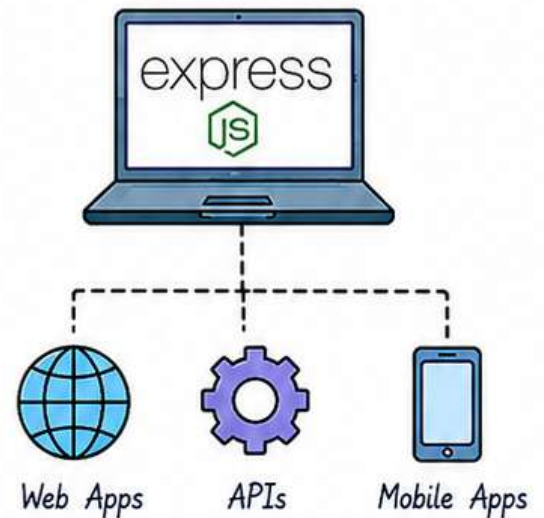


# Chapter 13: Express.js

## 13.1 Introduction to Express.js

### 13.1.1 What is Express.js?

- \* Express.js is a **fast, unopinionated, minimalist** web application framework for **Node.js**.
- \* It provides a thin layer of fundamental web application features.
- \* It is designed to build **web applications** and **APIs** quickly and efficiently.
- \* Express simplifies the process of handling **HTTP requests, routing, middleware** and **responses**.



### 13.1.2 Why Use Express.js?

- \* Simplifies **routing** and handling of **HTTP requests**.
- \* **Middleware** support makes it very flexible.
- \* Large **ecosystem** with many third-party packages.
- \* Easy to integrate with **databases, templating engines** and other tools.
- \* Ideal for building **RESTful APIs** and **full-stack applications**.



### 13.1.3 Features of Express.js



**Routing** - Easy and flexible routing to handle requests.



**Middleware** - Powerful support for using middleware functions.



**Template Engines** - Supports various template engines like EJS, Pug, etc.



**Database Integration** - Easy integration with MongoDB, MySQL and more.



**Rich Ecosystem** - Huge number of third-party packages available via npm.



**High Performance** - Built on top of Node.js, so it is fast and efficient.

### Key Points:

- ✓ Express.js is a web framework for Node.js.
- ✓ It helps in building web applications and APIs quickly.
- ✓ It provides robust features like routing, middleware and template engine support.
- ✓ Express is minimal yet powerful and widely used in modern web development.



### Key Takeaway:



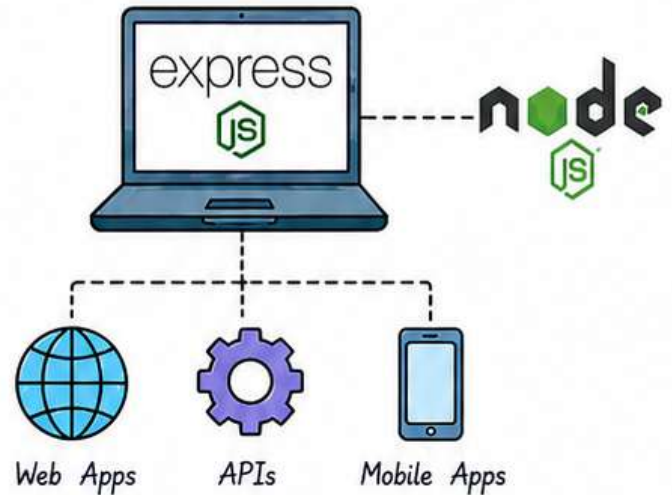
Express.js makes Node.js development **easier** and **faster** by providing a simple and flexible way to build **web applications** and **APIs**.



## 13.2 Express Installation

### 13.2.1 What is Express?

- \* **Express.js** is a fast, minimal and flexible **Node.js** web application framework.
- \* It provides a robust set of features for web and mobile applications.
- \* It makes it easy to build APIs and web applications.
- \* Express is **not a part of Node.js**, so we need to install it separately.



### 13.2.2 Prerequisites

- \* **Node.js** and **npm** (Node Package Manager) must be installed.
- \* You can check by running the following commands.

```
$ node -v
```

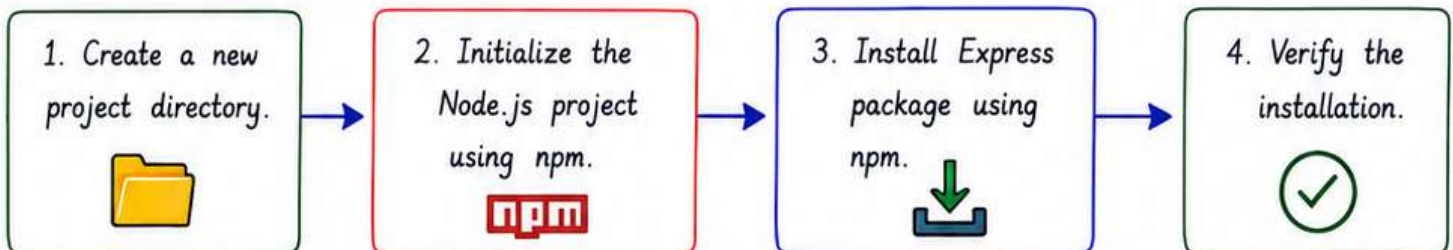
```
v20.11.1
```

```
$ npm -v
```

```
10.2.4
```



### 13.2.3 Steps to Install Express



Commands:

- |                         |                              |
|-------------------------|------------------------------|
| 1. mkdir my-express-app | # Create project directory   |
| 2. cd my-express-app    | # Move into the directory    |
| 3. npm init -y          | # Initialize Node.js project |
| 4. npm install express  | # Install Express package    |
| 5. npm list express     | # Verify installation        |



#### Key Points:

- ✓ Express is a third-party package and must be installed.
- ✓ npm is used to install and manage packages in Node.js.
- ✓ After installation, Express is available in the **node\_modules** folder.
- ✓ We can now use Express in our Node.js application.



#### Key Takeaway:



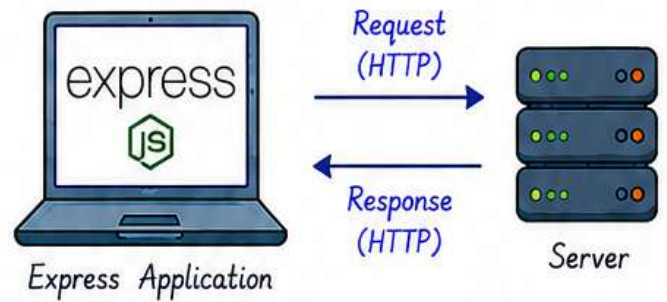
Installing Express is **quick and easy** using npm.  
Once installed, we can start building **powerful web applications** and **APIs**.



## 13.3 Express Application

### 13.3.1 What is an Express Application?

- \* An Express application is a **Node.js** application built using the **Express** framework.
- \* It provides a set of methods and features to handle **HTTP requests** and **responses**.
- \* It is used to build **web applications**, **RESTful APIs** and **backend services**.
- \* An Express app listens on a **port** and responds to client requests.



### 13.3.2 Creating a Simple Express Application

- \* Follow the steps below to create a basic Express application.

1. Import Express module.

2. Create an Express app.

3. Define a route for the home page.

4. Start the server and listen on a port.

Example:

```
1. const express = require('express');
2. const app = express();
3.
4. // Define a route
5. app.get('/', (req, res) => {
6.     res.send('Hello, Express!');
7. });
8.
9. // Start the server
10. const PORT = 3000;
11. app.listen(PORT, () => {
12.     console.log(`Server is running on http://localhost:${PORT}`);
13. });
```

How it Works?

Client makes a request to the server.

Express app matches the request to a route.

Handler function processes the request.

Response is sent back to the client.



Output in Terminal:

```
PS C:\my-express-app> node app.js
Server is running on http://localhost:3000
```



Open in Browser  
<http://localhost:3000>

Key Points:

- ✓ Express application is built using a few simple steps.
- ✓ Routing is used to handle different endpoints.
- ✓ `app.listen()` starts the server and makes the application accessible.
- ✓ We can create multiple routes for different functionalities.



Key Takeaway:



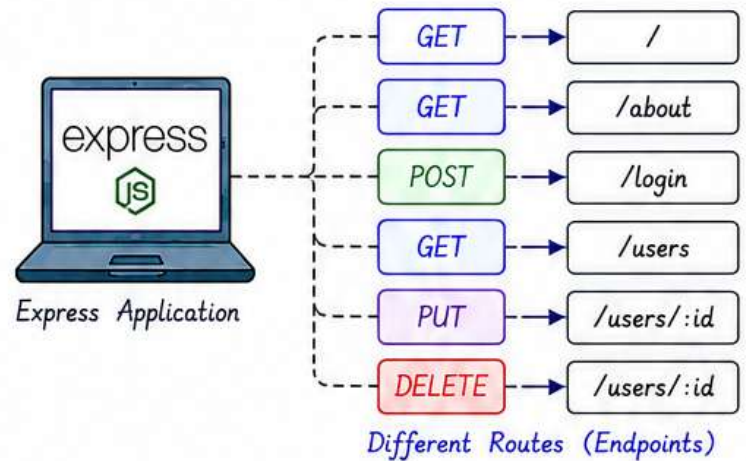
Using Express, we can quickly **create a web server**, define routes, and send responses to build **web applications** and **APIs** efficiently.



## 13.4 Routing

### 13.4.1 What is Routing?

- \* Routing refers to how an application's **endpoints** (URLs) respond to client requests.
- \* In **Express.js**, routes are defined using **HTTP methods** and **path patterns**.
- \* Each route can have a **handler function** that executes when the route is matched.
- \* Routing helps in organizing the application logic into different **endpoints**.



