# Introduction to RESTful Services

REST, which stands for Representational State Transfer.

RESTful services provide a scalable and flexible approach to building web applications.

RESTful services are a set of principles and constraints that guide the development of web services following the REST architecture.

**RESTful Operations:**

**1. CRUD Operations:**

Create (POST): Create a new resource.

Read (GET): Retrieve a representation of a resource.

Update (PUT/PATCH): Modify an existing resource.

Delete (DELETE): Remove a resource.

**2. HTTP Methods:**

HTTP methods (GET, POST, PUT, PATCH, DELETE) are used to perform CRUD operations on resources.

GET: GET https://api.example.com/users/123

POST: POST https://api.example.com/users

PUT: PUT https://api.example.com/users/123

DELETE: DELETE https://api.example.com/users/123

**3. Status Codes:**

HTTP status codes indicate the result of the server's attempt to process a request.

200 OK: Successful GET request.

201 Created: Successful POST request.

204 No Content: Successful DELETE request.

400 Bad Request: Invalid request from the client.

404 Not Found: Resource not found.

500 Internal Server Error: Server error.

# Introduction to Express JS

Express.js is a web application framework for Node.js, a server-side JavaScript runtime.

It is designed to simplify the process of building web applications and APIs by providing a set of features and tools that streamline common tasks.

Express.js is often described as fast, unopinionated, and minimalist, meaning it doesn't impose strict conventions, allowing developers flexibility in how they structure their applications.

**Features of Express JS**

- **Fast Server-Side Development**
  The features of node js help express saving a lot of time.
- **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.

**Advantages of Express.js**

- Makes Node.js web application development fast and easy.
- Easy to configure and customize.
- Allows you to define routes of your application based on HTTP methods and URLs.
- Includes various middleware modules which you can use to perform additional tasks on request and response.
- Allows you to define an error handling middleware.
- Easy to serve static files and resources of your application.
- Allows you to create REST API server.
- Easy to connect with databases such as MongoDB, MySQL

## Express JS Lifecycle

The Express.js lifecycle refers to the sequence of events that occur during the processing of an HTTP request in an Express application. Understanding the Express.js lifecycle is crucial for developers to comprehend how middleware functions, routes, and other components work together to handle incoming requests and produce responses.

Express.js Lifecycle involves seven stages

The initialization, middleware execution, routing, route handling, error handling, response generation, and termination together form a comprehensive sequence of events that enable developers to create robust and scalable applications using the Express framework.

**1. Initialization**:

At the beginning of your Express application, you typically set up essential configurations, middleware, and other settings. This includes creating an instance of the Express application:

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

**2. Middleware Execution:**

- Middleware functions play a central role in the Express.js lifecycle. They are functions that have access to the request (req), response (res), and the next middleware function in the application's request-response cycle.
- Middleware functions can perform various tasks such as logging, modifying the request or response objects, handling authentication, and more.
- They are executed in the order they are defined, and control is passed from one middleware to the next using the next function.

```
app.use((req, res, next) => {
  // Middleware logic
  next();
});
```

**3. Routing:**

- After middleware execution, the Express application processes routing. Routes define how the application responds to different HTTP methods and URIs.
- Routes are defined using methods like app.get(), app.post(), etc. Each route typically contains a callback function that is executed when a request matches the specified criteria.

```
app.get('/home', (req, res) => {
  // Route logic for handling GET requests to '/home'
});
```

**4. Route Handling:**

- When a request matches a defined route, the associated route handler is executed. This is where the main logic for processing the request and generating a response resides.
- Route handlers have access to the request and response objects and can send responses, render views, or perform other actions based on the client's request.

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  // Process the request and send a response
  res.send(`User with ID ${userId}`);
});
```

**5. Error Handling:**

- Express provides mechanisms for handling errors that may occur during the processing of a request. This includes errors thrown by middleware, route handlers, or other components.
- You can define error-handling middleware functions that take four arguments (err, req, res, next) to catch and handle errors.

```
app.use((err, req, res, next) => {
  // Handle errors
  res.status(500).send('Internal Server Error');
});
```

**6. Response:**

- Once the request has been processed, the server sends the response back to the client. This includes the HTTP status code, headers, and the body of the response.
- The response can be generated using methods like res.send(), res.json(), or by rendering views using a template engine.

