**UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAŞI**

**FACULTATEA DE INFORMATICĂ**

LUCRARE DE LICENŢĂ

# Typenet, un framework Node.js pentru aplicaţii scalabile

**propusă de**

***Sergiu Verciuc***

**Sesiunea:** *iulie, 2019*

**Coordonator ştiinţific**

**Drd. Florin Olariu**

**UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAŞI**

**FACULTATEA DE INFORMATICĂ**

# Typenet, un framework Node.js pentru aplicaţii scalabile

***Sergiu Verciuc***

**Sesiunea**: *iulie, 2019*

**Coordonator ştiinţific**

***Drd. Florin***

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

**DECLARAŢIE privind originalitatea conţinutului lucrării de licenţă**

Subsemntatul(a) ……..

domiciliul în ……..

născut(ă) la data de ……..…., identificat prin CNP ……..……..……..…., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iași, Facultatea de ……..…..specializarea ……..……, promoția ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal și dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul: _____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine și îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

# DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Typenet, un framework Node.js pentru aplicații scalabile*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, Absolvent,

Sergiu Verciuc

_____

# Cuprins

# Introducere

## Context

De la apariția sa în 2009, Node.js a avut o creștere mare în ceea ce privește rata de adopție a acestei tehnologii de către corporații, cele mai populare fiind IBM, LinkedIn, Microsoft și Netflix. Ceea ce se poate vedea și în cadrul diferitelor studii făcute de către „nodesource.com" care arată o creștere exponențială a numărului de descărcări dar și a numărului de contribuitori în ultimii ani, vezi Figura 1.

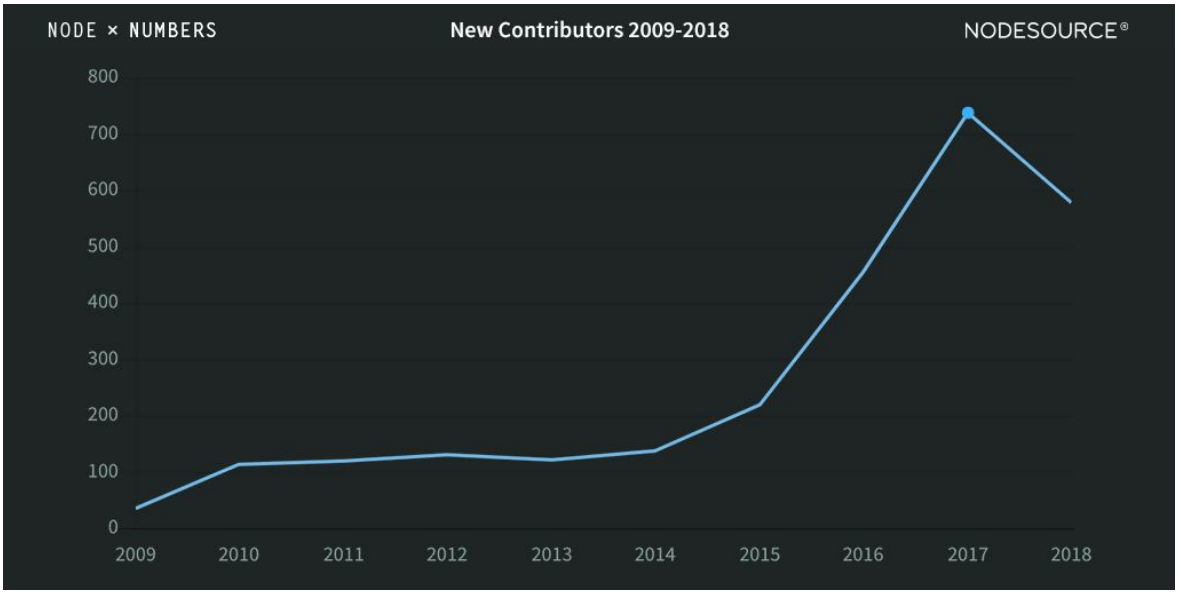


Figura 1 - Creșterea avută de Node.js din 2009 până în 2018 a numărului de contribuitori noi[1]

De asemenea, conform studiilor făcute de către site-ul „Stackoverflow" asupra utilizatorilor săi, Node.js este pe primul loc în topul celor mai folosite tehnologii și framework-uri, vezi Figura 2.