### 13.4.2 HTTP Methods in Routing

- \* Express.js supports all standard HTTP methods:

| Method | Description                                           |
|--------|-------------------------------------------------------|
| GET    | Used to retrieve data from the server.                |
| POST   | Used to send data to the server to create a resource. |
| PUT    | Used to update an existing resource completely.       |
| PATCH  | Used to update an existing resource partially.        |
| DELETE | Used to delete a resource from the server.            |

### 13.4.3 Route Path Patterns

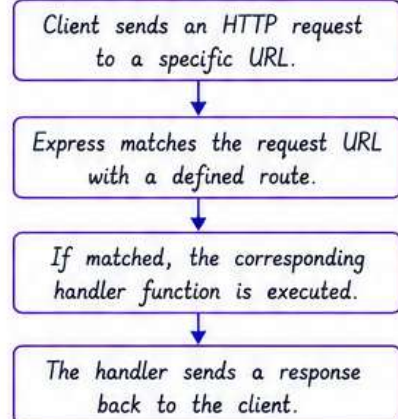
- \* Express supports dynamic routes using **parameters**.

| Pattern          | Example          | Description                |
|------------------|------------------|----------------------------|
| /                | /                | Root route                 |
| /about           | /about           | Static route               |
| /users/:id       | /users/101       | Route with parameter       |
| /users/:id/posts | /users/101/posts | Nested route               |
| /search/:query   | /search/nodejs   | Route with query parameter |

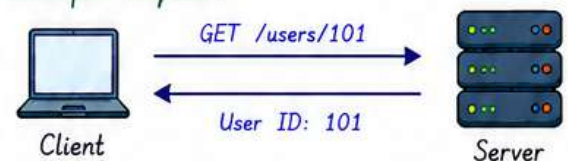
### 13.4.4 Routing Examples

```
1. const express = require('express');
2. const app = express();
3.
4. // Home Route
5. app.get('/', (req, res) => {
6.   res.send('Welcome to Express Routing!');
7. });
8.
9. // About Route
10. app.get('/about', (req, res) => {
11.   res.send('This is the about page.');
```

#### How Routing Works?



#### Example Request:



#### Key Points:

- ✓ Routing maps URLs and HTTP methods to handler functions.
- ✓ Express makes routing simple, flexible and powerful.
- ✓ We can create static routes, dynamic routes and nested routes.
- ✓ Routing is the backbone of any web application.

#### Key Takeaway:



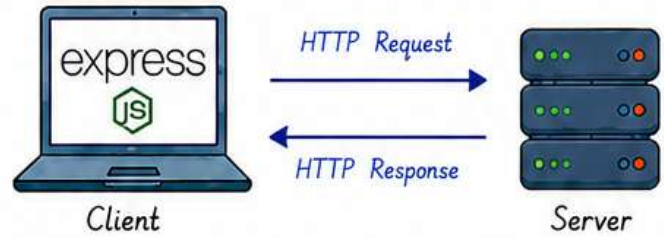
Routing in Express helps us handle different requests efficiently and build well-structured web applications and APIs.



# 13.5 HTTP Methods

## 13.5.1 What are HTTP Methods?

- \* HTTP methods are used to indicate the **action** to be performed on a resource.
- \* They are an essential part of **RESTful APIs**.
- \* Each method has a **specific purpose**.



## 13.5.2 Common HTTP Methods

| Method | Description                                           | Purpose                     | Request with Body | Idempotent | Safe |
|--------|-------------------------------------------------------|-----------------------------|-------------------|------------|------|
| GET    | Used to retrieve data from the server.                | Read a resource             | No                | Yes        | Yes  |
| POST   | Used to send data to the server to create a resource. | Create a resource           | Yes               | No         | No   |
| PUT    | Used to update an existing resource completely.       | Update a resource (replace) | Yes               | Yes        | No   |
| PATCH  | Used to update an existing resource partially.        | Update a resource (partial) | Yes               | No         | No   |
| DELETE | Used to delete a resource from the server.            | Delete a resource           | No                | Yes        | No   |

## 13.5.3 HTTP Methods with Examples

| Method | Example Request   | Example URL | Description                         |
|--------|-------------------|-------------|-------------------------------------|
| GET    | GET /users        | /users      | Retrieve the list of all users.     |
| POST   | POST /users       | /users      | Create a new user.                  |
| PUT    | PUT /users/101    | /users/101  | Update user with ID 101 completely. |
| PATCH  | PATCH /users/101  | /users/101  | Update user with ID 101 partially.  |
| DELETE | DELETE /users/101 | /users/101  | Delete user with ID 101.            |

## 13.5.4 Example: Using Different HTTP Methods in Express

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/users', (req, res) => res.send('Get all users'));
5. app.post('/users', (req, res) => res.send('Create a user'));
6. app.put('/users/:id', (req, res) => res.send('Update user'));
7. app.patch('/users/:id', (req, res) => res.send('Partial update'));
8. app.delete('/users/:id', (req, res) => res.send('Delete user'));
9.
10. app.listen(3000, () => console.log('Server running on port 3000'));
11.
```

### Flow of a Request

Client sends an HTTP request using a method (GET, POST, etc.).

Express matches the request to the correct route.

The corresponding handler function is executed.

A response is sent back to the client.



### Key Points:

- ✓ HTTP methods define the action to be performed on a resource.
- ✓ GET is used for reading data, POST for creating data.
- ✓ PUT updates completely, PATCH updates partially, DELETE removes a resource.
- ✓ Using the correct HTTP method makes APIs more meaningful and efficient.



### Key Takeaway:



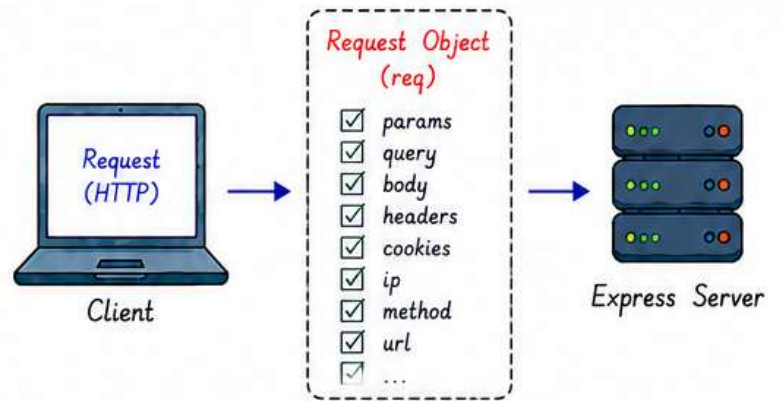
HTTP methods are the building blocks of **RESTful APIs**. They help us perform **Create**, **Read**, **Update** and **Delete** operations on resources.



# 13.6 Request Object

## 13.6.1 What is the Request Object?

- When a client makes a request, Express creates a **request object (req)**.
- This object contains information about the request, such as **parameters, body, headers, query**, etc.
- It is passed to the route handler function.
- It helps us **access and process** the data sent by the client.



## 13.6.2 Properties of Request Object

| Property                     | Description                               |
|------------------------------|-------------------------------------------|
| <code>req.params</code>      | Holds route parameters (from URL).        |
| <code>req.query</code>       | Holds query string parameters (from URL). |
| <code>req.body</code>        | Holds data sent in the request body.      |
| <code>req.headers</code>     | Holds HTTP headers sent with the request. |
| <code>req.cookies</code>     | Holds cookies sent by the client.         |
| <code>req.ip</code>          | Holds the IP address of the client.       |
| <code>req.method</code>      | Holds the HTTP method (GET, POST, etc.).  |
| <code>req.url</code>         | Holds the URL of the request.             |
| <code>req.originalUrl</code> | Holds the original URL of the request.    |
| <code>req.protocol</code>    | Holds the protocol used (http or https).  |
| ... (and more)               | Many other properties are available.      |

## 13.6.3 Example: Accessing Request Object

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/user/:id', (req, res) => {
5.   console.log('Route Parameter (id):', req.params.id);
6.   console.log('Query Parameter:', req.query);
7.   console.log('Request Body:', req.body);
8.   console.log('Headers:', req.headers);
9.   console.log('Client IP:', req.ip);
10.  console.log('Method:', req.method);
11.  console.log('URL:', req.url);
12.  res.send('Check console for request object data');
13. });
14.
15. app.listen(3000, () => console.log('Server running on port 3000'));
```

## 13.6.4 Example Request

Request URL:

GET http://localhost:3000/user/101?name=John&role=admin

Request Body (if POST):

```
{
  "name": "John",
  "email": "john@example.com"
}
```

Some Headers:

```
Content-Type: application/json
User-Agent: PostmanRuntime/7.x
Accept: */*
```

## 13.6.5 Accessing Request Data

| Access Using                           | Example            | Output             |
|----------------------------------------|--------------------|--------------------|
| <code>req.params.id</code>             | /user/101          | 101                |
| <code>req.query.name</code>            | ?name=John         | John               |
| <code>req.query.role</code>            | ?role=admin        | admin              |
| <code>req.body.name</code>             | { "name": "John" } | John               |
| <code>req.headers['user-agent']</code> | Header             | PostmanRuntime/7.x |
| <code>req.method</code>                | GET /user/101      | GET                |
| <code>req.ip</code>                    | Client IP          | 192.168.1.5        |

### Key Points:

- The request object (**req**) contains details about the incoming request.
- It helps access parameters, query strings, body data, headers and more.
- Different properties of **req** are used to handle client data effectively.
- Understanding the request object is key to building dynamic applications.



