Express.js

- Introduction
  - About Express.js
  - Features
  - Versioning
- Getting Started
  - Installation
  - Hello World example
  - Basic routing
  - Static files
- Guide
  - Routing
  - Middleware
  - Using template engines
  - Error handling
  - Debugging
  - Security best practices
  - Performance best practices
  - Deployment
- API Reference
  - Application
  - Request
  - Response
- Advanced Topics
  - Router
  - Middleware
  - Error handling
- Resources
  - Examples
  - Third-party middleware
  - Books and tutorials
  - Community resources

Introduction

- **Express.js**: Express.js is a popular web framework for Node.js thatifies the process of building web applications. It provides a set of tools and features that make it easy to handle web pages, forms, and more.
- **Features**: Express.js has many useful features, including:
    - **Routing**: Express.js allows you to define routes for different pages and actions in your app. You can use methods like get, post, put, and delete to handle different types of requests.
    - **Middleware**: Express.js allows you to use middleware functions to modify requests and responses, handle errors, and more. Middleware functions can be used to add functionality to your app without modifying the core code.
    - **Template engines**: Express.js allows you to use template engines like EJS, Handlebars, and Pug to generate HTML pages dynamically using data from your app.
    - **Static files**: Express.js allows you to serve static files like images, stylesheets, and JavaScript files.
    - **Error handling**: Express.js provides a set of tools for handling errors in your app, including middleware functions and error handling routes.
    - **Security**: Express.js provides a set of best practices for securing your app, including using HTTPS, validating user input, and using secure cookies.
    - **Performance**: Express.js provides a set of best practices for optimizing the performance of your app, including using caching, compressing responses, and minimizing the use of synchronous code.
- **Versioning:** Express.js has different versions, and they improve it over time. The latest version of Express.js as of May 2023 is version 5.0.0-alpha.8. New versions of Express.js are released periodically, and they often include bug fixes, performance improvements, and new features.

Getting Started

Installation

To use Express.js, you need to install it on your computer. You can do this using the Node.js package manager (npm) by running the following command in your terminal or command prompt:

```
npm install express
```

This command installs the Express.js package and makes it available for use in your project.

Hello World example

The following code creates a simple Express.js app that responds with "Hello, World!" when you visit the main page:

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

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

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

This code does the following:

1. Imports the Express.js package and creates a new Express.js app.
2. Defines a route for the main page ("/") using the `get` method. When someone visits the main page, the app sends "Hello, World!" as a response.
3. Starts the app listening on port 3000 and logs a message to the console.

Basic routing

The following code defines a route for an "About" page:

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

This code defines a route for the "About" page ("/about") using the `get` method. When someone visits the "About" page, the app sends "About page" as a response.

Static files

The following code tells Express.js to serve static files from a folder named "public":

```
app.use(express.static('public'));
```

This code tells Express.js to serve static files from a folder named "public". When someone requests a file, Express.js looks for it in the "public" folder and sends it as a response.

Guide

Routing

Routing is a way to define different pages and actions in your. In Express.js, you can use methods like `get`, `post`, `put`, and `delete` to handle different types of requests.

Here's an example of a route for a "Contact" page that uses the `post` method to handle form submissions:

```
app.post('/contact', (req, res) => {
  const name = req.body.name;
  const email = req.body.email;
  const message = req.body.message;
  // Do something with the form data
  res.send('Thanks for contacting us!');
});
```

This code defines a route for the "Contact" page ("/contact") using the `post` method. When someone submits a form on the "Contact" page, the app receives the form data in the `req.body` object. In this example, we're extracting the name, email, and message fields from the form data and doing something with them. Finally, the app sends a "Thanks for contacting us!" message as a response.

Middleware

Middleware are functions that can do things before or after your app does something. In Express.js, middleware functions can be used to modify requests and responses, handle errors, and more.

Here's an example of a middleware function that logs the time of each request:

```
const logger = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
};

app.use(logger);
```

This code defines a middleware function named "logger" that takes three arguments: `req` (the request object), `res` (the response object), and `next` (a function that tells Express.js to move on to the next middleware function). In this example, the logger function logs the current time, the HTTP method, and the URL of each request. The `app.use(logger)` line tells Express.js to use the "logger" middleware function for all requests.

Using template engines

Template engines help you generate HTML pages dynamically using data from your app. In Express.js, you can use template engines like EJS, Handlebars, and Pug.

Here's an example of an EJS template that displays a list of items:

```
<ul>
  <% items.forEach((item) => { %>
    <li><%= item %></li>
  <% }); %>
</ul>
```

This code defines an unordered list in HTML and uses EJS syntax to loop through an array of items and generate an HTML list item for each item.

To use this template in your Express.js app, you need to set the view engine to EJS and render the template with data from your app. Here's an example:

```
app.set('view engine', 'ejs');

app.get('/items', (req, res) => {
  const items = ['apple', 'banana', 'orange'];
  res.render('items', { items: items });
});
```

This code sets the view engine to EJS and defines a route for an "Items" page ("/items") using the `get` method. In this example, we're generating a list of items and passing it to the "items" template using the `res.render` method. The `items` variable is passed to the template as an object property.

API Reference

The Express.js API reference provides a comprehensive list of all the methods and properties available in the Express.js framework. Here are some examples of commonly used methods and properties:

`app.get()`

The `app.get()` method is used to define a route for handling GET requests. Here's an example:

```
app.get('/users', (req, res) => {
  // Handle GET request for /users
});
```

This code defines a route for handling GET requests to the "/users" URL. When a user visits the "/users" URL, the app executes the callback function and sends a response.

