Complete Express.js Guide

A Comprehensive Tutorial with Practical Examples

Table of Contents

---

1. Introduction to Express.js

What is Express.js?

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. Think of it as a toolkit that makes creating web servers much easier than using plain Node.js.

Key Benefits:

- Simple and easy to learn
- Fast development
- Large community and ecosystem
- Flexible and unopinionated
- Great for building APIs and web applications

Real-World Analogy: If Node.js is like having basic building blocks, Express.js is like having pre-made templates and tools that help you build faster and better.

---

2. Creating Your First Express App

2.1 Project Setup

Step 1: Create Project Directory

```bash
# Create a new folder for your project
mkdir my-express-app
cd my-express-app

# Initialize Node.js project
npm init -y
```

What this does:

- Creates a new folder
- Initializes a package.json file (stores project information)

Step 2: Install Express

```bash
npm install express
```

This downloads Express and adds it to your project's dependencies.

2.2 Understanding Dependencies

When you install Express, your `package.json` looks like this:

```json
```

```
{
  "name": "my-express-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Additional Useful Dependencies:

```bash
# For auto-reloading during development
npm install --save-dev nodemon

# For parsing JSON data
npm install body-parser

# For handling file uploads
npm install multer

# For environment variables
npm install dotenv
```

2.3 Basic App Initialization

Create a file named `app.js`:

```javascript
```

```javascript
// Import Express
const express = require('express');

// Create Express application
const app = express();

// Define a port number
const PORT = 3000;

// Create a simple route
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Explanation Line by Line:

1. `require('express')`: Imports the Express library
2. `express()`: Creates a new Express application instance
3. `PORT = 3000`: Defines which port the server will listen on
4. `app.get()`: Creates a route that responds to GET requests
5. `app.listen()`: Starts the server and listens for requests

Run Your Server:

```bash
node app.js
```

Visit `http://localhost:3000` in your browser to see "Hello, Express!"

2.4 Understanding the Listen Method

The `listen()` method is what actually starts your server.

```javascript
```

```javascript
// Basic syntax
app.listen(port, [hostname], [callback]);

// Example 1: Simple
app.listen(3000);

// Example 2: With callback
app.listen(3000, () => {
  console.log('Server started successfully!');
});

// Example 3: With hostname
app.listen(3000, 'localhost', () => {
  console.log('Server is running on localhost:3000');
});

// Example 4: Using environment variable
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

2.5 App Object Methods

The `app` object has many useful methods:

```javascript
```

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

// 1. HTTP Methods
app.get('/users', (req, res) => {
  res.send('Get all users');
});

app.post('/users', (req, res) => {
  res.send('Create a new user');
});

app.put('/users/:id', (req, res) => {
  res.send('Update user');
});

app.delete('/users/:id', (req, res) => {
  res.send('Delete user');
});

// 2. use() - Apply middleware
app.use(express.json()); // Parse JSON bodies

// 3. set() - Configure application settings
app.set('view engine', 'ejs');
app.set('port', 3000);

// 4. get() - Retrieve setting value
const port = app.get('port');

// 5. locals - Store application-wide variables
app.locals.title = 'My Express App';
app.locals.email = 'admin@example.com';

// 6. route() - Create chainable routes
app.route('/book')
  .get((req, res) => res.send('Get book'))
  .post((req, res) => res.send('Add book'))
  .put((req, res) => res.send('Update book'));

app.listen(3000);
```

3. Understanding Routes

3.1 Route Definition

A route is a way to respond to client requests at specific endpoints (URLs).

Basic Structure:

```
javascript

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

- **METHOD**: HTTP method (get, post, put, delete, patch)
- **PATH**: URL path
- **HANDLER**: Function to execute when route is matched

3.2 GET Routes

GET requests retrieve data from the server.