```
app.get('/hello', (req, res) => {
  res.send('Hello, Express!');
});
```

**7. Termination:**

- After sending the response, the Express application has completed the request-response cycle. The server is still active and ready to handle new incoming requests.
- If there are any asynchronous operations (e.g., database queries) initiated during the request processing, they may continue in the background.

## Building First Web Server:

Getting Started:

**1.Installation:**

Express.js can be installed using npm (Node Package Manager), a package manager for Node.js. The installation command is npm install express.

```
npm install express
```

**2. Hello World Example:**

The basic "Hello World" example showcases the simplicity of setting up an Express application. It involves creating an instance of Express, defining a route for the root URI ('/'), and listening on a specific port (e.g., 3000) for incoming requests.

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

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

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

**3. Routing:**

Express simplifies the creation of routes using methods like app.get() and app.post(). Routes define how the server responds to different HTTP methods and URIs. For example, app.get('/about', ...) sets up a route for handling GET requests to the '/about' URI.

Access the application in a web browser at http://localhost:3000.

```
app.get('/about', (req, res) => {
  res.send('About Us');
});

app.post('/contact', (req, res) => {
  res.send('Contact Us');
});
```

**4. Middleware:**

Middleware functions can be added to the application using app.use(). These functions execute in the order they are added, providing a powerful mechanism for handling tasks like logging, authentication, or modifying the request and response objects.

```
// Example middleware
app.use((req, res, next) => {
  console.log('Middleware function');
  next();
});
```

## Nodemon

Nodemon is a tool that helps develop Node.js based applications by automatically restarting the node application when file changes in the directory are detected.

**Installing nodemon:**

```
npm install -g nodemon
```

To use Nodemon with your project's main file, such as index.js or app.js, you can run the following command in your terminal:

```
nodemon index.js
```

This will start Nodemon and monitor index.js for changes. You can replace index.js with the name of your project's main file.

This means that Nodemon is running and monitoring your project files for changes. If you make a change to your code, such as changing the response text in your server, Nodemon will automatically restart your server.

To stop Nodemon, you can press ctrl + c in your terminal. This will stop Nodemon and close your server.

Using Nodemon can save you a lot of time when developing your Node.js applications, as it eliminates the need to manually restart your server every time you make a change to your code.

## Using Environmental Variables:

Reading configuration parameters in Express.js is a common task that allows you to customize the behavior of your application based on various settings. Configuration parameters are often used to store sensitive information like API keys, database connection strings, and other environment-specific variables.

Environment Variables: Storing configuration parameters externally, often as environment variables, is a best practice. This ensures sensitive information is kept secure and allows for easy deployment across different environments (development, staging, production).

**1.Setting Up Configuration:**

Create a Configuration File:

Start by creating a configuration file (e.g., config.js or config.json) to store your parameters.

```
// config.js
module.exports = {
  port: process.env.PORT || 3000,
  databaseUrl:                process.env.DATABASE_URL                ||
'mongodb://localhost:27017/mydatabase',
  apiKey: process.env.API_KEY,
};
```

**2. Accessing configuration:**

In your Express.js application, require the configuration file at an appropriate place (e.g., at the beginning of your main application file).

```
const config = require('./config');
Using configuration paramaters
const port = config.port;
const databaseUrl = config.databaseUrl;
const apiKey = config.apiKey;
```

or