`app.post()`

The `app.post()` method is used to define a route for handling POST requests. Here's an example:

```
app.post('/users', (req, res) => {
  // Handle POST request for /users
});
```

This code defines a route for handling POST requests to the "/users" URL. When a user submits a form to the "/users" URL, the app executes the callback function and sends a response.

`req.params`

The `req.params` property is an object containing properties mapped to the named route “parameters”. Here's an example:

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  // Handle GET request for /users/:id
});
```

This code defines a route for handling GET requests to the `/users/:id` URL. When a user visits a URL like `/users/123`, the app extracts the value `123` from the URL and stores it in the `req.params` object. In this example, we're extracting the `id` parameter from the URL and using it to handle the request.

`res.send()`

The `res.send()` method is used to send a response to the client. Here's an example:

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

This code defines a route for handling GET requests to the "/users" URL. When a user visits the "/users" URL, the app sends the string "Hello, World!" as a response.

`res.json()`

The `res.json()` method is used to send a JSON response to the client. Here's an example:

```
app.get('/users', (req, res) => {
  const users = [{ name: 'John', age: 30 }, { name: 'Jane', age: 25 }];
  res.json(users);
});
```

This code defines a route for handling GET requests to the "/users" URL. When a user visits the "/users" URL, the app sends a JSON response containing an array of user objects.

Advanced Topics

Middleware

Middleware functions are functions that have access to the request and response objects, and the next middleware function in the application's request-response cycle. Middleware functions can be used to modify requests and responses, handle errors, and more.

Here's an example of a middleware function that logs the time of each request:

```
const logger = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
};

app.use(logger);
```

This code defines a middleware function named "logger" that takes three arguments: `req` (the request object), `res` (the response object), and `next` (a function that tells Express.js to move on to the next middleware function). In this example, the logger function logs the current time, the HTTP method, and the URL of each request. The `app.use(logger)` line tells Express.js to use the "logger" middleware function for all requests.

Error handling

Express.js provides a way to handle errors using middleware functions. Here's an example of an error handling middleware function:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

This code defines an error handling middleware function that takes four arguments: `err` (the error object), `req` (the request object), `res` (the response object), and `next` (a function that tells Express.js to move on to the next middleware function). In this example, the error handling function logs the error stack trace to the console and sends a "Something broke!" message as a response with a 500 status code.

Routing

Routing is a way to define different pages and actions in your app. In Express.js, you can use methods like `get`, `post`, `put`, and `delete` to handle different types of requests.

Here's an example of a route for a "Contact" page that uses the `post` method to handle form submissions:

```
app.post('/contact', (req, res) => {
  const name = req.body.name;
  const email = req.body.email;
  const message = req.body.message;
  // Do something with the form data
  res.send('Thanks for contacting us!');
});
```

This code defines a route for the "Contact" page ("/contact") using the `post` method. When someone submits a form on the "Contact" page, the app receives the form data in the `req.body` object. In this example, we're extracting the name, email, and message fields from the form data and doing something with them. Finally, the app sends a "Thanks for contacting us!" message as a response.

Template engines

Template engines help you generate HTML pages dynamically using data from your app. In Express.js, you can use template engines like EJS, Handlebars, and Pug.

Here's an example of an EJS template that displays a list of items:

```
<ul>
  <% items.forEach((item) => { %>
    <li><%= item %></li>
  <% }); %>
</ul>
```

This code defines an unordered list in HTML and uses EJS syntax to loop through an array of items and generate an HTML list item for each item.

To use this template in your Express.js app, you need to set the view engine to EJS and render the template with data from your app. Here's an example:

```
app.set('view engine', 'ejs');

app.get('/items', (req, res) => {
  const items = ['apple', 'banana', 'orange'];
  res.render('items', { items: items });
});
```

This code sets the view engine to EJS and defines a route for an "Items" page ("/items") using the `get` method. In this example, we're generating a list of items and passing it to the "items" template using the `res.render` method. The `items` variable is passed to the template as an object property.

Resources

Official documentation

The official Express.js documentation provides a comprehensive guide to using the framework, including installation instructions, API reference, and examples.

You can access the official documentation at the following URL: https://expressjs.com/

Community resources

There are many community resources available for learning and using Express.js, including blogs, forums, and tutorials.

Here are some examples of community resources:

- The Express.js GitHub repository: https://github.com/expressjs/express
- The Express.js subreddit: https://www.reddit.com/r/expressjs/
- The Express.js tag on Stack Overflow: https://stackoverflow.com/questions/tagged/express

Example projects

There are many example projects available that demonstrate how to use Express.js for different types of applications.

Here's an example of an Express.js project that implements a simple chat application:

```
const express = require('express');
const app = express();
const http = require('http').Server(app);
const io = require('socket.io')(http);

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', (socket) => {
  console.log('a user connected');
  socket.on('chat message', (msg) => {
    console.log('message: ' + msg);
    io.emit('chat message', msg);
  });
  socket.on('disconnect', () => {
    console.log('user disconnected');
  });
});

http.listen(3000, () => {
  console.log('listening on *:3000');
```

```
});
```

This code defines an Express.js app that serves static files from a "public" folder and defines a route for the main page ("/") that sends an HTML file. The app also uses the Socket.IO library to implement real-time communication between clients.

When a user connects to the app, the app logs a message to the console and sets up event listeners for sending and receiving chat messages. When a user sends a chat message, the app logs the message to the console and emits the message to all connected clients. When a user disconnects, the app logs a message to the console.