

# NodeJS

Express

Middleware

Express & Typescript

MySQL2

# pnpm vs npm

pnpm תחליף לnpm

npm יוצר תיקית node\_modules

ומוריד לתוכה את כל הספריות שביקשנו בdependencies

pnpm יוצר תיקית node\_modules

ומוריד לתיקיה ראשית במחשב שלנו את הספריות

ובתיקיה node\_modules רק מצביע לתיקיה הראשית עם symbolic links

# התקנה של pnpm

## Using npm

We provide two packages of pnpm CLI, `pnpm` and `@pnpm/exe`.

- `pnpm` is a ordinary version of pnpm, which needs Node.js to run.
- `@pnpm/exe` is packaged with Node.js into an executable, so it may be used on a system with no Node.js installed.

```
npm install -g pnpm
```

התקנה פעם אחת למחשב:

אנחנו מתקינים ספרייה באופן גלובלי - כך שתהיה נגישה בטרמינל מכל מקום במחשב.

```
npm i -g pnpm
```

```
sudo npm i -g pnpm
```

במאק - אם קיבלתם הודעה שאין הרשאות - תוסיפו `sudo`

# שימוש בpnpm

הוספת מודול:

```
pnpm add express
```

```
pnpm add nodemon -D
```

הסרת מודול:

```
pnpm remove express
```

הרצת סקריפטים:

```
pnpm test  
pnpm start
```

# ניצור תיקית src וקובץ index.js בתוכה:

```
{
  "name": "lec3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "watch": "nodemon src/index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^3.0.2"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```



pnpm watch

# הפסקה של אפליקציה שרצה על פורט מסוים: (לפעמים שוכחים תוכנה שרצה על פורט מסויים)

```
pnpx kill-port 8080
```



pnpx הורדה מnpm והרצה של החבילה kill-port.

<https://www.npmjs.com/package/kill-port>

# פרוייקט express חדש:

```
const express = require('express');  
  
const app = express();  
  
app.get("/", (req, res) => {  
  res.json({ message: "Hello, World" });  
})  
  
app.listen(8080)
```

# מודולריות - routes

routes

users.js

קובץ אחד מטפל בusers  
בהמשך ניצור קובץ נוסף שמטפל בcards

```
//~ import {Router} from 'express';  
const { Router } = require('express');  
  
// create a Router object:  
const usersRouter = Router();  
  
//router.get("/") ~ app.get("/")  
usersRouter.get("/", (req, res) => {  
  res.json({ "users": [] });  
})  
  
module.exports = {usersRouter};
```



# מודולריות - routes

src/index.js

```
const express = require('express');  
const { usersRouter } = require('./routes/users');  
  
const app = express();  
  
app.use(express.json());  
app.use("/api/v1/users", usersRouter);  
  