[1] https://nodesource.com/node-by-numbers

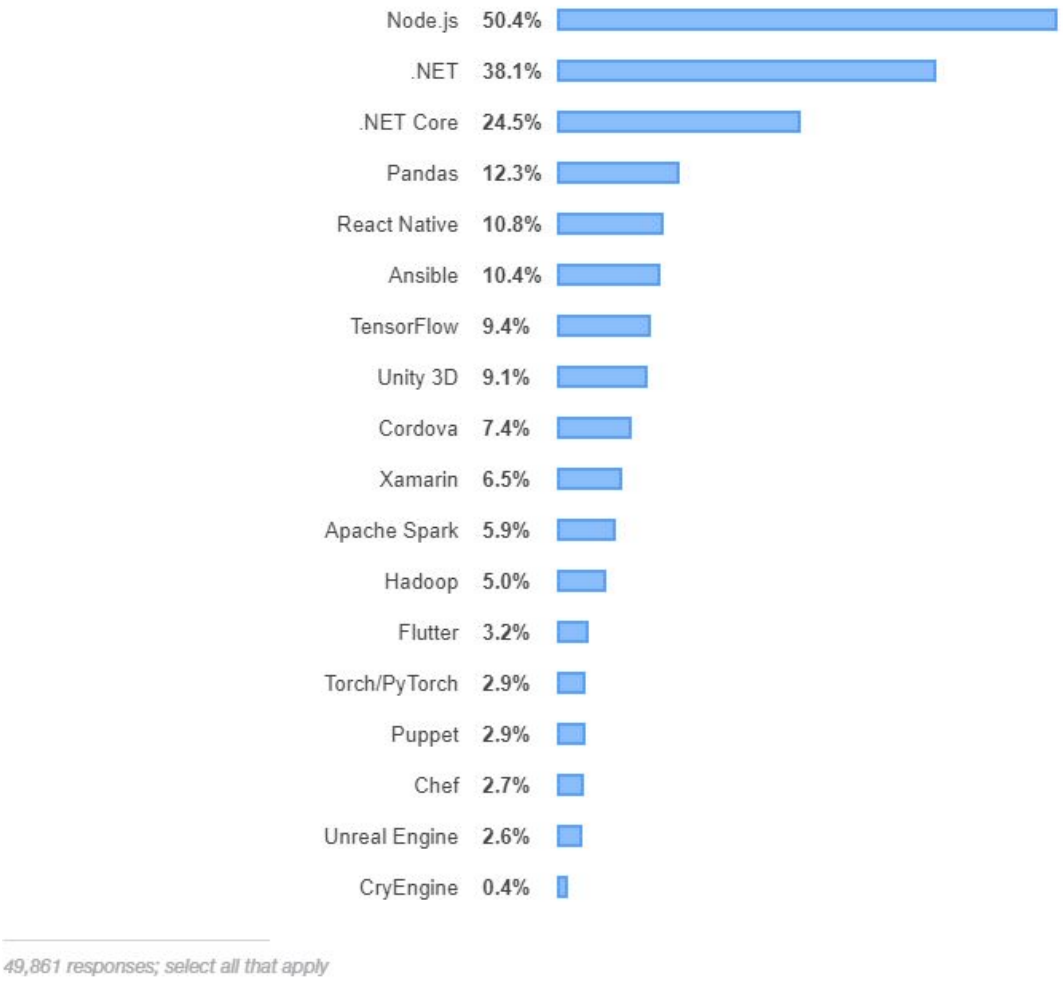


Figura 2 - Statistică făcută de către Stackoverflow referitoare la tehnologiile folosite de către programatori în anul 2019 (49,861 participanţi la statistică)[2]

Node.js oferă o suită de pachete pentru a realiza aplicaţii scrise JavaScript destinate aplicaţiilor server-side, iar comunitatea se foloseşte de aceste pachete pentru a crea diferite framework-uri şi librării care să rezolve diferite probleme des întâlnite. Cu timpul au apărut framework-uri open-source, precum Express, care este unul dintre cele mai populare framework-uri de Node.js, ce oferă posibilitatea de a crea servicii web într-un mod rapid şi uşor şi care sa fie viabile, rapide şi scalabile.