### Key Takeaway:



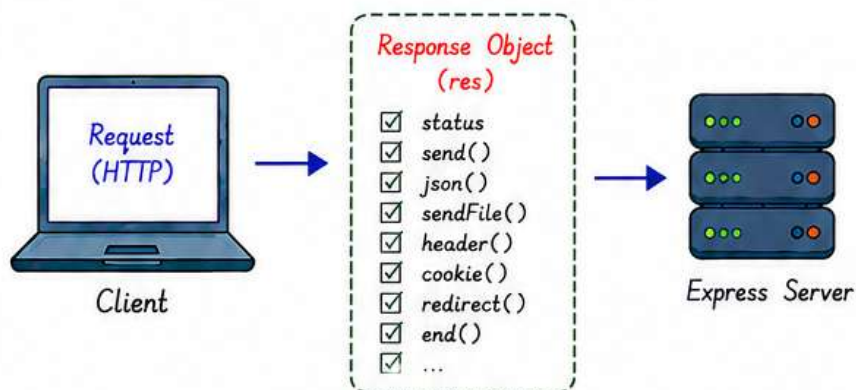
The **request object (req)** provides all the information about the client request. It is used to read and process data, making our APIs and web applications **dynamic and powerful**.



# 13.7 Response Object

## 13.7.1 What is the Response Object?

- \* The **response object (res)** is used by Express to send a **response** back to the client.
- \* It contains methods and properties that help send data, set **headers**, **status codes**, **cookies**, and more.
- \* It is passed to the route handler function along with the **request object**.
- \* It helps in completing the **request-response cycle**.



## 13.7.2 Properties of Response Object

| Property                       | Description                                            |
|--------------------------------|--------------------------------------------------------|
| <code>res.app</code>           | Reference to the Express application instance.         |
| <code>res.headersSent</code>   | Boolean that indicates if the headers have been sent.  |
| <code>res.locals</code>        | Object used to store local variables.                  |
| <code>res.statusCode</code>    | Holds the status code that will be sent to the client. |
| <code>res.statusMessage</code> | Holds the status message (e.g., OK, Not Found).        |
| <code>res.writableEnded</code> | Boolean that indicates if the response has been ended. |
| ... (and more)                 | Many other properties are available.                   |

## 13.7.3 Example: Using Response Object

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/', (req, res) => {
5.   res.status(200); // Set status code
6.   res.set('Content-Type', 'text/plain'); // Set header
7.   res.send('Hello! This is a response from Express.');
```

8. });

```
9.
10. app.listen(3000, () => {
11.   console.log('Server is running on http://localhost:3000');
```

12. });

14.  Set Status     Set Header     Send Data     Response Sent

## 13.7.4 Common Methods of Response Object

| Method                                | Description                            |
|---------------------------------------|----------------------------------------|
| <code>res.status(code)</code>         | Sets the HTTP status code.             |
| <code>res.send(data)</code>           | Sends the response data to the client. |
| <code>res.json(data)</code>           | Sends a JSON response.                 |
| <code>res.sendFile(path)</code>       | Sends a file as a response.            |
| <code>res.set(field, value)</code>    | Sets a response header field.          |
| <code>res.header(field, value)</code> | Alias for <code>res.set()</code> .     |
| <code>res.redirect(url)</code>        | Redirects the client to another URL.   |
| <code>res.cookie(name, value)</code>  | Sends a cookie to the client.          |
| <code>res.end()</code>                | Ends the response process.             |

## 13.7.5 Example: More Methods

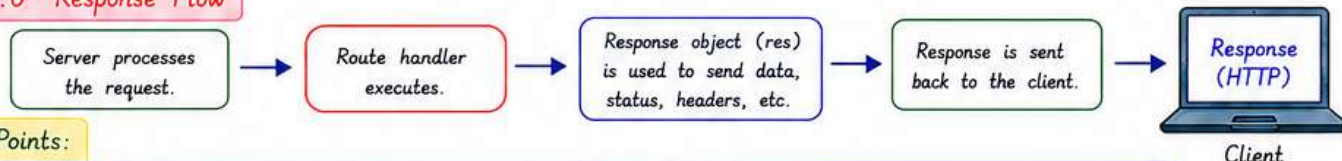
```
1. app.get('/data', (req, res) => {
2.   res.status(201).json({
3.     message: 'Data fetched successfully',
4.     status: 'success',
5.     data: [1, 2, 3, 4, 5]
6.   });
7. });
8. app.get('/file', (req, res) => {
9.   res.sendFile(__dirname + '/sample.txt');
```

10. });

12.



## 13.7.6 Response Flow



### Key Points:

- ✓ The response object (**res**) is used to send a response back to the client.
- ✓ It provides many methods to set status codes, headers, send data, files, cookies and more.
- ✓ Methods like `send()`, `json()`, `status()`, and `redirect()` are frequently used in web development.
- ✓ A proper response ensures smooth communication between the server and the client.



### Key Takeaway:



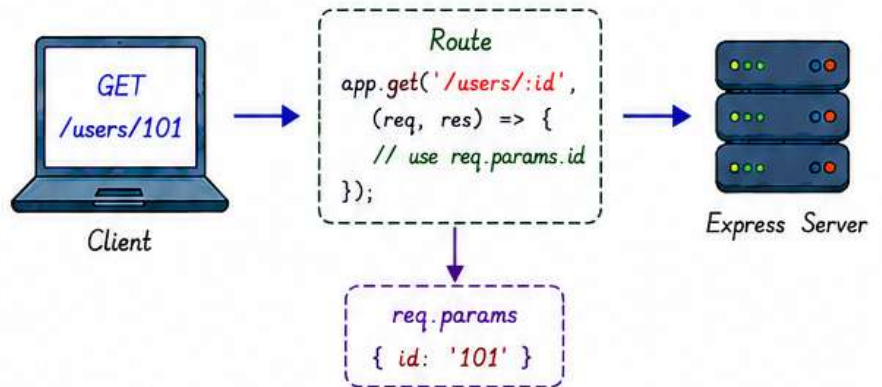
The response object (**res**) is essential for sending **meaningful** and **structured** responses to clients in Express applications.



# 13.8 Route Parameters

## 13.8.1 What are Route Parameters?

- \* Route parameters are **named segments** in the **URL**.
- \* They allow us to **capture values** from the **URL** and use them in our application.
- \* In Express.js, route parameters are defined using a **colon (:) before the parameter name**.
- \* They are accessed using **req.params**.



## 13.8.2 Syntax

```
app.METHOD('/path/:paramName', (req, res) => {
  // access using req.params.paramName
});
```

## 13.8.3 Examples of Route Parameters

| Route                    | Example URL          | req.params                        | Description                         |
|--------------------------|----------------------|-----------------------------------|-------------------------------------|
| /users/:id               | /users/101           | { id: '101' }                     | Captures user ID                    |
| /products/:pid           | /products/56         | { pid: '56' }                     | Captures product ID                 |
| /posts/:id/comments/:cid | /posts/10/comments/5 | { id: '10',<br>cid: '5' }         | Captures multiple parameters        |
| /files/:name.:ext        | /files/report.pdf    | { name: 'report',<br>ext: 'pdf' } | Captures parameters with extensions |

## 13.8.5 Multiple Route Parameters Example

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/posts/:id/comments/:cid', (req, res) => {
5.   const postId = req.params.id;
6.   const commentId = req.params.cid;
7.   res.send('Post ID: ${postId}, Comment ID: ${commentId}');
8. });
9.
10. app.listen(3000, () => {
11.   console.log('Server is running on http://localhost:3000');
12. });
```

## 13.8.6 Parameter Rules

| Rule                                               | Description                                      |
|----------------------------------------------------|--------------------------------------------------|
| ✓ Must start with a colon (:)                      | :id is a valid parameter, id is not.             |
| ✓ Can contain letters, numbers, and underscores    | :user_id, :post1 are valid.                      |
| ✓ Case sensitive                                   | :UserId and :userid are different.               |
| ✓ Can be used multiple times in a route            | /a/:x/b/:y is valid.                             |
| ✓ Optional parameters are not supported by default | Use <b>query parameters</b> for optional values. |

### Key Points:

- ✓ Route parameters are used to capture values from the URL.
- ✓ They are defined using a colon (:) before the parameter name.
- ✓ Access them using **req.params** in the route handler.
- ✓ Useful for building dynamic routes like **/users/:id**, **/posts/:id/comments/:cid**.



### Key Takeaway:



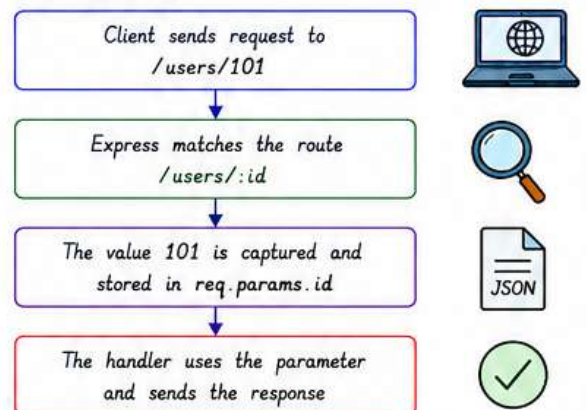
Route parameters make our Express applications **dynamic** by allowing us to capture and use values from the URL, enabling powerful and **flexible routing**.



## 13.8.4 Example: Using Route Parameters

```
1. const express = require('express');
2. const app = express();
3.
4. // Get user by ID
5. app.get('/users/:id', (req, res) => {
6.   const id = req.params.id; // Access parameter
7.   res.send('User ID is: ' + id);
8. });
9.
10. app.listen(3000, () => {
11.   console.log('Server is running on http://localhost:3000');
12. });
```

### How it Works?



### Note:

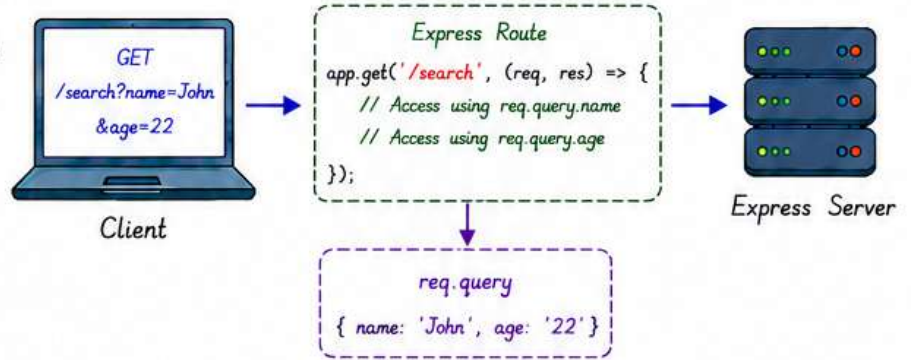
Route parameters help us build dynamic and flexible routes. They are powerful in creating RESTful APIs and handling dynamic data.



