בס"ד

קורס NodeJS תשפה

Express

express היא חבילת npm שמאפשרת כתיבת server בnode בצורה קלה וגמישה. החבילה כוללת סט כלים רחב לניהול בקשות HTTP, כולל routing, קבלת פרמטרים, middlewares ועוד, תוך שמירה על קוד פשוט ונקי.

התקנה

```
npm i express
npm i -D @types/express // express type definitions
```

יצירת server בסיסי

```
import express, { Request, Response } from 'express';

const HOST = "127.0.0.1";
const PORT = 3000;

const app = express();

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

app.listen(PORT, HOST, () => {
  console.log(`App listening on http://${HOST}:${PORT}/`);
});
```

יצירת application של express נעשית באמצעות קריאה לפונקציה ()express.

לאחר יצירת הapp ניתן להוסיף לו פונקציות לניהול בקשות שונות – לדוגמא, בקטע הקוד קיימת פונקציה לניהול בקשות המתקבלות בניתוב '/' עם method של GET, ומחזירה 'Hello World!'.

כדי להרים את הapp ולגרום לו להאזין לבקשות שיתקבלו בserver, יש לקרוא לפונקציה listen שמקבלת פרמטרים של host, port וcallback שירוץ כשהapp יתחיל להאזין לבקשות.

Routing

הגדרת הroutingכוללת לכל HTTP Method וURL פונקציונליות ייחודית.

בexpress, הגדרת route תיראה כך:

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

כאשר:

- app – application קיים של express.
- **METHOD** – HTTP method באותיות קטנות (לדוגמא: get, post, put, delete).
- **PATH** – הניתוב בserver.
- **HANDLER** – פונקציה שתרוץ כאשר תגיע לשרת בקשה שתואמת את הmethod והpath שהוגדרו.

דוגמאות:

ניהול של בקשות POST בניתוב '/':

```
app.post('/', (req: Request, res: Response) => {
    res.send('Got a POST request');
});
```

ניהול של בקשות PUT בניתוב 'user':

```
app.put('/user', (req: Request, res: Response) => {
    res.send('Got a PUT request at /user');
});
```

Request Parameters

Route Parameters

כדי להגדיר route שכולל פרמטרים (כמו ID של resource בserver), נוסיף את שם הפרמטר כשלפניו ':' לניתוב. קבלת הפרמטרים בhandler היא באמצעות req.params:

```
app.get('/users/:userId/books/:bookId', (req: Request, res:
Response) => {
    const routeParams = req.params;
    const userId = routeParams.userId;
    const bookId = routeParams.bookId;

    console.log(`User: ${userId}, Book ${bookId}.`);
    res.send({userId, bookId});
});
```

לדוגמא - URL מתאים לroute:

GET http://localhost:3000/users/34/books/8989

התשובה שתחזור:

```
{ "userId": "34", "bookId": "8989" }
```

Query Parameters

פרמטרים שנשלחים על הquery (כמו פרמטרים לfilter או pagination) ניתן לקרוא בhandler באמצעות req.query:

```
app.get('/books', (req: Request, res: Response) => {
    const category = req.query.category;
    res.send({books: [], category});
});
```

לדוגמא - URL מתאים לroute:

GET http://localhost:3000/books?category=History

התשובה שתחזור:

```
{ "books": [], "category": "History" }
```

Response

אובייקט הresponse מכיל פונקציות שונות שמעדכנות את הresponse שיחזור מהAPI.

כל אחת מהפונקציות גורמות לניהול הבקשה להסתיים. ללא קריאה לאחת הפונקציות, הבקשה תישאר פתוחה ולא תיסגר – לא תוחזר תשובה לclient.

res.download()

הורדת קובץ:

```
res.download('/report-12345.pdf');
```

res.send()

שליחת הודעה כתשובה, תומך בtypes שונים של תשובות כמו string, json, array, boolean:

```
res.send({ some: 'json' });
```

ניתן לשלב עם res.status() שמעדכן את status הresponse:

```
res.status(404).send('Sorry, we cannot find that!');
```

```
res.status(500).send({ error: 'something blew up' });
```

res.redirect()

הפניה של הבקשה לכתובת אחרת:

```
res.redirect('http://example.com');
```

הstatus code של redirect יהיה 302 בדרך כלל. כדי לשנות אותו ניתן להעביר status code אחר כפרמטר ראשון לפונקציה, לפני הurl.

res.end()

מסיים את ניהול הבקשה מבלי לשלוח הודעה בbody.

Router

כדי להפריד routes שונים לפי הprefix של הurl ולקבץ routes ששייכים לאותו controller יחד, ניתן להשתמש בrouter. הrouter מתפקד כmini-app – ניתן להגדיר עליו route handlers כפי שמגדירים על הapp עצמו.