## Motivaţie

Am simţit o lipsă de maturitate în ceea ce priveşte framework-urile existente în lumea Node.js, comparând cu altele mai mature precum cele de la Microsoft şi anume .NET, în

[2] https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools

principal ASP.NET Core. În general, framework-urile rezolvă o problemă specifică şi oferă o serie de funcţionalităţi, doar că neglijează partea de cum anume îşi vor arhitectura proiectele consumatorii respectivelor framework-uri. Nu multe framework-uri oferă posibilitatea de a auto-genera:

- proiecte de la zero
- componente
- module specifice framework-ului

Fără cele menţionate mai sus, ne confruntam cu o scădere în productivitate atunci când se începe un nou proiect deoarece mult timp se petrece în a crea diferite componente şi de a lua decizii de structurare a proiectului.

Deoarece Node.js este un runtime environment pentru JavasScript, acesta vine atât cu avantajele, cât şi cu dezavantajele limbajului. JavaScript-ul este un limbaj dinamic, programatorii pot scrie cod mult mai rapid fără să se gândească la definirea tipurilor de date, ceea ce, într-adevăr, creşte productivitatea, doar că odată cu creşterea în dimensiune a proiectului scade productivitatea, fiind foarte greu de întreţinut codul. Lipsa unui instrument care sa le ofere posibilitatea de a analiza posibile probleme ale codului precum nerespectarea signaturii unei metode sau accesarea unei proprietăţi inexistente de pe un obiect creşte numărul bugurilor introduse in aplicaţie din cauza erorilor umane. Dar odată cu apariţia supersetului TypeScript toate aceste probleme sunt rezolvate. Despre TypeScript şi cum anume rezolvă problemele existente în JavaScript voi discuta mai multe în Capitolul 2. Descrierea soluţiei.

Datorită dificultăţilor pe care le-am întâlnit în dezvoltarea serviciilor web folosind framework-uri precum Express, am decis să experimentez prin crearea unu framework de Node.js folosind TypeScript. Prin acest framework doresc să creez un mediu care să ajute programatorii în a îşi arhitectura aplicaţia într-un mod uşor şi rapid, care va conduce la o creştere a productivităţii atunci când se începe un proiect nou, accelerând prima iteraţie a proiectului. Totodată, framework-ul nu ar trebui să afecteze performanţa cu care vine Node.js şi să fie rapid, viabil şi uşor de scalat.

Încă de la începutul dezvoltării framework-ului am decis ca pachetele pe care le voi crea nu trebuie să aibă multe dependinţe care să îngreuneze procesul de dezvoltare. Ca inspiraţie în ceea ce priveşte arhitecturarea framework-ului am urmărit acelaşi şablon pe care îl are Angular 8, deoarece este modularizat bine şi gândit pentru aplicaţii scalabile. De asemenea, făcând comparaţie între framework-uri precum ASP.NET Core, am decis să preiau ideile cele

mai bune care ar aduce valoare şi ar îmbunătăţi la fel de mult productivitatea programatorilor şi să le adaptez framework-ului şi mediului Node.js.

Lucrarea este împărţită în trei capitole, în care în primul capitol: Capitolul 1. Stadiul framework-urilor Node.js voi descrie problemele şi abordările anterioare ale framework-urilor existente în lumea Node.js pentru aplicaţii server-side. Al doilea capitol: Capitolul 2. Descrierea soluţiei voi prezenta sumar soluţia mea pentru probleme discutate în capitolul întâi; de asemenea, voi prezenta în detaliu tehnologiile folosite şi cum anume influenţează şi îmbunătăţesc experienta programatorilor. În al treilea capitol: Capitolul 3. Arhitectura typenet voi discuta arhitectura aleasă pentru a implementa soluţia descrisă în capitolul anterior şi motivarea alegerii ei, voi prezenta pachetele importante ale framework-ului, ce probleme rezolvă şi cum anume pot fi folosite într-o aplicaţie.

# Vocabular

Acest capitol are rolul de a defini termenii din limba engleză care vor fi folosiţi pe parcursul documentaţiei şi nu au o traducere directă, sau care îşi pierd din înţeles odată traduşi.

