SD540 Server-Side Programming

Maharishi University of Management

Masters of Software Development

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

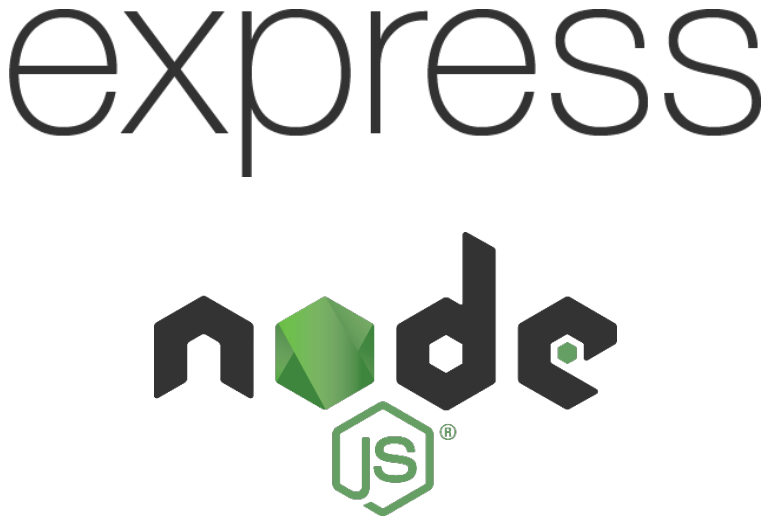
Express

Express is a web framework that wraps and extends Node core API.

Built-in parser for HTTP request params, query params, and body via middlewares (JSON and URL Encoded only).

Determining proper response headers based on data types.

Handling errors gracefully.



Layered Architecture

Router layer: façade to accept requests and forward them to controllers. You apply all middleware and verification here.

Controller layer: controller logic and API request handling.

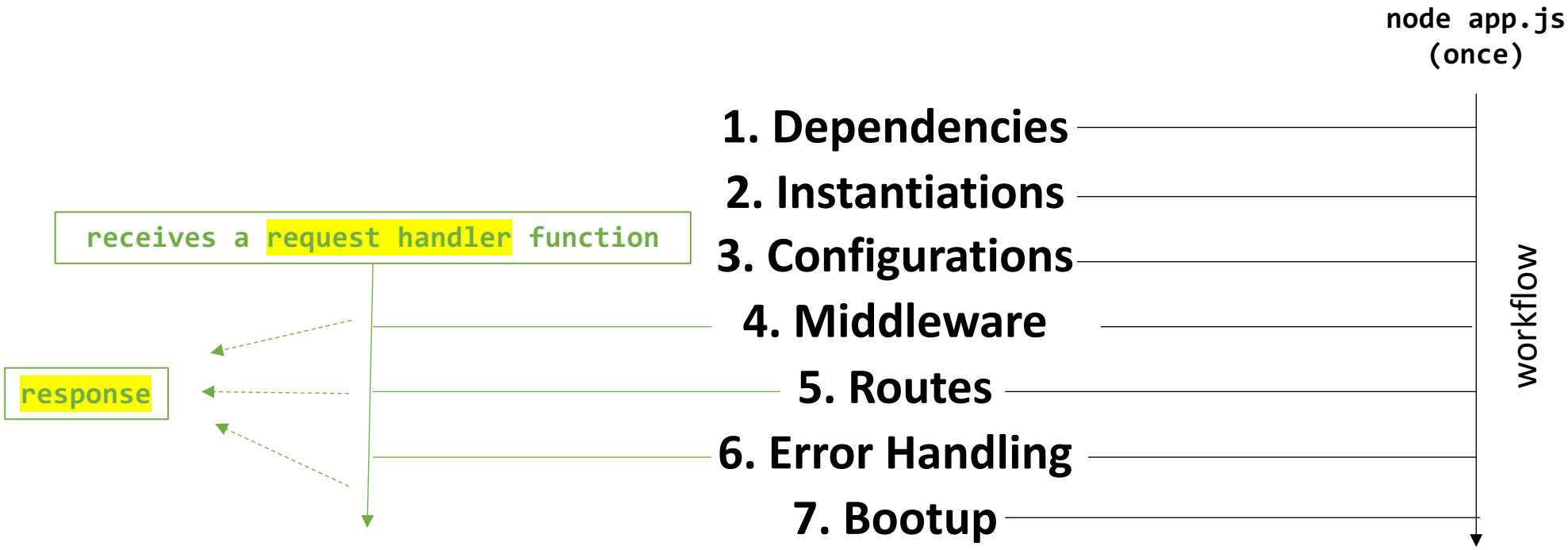
Service layer: business logic concerns, composed of modular components that handle a piece of the business logic.