```
javascript
```

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

// Simple GET route
app.get('/', (req, res) => {
  res.send('Welcome to Homepage');
});

// GET route with HTML response
app.get('/about', (req, res) => {
  res.send('<h1>About Us</h1><p>This is the about page</p>');
});

// GET route with JSON response
app.get('/api/users', (req, res) => {
  const users = [
    { id: 1, name: 'John Doe', email: 'john@example.com' },
    { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
  ];
  res.json(users);
});

// GET route with status code
app.get('/success', (req, res) => {
  res.status(200).send('Operation successful');
});

app.listen(3000);
```

3.3 POST Routes

POST requests send data to create new resources.

```javascript
```

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

// Middleware to parse JSON bodies
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// POST route to create user
app.post('/api/users', (req, res) => {
  const newUser = {
    id: Date.now(),
    name: req.body.name,
    email: req.body.email
  };

  console.log('Received:', newUser);

  res.status(201).json({
    message: 'User created successfully',
    user: newUser
  });
});

// POST route with validation
app.post('/api/login', (req, res) => {
  const { username, password } = req.body;

  if (!username || !password) {
    return res.status(400).json({
      error: 'Username and password are required'
    });
  }

  // Check credentials (simplified example)
  if (username === 'admin' && password === 'password') {
    res.json({
      message: 'Login successful',
      token: 'abc123xyz'
    });
  } else {
    res.status(401).json({
      error: 'Invalid credentials'
    });
```

```
  }
});

app.listen(3000);
```

Test with curl:

```bash
curl -X POST http://localhost:3000/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"John Doe","email":"john@example.com"}'
```

3.4 PUT Routes

PUT requests update existing resources.

```javascript
```

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

app.use(express.json());

// Sample data
let users = [
  { id: 1, name: 'John', email: 'john@example.com' },
  { id: 2, name: 'Jane', email: 'jane@example.com' }
];

// PUT route to update user
app.put('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);

  if (userIndex === -1) {
    return res.status(404).json({ error: 'User not found' });
  }

  // Update user
  users[userIndex] = {
    id: userId,
    name: req.body.name,
    email: req.body.email
  };

  res.json({
    message: 'User updated successfully',
    user: users[userIndex]
  });
});

app.listen(3000);
```

3.5 DELETE Routes

DELETE requests remove resources.

```javascript
```

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

let users = [
  { id: 1, name: 'John', email: 'john@example.com' },
  { id: 2, name: 'Jane', email: 'jane@example.com' }
];

// DELETE route
app.delete('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const userIndex = users.findIndex(u => u.id === userId);

  if (userIndex === -1) {
    return res.status(404).json({ error: 'User not found' });
  }

  const deletedUser = users.splice(userIndex, 1)[0];

  res.json({
    message: 'User deleted successfully',
    user: deletedUser
  });
});

app.listen(3000);
```

3.6 PATCH Routes

PATCH requests partially update resources.

```javascript
```

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

app.use(express.json());

let users = [
  { id: 1, name: 'John', email: 'john@example.com', age: 30 }
];

// PATCH route - update only specific fields
app.patch('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find(u => u.id === userId);

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  // Update only provided fields
  if (req.body.name) user.name = req.body.name;
  if (req.body.email) user.email = req.body.email;
  if (req.body.age) user.age = req.body.age;

  res.json({
    message: 'User updated partially',
    user: user
  });
});

app.listen(3000);
```

Difference between PUT and PATCH:

- **PUT**: Replaces entire resource (all fields required)
- **PATCH**: Updates specific fields (only changed fields needed)

3.7 Route Parameters

Route parameters are named URL segments used to capture values.

```javascript
```

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

// Single parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});

// Multiple parameters
app.get('/users/:userId/posts/:postId', (req, res) => {
  const { userId, postId } = req.params;
  res.json({
    message: 'Fetching post',
    userId: userId,
    postId: postId
  });
});

// Optional parameters using regex
app.get('/users/:id(\\d+)', (req, res) => {
  // Only matches if id is a number
  res.send(`User ID (number only): ${req.params.id}`);
});

// Parameter with custom name
app.get('/files/:filename.:format', (req, res) => {
  const { filename, format } = req.params;
  res.send(`File: ${filename}.${format}`);
});