# 13.9 Query Parameters

## 13.9.1 What are Query Parameters?

- \* Query parameters are **key-value pairs** appended to the **URL** after a question mark (?).
- \* They are used to send **non-sensitive data** to the server.
- \* In Express.js, query parameters are accessed using **req.query**.
- \* Multiple query parameters are separated by an ampersand (&).



## 13.9.2 Syntax

```
app.METHOD('/path', (req, res) => {
  // Access query parameters using req.query
  const value = req.query.keyName;
});
```

## 13.9.3 Examples of Query Parameters

| URL                                                           | req.query                                                  | Description                                     |
|---------------------------------------------------------------|------------------------------------------------------------|-------------------------------------------------|
| <code>/search?name=John&amp;age=22</code>                     | <code>{ name: 'John', age: '22' }</code>                   | Two query parameters: name and age              |
| <code>/products?category=books&amp;sort=asc&amp;page=2</code> | <code>{ category: 'books', sort: 'asc', page: '2' }</code> | Three query parameters: category, sort and page |
| <code>/user?active=true</code>                                | <code>{ active: 'true' }</code>                            | One query parameter: active                     |
| <code>/search</code>                                          | <code>{}</code> (empty object)                             | No query parameters                             |

## 13.9.5 Multiple Query Parameters Example

```
1. app.get('/products', (req, res) => {
2.   const category = req.query.category;
3.   const sort = req.query.sort;
4.   const page = req.query.page;
5.
6.   res.send('Category: ${category}, Sort: ${sort}, Page: ${page}');
7. });
```

## 13.9.6 Query Parameter Rules

| Rule                    | Description                                      |
|-------------------------|--------------------------------------------------|
| ✓ Start with ?          | Query string starts after ? in the URL.          |
| ✓ Key=Value format      | Each parameter is in key=value format.           |
| ✓ Multiple parameters   | Use & to separate multiple key=value pairs.      |
| ✓ Values are strings    | All values are received as strings in req.query. |
| ✓ Order does not matter | Parameters can be in any order.                  |
| ✓ Optional              | Query parameters are optional.                   |

### Key Points:

- ✓ Query parameters are used to pass data in the URL.
- ✓ Access them in Express using **req.query**.
- ✓ Multiple query parameters are separated by &.
- ✓ Useful for filtering, searching, pagination, sorting, etc.



### Key Takeaway:



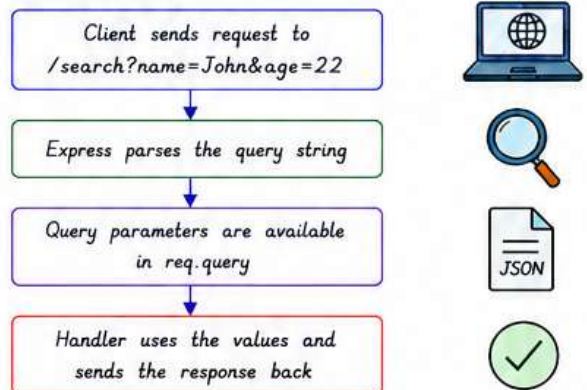
Query parameters help us send additional data with the request through the URL. They make our APIs **flexible**, **dynamic** and **easy to use**.



## 13.9.4 Example: Using Query Parameters

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/search', (req, res) => {
5.   const name = req.query.name;
6.   const age = req.query.age;
7.   const city = req.query.city;    // may be undefined
8.
9.   res.send('Name: ${name}, Age: ${age}, City: ${city}');
10. });
11.
12. app.listen(3000, () => {
13.   console.log('Server is running on http://localhost:3000');
14. });
```

### How it Works?



### Example URLs

- `/search?name=John`
- `/search?name=John&age=22`
- `/search?name=John&city=Hyd&age=22&page=1`
- `/search`

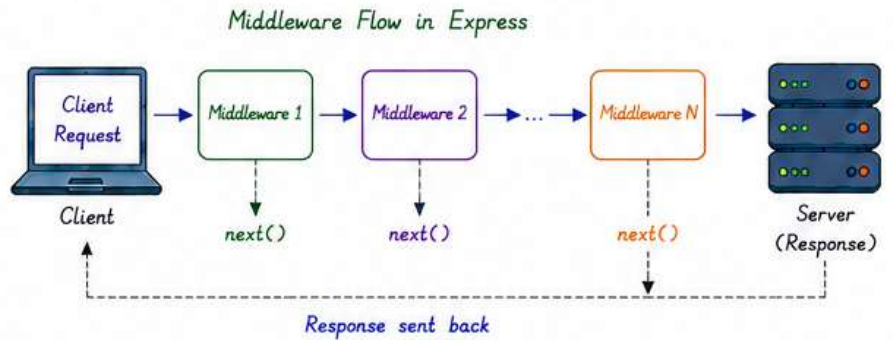
**Note:** Keys are case-sensitive.

**Example:** name and Name are different.

# 13.10 Middleware

## 13.10.1 What is Middleware?

- \* Middleware functions have access to the request object (**req**), response object (**res**), and the **next** middleware function in the application's request-response cycle.
- \* They can execute any code, make changes to the request and response objects, end the request-response cycle, or call the **next** middleware function.
- \* Middleware functions are executed in the **order** they are defined.



## 13.10.2 Syntax

```
app.use((req, res, next) => {  
  // Code to be executed  
  next(); // Pass control to the next middleware  
});
```

## 13.10.3 Types of Middleware

| Type                      | Description                                                                      |
|---------------------------|----------------------------------------------------------------------------------|
| Application Middleware    | Bound to an instance of app object. Used with app.use().                         |
| Router Middleware         | Bound to an express.Router() instance. Used for specific routes.                 |
| Built-in Middleware       | Built-in functions provided by Express. (e.g., express.json(), express.static()) |
| Third-party Middleware    | Created by the community. (e.g., morgan, cors, multer)                           |
| Error-handling Middleware | Handles errors in the application. (Must have 4 parameters: err, req, res, next) |

## 13.10.6 Example: Built-in Middleware

```
1. const express = require('express');  
2. const app = express();  
3.  
4. app.use(express.json()); // To parse JSON data  
5. app.use(express.urlencoded({ extended: true })); // To parse form data  
6. app.use(express.static('public')); // To serve static files  
7.  
8. app.get('/', (req, res) => {  
9.   res.send('Hello from Express Middleware!');  
10. });
```

## 13.10.8 Key Points

- ✓ Middleware functions have access to req, res and next.
- ✓ They are executed in the **order** they are defined.
- ✓ Use next() to pass control to the next middleware.
- ✓ Middleware can modify the request and response objects.
- ✓ Error-handling middleware must have 4 parameters.
- ✓ Middleware makes Express applications modular and powerful.



## Key Takeaway:



Middleware is the heart of Express.js. It helps us handle **requests**, **process data**, manage **authentication**, handle **errors** and build **scalable** and **maintainable** applications.



## 13.10.4 Example: Using Middleware

```
1. const express = require('express');  
2. const app = express();  
3.  
4. // Middleware 1: Logger  
5. app.use((req, res, next) => {  
6.   console.log('Request URL: ${req.url}');  
7.   next(); // Pass control to next middleware  
8. });  
9.  
10. // Middleware 2: Authentication Check  
11. app.use((req, res, next) => {  
12.   console.log('Checking authentication...');  
13.   next();  
14. });
```

## 13.10.5 Built-in Middleware Examples

| Middleware                             | Purpose                               |
|----------------------------------------|---------------------------------------|
| express.json()                         | Parses incoming JSON requests.        |
| express.urlencoded({ extended: true }) | Parses URL-encoded data.              |
| express.static('public')               | Serves static files from a directory. |
| express.text()                         | Parses incoming text data.            |
| express.raw()                          | Parses incoming raw data.             |

## 13.10.7 Error-handling Middleware

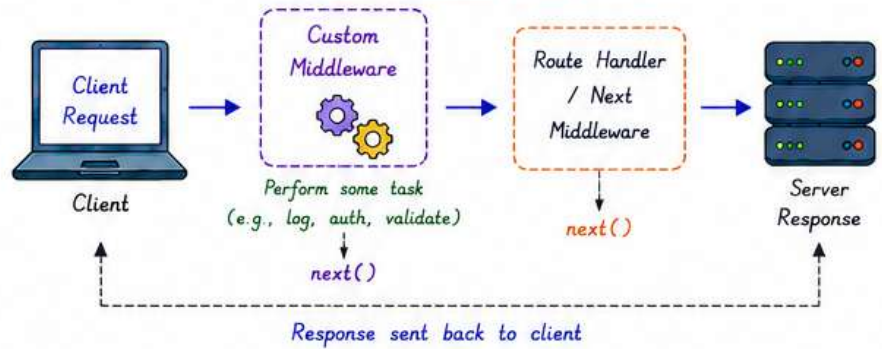
```
1. // Error-handling middleware (must have 4 parameters)  
2. app.use((err, req, res, next) => {  
3.   console.error(err.stack);  
4.   res.status(500).send('Something went wrong!');  
5. });  
6.  
7. app.listen(3000, () => {  
8.   console.log('Server is running on http://localhost:3000');  
9. });
```

# 13.11 Custom Middleware

## 13.11.1 What is Custom Middleware?

- \* Custom middleware is a user-defined function in Express.js.
- \* It allows us to perform specific tasks such as **logging, authentication, validation, etc.**
- \* It has access to **req, res** and **next** objects.
- \* It must call **next()** to pass control to the next middleware or route handler.

## How Custom Middleware Works



## 13.11.2 Syntax

