

Express.js Project – Step-by-Step Implementation Guide

1. Project Initialization

Creating package.json

To initialize the Node.js project and create a `package.json` file, run:

```
npm init -y
```

This command generates a default `package.json` file that stores project metadata and dependencies.

Installing Express.js

Express is a fast and minimal Node.js framework used to build web servers and APIs.

```
npm install express
```

Installing Nodemon (Development Dependency)

Nodemon automatically restarts the server whenever file changes are detected, improving development productivity.

```
npm install --save-dev nodemon
```

Add the following script inside `package.json`:

```
"scripts": {  
  "devStart": "nodemon server.js"  
}
```

2. Creating and Running the Server

Creating `server.js`

Create a file named `server.js` and add the following basic Express server code:

```
const express = require("express")
const app = express()

app.listen(3000)
```

Start the server using:

```
npm run devStart
```

If you add:

```
console.log("hello")
```

It will appear in the terminal, confirming the server is running.

3. Understanding Routes and Responses

Accessing the Server

Visit:

```
http://localhost:3000
```

If the browser shows “**Cannot GET /**”, it means no route has been defined for `/`.

Creating a GET Route

```
app.get("/", (req, res) => {
  console.log("shows in console")
  res.send("Hi")
})
```

- Displays "Hi" in the browser
 - Logs a message in the terminal
-

Sending Status Codes

```
res.sendStatus(500)
```

Displays **Internal Server Error** in the browser.

```
res.status(500).send("500 status code!")
```

Returns custom text along with the status code.

```
res.status(500).json({ error: "Server Error 500" })
```

Returns a JSON response.

For successful requests, Express automatically sends status code **200**.

4. Sending Files to the Client

To allow users to download a file:

```
res.download("server.js")
```

Refreshing the browser triggers the download.

5. Using Template Engines (EJS)

Installing EJS

```
npm install ejs
```

Set EJS as the view engine:

```
app.set("view engine", "ejs")
```

Rename `index.html` to `index.ejs` and place it inside the `views` folder.

EJS Expression Example

```
<h3>Given text: <%= 4 + 4 %></h3>
```

Output:

```
Given text: 8
```

Passing Data from Server to View

server.js

```
app.get("/", (req, res) => {
  res.render("index", { text: "How are you!" })
})
```

index.ejs

```
<h4>Text from server: <%= text %></h4>
```

Handling Missing Variables

If the variable is not passed correctly:

```
res.render("index", { textnew: "How are you!" })
```

Use a default value in EJS:

```
<h4>Text from server: <%= locals.text || "Default Value!" %></h4>
```

6. Route Modularization (Using Router)

When routes become complex, they should be separated.

Creating routes/users.js

```
const express = require("express")
const router = express.Router()

router.get("/", (req, res) => {
  res.send("User Lists")
})

router.get("/new", (req, res) => {
  res.send("User New Form")
})

module.exports = router
```

Connecting Router in server.js

```
const userRouter = require("./routes/users")
app.use("/users", userRouter)
```

Routes:

- /users → User Lists
 - /users/new → User New Form
-

7. Dynamic Route Parameters

```
router.get("/:userId", (req, res) => {
  res.send(`Got a user ID: ${req.params.userId}`)
})
```

⚠️ Important:

Dynamic routes must always be placed **below static routes** (`/new`), because Express matches routes top-to-bottom.

8. Chained Routes (GET, PUT, DELETE)

```
router.route("/:userId")
  .get((req, res) => {
    res.send(`Get user with ID ${req.params.userId}`)
  })
  .put((req, res) => {
    res.send(`Put user with ID ${req.params.userId}`)
  })
  .delete((req, res) => {
    res.send(`Delete user with ID ${req.params.userId}`)
  })
```

9. Using `router.param()`

```
const users = [{ name: "Routh" }, { name: "Engineer" }]

router.param("userId", (req, res, next, userId) => {
  req.user = users[userId]
  next()
})
```

Now `req.user` is accessible in routes:

```
console.log(req.user)
```

10. Middleware Concept

Middleware executes **between request start and response end**.

Logger Middleware

```
function logger(req, res, next) {  
  console.log(req.originalUrl)  
  next()  
}
```

Apply globally:

```
app.use(logger)
```

Or apply to a specific route:

```
app.get("/", logger, logger, (req, res) => {  
  res.render("index", { text: "How are you!" })  
})
```

11. Static Files Using `express.static`

For static HTML files:

```
app.use(express.static("public"))
```

Folder structure:

```
public/  
  └── index.html  
  └── test/  
      └── tt.html
```

Routes:

- `/` → index.html
 - `/test/tt.html` → tt.html
-

12. Forms and POST Requests

users/new.ejs

```
<form action="/users" method="POST">
  <label>
    First Name
    <input type="text" name="firstName" value="<%=
      locals.firstName
    %>">
  </label>
  <button type="submit">Submit</button>
</form>
```

Handling POST Request

```
router.post("/", (req, res) => {
  res.send("Create user!")
})
```

13. Handling Form Data

Enable body parsing in `server.js`:

```
app.use(express.urlencoded({ extended: true }))
```

Access input value:

```
console.log(req.body.firstName)
```

14. Redirect After Form Submission

```
router.post("/", (req, res) => {
  const isValid = true

  if (isValid) {
    users.push({ firstName: req.body.firstName })
    res.redirect(`/users/${users.length - 1}`)
  } else {
    res.render("users/new", { firstName: req.body.firstName })
  }
})
```

15. Query Parameters

URL:

<http://localhost:3000/users?name=userName>

Code:

```
console.log(req.query.name)
```

Output in console:

userName

Conclusion

This project demonstrates:

- Express server setup

- Routing and modular architecture
- Middleware usage
- Static file handling
- Template engines (EJS)
- Form handling and redirects
- RESTful routing concepts

It reflects real-world backend development practices and clean Express.js design.