app.listen(8080);
```

הראוטר של users

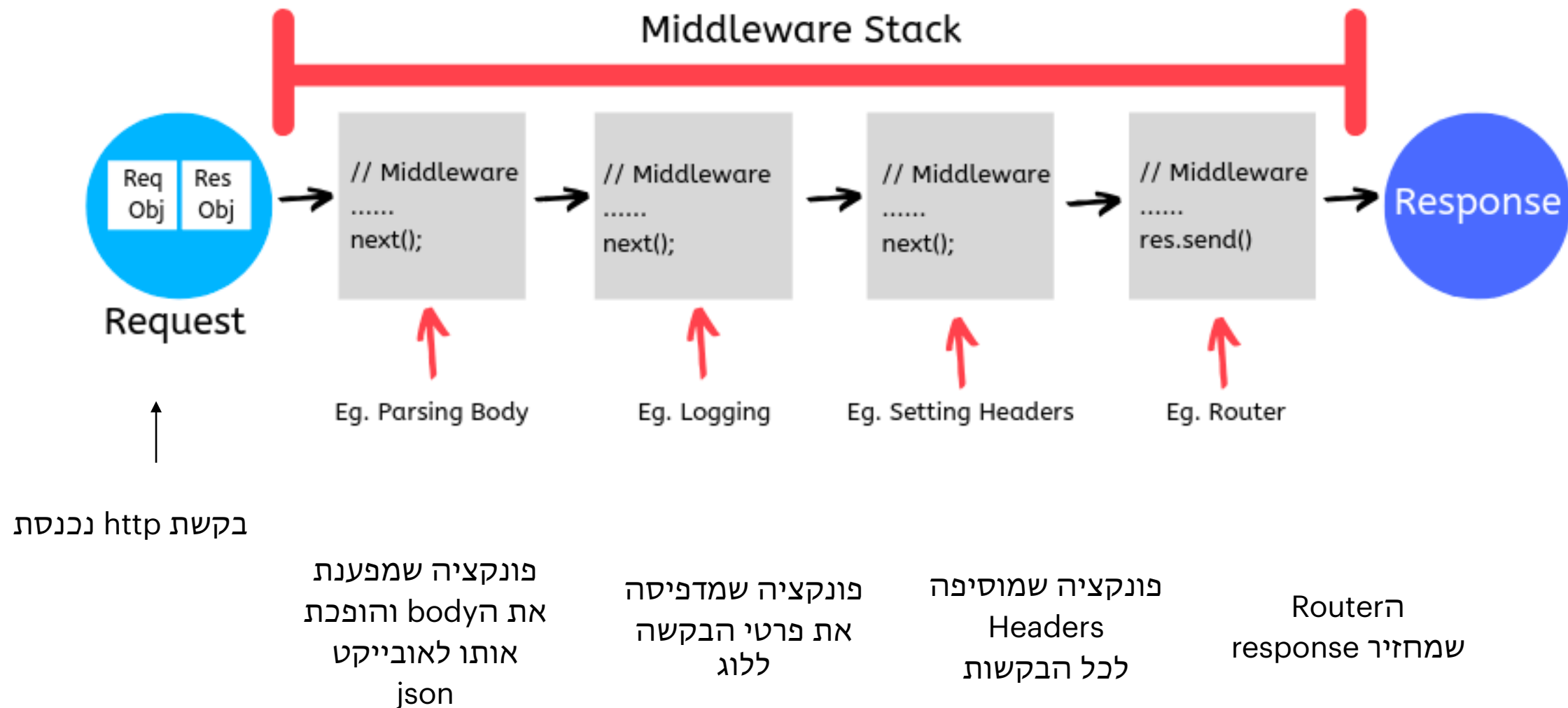
יופעל על כל נתיב שמתחיל ב/api/v1/users

## REST API Versioning

/v1/employees	Feature Flags
/v2/employees	Code structuring
.	Cleanup
.	
/v[n]/employees	

# Middleware Express

שרשרת של פונקציות שמקבלות את הבקשה - אחת אחרי השניה.



# Middleware בסיסי - דוגמא ראשונה:

src/middleware/logger.js

```
const logger = (req, res, next) => {  
  console.log(req.method, req.url);  
  
  //continue to next function in the chain.  
  next();  
}  
  
module.exports = { logger }
```

בExpress נקרא ל next() כדי להמשיך את השרשרת.

src/index.js

```
app.use(express.json());  
app.use(logger);  
app.use("/api/v1/users", usersRouter);
```

אחרת נשלח res.send()

כדי להחזיר תשובה ללקוח - ולעצור את השרשרת.

# Middleware בסיסי - דוגמא שנייה:

src/middleware/not-found.js

```
// The last middleware in the chain:
const notFound = (req, res, next) => {
  res.status(404).json({ message: "Not Found" })
}

module.exports = { notFound };
```

src/index.js

```
app.use(express.json());
app.use(logger);
app.use("/api/v1/users", usersRouter);
app.use(notFound);
```

אם הבקשה מתחילה ב/api/v1/users  
וגם ממשיכה ב/  
הבקשה מגיעה לusersRouter

usersRouter  
לא קורא לnext()

# שימוש בספריה chalk

## pnpm add chalk

בעיה: הספריה דורשת שימוש בES Modules  
מערכת המודולציה שמובנית בJS

NodeJS

משתמשת בברירת מחדל בCommonJS modules  
(מערכת שקיימת לפני שהיתה מודלציה בJS)