```
function myMiddleware(req, res, next) {
  // Do some task
  next(); // Pass control to the next middleware
}

app.use(myMiddleware); // Use the middleware
```



## 13.11.3 Example Use Cases

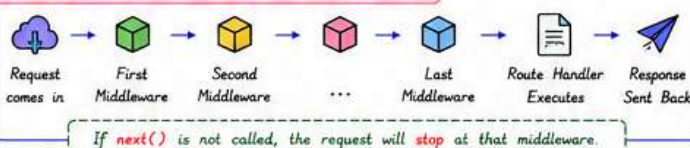
| Use Case                                                                                                   | Purpose                                     |
|------------------------------------------------------------------------------------------------------------|---------------------------------------------|
|  Logging requests          | Log details of incoming requests.           |
|  Authentication          | Verify if the user is authenticated.        |
|  Input validation        | Validate request data before processing.    |
|  Rate limiting           | Limit the number of requests from a client. |
|  Modify request/response | Add or change properties in req or res.     |

## 13.11.5 Example: Simple Logger Middleware

```
1. const express = require('express');
2. const app = express();
3.
4. // Logger Middleware
5. function logRequests(req, res, next) {
6.   console.log(`${req.method} request made to ${req.url}`);
7.   next();
8. }
9.
10. app.use(logRequests); // Apply middleware
11.
12. app.get('/about', (req, res) => {
13.   res.send('About Page');
14. });
15.
16. app.listen(3000, () => {
17.   console.log('Server is running on port 3000');
18. });
```



## 13.11.6 Middleware Execution Order



### Key Points

- ✓ Custom middleware is a regular JavaScript function.
- ✓ It receives **req, res** and **next** as arguments.
- ✓ Always call **next()** to pass control to the next middleware/route.
- ✓ It can be used globally (**app.use**) or for specific routes.
- ✓ Useful for logging, authentication, validation, rate limiting and more.



### Key Takeaway:

Custom middleware gives us the power to handle requests in a structured way, perform necessary tasks, and ensure our Express applications are **secure, scalable** and **maintainable**.



## 13.11.4 Example: Creating Custom Middleware

```
1. const express = require('express');
2. const app = express();
3.
4. // Custom Middleware: Logger
5. function logger(req, res, next) {
6.   const now = new Date().toLocaleString();
7.   console.log(`[${now}] ${req.method} ${req.url}`);
8.   next(); // Pass control to next middleware
9. }
10.
11. // Custom Middleware: Authentication Check
12. function authCheck(req, res, next) {
13.   const token = req.headers['x-auth-token'];
14.   if (!token) {
15.     return res.status(401).send('Unauthorized: No token provided');
16.   }
17.   // Token is valid, continue
18.   next();
19. }
20.
21. // Using Custom Middleware
22. app.use(logger); // All requests will be logged
23. app.use(authCheck); // All requests will be checked
24.
25. app.get('/', (req, res) => {
26.   res.send('Welcome to the Home Page!');
27. });
28.
29. app.listen(3000, () => {
30.   console.log('Server is running on http://localhost:3000');
31. });
```



Logger



Auth Check



Validation



Rate Limit



Modify Data

## 13.11.7 Example: Using on Specific Routes

```
1. // Middleware for a specific route
2. function checkAdmin(req, res, next) {
3.   if (req.query.role === 'admin') {
4.     next();
5.   } else {
6.     res.status(403).send('Access Denied. Admins only!');
7.   }
8. }
9.
10. app.get('/admin', checkAdmin, (req, res) => {
11.   res.send('Welcome Admin!');
12. });
```



### Benefits of Custom Middleware

- ✓ Keeps code modular and reusable.
- ✓ Improves security and data validation.
- ✓ Helps in maintaining clean and organized code.
- ✓ Enhances performance and user experience.

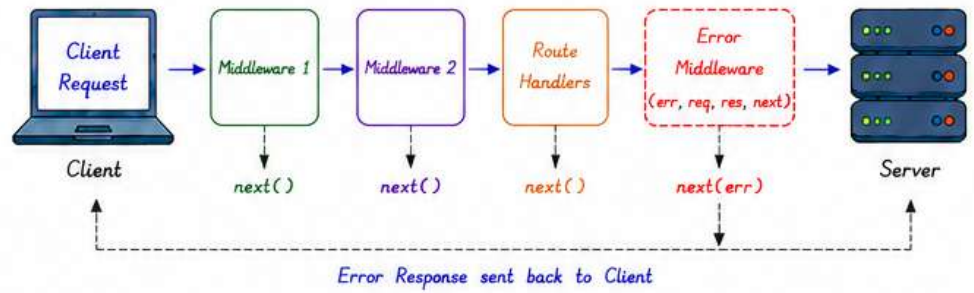


# 13.12 Error Middleware

## 13.12.1 What is Error Middleware?

- \* Error middleware is used to handle errors in an Express application.
- \* It has four parameters: `err, req, res, next`.
- \* It is placed after all other middleware and route handlers.
- \* It helps send a **proper error response** to the client without crashing the server.

### Where Error Middleware Fits?



## 13.12.2 Syntax

```
app.use((err, req, res, next) => {
  // Handle the error
  res.status(500).send('Something went wrong!');
});
```



## 13.12.3 When is Error Middleware Called?

| Scenario                    | Description                                            |
|-----------------------------|--------------------------------------------------------|
| Synchronous Errors          | Errors thrown in route handlers or middleware.         |
| Asynchronous Errors         | Errors passed to <code>next(err)</code> .              |
| Database / Operation Errors | Errors from DB queries, file operations, etc.          |
| Custom Errors               | Manually created errors (e.g., not found, validation). |

## 13.12.4 Example: Handling Sync Error

```
1. const express = require('express');
2. const app = express();
3.
4. app.get('/divide/:a/:b', (req, res, next) => {
5.   const a = parseInt(req.params.a);
6.   const b = parseInt(req.params.b);
7.   if (b === 0) {
8.     // Pass error to error middleware
9.     return next(new Error('Cannot divide by zero'));
10.  }
11.  res.send(`Result: ${a / b}`);
12. });
```



## 13.12.5 Example: Handling Async Error

```
1. app.get('/user/:id', async (req, res, next) => {
2.   try {
3.     const user = await getUserFromDatabase(req.params.id);
4.     if (!user) {
5.       return next(new Error('User not found'));
6.     }
7.     res.json(user);
8.   } catch (err) {
9.     next(err); // Pass error to error middleware
10.  }
11. });
```



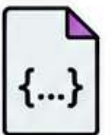
## 13.12.6 Example: Error Middleware in Express

```
1. const express = require('express');
2. const app = express();
3.
4. // Route that throws an error
5. app.get('/error', (req, res, next) => {
6.   throw new Error('Something went wrong!');
7. });
8.
9. // Another route
10. app.get('/', (req, res) => {
11.   res.send('Welcome to Error Middleware Demo');
12. });
13.
14. // ----- Error Middleware (must be after all routes) -----
15. app.use((err, req, res, next) => {
16.   console.error(err.stack); // Log error in server console
17.   res.status(500).json({
18.     success: false,
19.     message: err.message,
20.     stack: process.env.NODE_ENV === 'development' ?
21.       err.stack : {}
22.   });
23. });
24.
25. app.listen(3000, () => {
26.   console.log('Server is running on http://localhost:3000');
27. });
```



## 13.12.7 Error Object Properties

| Property                               | Description                                                      |
|----------------------------------------|------------------------------------------------------------------|
| <code>err.message</code>               | Detailed error message.                                          |
| <code>err.name</code>                  | Error type (e.g., <code>Error</code> , <code>TypeError</code> ). |
| <code>err.stack</code>                 | Stack trace showing where the error occurred.                    |
| <code>err.status/err.statusCode</code> | HTTP status code (custom).                                       |
| <code>err.custom</code>                | Any custom property you add.                                     |



## 13.12.8 Best Practices

- ✓ Always define error middleware as the last `app.use()`.
- ✓ Always send a meaningful error message to the client.
- ✓ Do not expose stack trace in production.
- ✓ Use proper status codes (400, 404, 500, etc.).
- ✓ Log the error for debugging.



## 13.12.9 Error Middleware Execution Flow



### Key Points:

- ✓ Error middleware handles runtime errors and sends a controlled response.
- ✓ It must have four parameters: `(err, req, res, next)`.
- ✓ It is placed after all routes and other middleware.
- ✓ Use `next(err)` to pass errors to the error handling middleware.
- ✓ It improves the reliability and user experience of the application.



### Key Takeaway:

Error middleware is essential for handling errors **gracefully** in Express applications, preventing crashes and providing **useful feedback** to users.

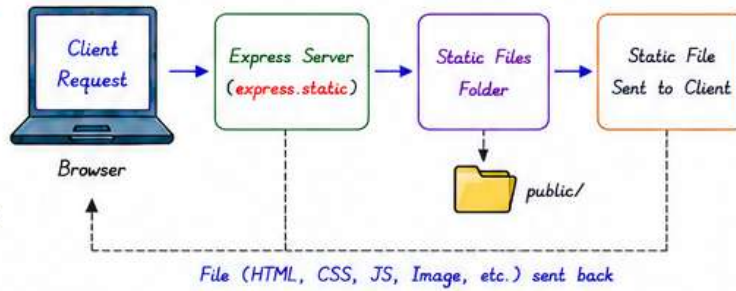


# 13.13 Static Files

## 13.13.1 What are Static Files?

- \* Static files are files that are **sent to the client** without any changes.
- \* Examples: **HTML, CSS, JavaScript, Images, Fonts, Videos, Documents, etc.**
- \* These files **do not need** to be processed by the server.
- \* Express provides the **express.static** middleware to serve static files.

## How Static Files Work in Express

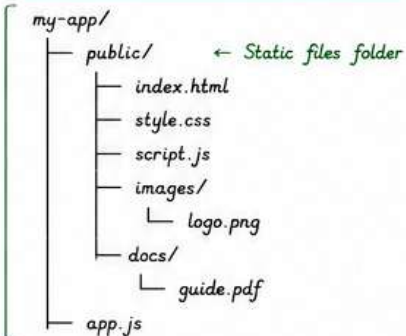


## 13.13.2 Syntax

