# **Node.js**

**Node.js**

סביבת הרצה של JavaScript מחוץ לדפדפן. לרוב נמצאת בשימוש עבור מערכות Backend.

אתר: <https://nodejs.org/en>

**NPM**

Node Package Manager – המערכת של Node.js לשיתוף ספריות קוד פתוח ע"י מתכנתים מכל העולם.

**Package**

פרויקט אחד שנכתב ב-Node.js. Package בנוי מקבצי js. כל קובץ js. נקרא Module.

**Module**

זהו קובץ js. אחד ויחיד המרכיב פרויקט.

**package.json**

זהו קובץ קונפיגורציה המתאר פרויקט Node.js אחד ויחיד (Package אחד)

מכיל את שם הפרויקט, גרסה, תאור, במה הפרויקט תלוי (פרויקטים אחרים).

בכל פרויקט Node.js חובה קובץ package.json אחד ויחיד.

פקודה ליצירת קובץ כזה: npm init או npm init -y עבור הגדרות ברירת מחדל

**Nodemon**

זהו קיצור ל- Node Monitor – זו ספרייה שמתקינים פעם אחת ברמת המחשב שמסוגלת להריץ פרויקט Node.js כולל Watch על קבצי הפרויקט (האזנה). כל שינוי יגרום להרצה מיידית נוספת של הפרויקט.

התקנה: npm i -g nodemon

הרצה: nodemon <file-to-run.js>

**global**

זהו האובייקט הראשי של Node.js.

כאשר אנו נמצאים במסגרת דפדפן, האובייקט הראשי הוא ה-window שמכיל גם document, alerts וכו'.

אין את כל אלו ב-Node.js.

האובייקט הגלובלי נקרא global ומכיל תשתיות בסיסיות של Node.js.

לדוגמה global.process.env

לדוגמה global.setInterval

כמו בצד ה-Front ששם לא היה צריך לרשום window, אפשר לא לרשום את המילה global.

**סוגי Modules**

ישנם שלושה סוגים של Modules שניתן לבנות / להשתמש:

1. Custom Module – זהו Module שאנו בונים בפרויקט – קובץ js. (או ts.) אחד ויחיד.

ניתן להחצין משתנים / פונקציות / מחלקות וכו' מ-Module לצורך שימוש במודולים אחרים.

Module תמיד סגור ב-IIFE אוטומטית (זה מה שתוכנת Node.exe שהינה Node.js עושה לכל קובץ שלכם)

1. Built-In Module – זהו Module מובנה הקיים כחלק מהתקנת Node.js.
2. npm Module – זהו Module שהותקן משרתי npm (אפשר גם משרתים אחרים)

**Node.js Versioning System**

פורמט הגרסאות של פרויקטים ב-Node.js

כל גרסה מחולקת לשלושה חלקים:

מספר ימני – נקרא patch – מתאר גרסה המכילה שינויים מינוריים בלבד ותיקוני באגים.

מספר אמצעי – נקרא minor – מתאר גרסה המכילה שינויים ותוספות שלא שוברות מערכות קיימות שעובדות עם הגרסה הקודמת. כשמעלים מספר זה, מאפסים את מספר ה-patch.

מספר שמאלי – נקרא major – מתאר גרסה המכילה שינויים ותוספות שכן עלולים לשבור מערכות קיימות שעובדות עם הגרסה הקודמת. כשמעלים מספר זה, מאפסים את מספר ה-patch ואת מספר ה-minor.