```
//require => commonJS modules (NodeJS Built modulation system)
const chalk = require('chalk');

const logger = (req, res, next) => {
  console.log(req.method, req.url);

  //continue to next function in the chain.
  next();
}

module.exports = { logger }
```

פתרון: נחליף את מערכת המודולים בפרוייקט - במקום להשתמש בrequire  
בפרוייקט חדש - נשתמש בimport/export  
בפרוייקט קיים - יש פרוייקטים שעדיין משתמשים בcommonJS

```
{  
  "type": "module",  
  "name": "lec3",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "watch": "nodemon src/index.js",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "nodemon": "^3.0.2"  
  },  
  "dependencies": {  
    "chalk": "^5.3.0",  
    "express": "^4.18.2"  
  }  
}
```

שינוי ברירת המחדל לES modules

# מעבר לimport/export

logger.js

```
//require => commonJS modules (NodeJS Built modulation system)
```

```
import chalk from 'chalk';
```

```
const logger = (req, res, next) => {  
  console.log(req.method, req.url);
```

```
  //continue to next function in the chain.  
  next();  
}
```

```
export { logger }
```

not-found.js

```
// The last middleware in the chain:
```

```
const notFound = (req, res, next) => {  
  res.status(404).json({ message: "Not Found" })  
}
```

```
export { notFound };
```

users.js

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

```
// create a Router object:
```

```
const usersRouter = Router();
```

```
//app.use("/api/v1/users", usersRouter);  
/*
```

```
  /api/v1/users/  
*/
```

```
usersRouter.get("/", (req, res) => {  
  res.json({ "users": [] });  
})
```

```
export { usersRouter };
```

# מעבר לimport/export

index.js

```
import express, { json } from 'express';  
import { usersRouter } from './routes/users.js';  
import { logger } from './middleware/logger.js';  
import { notFound } from './middleware/not-found.js';
```

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

```
// middleware chain:
```

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

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

```
app.use("/api/v1/users", usersRouter);
```

```
app.use(notFound);
```

```
app.listen(8080);
```

שימו לב ל.js. בייבוא של המודולים של JS



# שימוש בספרייה chalk

<https://www.npmjs.com/package/chalk>

logger.js

```
//require => commonJS modules (NodeJS Built modulation system)
import chalk from 'chalk';

const logger = (req, res, next) => {
  console.log(chalk.blue(req.method, req.url));

  //continue to next function in the chain.
  next();
}

export { logger }
```

# שימוש בספרייה chalk

<https://www.npmjs.com/package/chalk>

```
import chalk from 'chalk';

const log = console.log;

// Combine styled and normal strings
log(chalk.blue('Hello') + ' World' + chalk.red('!'));

// Compose multiple styles using the chainable API
log(chalk.blue.bgRed.bold('Hello world!'));

// Pass in multiple arguments
log(chalk.blue('Hello', 'World!', 'Foo', 'bar', 'biz', 'baz'));

// Nest styles
log(chalk.red('Hello', chalk.underline.bgBlue('world') + '!'));

// Nest styles of the same type even (color, underline, background)
log(chalk.green(
  'I am a green line ' +
  chalk.blue.underline.bold('with a blue substring') +
  ' that becomes green again!'
));
```

# עבודה עם Typescript

```
const fn = (name)=>{  
  name.slice(-1)  
}
```



fn(1900) →

# מעבר לtypescript בפרוייקט:

```
pnpm add typescript -D
```

התקנה של typescript בפרוייקט

כשסיימנו את הפרוייקט - נהפוך את הקוד לJS  
בזמן פיתוח עבדנו עם השלמה אוטומטית.

בזמן אמת - הפרוייקט כבר כתוב (וכתוב בלי טעויות של types)  
נהפוך אותו לJS תקין - יותר יעיל.

בprod אין צורך בשלב של קומפילציה והרצה.  
אפשר לעבור ישירות להרצה - יותר יעיל.