Data layer: unified data access models, communicates with the service layer using data models.

Install Express with Type Definition

```
$ npm install express
$ npm install @types/express -D
```

Express Application Structure



Every request handler receives the **`request`**, **`response`**,**`next`** objects and may manipulate **`req`**, **`res`** objects, pass data to the next handler, or send the response. **next** is usually the request handler defined by your order.

Request Handlers

Request handlers are functions with the following signature:

```
const request_handler : RequestHandler<T, S, U, V> = function(req, res, next) {
        // you may read/manipulate the req
        // you may send/manipulate the response
        // you may trigger/invoke the next request handler (passing the same req,res)
};
```

Generic types:

1. **T** request params
2. **S** response body
3. **U** request body
4. **V** request query params

The `RequestHandler` Function Types

	`T(Params)`	`S(Response Body)`	`U(Request Body)`	`V(QueryParams)`
GET	*When accepting Params*	Always	**No**	*When accepting Query Params*
POST		Always	**Yes**	
PUT/PATCH		Always	**Yes**	
DELETE		Always	**No**	

Your First Express App

```
import express from 'express';
const app = express();

app.get('*', function(request, response){
        response.status(200).json({ msg: `Welcome my dears` })
});

app.listen(3000, ()=> console.log('The server is running'));
```

The `app.httpVerb()` function supports Regular Expressions of the URL patterns in a string format.
The second parameter to the app.get() method is a Request Handler function.
You may pass multiple request handler functions and they will be invoked by the scheduling order, one after the other. (only if **`next()`** is called)

How Express Works – Part 1

Express starts by reading all your middleware functions, routes, and error handlers and **registers/schedule** them **by order**, so it knows exactly the execution order.

When a request arrives, Express invokes the first request handler function that matches the Route (HTTP verb and URL) and passes the **`request`**, **`response`**, and a reference to the **`next`** request handler function in order.

How Express Works – Part 2

A request handler may perform one of these actions:

- call **`next()`** to invoke the next scheduled request handler function in order, that matches the Route, which will also receive the same **`request`**, **`response`** objects, along with a reference to the **`next`** scheduled request handler function in order.
- call **`next(something)`** to skip all upcoming request handlers and invoke the error handler function, passing the same **`request`**, **`response`** objects.
- **send out the response**, when you call **`res.json()`**, you send out the response and you cannot call **`next()`** afterward. You may need to use a **`return`** statement to stop the execution flow.

App Configurations

There are two ways to configure our application instance:

app.set()/app.get()

```
app.set('port', process.env.PORT || 3000);
const port = app.get('port);
```

app.enable()/app.disable()

```
app.enable('etag') === app.set('etag', true)
app.disable('etag') === app.set('etag', false)
```

Request Object

request.headers Example: `req.headers['authorization'];`

request.params

request.query

request.route Returns currently-matched route

request.body You need to use a middleware to parse the request body

Other Request Properties/Methods https://expressjs.com/en/5x/api.html#req

Request Object Examples

request.query

Optional

```
http://localhost:3000/search?q=nodejs&lang=eng
{"q": "nodejs", "lang": "eng"}
```

request.params

Mandatory

```
app.get('/api/:id/:name/:city',
        function(req, res) {
                res.end(req.params);
        }); // //
http://localhost:3000/api/1/Asaad/Fairfield
{ id: 1, name: 'Asaad', city: 'Fairfield' }
```

request.body

```
app.use(express.json());
app.post('/api', function(req, res){
        res.end(req.body);
});
```