בדוגמא הבאה, ניצור router לניהול routes שכלולים בcontroller של birds:

```
import { Router } from 'express';

export const birdsRouter = Router();
birdsRouter.get('/', (req, res) => {
    res.send('Get all birds');
});

birdsRouter.get('/:id', (req, res) => {
    res.send('Get specific bird');
});

// all REST
```

לאחר יצירת הrouter, נגדיר את הroute אליו בapp:

```
app.use('/birds', birdsRouter);
```

כעת, כל הבקשות שיגיעו לשרת עם כשהprefix url הוא /birds, ינווטו לbirdsRouter וינוהלו בעזרת ה route handlers שהוגדרו עליו.

בעזרת routers, ניתן לנהל controllers שונים בשרת תוך הפרדה לוגית של resources וקיבוץ route handlers ששייכים לאותו הresource יחד.

Middlewares

בהגדרת הhandlers לניהול בקשות שמגיעות לserver, נרצה לעיתים להגדיר middlewares – פונקציות אחידות לניהול גנרי של בקשות שונות לפני או אחרי הניהול הפרטני.

הגדרת middleware שימושית לניהול פרמטרים ובדיקות התקינות שלהם, logging של בקשות ופרטים שלהן, בדיקת הרשאות, הכנת תשובה ושליחה שלה ועוד.

בexpress, middleware היא פונקציה שמקבלת שלושה פרמטרים – request, response וnext – שמייצג את הפונקציה הבאה בrequest chaining. הmiddleware יכול לקרוא את הבקשה ולבצע בה שינויים, לעדכן את הresponse, ולזמן את הפונקציה הבאה (שיכולה להיות הmiddleware הבא או הroute handler עצמו).

הוספת middleware היא באמצעות קריאה לפונקציה app.method – לניהול בקשות שמתקבלות בניתוב ספציפי וhttp method מסוים, או app.use לניהול גנרי על כל הבקשות שמגיעות לשרת:

```
app.get('/', (req, res, next: NextFunction) => {
    console.log(`[${req.method}] ${req.url}`);
    next();
});
```

פונקציית הmiddleware לעיתים קרובות תוגדר כפונקציה בקובץ utils, ותיקרא לניהול routes שונים בפרויקט:

```
export function parseUserId(req, res, next: NextFunction) {
    const userId = parseInt(req.params.userId);
    if (!userId) {
        res.status(400).end(`User id must be a number`);
        return;
    }

    req.userId = userId;
    next();
}
```

והשימוש:

```
app.use(parseUserId);
```

הmiddleware בדוגמא מוסיף משתנה של userId לrequest. כל route שיוגדר לאחר מכן על הapp, ייקרא לאחר הפעלת הmiddleware על הrequest ולכן יכיל את הפרמטר userId ויוכל לגשת אליו.

בנוסף, ניתן להגדיר פונקציית middleware גנרית שמקבלת פרמטרים, באופן הבא:

```
export function logContext(context: string) {
    return function (req, res, next) {
        console.log(`New request on context: ${context}`);
        next();
    }
}
```

והשימוש:

```
app.get('/users', logContext('usersController'));
```

צורת שימוש נוספת בmiddleware, היא זימון של פונקציית middleware אחת או יותר לפני הקריאה לroute handler, באותה קריאה לapp.method. בשעת הקריאה לroute, הmiddlewares המוגדרים ירוצו לפי הסדר כשכל אחד רץ רק כשהmiddleware הקודם קורא ל()next. לדוגמא:

```
export function logRequest(req, res, next) {
    console.log(`[${req.method}] ${req.url}`);
    next();
}
```

```
app.get('/login', logRequest, (req: Request, res: Response) => {
    if (req.query.password == "abcd") {
        res.send("Success");
    } else {
        res.send("Denied")
    }
});
```

בכל בקשה לlogin/, הmiddleware של logRequest ירוץ ראשון על הבקשה, ורק כשיקרא לnext ה route handler עצמו ירוץ כhandler הבא של הבקשה.

אם הmiddleware לא קורא לnext ובעצם מסיים את הטיפול בבקשה, היא לא תועבר לניהול ע"י הmiddlewares הבאים. לדוגמא:

```
export function isAuthorized(req, res, next) {
    if (req.userRole == "admin") {
        next();
    }

    res.status(403).end('User is not authorized to this resource');
}
```

```
usersRouter.put('/:userId', parseUserId, isAuthorized, (req, res) =>
{
    db.update(req.body.user);
    res.end();
});
```

הmiddleware isAuthorized מסיים את ניהול הבקשה ולכן במידה והuser לא מורשה לגשת לresource, היא תסתיים בשגיאה והroute handler לא ירוץ.

במידה והוגדרו יותר מmiddleware או route handler אחד לאותו route, הם ייקראו לפי סדר הגדרתם.