- Framework - reprezintă o serie de funcţionalităţi destinate programatorilor ce au ca scop rezolvarea unor probleme des întâlnite. Aceste funcţionalităţi sunt de tip generic, de aceea pot fi extinse prin adăugare de cod nou. De asemenea, un framework are o serie de caracteristici în plus faţă de o librărie, şi anume: un framework este extensibil, codul sursă nu poate fi modificat şi fluxul de control este dictat de către framework, nu de către apelant (inversion of control).
- Runtime environment - este un mediu de execuţie a unei aplicaţii în cadrul unui sistem de operare. Prin acest mediu, aplicaţiile pot trimite instrucţiuni sau comenzi către procesor şi au acces la alte resurse ale sistemului de operare precum memoria RAM, care în mod normal nu sunt accesibile în limbaje de programare, deoarece majoritatea limbajelor de programare sunt de nivel înalt.
- Command line interface - reprezintă o metodă de interacţiune cu un program în care un user emite o serie de comenzi ce reprezintă o înşiruire de cuvinte/numere.
- Superset - reprezintă un limbaj de programare care conţine toate funcţionalităţile unui alt limbaj de programare, pe care îl extinde sau îl îmbunătăţeşte cu alte funcţionalităţi.
- Open-source - reprezintă o metodologie pragmatică de distribuire a aplicaţii sau cod sursă fără restricţii, de acces, citire sau copiere. Aplicaţiile open-source permit accesul oricui să contribuie la dezvoltarea şi îmbunatăţirea produsului.
- Query string - este o parte a URL ce urmează după primul caracter de semnul întrebării. De obicei este de forma: cheie1=valoare1&cheie2=valoare2… . Exemplu de URL ce conţine un query string: „http://localhost/api/students?name=’Andrei’&year=3”.
- Continuous deployment - este o metodologie asociată cu practicile Agile ce presupune lansarea codului sursă în medii de dezvoltare, testare sau producţie după ce codul trece de verificările făcute de către o serie de teste automate, dacă este cazul.

# Contribuţie

În lipsa framework-urilor orientate spre a îmbunătăţi experienţa şi productivitatea programatorilor am decis să facilitez aceste aspecte prin crearea typenet, un framework Node.js pentru aplicaţii scalabile. Am analizat framework-uri de succes din lumea Node.js care au ca focus crearea de aplicaţii client-side, dar şi framework-uri pentru crearea de aplicaţii server-side din lumea ASP.NET. Am observat cum anume influenţează şi îmbunătăţeşte Angular[3] experienţa şi productivitatea programatorilor lucrând cu el de aproape doi ani, şi am ajuns la concluzia că o arhitecturare corectă a framework-ului are un impact mare asupra codului scris de către programatori, dar şi limbajul TypeScript, fiind un limbaj cu tipuri de date, creşte productivitatea odată cu creşterea în dimensiuni a proiectului. Angular fiind concentrat asupra problemelor ce ţin de aplicaţii client-side, am fost nevoit să analizez un framework destinat aplicaţiilor server-side, şi anume ASP.NET Core[4]. Acesta vine cu multe idei bune precum middleware system, despre care voi discuta în Capitolul 3. Arhitectura typenet, subcapitolul Pachetul „Core”.

[3] https://angular.io/docs
[4] https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2

# Capitolul 1. Stadiul framework-urilor Node.js

Din ce în ce mai mulţi programatori se orientează spre a face aplicaţii folosind JavaScript, în special pentru aplicaţii client-side. Acest lucru a condus la o creştere mare a numărului de framework-uri făcute pentru a crea aplicaţii web într-un mod rapid şi cu minimul de cod necesar. Folosirea limbajului JavaScript atât pentru aplicaţii web, cât şi pentru servicii web, a condus la o adopţie rapidă a Node.js şi, totodată, la apariţia de framework-uri. Dar se simte o diferenţă când vine vorba de implicarea companiilor mari în dezvoltarea de framework-uri pentru servicii web în Node.js, lucru ce nu se observă în framework-urile pentru aplicaţii client-side. Companii precum Google şi Facebook, împreună cu comunitatea, au creat framework-uri precum Angular şi respectiv React.js. Aceste framework-uri vin cu îndrumări de cum să se implementeze anumite soluţii de probleme, respectă practicile bune din programare şi cresc productivitatea programatorilor odată ce ajung să le stăpânească.