app.listen(3000);
```

Examples:

- `/users/123` → userId = "123"
- `/users/5/posts/42` → userId = "5", postId = "42"
- `/files/report.pdf` → filename = "report", format = "pdf"

3.8 Query Strings

Query strings pass data via URL after the `?` symbol.

javascript

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

// Simple query string
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`Searching for: ${query}`);
});
// URL: /search?q=express

// Multiple query parameters
app.get('/products', (req, res) => {
  const { category, minPrice, maxPrice, sort } = req.query;

  res.json({
    category: category || 'all',
    priceRange: {
      min: minPrice || 0,
      max: maxPrice || 'unlimited'
    },
    sortBy: sort || 'name'
  });
});
// URL: /products?category=electronics&minPrice=100&maxPrice=500&sort=price

// Query with defaults
app.get('/api/users', (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const sortBy = req.query.sortBy || 'name';

  res.json({
    message: 'Fetching users',
    pagination: {
      page: page,
      limit: limit,
      sortBy: sortBy
    }
  });
});
// URL: /api/users?page=2&limit=20&sortBy=email
```

```
app.listen(3000);
```

Difference between Params and Query:

- **Params**: `/users/:id` → Part of the path, required
- **Query**: `/users?id=123` → After `?`, optional

3.9 Route Matching

Express matches routes in the order they are defined.

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

// Specific route (checked first)
app.get('/users/admin', (req, res) => {
  res.send('Admin user page');
});

// Parameter route (checked second)
app.get('/users/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});

// Wildcard route (checked last)
app.get('/users/*', (req, res) => {
  res.send('Some user-related page');
});

// Pattern matching with regex
app.get(/^\/users\/([0-9]+)$/, (req, res) => {
  res.send(`User with numeric ID: ${req.params[0]}`);
});

// Multiple paths for same handler
app.get(['/about', '/about-us'], (req, res) => {
  res.send('About Us page');
});

app.listen(3000);
```

Important Rule: Always put specific routes before generic ones!

3.10 Method Chaining

Handle multiple HTTP methods on the same route.

```javascript
```

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

app.use(express.json());

// Method 1: Individual routes
app.get('/user', (req, res) => res.send('Get user'));
app.post('/user', (req, res) => res.send('Create user'));
app.put('/user', (req, res) => res.send('Update user'));
app.delete('/user', (req, res) => res.send('Delete user'));

// Method 2: Using app.route() - Cleaner!
app.route('/product')
  .get((req, res) => {
    res.json({ message: 'Get product' });
  })
  .post((req, res) => {
    res.json({ message: 'Create product' });
  })
  .put((req, res) => {
    res.json({ message: 'Update product' });
  })
  .delete((req, res) => {
    res.json({ message: 'Delete product' });
  });

// Method 3: With parameters
app.route('/books/:id')
  .get((req, res) => {
    res.send(`Get book ${req.params.id}`);
  })
  .put((req, res) => {
    res.send(`Update book ${req.params.id}`);
  })
  .delete((req, res) => {
    res.send(`Delete book ${req.params.id}`);
  });

app.listen(3000);
```

Benefits of Method Chaining:

- Less code repetition

- Better organization
- Easier to maintain

3.11 Complete CRUD Operations Example

javascript

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

app.use(express.json());

// In-memory database (array)
let books = [
  { id: 1, title: '1984', author: 'George Orwell', year: 1949 },
  { id: 2, title: 'To Kill a Mockingbird', author: 'Harper Lee', year: 1960 }
];

// CREATE - Add new book
app.post('/api/books', (req, res) => {
  const newBook = {
    id: books.length + 1,
    title: req.body.title,
    author: req.body.author,
    year: req.body.year
  };

  books.push(newBook);

  res.status(201).json({
    message: 'Book created successfully',
    book: newBook
  });
});

// READ - Get all books
app.get('/api/books', (req, res) => {
  res.json({
    count: books.length,
    books: books
  });
});

// READ - Get single book
app.get('/api/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));

  if (!book) {
    return res.status(404).json({ error: 'Book not found' });
  }
```

```
  res.json(book);
});

// UPDATE - Update entire book
app.put('/api/books/:id', (req, res) => {
  const bookIndex = books.findIndex(b => b.id === parseInt(req.params.id));

  if (bookIndex === -1) {
    return res.status(404).json({ error: 'Book not found' });
  }

  books[bookIndex] = {
    id: parseInt(req.params.id),
    title: req.body.title,
    author: req.body.author,
    year: req.body.year
  };

  res.json({
    message: 'Book updated successfully',
    book: books[bookIndex]
  });
});