ניתן להגדיר middleware גם ברמת הrouter, כך שירוצו על כל הroutes שמוגדרים בrouter או על חלקם. אפשרות זו שימושית להגדרת middlewares על controller ספציפי כך שכל הroutes תחתיו ינוהלו על ידו, לדוגמא הגדרת הרשאות לפי controller – ניהול הauthorization לפי רמת ההרשאה של המבקש, ובדיקה אם הוא מורשה לresource הספציפי שמנוהל ע"י הcontroller.

Built-In Middlewares

בexpress ישנם middlewares שהם built-in ושימושיים להגדרות שונות על הexpress app.

לדוגמא, כדי לקרוא את הbody של הrequest, צריך לקבל את הbody כstream, ולהמיר אותו מjson לjavascript object. פעולה זו קיימת כbuilt-in middleware בexpress – express.json(). רק לאחר זימון הmiddleware נוכל לקרוא לreq.body בroute handler ולקרוא את תוכן הבקשה:

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

app.post('/products', (req, res) => {
    const product = req.body as Product;
    // handle the create opration
});
```

Built-in middlewares נוספים:

- express.static() – משמש להנגשת קבצים סטטיים מהשרת.
- express.urlencoded() – קורא request body שנשלח בפורמט של urlencoded.

Error Handling Middlewares

Express כולל ניהול שגיאות by default, כך שאם נזרקות שגיאות בroute handlers הן תחזורנה לuser עם הstatus code המתאים והserver לא ייפול. ניתן להגדיר middleware שתופס את השגיאות הללו ומנהל אותן בהתאמה אישית.

הגדרת הmiddleware היא באמצעות פונקציה שמקבלת ארבעה פרמטרים – כאשר הראשון הוא הerr שנתפס, והאחרים הם req, res וnext כרגיל:

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

הerror handling middleware תופס שגיאות שנזרקות בhandlers שהוגדרו לפניו, ולכן כדי לנהל שגיאות באופן גלובאלי בapp יש לזמן אותו אחרון.

ניתן לכתוב error handling middleware ספציפי לrouter מסוים, ולקרוא לו לאחר הגדרת הroutes בrouter. במקרה כזה השגיאה תיפול בmiddleware הפנימי ותנוהל בו, ולא תגיע לחיצוני גם אם קיים.

ניהול שגיאות נכון בserver הוא קריטי כדי לתעד את השגיאות, לנהל אותן ולהחזיר לuser כולל ה status code והstatus message המתאימים. באמצעות שימוש בerror handling middleware ניתן למנוע שגיאות לא מנוהלות והחזרה של מידע לא מותאם לuser – מפורט מידי או לא אינפורמטיבי מספיק.

Third-Party Middlewares

ישנם middlewares שימושיים לפעולות נפוצות שניתן להתקין באמצעות חבילות npm ולהשתמש על הexpress app.

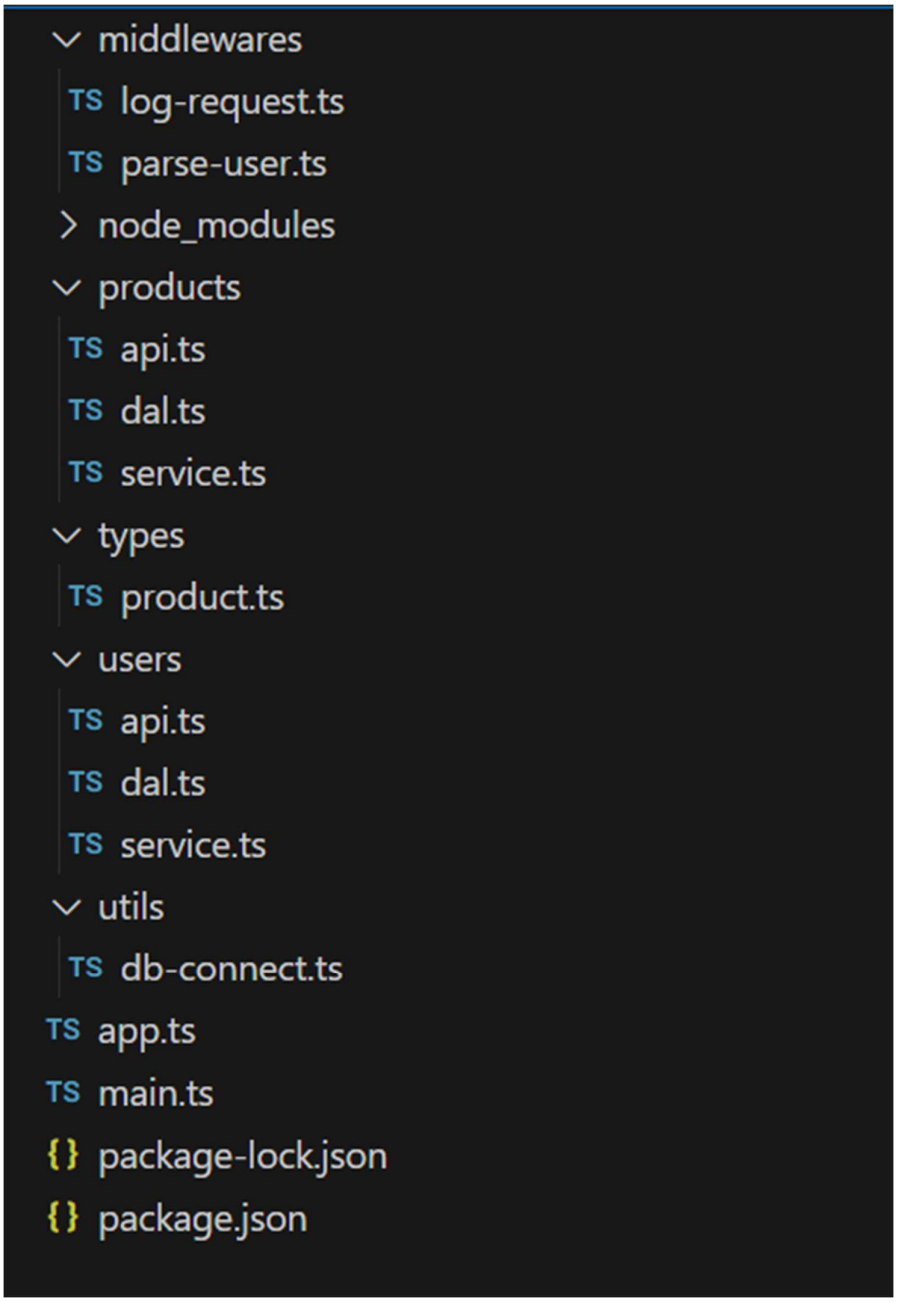
דוגמא נפוצה לmiddleware כזה היא cookie-parser לחילוץ cookies מתוך הrequest.

Third-party middlewares נוספים: connect-timout, cookie-session, cors.

מבנה פרויקט express

בכתיבת פרויקט nodejs עם typescript, הגדרת הserver עם express דורשת ניהול מושכל של מבנה התיקיות, הקבצים והשכבות. דרך נפוצה לחלוקה כזו היא לפי resources – יצירת תיקיה לכל resource, כאשר תחת התיקיה יושבים קבצים לapi, service, dal. חוץ מתיקיות הresources, ניתן להגדיר תיקיות שונות לtypes (לעיתים יושב תחת כל resource), consts (כנ"ל), middlewares וutilities.

מבנה התיקיות בפרויקט לדוגמא:



קונבנציה נוספת שכדאי להיצמד אליה, היא שימוש בclasses בלבד – ללא קבצים שכוללים קוד להרצה או פונקציות בודדות. כך, קובץ app לדוגמא נראה כך:

```
import express from 'express';
import DbConnect from './utils/db-connect';
import ProductsDal from './products/dal';
import ProductsService from './products/service;
import ProductApi from './products/api';

const PORT = 5060;

export default class App {
    private app: express.Application;
    private productsApi: ProductApi;
    constructor() {}

    public async init() {
        this.app = express();
        this.app.use(express.json());

        const dbConn = new DbConnect();
        await dbConn.init();

        const productsDal = new ProductsDal(dbConn);
        await productsDal.init();
        const productsService = new ProductsService(productsDal);
        this.productsApi = new ProductApi(productsService);

        this.setRoutes();
    }

    private async setRoutes() {
        this.productsApi.setRoutes();
        this.app.use('/api/products', this.productsApi.router);

        this.app.listen(PORT, () => {
            console.log(`listening on port ${PORT}`);
        });
    }
}
```

כשקובץ הRouter:

```
import { Request, Response, Router } from "express";
import ProductsService from "./service";
import { Product } from "../types/product";
import { parseUser } from "../middlewares/parse-user";

export default class ProductApi {
    public router: Router;
    constructor(private productsService: ProductsService) {
        this.router = Router();
    }

    public setRoutes() {
        this.router.use(parseUser);
        this.router.get('/', async (req, res) => {
            const products = await
this.productsService.getAllProducts();
            res.send(products);
        });

        // all REST
    }
}
```

כדי להריץ את קובץ הapp, ניצור קובץ main שיוצר מופע של classה ומריץ אותו. לדוגמא:

```
import App from "./app"

(async () => {
    const app = new App();
    await app.init()
})();
```

כעת, בהרצת קובץ הmain הapp יעלה עם כל הrouters המוגדרים עליו, וניתן יהיה לשלוח בקשות לserver ולקבל תשובות מהroute המתאים.