Unul dintre cele mai populare framework-uri pentru servicii web în node este Express.js[5]. Sunt câteva framework-uri Node.js care sunt construite în jurul acestui framework şi pe care îl extind sau care cel puţin se inspiră din abordările lui, unul dintre ele ar fi Fastify[6], gândit la fel ca şi Express.js, doar că pune mai mult accentul pe eficienţă. Acesta oferă funcţionalităţi minimaliste pentru a crea servicii web. Acesta pune la dispoziţie un sistem de rutare prin care programatorii îşi pot crea endpoint-uri corect semantice din punct de vedere REST. În Figura 3 se poate evidenţia simplitatea creării unei aplicaţii simple folosind Express.js, care are un singur endpoint.

```
const express = require('express')
const app = express()

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

app.listen(80)
```

Figura 3 - exemplu de aplicaţie folosind Express.js care expune un endpoint „hello” la care va răspunde cu mesajul „Hello World!”

Odată cu creşterea în dimensiuni a proiectului lucrurile se vor complica, întrucât Express nu pune la dispoziţie un sistem de separare a endpoint-urilor pe controller care să aibă aceeaşi întrebuinţare, şi anume să grupeze operaţiile posibile pentru o anumită resursă într-un singur fişier, aceste lucruri îi revin programatorilor să îşi creeze singuri mecanisme asemănătoare sau

[5] https://expressjs.com
[6] https://www.fastify.io

să păstreze într-un singur fişier toate endpoint-urile expuse. De asemenea, lipsa unui sistem de injectare a dependinţelor nu facilitează realizarea şi separarea codului pe module granulare care să aibă o singură întrebuinţare în toată aplicaţia.

Odată cu adopţia micro-serviciilor ca şablon arhitectural, serviciile web de tip REST devin mici ca dimensiuni, devin foarte decuplate de celelalte servicii şi în general au responsabilitatea de a rezolva o singură problemă de business. Dar datorită numărului mare de dependinţe pe care îl are express-ul cu alte pachete Node.js, acesta nu este foarte eficient pentru micro-servicii, el având aproape 50 de dependinţe pe arborele dependinţelor,[7] ceea ce îngreunează procesul de dezvoltare şi afectează şi performanţa.

Aşadar, framework-urile server-side Node.js se concentrează pe aspecte care nu aduc valoare programatorilor. În loc să ofere o experientă bună, câteodată complică lucrurile datorită lipsei de funcţionalităţi care să ajute la realizarea de bune practici, precum inversarea dependinţelor printr-un sistem de injectare a dependinţelor. Dar un framework care să crească productivitatea şi experienţa nu ar trebui să ofere doar un sistem de rutarea requesturilor şi un sistem de injectare a dependinţelor, el ar trebui să faciliteze şi realizarea de arhitecturi complexe precum onion sau micro-servicii şi, în acelaşi timp, să fie performant şi uşor de scalat.

[7] https://npm.anvaka.com/#/view/2d/express

# Capitolul 2. Descrierea soluţiei

Pentru a rezolva problemele existente în framework-urile Node.js am decis să realizez un framework împărţit cât mai granular, astfel încât fiecare proiect să fie nevoit să instaleze doar de ce anume are nevoie şi chiar foloseşte. Datorită experienţei avute de către Express.js am optat pentru a crea pachete Node.js care să aibă dependinţe directe cât mai puţine.
Pentru a creşte productivitatea şi a îmbunătăţi experienţa programatorilor am decis să dezvolt framework-ul folosind TypeScript, care este folosit şi de către Angular. De asemenea, pentru a face cât mai uşoară trecerea programatorilor care sunt familiarizaţi cu Angular, am adaptat structurarea şi unele concepte folosite, care se pretează şi într-un framework server-side precum „Module" sau „Injectable". TypeScript oferă posibilitatea de a adăuga metadata peste codul scris folosind decoratori, lucru întâlnit şi în C# prin atribute, dar şi în JAVA prin adnotări, ceea ce a făcut posibilă împrumutarea modului de a declara endpointuri, controllere şi module în framework-urile mature.