```
app.use(express.static(root, [options]));
// root : Directory to serve static files from
// options : Optional configuration
```



## 13.13.3 Example Folder Structure



## 13.13.4 Example: Basic Usage

```
1. const express = require('express');
2. const app = express();
3.
4. // Serve static files from 'public' folder
5. app.use(express.static('public'));
6.
7. app.get('/', (req, res) => {
8.   res.sendFile(__dirname + '/public/index.html');
9. });
10.
11. app.listen(3000, () => {
12.   console.log('Server is running on http://localhost:3000');
13. });
```



## 13.13.5 Accessing Static Files

| URL in Browser                        | File Served            |
|---------------------------------------|------------------------|
| http://localhost:3000/index.html      | public/index.html      |
| http://localhost:3000/style.css       | public/style.css       |
| http://localhost:3000/script.js       | public/script.js       |
| http://localhost:3000/images/logo.png | public/images/logo.png |
| http://localhost:3000/docs/guide.pdf  | public/docs/guide.pdf  |



## Key Points:

- ✓ Static files are files that do not change dynamically.
- ✓ Express provides **express.static** middleware to serve static files.
- ✓ Place static files in a folder (commonly **'public'**).
- ✓ Access files directly through their URL.
- ✓ Use options for caching, default files, and other configurations.

## Key Takeaway:



Static files in Express make it easy to serve **front-end assets** like HTML, CSS, JavaScript, images, and documents directly to the client, improving **speed and efficiency**.



## 13.13.6 Example: Using a Custom Path

```
1. const express = require('express');
2. const path = require('path');
3. const app = express();
4.
5. // Serve static files from 'assets' folder at '/assets' route
6. app.use('/assets', express.static(path.join(__dirname, 'assets')));
7.
8. app.get('/', (req, res) => {
9.   res.sendFile(path.join(__dirname, 'views', 'home.html'));
10. });
11.
12. app.listen(3000, () => {
13.   console.log('Server is running on http://localhost:3000');
14. });
```



## 13.13.7 Options

| Option     | Description                                                |
|------------|------------------------------------------------------------|
| dotfiles   | Serve dotfiles (e.g., .gitignore, .env). Default: 'ignore' |
| etag       | Enable ETag generation. Default: true                      |
| extensions | Serve files by trying extensions. Default: false           |
| index      | Send index file. Default: 'index.html'                     |
| maxAge     | Set Cache-Control max-age. Default: 0                      |
| redirect   | Redirect to trailing '/' when directory. Default: true     |



Caching

## 13.13.8 Example: With Options

```
1. app.use(express.static('public', {
2.   dotfiles: 'ignore',
3.   etag: true,
4.   extensions: ['html', 'htm'],
5.   index: 'home.html',
6.   maxAge: '7d', // 7 days
7.   redirect: true
8. }));
```



## 13.13.9 Notes

- ✓ Static files are served as-is, without processing.
- ✓ Place static files in a public (or any) directory.
- ✓ Make sure the path is correct.
- ✓ Use **express.static()** before defining routes that may serve dynamic content.
- ✓ Use caching (**maxAge**) to improve performance.



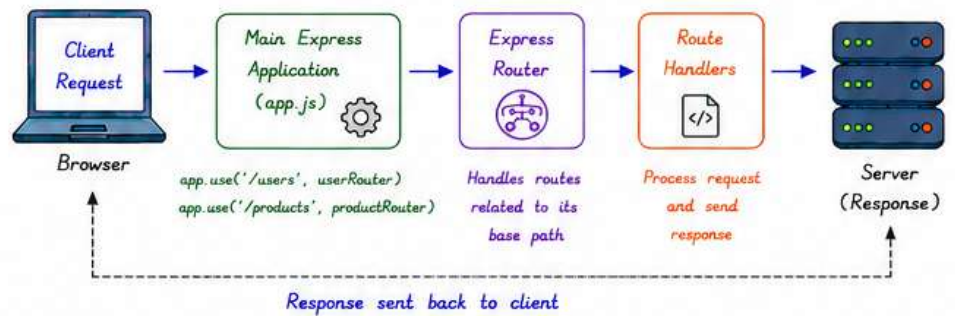
NOTES

# 13.14 Express Router

## 13.14.1 What is Express Router?

- \* Express Router is a **mini application** instance used to define routes and middleware.
- \* It helps to **organize routes** into separate files based on functionality.
- \* It makes the code **modular, clean** and **easy to maintain**.
- \* Routers can be used with different **base paths** in the main application.

## How Express Router Works

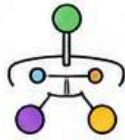


## 13.14.2 Syntax

```
const express = require('express');
const router = express.Router();

// Define routes on the router
router.get('/', (req, res) => {
  res.send('Router Home');
});

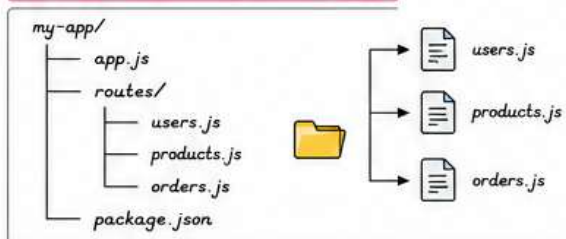
module.exports = router; // Export router
```



## 13.14.3 Benefits of Using Router

| Benefit             | Description                                          |
|---------------------|------------------------------------------------------|
| Modularity          | Breaks the application into smaller, reusable parts. |
| Better Organization | Group related routes together in separate files.     |
| Easier Maintenance  | Easy to update and maintain routes independently.    |
| Reusability         | Routers can be reused in different projects.         |
| Scalability         | Helps in building large applications efficiently.    |

## 13.14.4 File Structure Example



## 13.14.5 Example: Using Express Router

```
// users.js (Router file)
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('Users Home');
});

router.get('/add', (req, res) => {
  res.send('Add New User');
});

module.exports = router;

// app.js (Main file)
const express = require('express');
const app = express();
const userRouter = require('./users');

app.use('/users', userRouter); // Mount router

app.get('/', (req, res) => {
  res.send('Welcome to Express Router Demo');
});

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



## 13.14.6 HTTP Methods in Router

| Method                       | Example                               |
|------------------------------|---------------------------------------|
| router.get(path, handler)    | router.get('/', handler)              |
| router.post(path, handler)   | router.post('/add', handler)          |
| router.put(path, handler)    | router.put('/update/:id', handler)    |
| router.delete(path, handler) | router.delete('/delete/:id', handler) |

GET

POST

PUT

DELETE

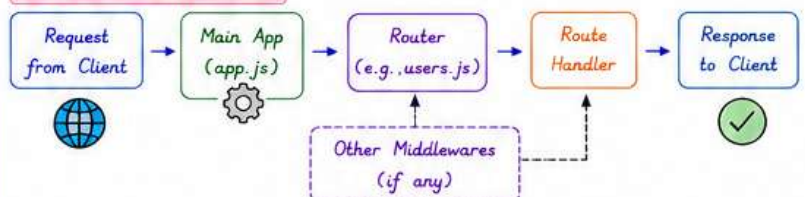
## 13.14.7 Example: Multiple Routers

```
const express = require('express');
const app = express();
const userRouter = require('./routes/users');
const productRouter = require('./routes/products');
const orderRouter = require('./routes/orders');

app.use('/users', userRouter);
app.use('/products', productRouter);
app.use('/orders', orderRouter);

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

## 13.14.8 Router Flow



## 13.14.9 Notes

- ✓ A router is created using `express.Router()`.
- ✓ Routes defined in a router are **not absolute**, they are relative to the path where the router is mounted.
- ✓ Use `module.exports = router` to export the router.
- ✓ Routers can also use middleware.

## Key Takeaway:

Express Router helps to divide the application into multiple route modules, making the code **cleaner, easier to manage** and **scalable**.

## Example URL Mapping

- /users → Handled by users.js
- /users/add → Handled by users.js
- /products → Handled by products.js
- /orders → Handled by orders.js

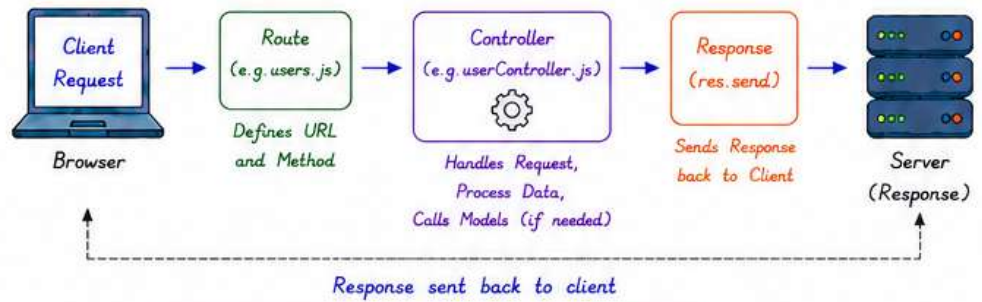


# 13.15 Controllers

## 13.15.1 What are Controllers?

- ✳ Controllers are functions that handle the **business logic** of an application.
- ✳ They receive the **request** from the routes, process the data, and send the **response**.
- ✳ Controllers help keep the code **organized, clean and easy to maintain**.
- ✳ They are usually placed in a separate **controller file**.

## How Controllers Fit in Express Application



## 13.15.2 Syntax

```
// controller function
const controllerFunction = (req, res) => {
  // Business logic
  // Process request and send response
};
```

## 13.15.3 Benefits of Using Controllers

| Benefit                                                                                                 | Description                                            |
|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
|  Separation of Concerns | Routes handle routing, controllers handle logic.       |
|  Code Organization      | Keeps code structured in separate files.               |
|  Reusability          | Controller functions can be reused in multiple routes. |
|  Easy Maintenance     | Easier to update and maintain business logic.          |
|  Better Scalability   | Easier to manage as application grows.                 |

## 13.15.4 Typical Controller Structure