// UPDATE - Partial update
app.patch('/api/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));

  if (!book) {
    return res.status(404).json({ error: 'Book not found' });
  }

  if (req.body.title) book.title = req.body.title;
  if (req.body.author) book.author = req.body.author;
  if (req.body.year) book.year = req.body.year;

  res.json({
    message: 'Book updated partially',
    book: book
  });
});

// DELETE - Remove book
```

```
app.delete('/api/books/:id', (req, res) => {
  const bookIndex = books.findIndex(b => b.id === parseInt(req.params.id));

  if (bookIndex === -1) {
    return res.status(404).json({ error: 'Book not found' });
  }

  const deletedBook = books.splice(bookIndex, 1)[0];

  res.json({
    message: 'Book deleted successfully',
    book: deletedBook
  });
});

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

---

4. Working with Middleware

4.1 What is Middleware?

Middleware functions are functions that have access to the request (`req`), response (`res`), and the next middleware function (`next`) in the application's request-response cycle.

Think of middleware as checkpoints: Just like security checkpoints at an airport, middleware checks and processes requests before they reach their final destination (your route handler).

Simple Example:

```
javascript
```

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

// This is middleware!
app.use((req, res, next) => {
  console.log('Request received at:', new Date().toISOString());
  next(); // Pass control to next middleware
});

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

app.listen(3000);
```

4.2 Understanding the Middleware Stack

Requests flow through middleware in order:

```javascript
```

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

// Middleware 1
app.use((req, res, next) => {
  console.log('1. First middleware');
  next();
});

// Middleware 2
app.use((req, res, next) => {
  console.log('2. Second middleware');
  next();
});

// Middleware 3
app.use((req, res, next) => {
  console.log('3. Third middleware');
  next();
});

// Route handler (final destination)
app.get('/', (req, res) => {
  console.log('4. Route handler');
  res.send('Response sent');
});

app.listen(3000);

// Console output when visiting /:
// 1. First middleware
// 2. Second middleware
// 3. Third middleware
// 4. Route handler
```

4.3 The next() Function

The `next()` function passes control to the next middleware.

```javascript
```

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

// Middleware that calls next()
app.use((req, res, next) => {
  console.log('Processing...');
  next(); // Continue to next middleware
});

// Middleware that stops the chain
app.use((req, res, next) => {
  if (req.query.stop === 'true') {
    return res.send('Stopped here!');
    // next() is NOT called, so chain stops
  }
  next();
});

app.get('/', (req, res) => {
  res.send('Reached the route!');
});

app.listen(3000);

// Try: http://localhost:3000/ → "Reached the route!"
// Try: http://localhost:3000/?stop=true → "Stopped here!"
```

Important Rules:

1. Always call `next()` unless you're sending a response
2. Don't call `next()` after sending a response
3. Don't call `next()` multiple times

4.4 Types of Middleware

4.4.1 Application-Level Middleware

Bound to the app object, runs for all routes.

```javascript
```

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

// Runs for ALL routes
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// Runs for routes starting with /api
app.use('/api', (req, res, next) => {
  console.log('API request received');
  next();
});

app.get('/', (req, res) => res.send('Home'));
app.get('/api/users', (req, res) => res.send('Users API'));

app.listen(3000);
```

4.4.2 Router-Level Middleware

Works same as application-level but bound to router.

```
javascript
```

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

// Middleware for this router only
router.use((req, res, next) => {
  console.log('Router middleware');
  next();
});

router.get('/products', (req, res) => {
  res.send('Products list');
});

router.get('/products/:id', (req, res) => {
  res.send(`Product ${req.params.id}`);
});

// Mount router
app.use('/api', router);

app.listen(3000);
```

4.4.3 Built-in Middleware

Express provides some built-in middleware:

```javascript
```

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

// 1. Parse JSON bodies
app.use(express.json());

// 2. Parse URL-encoded bodies (form data)
app.use(express.urlencoded({ extended: true }));

// 3. Serve static files
app.use(express.static('public'));

// 4. Serve static files with prefix
app.use('/assets', express.static('public'));

app.post('/api/data', (req, res) => {
  console.log(req.body); // Now you can access POST data
  res.json(req.body);
});

app.listen(3000);
```