```
pnpm add tsx -D
```

מריץ קבצי ts:

(מאחורי הקלעים מתרגם לJS ומריץ את JS)

# מעבר לtypescript בפרוייקט:

```
pnpm add @types/express -D
```

Express ספרייה שכתובה בJS

ההתקנה מוסיפה תמיכה בtypescript לספרייה express

```
pnpm add @types/node -D
```

NodeJS מיועדת במקור לJS

ההתקנה מוסיפה תמיכה בtypescript לNode

```
pnpm add typescript tsx @types/express @types/node -D
```

# שינויים בפרוייקט:

סיומת של קבצים: .ts

נסיר את "type": "module" מהקובץ package.json

בטרמינל: tsc --init

נסיר את nodemon


pnpm remove nodemon



"watch": "tsx watch src/index.ts"


נמחק את node\_modules ונתקין שוב



pnpm i

# שינוי סיומת ל.ts

▼  middleware

-  logger.ts
-  not-found.ts

▼  routes

-  users.ts
-  index.ts

TS index.ts ×

src > TS index.ts >  app

```
1 import express, { json } from 'express';
2 import { usersRouter } from './routes/users';
3 import { logger } from './middleware/logger';
4 import { notFound } from './middleware/not-found';
```

↑  
להסיר .js

# שינויים בpackage.json

נסיר את "type": "module"

נתקן את הסקריפט watch

נשנה את השם של הקובץ הראשי  
לsrc/index.ts

```
{
  "name": "lec3",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.ts",
  "scripts": {
    "watch": "tsx watch src/index.ts",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/express": "^4.17.21",
    "@types/node": "^20.10.3",
    "nodemon": "^3.0.2",
    "tsx": "^4.6.2",
    "typescript": "^5.3.2"
  },
  "dependencies": {
    "chalk": "^5.3.0",
    "express": "^4.18.2"
  }
}
```



# קובץ הגדרות ל Typescript בפרוייקט

בטרמינל:  
`tsc --init`

יוצר קובץ הגדרות עבור typescript בפרוייקט:

שם הקובץ שנוצר אחרי הפקודה:

`tsconfig.json`

# קובץ הגדרות ל Typescript בפרוייקט

הגדרות לקומפילציה מ TS ל JS

tsconfig.json

```
{
```

```
  "compilerOptions": {
```

```
    "target": "es2016",
```

```
    "module": "commonjs",
```

```
    "esModuleInterop": true,
```

```
    "forceConsistentCasingInFileNames": true,
```

```
    "strict": false,
```

```
    "skipLibCheck": true
```

```
  }
```

```
}
```

הגירסה של JS אחרי קומפילציה

JS אחרי קומפילציה יכול  
מודלים של Node

מאפשר לעבוד עם מודלים של ES

יש חשיבות לאותיות קטנות/גדולות  
בשמות הקבצים

חובה שכל קובץ לא יכול פונקציות או משתנים ללא הגדרת טיפוס

כדי לזרז קומפילציה:  
אין צורך לבדוק ספריות npm  
(ההנחה - הספריות תקינות).

# כסף עם Typescript

```
//require => commonJS modules (NodeJS Built modulation system)  
import chalk from "chalk";
```

```
import { Request, Response, NextFunction } from "express";
```

טיפוס של כל פרמטר בפונקציה

↓ ↓ ↓

```
const logger = (req: Request, res: Response, next: NextFunction) => {  
  console.log(chalk.blue(req.method, req.url));
```

```
  //continue to next function in the chain.  
  next();  
};
```

```
export { logger };
```

# כסף עם Typescript

במקום לציין טיפוס לכל פרמטר בפונקציה:  
אפשר לציין טיפוס לפונקציה עצמה:

```
//require => commonJS modules (NodeJS Built modulation system)
import chalk from "chalk";
import { RequestHandler } from "express";

const logger: RequestHandler = (req, res, next) => {
  console.log(chalk.blue(req.method, req.url));

  //continue to next function in the chain.
  next();
};

export { logger };
```

# תזכורות לגבי ts

אם יש למשתנה ערך - אין חובה לציין טיפוס

```
const x:string = "";
```

```
const x = "";
```

```
const what = 4;
```

```
const bi = { message: "bi" };
```

```
interface Person{  
  name: string  
  lastName: string  
}
```

```
const p: Person = {  
  name: "",  
  lastName: ""  
}
```


הגדרת טיפוס לPerson

דרישות מPerson

סכמה של Person

הגדרת אובייקט מסוג Person  
חייב לעמוד בדרישה של הממשק.


# תזכורות לגבי ts

```
5  interface Person{  
6      name: string  
7      lastName: string  
8  }  
9  
10 const fn = (p:Person)=>{  
11      p.|
```

 lastName

 name

# תזכורות לגבי ts

```
5  type Person = {  
6    name: string;  
7    lastName: string;  
8  };  
9  
10 const fn = (p:Person)=>{  
11   p.|
```

 lastName

 name

דרך נוספת להגדיר טיפוס  
לאובייקט שאנחנו יצרנו:

# תזכורות לגבי ts

```
type myFunction = (name: string) => void;  
  
const fn: myFunction = (n) => {  
  console.log(n.slice(-1));  
};
```

במקרה הנוכחי - היה יותר קל  
לכתוב:

```
const fn = (n:string) => {  
  console.log(n.slice(-1));  
};
```

הגדרת טיפוס לפונקציה:  
הפונקציה מקבלת פרמטר אחד  
- מחרוזת

הפונקציה לא מחזירה ערך -  
void



# תזכורות לגבי ts

נגדיר טיפוס לפונקציה:

כשיש הרבה פרמטרים:  
או שרוצים לעשות שימוש חוזר  
בהגדרה של פרמטרים  
לפונקציה

```
type myFunction = (name: string, lastName: string, age: number, id: string)=>void  
  
const fn:myFunction = (name, lastName, age) => {  
  console.log(name.slice(-1));  
};
```

# עבודה עם mysql

pnpm add mysql2

```
import { Router } from "express";  
import mysql2 from "mysql2";  
  
const connection = mysql2.createConnection({  
  //host, port, user, password, database  
  host: "localhost",  
  port: 3306,  
  user: "root",  
  password: "1qazxsw2",  
  database: "sakila",  
});  
  
const usersRouter = Router();  
  
usersRouter.get("/", (req, res) => {  
  res.json({ users: [] });  
});  
  
