn more about routing in express [here](#).

Creating a basic REST API using Express

Step 1 — Build and Run an Express Server with Node.js

In this step, you will write up an Express server with Node.js and run it locally.

First, Setup [Express](#). After the setup you should have one directory `node_modules` and two files `package.json` & `package-lock.json`.

Next, you will write a simple “Hello World” app. Create an `index.js` file and add the code below to it.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.json('Hello World!');
});

app.listen(port, () => {
  console.log(`app listening at http://localhost:${port}`)
});
```

Finally, you should run the server with a command that tells Node.js to run your Express sever:

```
node index.js
```

Opening `http://localhost:3000/` should return a “Hello World” message as expected.

Stop the server by running `CTRL + C`.

In the next step, you will add an API endpoint to your server.

Step 2 — Create a GET Endpoint

In this step, you are going to create an endpoint that returns a list of all the movies stored in a JSON file.

Let's consider that you have a JSON database of movies in a file named `movies.json` that lies in the same directory as your `index.js` file.

```
[
  {
    "id": 1,
    "title": "Star girl"
  },
  {
    "id": 2,
    "title": "Five feet apart"
  },
  {
    "id": 3,
    "title": "Fifty shades of Grey"
  }
]
```

```

    },
    {
      "id": 4,
      "title": "Fifty shades of Grey"
    }
  ]
}

```

First, you would need to read the file contents and then return the data obtained to the client that made the GET request to your server.

We will make use of the [fs module](#) to read the file. How to do this is shown below.

```

const express = require('express')
const app = express()
const port = 3000
const fs = require('fs')

app.get('/', (req, res) => {
  res.end('Hello World!');
});

app.get("/list_movies", (req, res) => {
  fs.readFile(__dirname + '/' + 'movies.json', 'utf8', (err, data) => {
    res.json(data);
  });
});