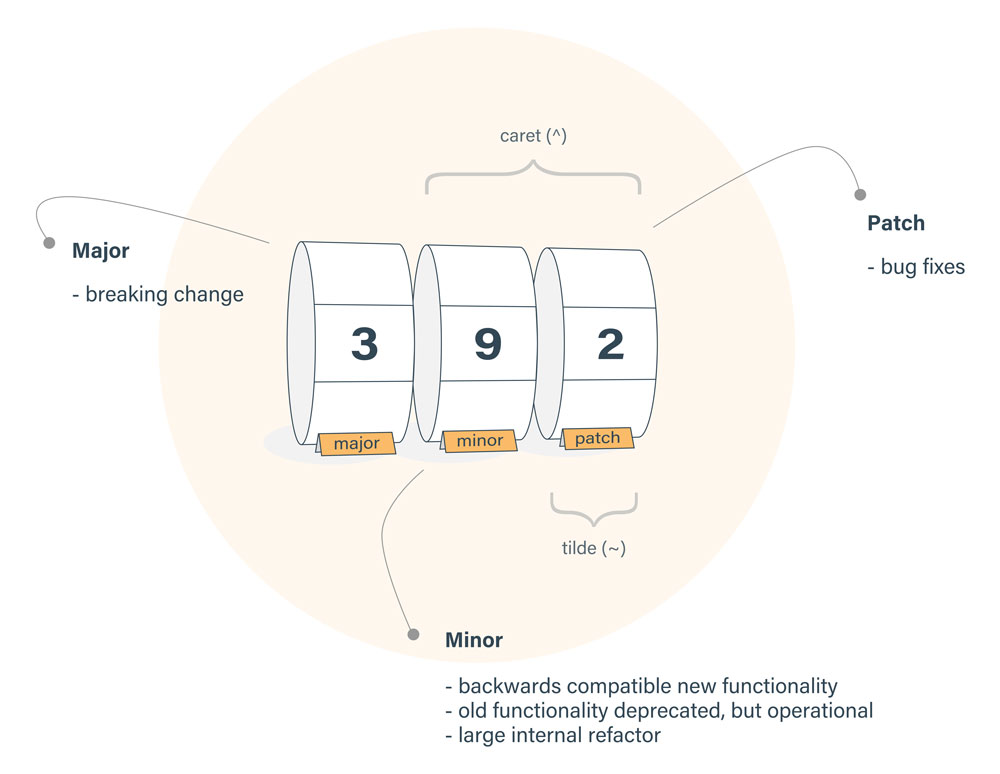
Wild Cards

~ אם יתבצע עדכון – יש להביא את הגרסה החדשה ביותר מבחינת ה-Patch.

^ אם יתבצע עדכון – יש להביא את הגרסה החדשה ביותר מבחינת ה-Minor ו-Patch

\* אם יתבצע עדכון – יש להביא את הגרסה העדכנית ביותר (Major, Minor ו-Patch)

אם לא מגדירים שום Wild Card – אם יתבצע עדכון תתקבל אותה הגרסה בדיוק.



**העלאת Package שאנו בונים לשרתי NPM**

יש שני סוגי Packages:

1. לוקלי – ניתן לשימוש מתוך פרויקט אחר.

לדוגמה, התקנת npm i axios לתוך פרויקט React שאנו בונים, ואז שימוש בספריית axios מתוך הפרויקט הזה.

1. גלובלי – ניתן להרצה ע"י פקודת CLI (Command-Line Interface) – פקודות בטרמינל.

לדוגמה, התקנת npm i -g northwind-back-end ואז שימוש בספרייה הזו ע"י הרצת הפקודה בטרמינל: northwind

בספרייה גלובלית, כל קובץ JavaScript חייב להכיל כשורה ראשונה את הפקודה הבאה:

#!/usr/bin/env node

פקודה זו מציינת שכאשר הקובץ יורץ מטרמינל, על מערכת ההפעלה להריץ אותו ע"י Node.js.