export { usersRouter };
```

# עבודה עם mysql

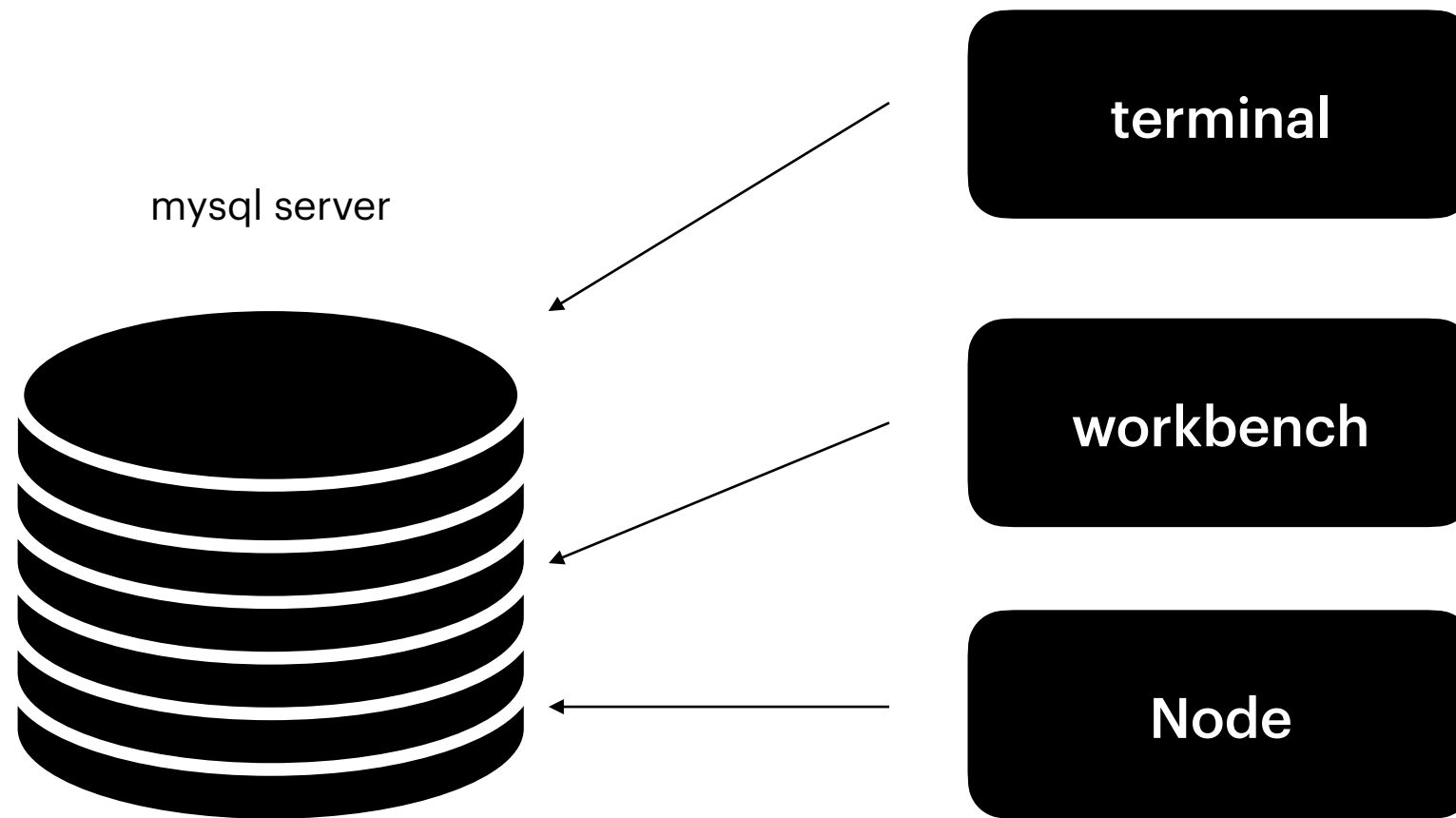
```
import mysql2 from "mysql2";

const connection = mysql2.createConnection({
  //host, port, user, password, database
  host: "localhost",
  port: 3306,
  user: "root",
  password: "1qazxsw2",
  database: "sakila",
});
```

```
usersRouter.get("/", (req, res) => {
  //sql query to fetch all customers
  connection.query("SELECT * FROM customer", (err, result) => {
    if (err) {
      res.status(500).json(err); ← 500 - internal server error
    } else {
      res.json(result);
    }
  });
});
```

<http://localhost:8080/api/v1/users>

# עבודה עם mysql



ההתקנה מתקינה לנו גם server וגם workbench

# עבודה עם mysql

```
usersRouter.post("/", (req, res) => {                                בקשת POST להוספת customer
  const { firstName, lastName, email } = req.body;

  const sql = `
    INSERT INTO customer(
      store_id, first_name, last_name, email, address_id
    )
    VALUES (1, '${firstName}', '${lastName}', '${email}', 1)
  `;
  //INSERT INTO customer(...) VALUES(...)
  connection.query(sql, (err, result) => {
    if (err) {
      res.status(500).json(err);
    } else {
      res.json(result);
    }
  });
});
```