```
const { port, databaseUrl, apiKey } = config;
```

Example:

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

const config = require('./config');

const port = config.port;

app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

## Routing of Express App

- Routing is made from the word route. It is used to determine the specific behavior of an application.
- ***Routing*** refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

**app.METHOD(PATH, HANDLER) or app.METHOD(path, callback [, callback ...])**

**Where:**

**app** is an instance of express.

**METHOD** is an HTTP request method .METHOD is the HTTP method of the request, such as GET, PUT, POST, and so on, in lowercase. Thus, the actual methods are app.get(), app.post(), app.put(), and so on.

**app.get()** to handle GET requests

**app.post** to handle POST requests

**app.all()** to handle all HTTP methods

**app.use()** to specify middleware as the callback function

**PATH** is a path on the server.

**HANDLER** is the callback function executed when the route and method are matched. In fact, the routing methods can have more than one callback function as arguments. With multiple callback functions, it is important to provide next as an argument to the callback function and then call next() within the body of the function to hand off control to the next callback.

## HTTP Methods:

| | |
|---|---|
| GET | Read Data |
| POST | Insert Data |
| PUT or PATCH | Update Data |
| DELETE | Delete Data |

**Route Method:**

- A route method is derived from one of the HTTP methods, and is attached to an instance of the express class.
- The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

- GET and POST are Standard HTTP request used to create REST API's

- When sending or receiving a significant amount of data, POST and GET request are needed


## Handling HTTP GET Requests:

- ❖ GET method is used to request data from the server but mainly this method is used to read data.
- ❖ Get requests are to be cache and remain in browser history, and GET Method is not used because of sensitive data.
- ❖ To get the data, we add parameters in the URL and get data in response.
- ❖ The **Get** method is used to handle the get requests in Express. It takes two arguments, the first one is the route(i.e. path), and the second is the callback. Inside the callback, we send the response when a user requests that path.
- ❖ It is more efficient and popular than the POST Method.

Handling GET requests in Express is pretty straightforward. You have to use the instance of Express to call the method **Get**.

**Server.js**

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

## Handling HTTP POST Requests:

- Post Method is used to create new or to edit already existing data.
- This method allows you to communicate massive amounts of data.
- The Post method is secure because the data is not visible(Even if you can't bookmark) in the URL bar it is not as widely utilized as the GET method. Basically we use the GET method mostly but sensitive data we use the POST Method.
- The data sent through the POST request is stored in the request object.

**Server.js**

```
const express = require('express')
const app = express()
//POST Method Requires POSTMAN/TUNDER
app.post('/request', (req, res) => {
```

```
  res.send('POST Request!')
})
app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

## Calling Endpoints using Postman/Thunder client:

- **Thunder client:** Thunder Client is an alternative to the famous Postman tool used for testing client APIs. The Thunder Client VS Code extension is lightweight and lets you test APIs on the fly within the editor.
- Install From Marketplace Extension

## Handling HTTP PUT Requests:

- The app.put() function routes the HTTP PUT requests to the specified path with the specified callback functions.
- It helps to Update the data.

**Syntax**:
app.put(path, callback [, callback ...])

**Arguments:**

**Path:** The path for which the middleware function is invoked and can be any of the:

- A string represents a path.
- A path pattern.
- A regular expression pattern to match paths.
- An array of combinations of any of the above.

**Callback:** Callback functions can be:

- A middleware function.
- A series of middleware functions (separated by commas).
- An array of middleware functions.
- A combination of all of the above.

**Code.js**

```
const express = require('express');
const app = express();
const PORT = 3000;
 app.put('/', (req, res) => {
    res.send("PUT Request Called")
})
 app.listen(PORT, function (err) {
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

## Handling HTTP Delete Requests

- The app.delete() method routes all the HTTP DELETE requests to the specified path with the specified callback functions.

**Syntax:**
**app.delete(path, callback, [callback])**

**Parameters:**

**path –** This is the path for which the middleware function is invoked. A path can be a string, path pattern, a regular expression, or an array of all these.
**callback** − These are the middleware functions or a series of middleware functions that act like a middleware except that these callbacks can invoke next (route).

## // Execution Flow:

**Create a Folder in VS code (Express_code)**
**$ npm init**
**Creates package.json & index.js files in the Folder**
**$ npm install express –save**
**It creates node_modules folder**
**$ npm install -D nodemon**
**If any changes in data automatically saves no need to close & run**

**Paste Below code & run the server using node index.js**

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

**Paste Below code & run the server using node index.js**