Local Package:

1. Install:

npm i display-time-by-assaf-70

2. Use:

a. Open some node.js project

b. Use functions from the library:

const display = require("display-time-by-assaf-70");

display.displayTime();

Global Package:

1. Install:

npm i -g something-beautiful-by-assaf-70

2. Use:

a. Open termial

b. Write the specific word to run it

make-my-day

עבודה עם TypeScript ב-Node.js

1. התקנת npm i -g typescript
2. התקנת npm i -g ts-node
3. בניית פרויקט Node.js הבנוי בצורה ספציפית (מצורף מסמך המראה זאת)
4. התקנה: npm i @types/node -D

**הבדל בין dependencies לבין devDependencies**

For native JavaScript:

npm i colors

const colors = require("colors");

console.log(colors.green("cool"));

-------------------------------

For TypeScript:

npm i colors --> dependencies

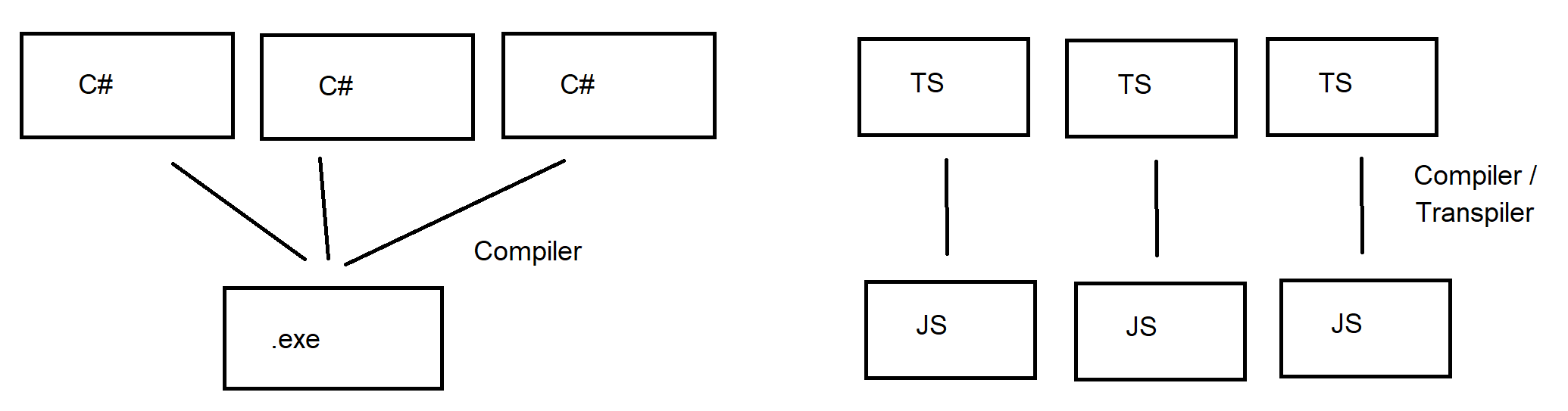
npm i @types/colors --save-dev --> devDependencies

npm i @types/colors -D --> devDependencies

import colors from "color";

console.log(colors.green("cool"));

**הבדל בין Compiler ל-Transpiler**



**הפיכת פרויקט TypeScript לקוד JavaScript עבור Production**

1. הרצת הפקודה tsc שמפעילה את הקומפיילר של TypeScript, מייצרת JavaScript ושומרת את קוד ה-JavaScript בתיקייה build שהגדרנו בקובץ tsconfig.json בפקודה outDir.
2. שכפול קובץ ה-package.json לעותק נפרד והכנסתו לתיקייה build בנוסף לקבצי ה-JavaScript.
3. שינוי קובץ ה-package.json הזה כך שיתאים ל-Production, לדוגמה שינוי ה-start לפקודה: node app.js (או פקודה מתאימה), מחיקת devDependencies.