## Tehnologii folosite

În acest subcapitol voi detalia tehnologiile folosite, voi discuta despre motivul alegerii lor, şi cum reuşesc aceste tehnologii să aducă valoare framework-ului şi sa îmbunătăţească productivitatea programatorilor.

### Node.js

Node.js este un runtime environment pentru cod JavaScript construit în engine-ul V8 de la chrome.[8] Node.js oferă posibilitatea programatorilor de a-şi crea command line tools şi aplicaţii server-side, fie servicii web de tip RESTful[9], fie aplicaţii real-time folosind protocolul WebSocket.
V8 este un mediu de execuţie de cod JavaScript sau WebAssembly care a fost iniţial construit de către Google pentru Google Chrome[10], iar mai apoi a devenit open source. Acesta este scris în C++, de aceea este rapid şi performant când vine vorba de viteza de execuţie a codului,

[8] https://nodejs.org/en/
[9] https://restfulapi.net
[10] https://v8.dev/docs

întrucât compilează codul JavaScript direct în cod maşină înainte de a-l executa. Acesta se ocupă de alocarea obiectelor în memorie şi de colectarea obiectelor care nu sunt folosite. V8 implementează atât ECMAScript, cât şi WebAssembly, şi poate fi executat pe maşini ce au ca sisteme de operare:

- Windows 7, 8 sau 10
- macOS 10.12+
- Linux

care pot folosi ca arhitectura de procesor una dintre x64, IA-32, ARM sau MIPS.

Node.js este executat într-un singur thread, folosind event loop pentru a realiza operaţii ne-blocante de I/O, de aceea toate apelurile de tip I/O folosesc un callback pentru a gestiona răspunsul operaţiei.

Node.js pune la dispoziţie o serie de pachete de bază cu care se pot realiza diferite interacţiuni precum:

- Pachetul „child_process” facilitează realizarea de procese copil ce au ca scop, în general, de a executa o metodă costisitoare din punct de vedere a resurselor sau al timpului de execuţie, dar a cărei răspuns nu afectează aplicaţia. Cea mai deasă întrebuinţare este de a trimite email-uri, deoarece în majoritatea aplicaţiilor trimiterea de emailuri nu ar trebui să blocheze sau să ofere un feedback despre starea trimiterii.
- Pachetul „http” pune la dispoziţie o serie de funcţionalităţi prin care se poate crea un server care sa asculte request-urile de la un anumit port. În Figura 4 se poate vedea simplitatea cu care se poate crea un server folosind acest pachet, în schimb, pentru cazuri mai complexe, ce presupun rutarea cererilor, este nevoie de mai multe mecanisme, pe care le voi discuta în Capitolul 2. Descrierea soluţiei.

```
const http = require('http');

const server = http.createServer((req, res) => {
 res.statusCode = 200;
 res.setHeader('Content-Type', 'text/plain');
 res.end('Hello World\n');
});

server.listen(80);
```

Figura 4 - exemplu de cod care creează un server la portul 80 şi răspunde cu „Hello World” pentru fiecare request

- Pachetul „fs” (File system) uşurează interacţiunea cu fişierele/directoarele de pe server. Toate operaţiile au implementări atât blocante cât şi ne-blocante.

Node.js pune la dispoziţie tot ce este necesar pentru a crea un framework rapid, scalabil, care să poată accepta un număr foarte mare de conexiuni. Aceste aspecte m-au ajutat să mă concentrez asupra problemelor pe care am vrut să le rezolv prin acest framework, în loc să ma gândesc la îmbunatăţirea performanţei şi a scalabilităţii ci doar la menţinerea lor, ele venind odată cu Node.js.

## TypeScript

TypeScript este un superset al limbajului JavaScript, care are ca principală caracteristică introducerea de tipuri de date în limbajul de scripting. TypeScript este o soluţie open-source dezvoltată şi întreţinută de către Microsoft, începând cu anul 2012 până în prezent.