```
const express = require('express')
const app = express()
//POST Method Requires POSTMAN/THUNDER
app.post('/request', (req, res) => {
  res.send('POST Request!')
})
app.listen(3000, () => {
  console.log('Example app listening on port 3000!')
})
```

## Build an API

**Create a Basic Application for Student data using Routing Handler Methods**

1. Create a Folder Name called **Application**
2. Change the Directory using command **cd Application**
3. Initialize the init Method using **npm init - -yes (Default values its creates)**
4. **T**o Open VS code, the use command **code .**
5. Install Express Plugin using command **npm install express (Req internet)**
6. Create a File called **index.js**
7. Use nodemon for any changes automatically it will updates use command **npm install nodemon -g**

Install Chrome Extension: **JSON Formatter (In Browser to see JSON Formate )**

Fake data Creation in JSON Format: **mockaroo.com**

**students.js** **//data is taken from mockaroo**.com

```
const express = require('express');
students=[{
  "id": 1,
  "first_name": "Alejandra",
  "last_name": "Byatt",
  "email": "abyatt0@geocities.com"
}, {
  "id": 2,
  "first_name": "Brigitta",
  "last_name": "Jindracek",
  "email": "bjindracek1@g.co"
}, {
  "id": 3,
  "first_name": "Steffi",
  "last_name": "Gonsalo",
  "email": "sgonsalo2@oracle.com"
}, {
  "id": 4,
  "first_name": "Emanuele",
  "last_name": "Starbeck",
  "email": "estarbeck3@alexa.com"
}, {
  "id": 5,
  "first_name": "Trenton",
  "last_name": "Johananoff",
  "email": "tjohananoff4@cnn.com"
}, {
  "id": 6,
  "first_name": "Ariella",
  "last_name": "Champain",
  "email": "achampain5@about.com"
}, {
  "id": 7,
  "first_name": "Florri",
  "last_name": "Mead",
  "email": "fmead6@ask.com"
}, {
  "id": 8,
```

```
  "first_name": "Tamiko",
  "last_name": "Noakes",
  "email": "tnoakes7@t-online.de"
}, {
  "id": 9,
  "first_name": "Tye",
  "last_name": "Cowper",
  "email": "tcowper8@ycombinator.com"
}, {
  "id": 10,
  "first_name": "Bonita",
  "last_name": "Karpets",
  "email": "bkarpets9@theguardian.com"
}]
module.exports =students  //export student data;
```

**Index.js**

```
const express = require('express');
const students = require('./students')

const app=express();
app.use(express.json())

app.get('/api',(req,res)=>{
  res.send({message: "API is Working"});
})

app.get('/api/students',(req,res)=>{
  res.json(students);
})
//POST Method Requires POSTMAN/TUNDER
app.post('/api/test',(req, res)=>{
  res.json("POST Request")
})

//Push the Data in JSON format in Tunder Client like
/*
{
  "first_name": "John",
  "last_name": "Doe",
  "email": "johndoe@kmit.in"
}
*/
app.post('/api/students',(req, res)=>{
  if(!req.body.email){                  //email is mandatory
    res.status(400)
    res.json({ error: "email is Required..."})
  }
  const user ={
    id:students.length +1,
    first_name: req.body.first_name,
    last_name: req.body.last_name,
    email : req.body.email
  }
  students.push(user);
  res.json(user);
})

//PUT Data used for Update the Value:
```

```
app.put('/api/students/:id', (req,res)=>{
  let id=req.params.id;
    let first_name= req.body.first_name
    let last_name= req.body.last_name
    let email = req.body.email

    let index =students.findIndex((students)=> {
      return (students.id== Number.parseInt(id))
    })
    if(index >=0){
      const stu =students[index]
      stu.first_name =first_name
      stu.last_name =last_name
      stu.email =email
    }else{
      res.status(404)
      res.end()
    }
  console.log(id);
  res.json(id);
})

app.delete("/api/students/:id", (req,res)=>{
  let id=req.params.id;
  const index =students.findIndex((students)=> {
    return (students.id== Number.parseInt(id))
  })
  if(index >=0){
    let stu =students[index]
    students.splice(index, 1) //1 referece 1 object d elete
    res.json(stu)
  }else{
    res.status(404)
  }