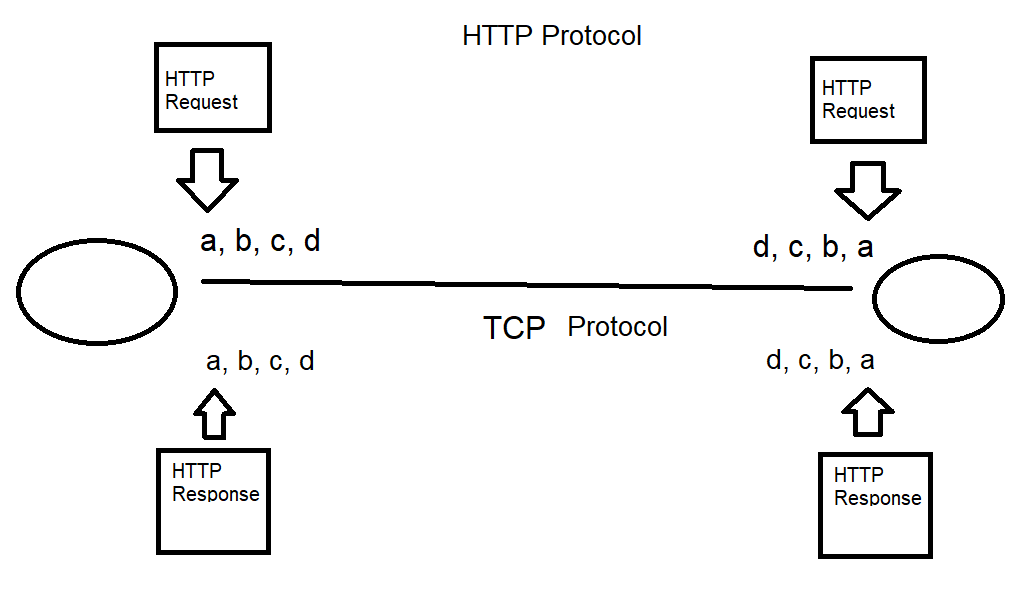
**TCP – Transmission Control Protocol**

פרוטוקול תקשורת להעברת מידע ברשת

**HTTP – Hypertext Transfer Protocol**

פרוטוקול מבנה שני מסמכים – Request המכיל בקשת דף אינטרנט ו-Response המכיל את דף האינטרנט.

הבקשה והתשובה מועברים ע"י פרוטוקול TCP



**Express**

ספרייה המאפשרת לבנות REST API ב-Node.js.

התקנה:

npm i express

npm i @types/express -D

כל Route שאנו בונים בעזרת Express מקבל:

1. אובייקט מסוג Request המכיל מידע שהתקבל ב-Request.
2. אובייקט מסוג Response המיועד להכיל מידע שהולך להישלח בחזרה ב-Reponse.
3. אובייקט מסוג NextFunction שנדבר עליו בהמשך

**Postman**

דפדפן ע"י שורת ה-URL מסוגל לשלוח אך ורק בקשות GET.

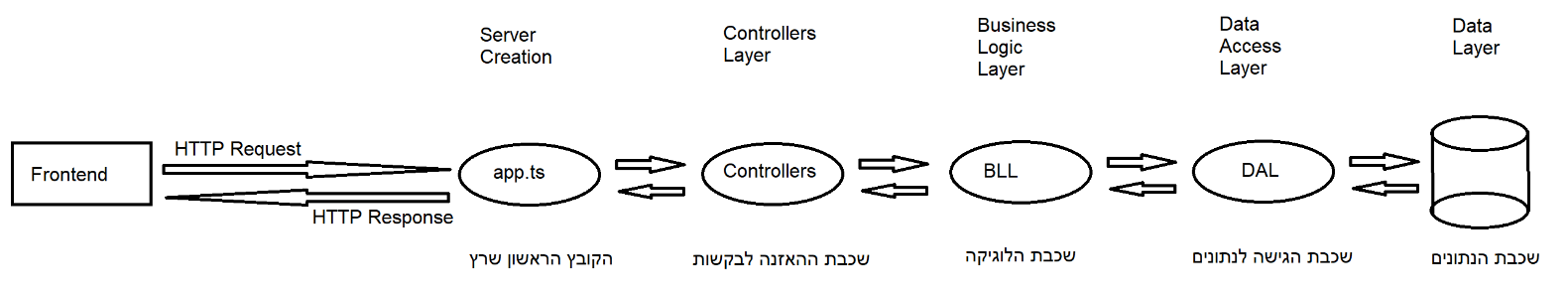
לכן יש מספר אפליקציות המאפשרות לבצע Request ולקבל Response לצורך בדיקת REST API שאנו בונים במקום לבצע זאת ע"י דפדפן שגם ככה לא יכול לבדוק POST/PUT/PATCH/DELETE (בשורת ה-URL).