```
// Example: userController.js
exports.getAllUsers = (req, res) => {
  try {
    // Logic to get all users
    res.status(200).json({ success: true, data: users });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
};
```

## 13.15.6 Example: User Controller Methods

```
1. // controllers/userController.js
2. const User = require('../models/userModel');
3.
4. // Get all users
5. exports.getAllUsers = async (req, res) => {
6.   try {
7.     const users = await User.find();
8.     res.status(200).json({ success: true, data: users });
9.   } catch (err) {
10.    res.status(500).json({ success: false, message: err.message });
11.   }
12. };
13.
14. // Get user by ID
15. exports.getUserById = async (req, res) => {
16.   try {
17.     const user = await User.findById(req.params.id);
18.     if (!user) return res.status(404).json({
19.       success: false, message: 'User not found' });
20.     res.status(200).json({ success: true, data: user });
21.   } catch (err) {
22.     res.status(500).json({ success: false, message: err.message });
23.   }
24. };
25.
26. // Create new user
27. exports.createUser = async (req, res) => {
28.   try {
29.     const user = new User(req.body);
30.     await user.save();
31.     res.status(201).json({ success: true, data: user });
32.   } catch (err) {
33.     res.status(500).json({ success: false, message: err.message });
34.   }
35. };
36.
37. // Create new user
```



Get All Users



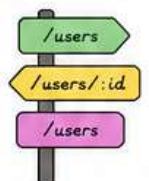
Get User By ID



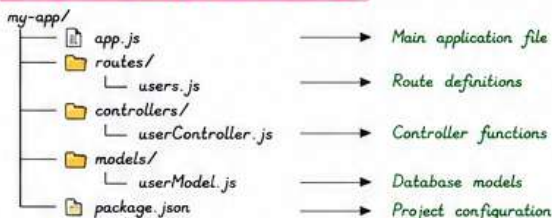
Create User

## 13.15.7 Using Controllers in Routes

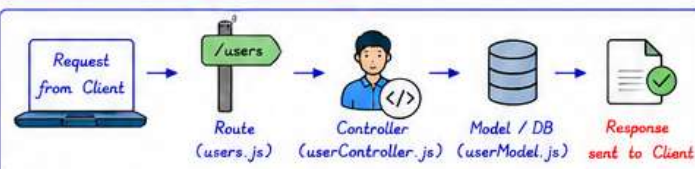
```
1. // routes/users.js
2. const express = require('express');
3. const router = express.Router();
4. const userController = require('../controllers/userController');
5.
6. router.get('/', userController.getAllUsers);
7. router.get('/:id', userController.getUserById);
8. router.post('/', userController.createUser);
9.
10. module.exports = router;
```



## 13.15.5 Example File Structure



## 13.15.8 Example: Full Flow



## 13.15.10 Notes

- ✓ Controllers make the code modular and easy to maintain.
- ✓ Each controller function should handle one action.
- ✓ Always handle errors and send meaningful messages.



## 13.15.9 Common Practices

| Practice                                                                                                     | Description                                       |
|--------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
|  Validate Input           | Validate req.body, req.params, req.query.         |
|  Handle Errors            | Use try...catch and send proper error responses.  |
|  Consistent Responses     | Always send consistent JSON responses.            |
|  Keep Controllers Focused | Controller should only handle logic, not routing. |

## Key Takeaway

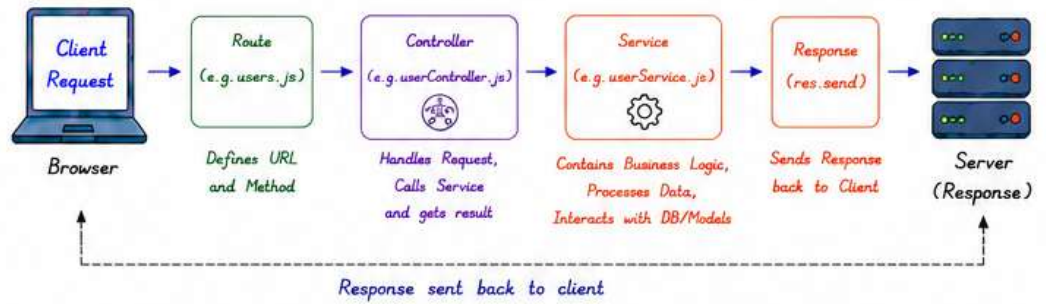
Controllers are the heart of your **application logic**. They process requests, interact with models, and send responses in a clean and organized way.



### 13.16.1 What are Services?

## How Services Fit in Express Application

- \* Services contain the business logic of an application.
- \* They are used by Controllers to perform operations like validation, calculations, data processing, and database actions.
- \* Services help keep controllers thin and focused only on request and response.
- \* They improve code reusability, readability and testability.



### 13.16.2 Syntax

```
// service function
const serviceFunction = async (data) => {
  // Business logic
  // Return result
};
```



### 13.16.3 Benefits of Using Services

| Benefit                                                                                                 | Description                                                       |
|---------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
|  Separation of Concerns | Controllers handle requests, services handle business logic.      |
|  Reusability            | Service functions can be reused across controllers and routes.    |
|  Clean Code           | Keeps code modular, organized and easy to maintain.               |
|  Testability          | Services can be tested independently from routing and HTTP layer. |
|  Scalability          | Business logic can grow without affecting controllers.            |

### 13.16.4 Typical Service Structure

```
// Example: userService.js
exports.getAllUsers = async () => {
  // Logic to get all users
};

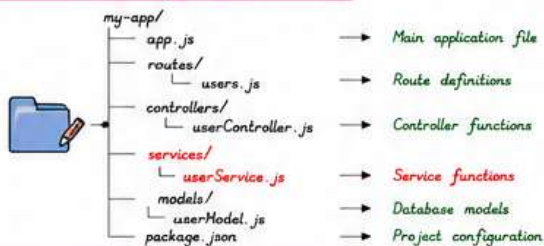
exports.getUserById = async (id) => {
  // Logic to get user by id
};

exports.createUser = async (userData) => {
  // Logic to create user
};
```

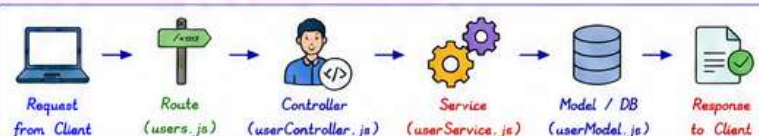


Service

### 13.16.5 Example File Structure



### 13.16.8 Flow of Request with Services



## 13.16.10 Notes

- ✓ Services are reused by multiple controllers and routes.
- ✓ They help in writing clean, maintainable and scalable applications.
- ✓ Business logic should always be in services, not in controllers.
- ✓ Services can also call other services internally if needed.



### 13.16.6 Example: User Service (userService.js)

```

1. const User = require('../models/userModel');
2.
3. // Get all users
4. exports.getAllUsers = async () => {
5.     const users = await User.find();
6.     return users;
7. };
8.
9. // Get user by id
10. exports.getUserById = async (id) => {
11.     const user = await User.findById(id);
12.     if (!user) {
13.         const err = new Error('User not found');
14.         err.statusCode = 404;
15.         throw err;
16.     }
17.     return user;
18. };
19.
20. // Create new user
21. exports.createUser = async (userData) => {
22.     const user = new User(userData);
23.     await user.save();
24.     return user;
25. };
26.
27. // Update user
28. exports.updateUser = async (id, data) => {
29.     const user = await User.findByIdAndUpdate(id, data, { new: true });
30.     return user;
31. };

```



## Get All Users



### Get User By ID



Create  
User



Update  
User

### 13.16.7 Example: Using Service in Controller

```
1. const userService = require('../services/userService');
2.
3. // Get all users
4. exports.getAllUsers = async (req, res, next) => {
5.   try {
6.     const users = await userService.getAllUsers();
7.     res.status(200).json({ success: true, data: users });
8.   } catch (err) {
9.     next(err);
10.  }
11. };
12.
13. // Other controller functions call services similarly...
14.
```



### Controller

### 13.16.9 Best Practices

- ✓ Keep controllers thin. Use services for business logic.
- ✓ Each service function should do one specific task.
- ✓ Handle errors in services and throw meaningful errors.
- ✓ Keep services independent of Express (no req, res).
- ✓ Use `async/await` for asynchronous operations.



### Key Takeaway

Services are the heart of your application's **business logic**. They keep your controllers focused, your code **clean**, and your application **scalable**.

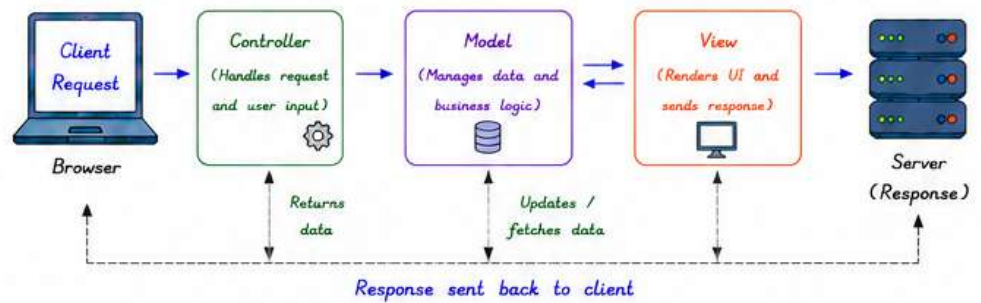


# 13.17 MVC Architecture

## 13.17.1 What is MVC?

- \* MVC stands for **Model, View, Controller**.
- \* It is an **architectural pattern** used to build web applications.
- \* It separates an application into three interconnected components.
- \* It improves code **organization, scalability, and maintainability**.

## How MVC Works in Express Application



## 13.17.2 Roles in MVC

| Component                                                                                   | Role                                                            | Example in Express           |
|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------|------------------------------|
|  Controller | Handles request, calls Model and returns response via View.     | userController.js            |
|  Model      | Handles data, business logic and interacts with database.       | userModel.js                 |
|  View       | Displays data to user (HTML/CSS/JS templates or JSON response). | EJS, Pug, Handlebars or JSON |

## 13.17.3 Benefits of MVC

-  Separation of Concerns  
Each layer has a specific responsibility.
-  Reusability  
Components can be reused across projects.
-  Ease of Maintenance  
Changes in one layer do not affect others.
-  Better Scalability  
Easy to add new features and modules.
-  Team Collaboration  
Different team members can work on different layers.

## 13.17.5 Typical Project Structure



## 13.17.4 Example: Simple MVC in Express