TypeScript are un compilator care transformă codul scris în cod JavaScript, care poate fi executat mai apoi. Unul dintre aspectele importante ale compilatorului este de a verifica respectarea tipurilor folosite în codul scris, ceea ce creşte productivitatea şi. de asemenea, scade numărul de bug-uri, după cum arată un studiu care evidenţiază faptul că numărul de bug-uri scade cu 15%[11] pe proiectele care au adoptat TypeScript.

TypeScriptul oferă plug-in-uri pentru diferite IDE-uri prin care facilitează sugestii de autocompletare a codului, navigare uşoară prin cod pentru a vedea implementarea unor clase sau definirea unor interfeţe, toate aceste aspecte cresc productivitatea programatorilor. TypeScript include un set de funcţionalităţi de programare orientată pe obiecte, printre care declararea de interfeţe, clase, metode şi clase generice, moştenire, încapsulare, polimorfism şi separarea codului sau a conceptelor prin module şi namespace-uri. De asemenea, oferă funcţionalităţi de adăugare de metadată peste codul scris ceea ce ajută la declarare de comportament pentru clase, metode şi parametrii funcţiilor. În Figura 5 se poate vedea un exemplu prin care un decorator marchează o clasă ca fiind un controller, prin astfel de abordări sunt nevoiţi cei ce vor folosi framework-ul să marcheze diferite concepte ale framework-ului. Mai multe despre cum anume vor fi folosiţi respectivii decoratori voi vorbi în Capitolul 3. Arhitectura typenet.

[11] https://blog.acolyer.org/2017/09/19/to-type-or-not-to-type-quantifying-detectable-bugs-in-javascript/

```
export function Controller(route: string): ClassDecorator {
   return (target: Object) => {
       Reflect.defineMetadata('Controller!', true, target);
   };
}

@Controller('api/person')
class PersonController {
   public name: string;
}
```

Figura 5 - exemplu de declarare și folosire a unui decorator pentru a specifica comportamentul unei clase

Toate aspectele menționate mai sus m-au motivat să aleg TypeScript-ul în locul JavaScript-ului, ajutându-mă să realizez un framework ușor de înțeles și întreținut, care să ajute mai departe programatorii care îl vor folosi, prin îmbunătățirea experienței și creșterea productivității.

# Capitolul 3. Arhitectura typenet

În acest capitol voi discuta despre arhitectura aleasă în realizarea framework-ului, motivaţia alegerii şi beneficiile cu care vine arhitectura aleasă, şi anume arhitectura de tip micro-kernel, însă mai multe detalii despre aceasta voi prezenta în subcapitolul următor: Arhitectura micro-kernel.

Încă din fazele incipiente ale framework-ului am decis să separ concepte, soluţii şi funcţionalităţile într-un mod cât mai granular, astfel încât un proiect să nu fie nevoit să îşi instaleze pachete de care el nu are nevoie. Un exemplu de o asemenea abordare se poate vedea în implementarea framework-ului Angular. Acesta este separat în 25 de pachete în versiunea a opta a lui, oferind şi pachete ce sunt folosite doar pe partea de dezvoltare ce au ca scop îmbunătăţirea scrierii de teste.

De asemenea, pentru a îmbunătăţi construirea de proiecte am concluzionat că un command line interface ar spori interacţiunea pe care o are un programator cu framework-ul oferindu-i funcţionalităţi care să îi uşureze munca şi, astfel, să se concentreze să scrie cod care să aducă mereu valoare proiectului, nu să se lupte cu probleme cu care nu ar trebui să se lupte, dar mai multe detalii despre aceste funcţionalităţi voi discuta în subcapitolul: Command line interface.

## Arhitectura micro-kernel

Analizând soluţia cu care au venit programatorii de la Google în realizarea framework-ului Angular am observat o separare riguroasă a funcţionalităţilor pe care le oferă în foarte multe pachete. Acest şablon arhitectural poarta numele de micro-kernel, care este folosit de majoritatea sistemelor de operare, de unde preia şi numele.

Arhitectura micro-kernel are la bază două componente: un sistem central numit core şi o serie de module.

Sistemul central de obicei conţine minimul de funcţionalităţi necesare pentru a face aplicaţia funcţională, dar şi funcţionalităţi prin care modulele externe se pot integra, pentru a extinde funcţionalităţile deja prezente în core. În Figura 6 se evidenţiază separarea care o presupune această arhitectură dar şi importanţa pe care o are sistemul central.