**מודל השכבות**

ארכיטקטורה של בניית מערכת המשלבת בתוכה מידע (המגיע מ-Database)

המודל בנוי משכבות (שכבה = Tier או Layer) לכן המודל נקרא מודל השכבות – N-Tiers.

שכבה = אזור במערכת המטפל בתחום מסוים.



**Middleware**

זו פונקציה שמתבצעת אוטומטית בין ה-Request ל-Response.

אנו בונים אותה (או משתמשים באחת קיימת במערכת) ומגדירים בדיוק מתי עליה לפעול)

פונקציה זו עושה פעולה כלשהי (שאנו רוצים לבצע אוטומטית).

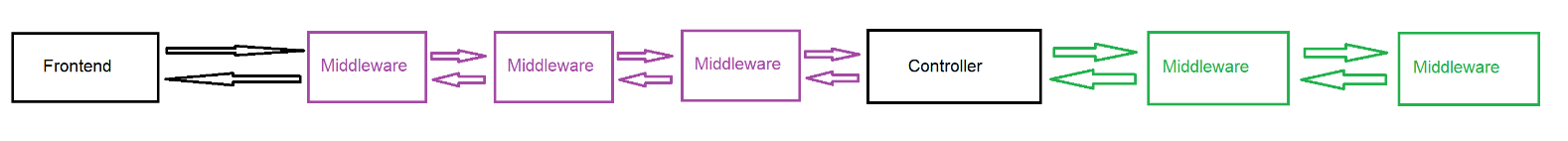
הפונקציה יכולה לעצור את ה-Request ומיד להחזיר Response מבלי להמשיך בזרימה הרגילה.

דוגמה: בדיקת חוקיות token. אם חוקי – נמשיך בזרימה. אם לא חוקי – נחזיר שגיאה מבלי להמשיך.

הפונקציה יכולה לבצע פעולה וכן להמשיך את הזרימה הרגילה.

דוגמה: log – שמירה בקובץ log שמשתמש גלש לשרת.

אפשר לומר שגם פונקציות ה-Route הרגילות שכתבנו ב-Controllers הינם Middleware כי הם מתבצעות אוטומטית כתלות ב-Method וב-Route בין ה-Request ל-Response.



אפשר לרשום Middleware בשלושה מקומות:

1. ברמת Route ספציפי – רק גלישה ל-Route הזה יגרום להפעלת ה-Middleware
2. ברמת Controller – גלישה לכל Route של ה-Controller יגרום להפעלת ה-Middleware – אך גם גלישות ל-Controller-ים שנרשמו אחריו. ולכן זה לא מומלץ לביצוע!!

(במילים אחרות – נא לא לבצע רישום Middleware ברמת Controller)

1. ברמת כל האפליקציה – בכל Route בכל Controller ה-Middleware יופעל.

**Error Middleware**

קיימת Middleware מיוחדת. היא מקבלת ארבעה ארגומנטים. הראשון זה אובייקט שגיאה. שלושת האחרים – הינם ה-request, response, next.

היא מיוחדת כי אם שולחים אובייקט ל-next – היא אוטומטית מופעלת (תוך קפיצה על כל ה-Middleware שהיו לפני).

אם לא נשלח אובייקט שגיאה ב-next – לא תופעל ה-Middleware הזו.

נכון יהיה לרשום Middleware זו תמיד אחרונה כי היא כמו catch-all.