Route

A route consists of the following parts:

- HTTP verb
- URL
- Params

Example:

`GET /users/:user_id`

will match any request to the following URL:

- `GET /users/1`
- `GET /users/2?name=asaad`

The the following requests will not match:

- `GET /users`
- `GET /users?name=asaad`

Response Object

`response.redirect(url)` Redirect to new path with status 302

`response.send(data)` Send response

`response.json(data)` Send JSON with proper headers

`response.sendFile(path, options, callback)`

`response.status(status)` Send status code

Other Response Properties/Methods https://expressjs.com/en/5x/api.html#res

The `response.send()` method conveniently outputs any data application thrown at it (such as strings, JavaScript objects, and even Buffers) with automatically generated proper HTTP headers (Content-Length, ETag, or Cache-Control).

Manipulating the Response Header

`res.set()` is used to set the headers of the response.

```
// single header
res.set('content-type', 'application/json');

// multiple headers can be set
res.set({
   'content-type': 'application/json',
   'content-length': '100’,
   'warning': "this course is the best course ever"
});
```

Middleware

A Middleware is a Request Handler, a useful pattern that allows developers to reuse code within their applications and even share it with others in the form of NPM modules.

The request (req) and response (res) objects are the same for the subsequent middleware, so you can add properties to them (req.user = 'Asaad') to access them later.

Use a Middleware

To use a middleware, we call the `app.use()` method which accepts:

- One **optional URL path**.
- One **request handler callback function**.

```
const middleware: RequestHandler<T, S, U, V> = function(req, res, next) {
        // ...
        return next();
}
// apply to all routes
app.use(middleware);

// apply to a specific route
app.verb('/', middleware, requestHandler);
```

Built-in Middlewares

Express comes with many built-in middlewares, some that we will use:

`express.json()`

`express.Router()`

`express.json()`

This is a built-in middleware function in Express. It parses incoming requests with JSON payloads and assigns them to **`req.body`**.

```
app.use(express.json())
```

Middleware Order

When using middleware, the order in which middleware functions are applied matters, because **this is the order in which they'll be executed**.

Useful 3rd-party middleware:

```
'morgan', '@types/morgan' // logger
'cors', '@types/cors' // accept CORS requests
'multer', '@types/multer' // upload files
'bcrypt', '@types/bcrypt' // hash and compare passwords
'jsonwebtoken', '@types/jsonwebtoken' // sign and verify JWT
'dotenv' // global environment variables via process.env
'helmet' // secure Express apps by setting HTTP response headers
```

`next()`

`next()`

Go to the next scheduled request handler function (middleware, route)

`next(something)`

Go to the Error handler

Throwing an Error

If you pass anything to the **`next()`** function, Express considers the current request as being in error and will skip any remaining non-error handling routing and middleware functions and passes the request/response to error handlers.

```
router.get('/user', request_handler);

const request_handler: RequestHandler =  async (req, res, next) => {
    try {
       // your logic here
       if(somethingWrong) throw new Error(`User Not found`)
    } catch (error) {
        next(error)
    }
  }
```

Error Handlers in Express

Define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three: `(err, req, res, next)`

```
app.use((error: Error, req: Request, res: Response<{msg: string}>, next:
NextFunction){
        console.error(err.stack);
        res.status(500).json({'msg': err.message });
});
```

IMPORTANT: You define error-handling middleware last, after other middleware and routes calls.

Extending the Error class with status code

```
export default class ErrorResponse extends Error {
  status?: number;

  constructor(message, statusCode) {
    super(message);
    this.status = statusCode;
  }

}
```