4.4.4 Third-Party Middleware

Popular middleware from npm:

```javascript
```

```javascript
const express = require('express');
const morgan = require('morgan'); // Logging
const cors = require('cors'); // Cross-origin requests
const helmet = require('helmet'); // Security headers
const compression = require('compression'); // Compress responses

const app = express();

// Install: npm install morgan cors helmet compression

// Logging middleware
app.use(morgan('dev'));

// Enable CORS
app.use(cors());

// Security headers
app.use(helmet());

// Compress responses
app.use(compression());

// Your routes here
app.get('/', (req, res) => {
  res.send('Hello with middleware!');
});

app.listen(3000);
```

4.4.5 Route-Specific Middleware

Apply middleware to specific routes only:

```javascript
```

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

// Authentication middleware
const authenticate = (req, res, next) => {
  const token = req.headers.authorization;

  if (token === 'secret-token') {
    next(); // Authenticated, continue
  } else {
    res.status(401).json({ error: 'Unauthorized' });
  }
};

// Logging middleware
const logRequest = (req, res, next) => {
  console.log(`${req.method} ${req.url} at ${new Date().toISOString()}`);
  next();
};

// Public route (no middleware)
app.get('/', (req, res) => {
  res.send('Public homepage');
});

// Protected route (with authentication)
app.get('/dashboard', authenticate, (req, res) => {
  res.send('Private dashboard');
});

// Route with multiple middleware
app.get('/admin', [authenticate, logRequest], (req, res) => {
  res.send('Admin panel');
});

// Route-specific middleware for POST
app.post('/api/data', logRequest, (req, res) => {
  res.send('Data received');
});

app.listen(3000);
```

4.5 Error-Handling Middleware

Error-handling middleware has 4 parameters: `err, req, res, next`.

```javascript
```

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

app.use(express.json());

// Regular routes
app.get('/', (req, res) => {
  res.send('Homepage');
});

app.get('/error', (req, res) => {
  throw new Error('Something went wrong!');
});

app.post('/api/user', (req, res, next) => {
  if (!req.body.name) {
    const error = new Error('Name is required');
    error.status = 400;
    return next(error); // Pass error to error handler
  }
  res.json({ message: 'User created' });
});

// 404 Handler (no route matched)
app.use((req, res, next) => {
  res.status(404).json({
    error: 'Route not found',
    path: req.url
  });
});

// Error handling middleware (MUST be last)
app.use((err, req, res, next) => {
  console.error('Error:', err.message);

  res.status(err.status || 500).json({
    error: err.message || 'Internal Server Error',
    path: req.url,
    timestamp: new Date().toISOString()
  });
});
```

```
app.listen(3000);
```

Key Points:

- Error middleware must have 4 parameters
- Must be defined AFTER all other routes and middleware
- Catches errors from all previous middleware and routes

4.6 Middleware Ordering

CRITICAL: Order matters! Middleware executes in the order it's defined.

```
javascript
```

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

// ❌ WRONG - This won't work!
app.get('/api/data', (req, res) => {
  res.json(req.body); // req.body is undefined!
});

app.use(express.json()); // Too late!

// ✅ CORRECT - Middleware before routes
const express = require('express');
const app = express();

// 1. Body parsing middleware first
app.use(express.json());

// 2. Logging middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// 3. Routes come after middleware
app.get('/api/data', (req, res) => {
  res.json(req.body); // Now req.body works!
});

// 4. 404 handler
app.use((req, res) => {
  res.status(404).send('Not found');
});

// 5. Error handler (always last!)
app.use((err, req, res, next) => {
  res.status(500).send('Server error');
});

app.listen(3000);
```

Correct Order:

1. Built-in middleware (express.json(), etc.)

2. Third-party middleware (morgan, cors, etc.)
3. Custom middleware
4. Routes
5. 404 handler
6. Error handler

4.7 Creating Custom Middleware

Example 1: Request Logger

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

// Custom logging middleware
const requestLogger = (req, res, next) => {
  const startTime = Date.now();

  // Log when request arrives
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

  // Log when response is sent
  res.on('finish', () => {
    const duration = Date.now() - startTime;
    console.log(`Response sent in ${duration}ms with status ${res.statusCode}`);
  });

  next();
};

app.use(requestLogger);

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

app.listen(3000);
```

Example 2: Authentication Middleware

```javascript
```

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

// Authentication middleware
const requireAuth = (req, res, next)
```