app.listen(port, () => {
  console.log(`app listening at http://localhost:${port}`)
});

```

Next, start up the server by running `node index.js` .

Finally, opening `http://localhost:3000/list_movies` should return the same data found in the `movies.json` file, as expected.

Stop the server by pressing `CTRL + C` .

Congratulations!!! You have managed to build and locally serve a Node.js REST API built with Express!

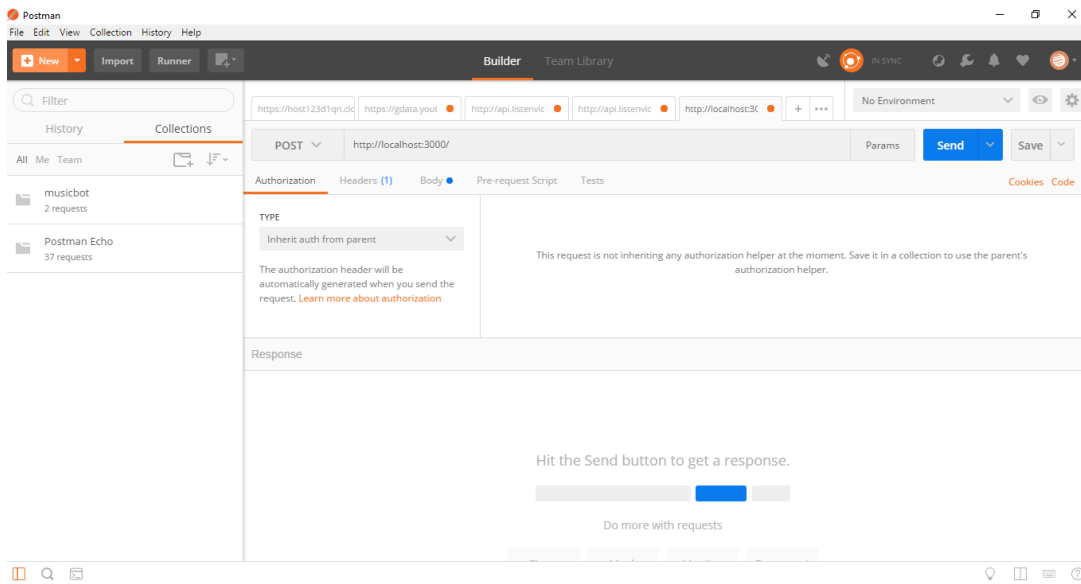
In the next step, you will learn how you can make API requests using POSTMAN.

Using Postman for testing API

In this step, you are going to use Postman to make test requests to your running server.

Postman is an API(application programming interface) development tool which helps to build, test and modify APIs. Almost any functionality that could be needed by any developer is encapsulated in this tool. It is used by over 5 million developers every month to make their API development easy and simple. It has the ability to make various types of HTTP requests(GET, POST, PUT, PATCH), saving environments for later use, converting the API code for various languages(like JavaScript, Python). **So, lets get started !! You can download Postman from [here](#).**

After downloading and installing the Postman, open the software.



Explaining the Interface

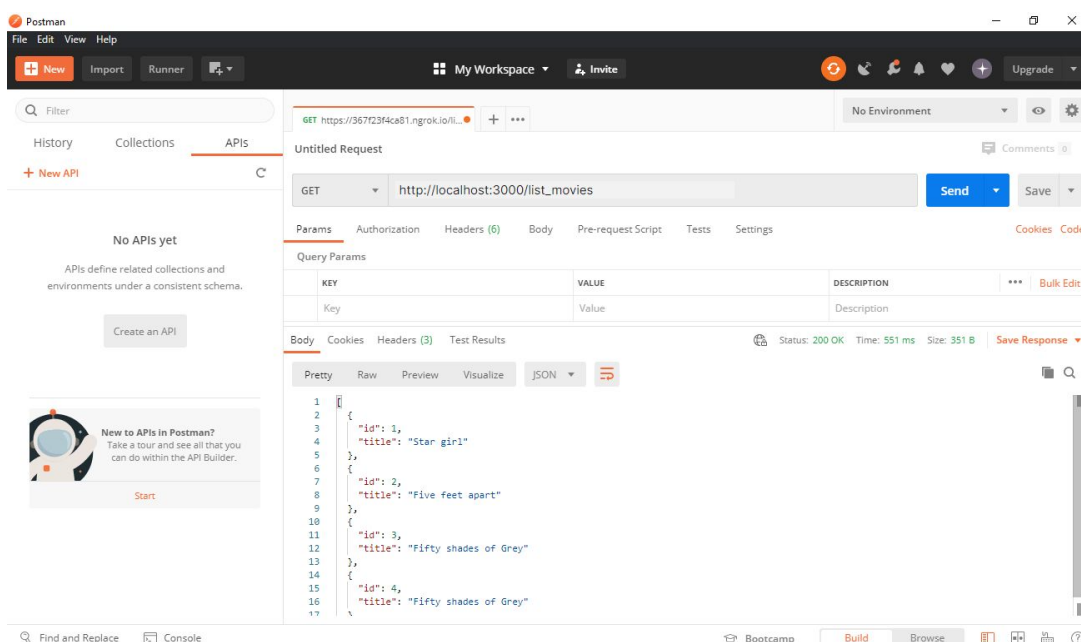
- The longest middle input field that looks something like a search bar is where the URL that we want to GET or POST or DELETE, etc. is fed.
- Just to the left of it, is a drop down button which has all the various HTTP methods as options. If you want to POST to the URL that you have specified, select POST.
- To the right of it is the params button. If you click on it, a new interface will appear. Params are basically the data that we want to send to the server with our request. We will use this params interface to **POST** to put app a new User.
- To the left of this button is the **Send** button which is used in sending the request to the server or the app in this case.

I think this much introduction to the interface is enough for us to get started. So, let's get started with sending and receiving requests through Postman.

Sending and receiving requests through Postman

- Enter the URL that you want to hit in the URL bar that is described above. We will put the url of api we just created above i.e. **http://localhost:3000/list_movies**

Let's select our HTTP method to send the request as GET in the left button. Now click on the Send button.



Conclusion

In this module we had an introduction to express and we saw how easy it is to create a web server using express. Further we saw how routing implemented in express and we also built a simple API to understand basics of express. In further modules we will learn about middlewares and error handling in express.

Interview Questions

What is routing and how routing works in Express.js?

Routing refers to determining how an application responds to a request.

Syntax:

```
app.METHOD(PATH, HANDLER);
```



Where:

- **app** is an instance of **express**.
- **METHOD** is an HTTP request method (get, post, put, etc.).
- **PATH** is a path/endpoint on the server.
- **HANDLER** is a function executed when the route is matched.

#Example: A route with path / and get method.

```
app.get('/', function (req, res) {  
  res.send('Express.js Interview Questions')  
})
```



How you can serve static files in Express.js?

Two possible ways:

```
app.use(express.static('public'))  
app.use('/static', express.static(path.join(__dirname, 'public')));
```



What are some Express.js key features

Following are the key features of express framework:

1. **Middlewares:** Express middleare are functions that access of - request (req), response (res) and next().
2. **Routing:** It is used to define routes in order to perform and handle HTTP requests/operations.
3. **Templates Engine:** It has SSR renders html template used to render the HTML on the browser.
4. **High Performance:** Express prepare a thin layer, therefore, the performance is adequate.
5. **Database Support:** Express supports RDBMS as well as NoSQL databases - MySQL, MongoDB, etc.
6. **MVC Support:** It is a web application which supports MVC architecture.
7. **ORM Support:** It support various ORM/ODM - Mongoose, Sequelize, etc.

Thank you !