**טיפול בשגיאות ב-REST API**

יש מספר סוגי שגיאות:

1. שגיאות לקוח – שגיאות שנגרמות עקב צד ה-Frontend:

* נתיב שלא קיים שהלקוח מנסה לגלוש אליו (404)
* משאב שלא קיים (Resource) לדוגמה, לקוח מבקש לעדכן ספר בעל קוד 85, אך אין ספר כזה (404)
* שגיאות ולידציה – הוספה או עדכון המכיל מידע לא חוקי (400)

עבור בדיקות ולידציה ניתן להשתמש בספרייה בשם Joi המאפשרת לבצע בדיקות ולידציה במחלקת ה-Model.

npm i joi

npm i @types/joi -D

* שגיאות אימות או הרשאות – משתמש מנסה למחוק עובד אך אין לו הרשאה למחיקת עובד (401/403)

1. שגיאות שרת – שגיאות שנגרמות עקב בעיה ב-Backend:

* קריסות – קריסה בתוכנית שאנו צריכים לתפוס ע"י catch (500)

**JWT**

התקנה:

npm i jsonwebtoken

npm i @types/jsonwebtoken -D

Authentication Levels

1. Anonymous – כל אחד יכול להיכנס, ללא ביצוע Login.
2. Logged In – בכדי להיכנס, חובה להיות Logged-In (אם המשתמש כרגע ביצע הרשמה הוא גם נחשב Logged-In) במילים אחרות – אם הבאת לי token חוקי – מותר לך להיכנס.
3. Role – בכדי להיכנס חובה להיות בעל תפקיד ספציפי, לדוגמה רק אדמין יכול לבצע פעולה מסוימת.

נוהל בניית Auth

1. מסד הנתונים צריך להכיל מידע לגבי משתמשים
2. בניית UserModel המכיל את המידע שיש לכל משתמש.
3. התקנת npm i jsonwebtoken וכן: npm i @types/jsonwebtoken -D
4. בניית util (נניח בשם cyber) המכיל פונקציה שמקבלת אובייקט UserModel ומחזירה token המכיל אותו.
5. בניית פונקציית register ב-logic המקבלת אובייקט משתמש חדש, מבצעת ולידציה (אם לא חוקי מעלה שגיאת 400) ומוסיפה אותו למסד הנתונים אם חוקי. על הפונקציה לייצר token ע"י ה-util שבנינו ולהחזיר ע"י return.
6. בניית route עבור register ב-controller הקורא לפונקציית register, שולח את אובייקט המשתמש שהתקבל ב-request.body, מקבל את ה-token ומחזיר ל-Front
7. שילוב ה-controller ב-app.ts
8. בניית CredentialsModel המכיל את המידע הדרוש עבור login
9. בניית פונקציית login ב-logic המקבלת אובייקט credentials, מבצעת ולידציה (אם לא חוקי מעלה שגיאת 400) ובודקת האם הפרטים נכונים (האם האדם אכן קיים במסד הנתונים). אם האדם לא קיים – מעלה שגיאת 401.
10. בניית route עבור login ב-controller הקורא לפונקציית login, שולח את אובייקט ה-Credentials שהתקבל ב-request.body, מקבל את ה-token ומחזיר ל-Front
11. הוספה ל-util פונקציה שמקבלת את ה-Authorization Header שאמור להכיל את ה-token ובודקת האם הוא חוקי או לא.
12. בניית Middleware לבדיקה האם המשתמש Logged In כן או לא, ע"י שימוש בפונקציה מהסעיף הקודם. אם המשתמש שלח token לא חוקי – להעלות שגיאת 401.
13. שילוב ה-Middleware הזה בכל route בו על המשתמש להיות Logged In.
14. הוספה ל-util פונקציה שמקבלת את ה-Authorization Header, שולפת את אובייקט ה-user שנמצא בתוך ה-token ומחזירה אותו.
15. בניית Middleware לבדיקה האם המשתמש הינו בעל Role מסוים (נניח אדמין) ע"י שליחת ה-Authorization Header לפונקציית ה-util שבנינו בסעיף הקודם, קבלת אובייקט ה-user ובדיקת ה-Role שלו. אם ה-Role שלו אינו מספק – העלאת שגיאת 403.
16. שילוב ה-Middleware הזה בכל מקום בו רוצים להגן מפני כניסת משתמשים שאינם בעלי ה-Role המתאים.