```
// 1. Model (models/userModel.js)
const users = [];
exports.addUser = (name) => {
  users.push({ id: users.length + 1, name });
  return users;
};
exports.getAllUsers = () => users;

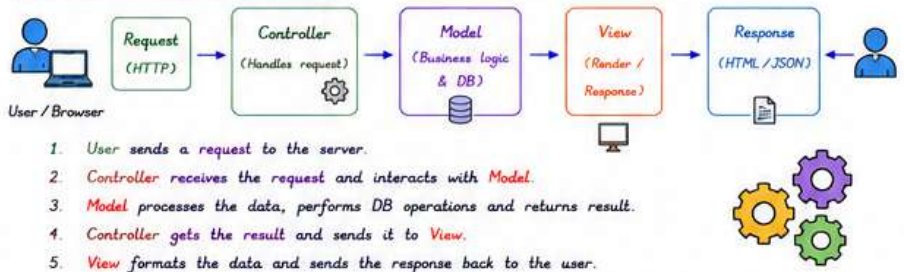
// 2. Controller (controllers/userController.js)
const userModel = require('../models/userModel');
exports.getAllUsers = (req, res) => {
  const users = userModel.getAllUsers();
  res.json({ success: true, data: users });
};
exports.createUser = (req, res) => {
  const { name } = req.body;
  const users = userModel.addUser(name);
  res.status(201).json({ success: true, data: users });
};

// 3. Routes (routes/userRoutes.js)
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

router.get('/', userController.getAllUsers);
router.post('/', userController.createUser);
module.exports = router;

// 4. View (Response sent as JSON)
// You can also render EJS, Pug, etc.
// Example JSON:
// { "success": true, "data": [ { "id": 1, "name": "John" } ] }
```

## 13.17.6 MVC vs in Detail



## 13.17.7 Example: Create and Get Users

```
// Create User (POST /users)
POST /users
Body: { "name": "Alice" }
Response: { "success": true, "data": [ { "id": 1, "name": "Alice" } ] }

// Get All Users (GET /users)
GET /users
Response: { "success": true,
  "data": [ { "id": 1, "name": "Alice" }, { "id": 2, "name": "Bob" } ] }
```

## 13.17.9 Notes

- ✓ MVC is not enforced by Express, it is a way to structure your code.
- ✓ You can use any template engine or even return JSON.
- ✓ Always keep controllers thin and move business logic to services/models.
- ✓ Follow MVC for clean, scalable and professional applications.



## Key Takeaway:

MVC separates your application into **Model** (data), **View** (UI) and **Controller** (logic). It makes your code **clean, organized, reusable** and **scalable**.



| Aspect            | With MVC                    | Without MVC                  |
|-------------------|-----------------------------|------------------------------|
| Code Organization | Well organized & structured | Spaghetti / Unorganized code |
| Maintainability   | Easy to maintain and update | Hard to maintain             |
| Scalability       | Highly scalable             | Difficult to scale           |
| Reusability       | High (components reusable)  | Low (duplicate code)         |
| Testing           | Easy to test each layer     | Difficult to test            |



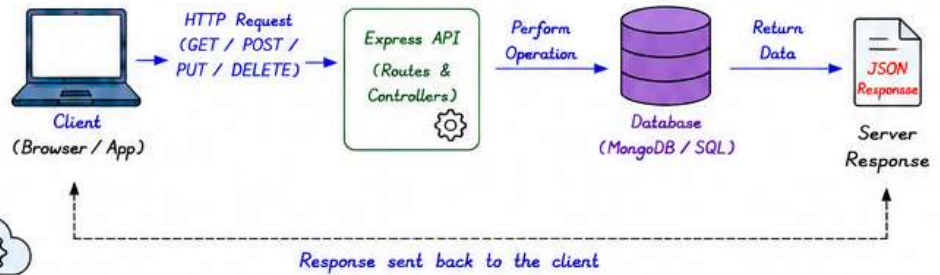
# 13.18 REST API Development

## 13.18.1 What is REST API?

- \* REST stands for **Representational State Transfer**.
- \* It is an architectural style for designing networked applications.
- \* REST APIs allow systems to communicate over HTTP using standard methods.
- \* Data is usually exchanged in **JSON** format.
- \* REST APIs are **lightweight**, **scalable** and **easy to use**.



## How a REST API Works in Express



## 13.18.2 HTTP Methods in REST

| Method | Purpose               | Example         |
|--------|-----------------------|-----------------|
| GET    | Retrieve data         | GET /users      |
| POST   | Create new data       | POST /users     |
| PUT    | Update existing data  | PUT /users/1    |
| PATCH  | Partially update data | PATCH /users/1  |
| DELETE | Delete data           | DELETE /users/1 |

## 13.18.3 Benefits of REST APIs

- \* **Stateless**: Each request from client contains all the information.
- \* **Scalable**: Can handle large number of requests.
- \* **Platform Independent**: Work across different platforms and languages.
- \* **Cacheable**: Responses can be cached to improve performance.
- \* **Easy Integration**: Can be consumed by web, mobile and third-party applications.



## 13.18.4 Typical API Structure

|                           |                          |
|---------------------------|--------------------------|
| my-api/                   |                          |
| ├── app.js                | → Main application file  |
| ├── routes/               |                          |
| │   └── userRoutes.js     | → Route definitions      |
| ├── controllers/          |                          |
| │   └── userController.js | → Controller functions   |
| ├── models/               |                          |
| │   └── userModel.js      | → Database models        |
| ├── services/             |                          |
| │   └── userService.js    | → Business logic         |
| ├── middlewares/          |                          |
| │   └── auth.js, error.js | → Middleware files       |
| ├── config/               |                          |
| │   └── db.js             | → Database configuration |
| └── package.json          | → Project configuration  |

## 13.18.5 Example: Build a Simple REST API (Users)

```
// app.js
1. const express = require('express');
2. const app = express();
3. app.use(express.json());
4.
5. const userRoutes = require('./routes/userRoutes');
6. app.use('/api/users', userRoutes);
7.
8. app.listen(3000, () => {
9.   console.log('REST API is running on http://localhost:3000');
10. });
```



```
// routes/userRoutes.js
1. const express = require('express');
2. const router = express.Router();
3. const userController = require('../controllers/userController');
4.
5. router.get('/', userController.getUsers); // GET /api/users
6. router.get('/:id', userController.getUserById); // GET /api/users/1
7. router.post('/', userController.createUser); // POST /api/users
8. router.put('/:id', userController.updateUser); // PUT /api/users/1
9. router.delete('/:id', userController.deleteUser); // DELETE /api/users/1
10.
11. module.exports = router;
```



```
// controllers/userController.js
1. const User = require('../models/userModel');
2.
3. exports.getUsers = async (req, res) => {
4.   const users = await User.find();
5.   res.status(200).json(users);
6. };
// (Other CRUD methods: getUserById, createUser, updateUser, deleteUser)
```



## 13.18.6 Example API Endpoints

| Endpoint     | Method | Description     | Sample Response                       |
|--------------|--------|-----------------|---------------------------------------|
| /api/users   | GET    | Get all users   | [ { "id": 1, "name": "Alice", ... } ] |
| /api/users/1 | GET    | Get user by ID  | { "id": 1, "name": "Alice" }          |
| /api/users   | POST   | Create new user | { "id": 2, "name": "Bob" }            |
| /api/users/1 | PUT    | Update user     | { "id": 1, "name": "Alice Smith" }    |
| /api/users/1 | DELETE | Delete user     | { "message": "User deleted" }         |



## 13.18.8 Status Codes

| Code | Meaning             | When to Use                   |
|------|---------------------|-------------------------------|
| 200  | OK                  | Request successful            |
| 201  | Created             | Resource created successfully |
| 400  | Bad Request         | Invalid request data          |
| 401  | Unauthorized        | Authentication required       |
| 404  | Not Found           | Resource not found            |
| 500  | InternalServerError | Server-side error             |



## 13.18.7 Sample Request & Response

### Sample Request (POST /api/users)

```
POST /api/users
Headers:
  Content-Type: application/json
Body (JSON):
{
  "name": "Charlie",
  "email": "charlie@example.com"
}
```



### Sample Response

```
Status: 201 Created
Headers:
  Content-Type: application/json
Body (JSON):
{
  "id": 3,
  "name": "Charlie",
  "email": "charlie@example.com"
}
```



## 13.18.9 Best Practices

- ✓ Use meaningful URIs (e.g., /api/users).
- ✓ Use appropriate HTTP methods for each operation.
- ✓ Always return proper status codes and messages.
- ✓ Validate request data before processing.
- ✓ Use middleware for authentication and error handling.
- ✓ Keep responses consistent and in JSON format.



## 13.18.10 Notes

- ✓ REST APIs are stateless and follow standard HTTP protocol.
- ✓ JSON is the most common format for request and response.
- ✓ Tools: Postman, Insomnia, Thunder Client can be used to test APIs.



## Key Takeaway:

REST APIs in Express allow you to build powerful and scalable backend services that can be easily consumed by any client application.