})

const port=1234;
app.listen(port,()=> {
  console.log('App Running in port: ${port}' );
});
```

# Response Methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

| Method | Description |
|---|---|
| **res.download()** | Prompt a file to be downloaded. |
| **res.end()** | End the response process. |
| **res.json()** | Send a JSON response. |
| **res.jsonp()** | Send a JSON response with JSONP support. |
| **res.redirect()** | Redirect a request. |
| **res.render()** | Render a view template. |
| **res.send()** | Send a response of various types. |
| **res.sendFile()** | Send a file as an octet stream. |
| **res.sendStatus()** | Set the response status code and send its string representation as the response body. |

## Methods:

```
//exploring response methods
const express=require("express");
const app=express();
const PORT=3000;
app.get("/",(req,res)=>{
    res.type( 'text/plain');
    res.send("Response Methods");
});
app.get("/status",(req,res)=>{
        res.status(404);
        res.send('Status');
});
//seting the name and value through set method
app.get("/set",(re,res)=>{
    res.set("WT","Web Technologies");
    res.send("Setter method");
});
//set a cookie from the server side
app.get("/cookie",(req,res)=>{
    res.cookie("cook","first",[]);
    res.clearCookie("cook");
    res.send("Cookie from the server side");
});
//Redirect to the browser to a URL
app.get( '/redirect', ( req, res ) => {
    res.redirect( 201, '/' )
    });
//Sends a JSON format to the client
app.get("/json",(re,res)=>{
    res.json({
        page:"WT",
        comment: "Thanks for choosing WT",
    })
});
//Terminate the conection without sending something to the client
app.listen(PORT,()=>{
    console.log(`Listening on ${PORT}`);
})
```

```
/*
res.download()
res.end()
res.render()
res.sendFile()
res.sendStatus()

*/
```

## Route Handlers

Multiple callback functions that behave like middleware to handle a request. The only exception is that these callbacks might invoke next('route') to bypass the remaining route callbacks. You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

Route handlers can be in the form of a function, an array of functions, or combinations of both.

### Route Handler with function

```
const express=require("express");
const app=express();
const PORT=3000;
//A route can be handled using a single callback function
app.get("/users/a",function(req,res){
    res.send("Message from A");
});
//A route can be handled using a more than one callback function
app.get("/users/b",function(req,res,next){
    console.log("Response will be sent by the next function...");
    next();
}, function(req,res){
    res.send("Message from B");
});
//A route can be handled using an array of callback functions
const b0=function(req,res,next){
    console.log("From b0");}
const b1=function(req,res,next){
    console.log("From b1");}
const b2=function(req,res,next){
    console.log("From b2");}
app.get("/users/c",[b0,b1,b2]);
app.listen(PORT,function(){
    console.log(`Listening on ${PORT}`)
});
```

### Route Handler with arrow function

```
const express=require("express");
const app=express();
const PORT=3000;
//A route can be handled using a single callback function
app.get("/users/a",(req,res)=>{
    res.send("Message from A");
});
//A route can be handled using a more than one callback function
app.get("/users/b",(req,res,next)=>{
    console.log("Response will be sent by the next function...");
    next();
```

```
}, (req,res)=>{
    res.send("Message from B");
});
//A route can be handled using an array of callback functions
const b0=(req,res,next)=>{
    console.log("From b0");}
const b1=(req,res,next)=>{
    console.log("From b1");}
const b2=(req,res,next)=>{
    console.log("From b2");}
app.get("/users/c",[b0,b1,b2]);
app.listen(PORT,()=>{
    console.log(`Listening on ${PORT}`)
});
```

**A combination of independent functions and arrays of functions can handle a route.**

```
const cb0 = function (req, res, next) {
    console.log('CB0')
    next();  }
  const cb1 = function (req, res, next) {
    console.log('CB1')
    next(); }
  app.get('/users/d', [cb0, cb1], (req, res, next) => {
    console.log('the response will be sent by the next function ...')
    next()
  }, (req, res) => {
    res.send('Message from D!')
  })
app.listen(PORT,()=>{
    console.log(`Listening on ${PORT}`)
});
```

**For more detailed references please visit below URLs:**
**https://nodejs.dev/en/learn**
**https://expressjs.com/en/starter/basic-routing.html**