```
### Add a user:                בדיקה:
POST http://localhost:8080/api/v1/users
Content-Type: application/json

{
  "firstName": "Bruce",
  "lastName": "Wayne",
  "email": "Bruce@batcave.com"
}
```

# שיעורי בית:

## צרו פרוייקט חדש Typescript חדש - לפי ההנחיות הבאות:

1) צרו תיקיה חדשה בשם lec3hw  
ופתחו את התיקיה עם VSCode  
בטרמינל של VSCode יש להריץ את הפקודה

2) בטרמינל של VSCode הריצו את הפקודה הבאה (אין צורך ב-y-):

```
pnpm init
```

3) בטרמינל של VSCode התקינו את הספריות הבאות:

```
pnpm add express mysql2 chalk
```

עד כאן - יצרנו פרוייקט רגיל בnode (ממש כמו בJS)

המשך בעמוד הבא:

# שיעורי בית - המשך מעמוד קודם:

## צרו פרוייקט חדש Typescript חדש - לפי ההנחיות הבאות:

4) בטרמינל של VSCode התקינו את הספריות הבאות:

```
pnpm add typescript tsx @types/express @types/node -D
```

כאן אנחנו בעצם מתקינים ספריות לעבודה עם typescript:

פירוט לגבי הספריות בשלב 4:

typescript

התקנה של typescript לפרוייקט

tsx

ספריה שמאפשרת לנו להריץ קבצי typescript

@types/express

ההתקנה מוסיפה תמיכה בtypescript לספריה express

@types/node

ההתקנה מוסיפה תמיכה בtypescript לNode

5) בטרמינל יש להריץ את הפקודה הבאה:

```
tsc --init
```

הפקודה יוצרת קובץ הגדרות לtypescript בפרוייקט.  
הקובץ שנוצר tsconfig.json

# שיעורי בית - המשך מעמוד קודם:

## צרו פרוייקט חדש Typescript חדש - לפי ההנחיות הבאות:

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": false,
    "skipLibCheck": true
  }
}
```

(6) מחקו את כל ההערות בקובץ tsconfig.json ושנו את התכונה strict לערך של false

(7) צרו תיקיה בשם src  
- בתיקיה צרו קובץ חדש בשם index.ts שמדפיס "lec3hw" לconsole.

(8) ערכו את הקובץ package.json הוסיפו סקריפט לwatch:

```
"watch": "tsx watch src/index.ts"
```



# שיעורי בית - Express:

צרו אובייקט של Express שמאזין בפורט 8081

צרו Router (בקובץ נפרד כמו שלמדנו בכיתה - ראו קובץ users.ts) שיטפל בכתובות הבאות

localhost: 8081/api/v1/films

localhost: 8081/api/v1/films/drama

יש להחזיר מידע תוך שימוש בMySQL  
ומסד הנתונים sakila

בכל בקשה לאחד הדפים הללו בדפדפו יש להחזיר JSON.

צרו Middleware עבור Not Found

אם הלקוח מנסה לנווט לדף שלא הוגדר יש להחזיר בתגובה אובייקט JSON

{message: "not found"}

יש לציין את הסטטוס - 404

ודאו שהתגובה נשלחת עם HTTP Header של Content-Type:Application/json

(כדי לראות את הHeader - יש להסתכל בpostman או בchrome בnetwork tab)

(אפשר לראות את הHeader גם בתגובה של התוסף rest client)

יש להוסיף Middleware שכותב ללוג את הכתובת והמתודה של הבקשות שנכנסו.  
השתמשו בספריה chalk כדי לצבוע את הטקסט בצבעים.

בדקו את הAPI עם Postman

בדקו את הAPI עם תוסף rest client לVSCode  
(צרו קובץ requests.http)

עבודה מהנה :-D