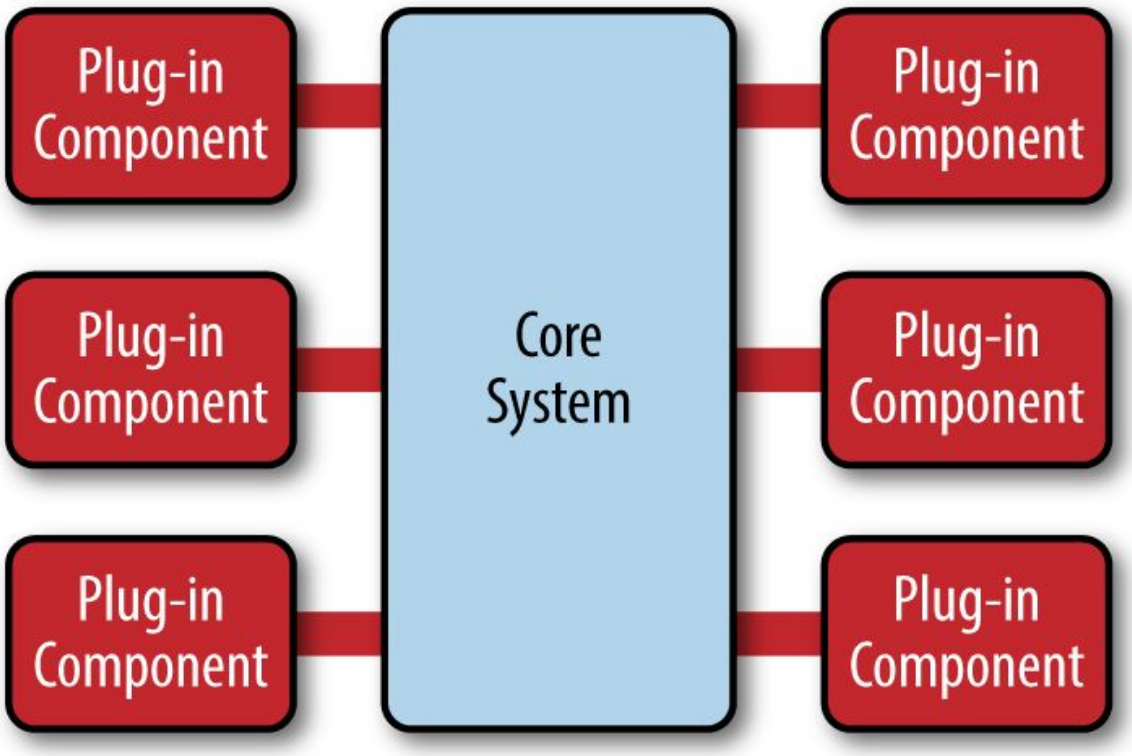


Figura 6 - Arhitecura micro-kernel[12]

Modulele sunt de obicei de sine stătătoare, independente de celelalte module, fiind dependente doar de core, specializate doar pe o singura funcţionalitate şi au ca scop extinderea capabilităţilor oferite de sistemul central.

De obicei sistemul central ştie despre existenţa modulelor disponibile, expunând o funcţionalitate de a înregistra fiecare modul.

Această arhitectură, ca orice ce altă arhitectură, vine cu avantaje şi dezavantaje. Un avantaj, dar în acelaşi timp şi un dezavantaj al acestei arhitecturi este separarea capabilităţilor într-un mod cât mai granular. Acest fapt îmbunătăţeşte performanţa, abilitatea de a scrie teste, şi este uşor de publicat şi de distribuit. Dar, în acelaşi, timp nu este uşor de gestionat proiectarea modulelor şi a sistemului central, complicând procesul de dezvoltare ce caută mereu soluţii pentru a integra cat mai uşor o funcţionalitate nouă fără a afecta codul deja existent.

În Figura 7 este exemplificată separarea conceptelor într-o serie de module independente care să faciliteze funcţionalităţi comune în aplicaţiile de tip server-side.

[12] https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html

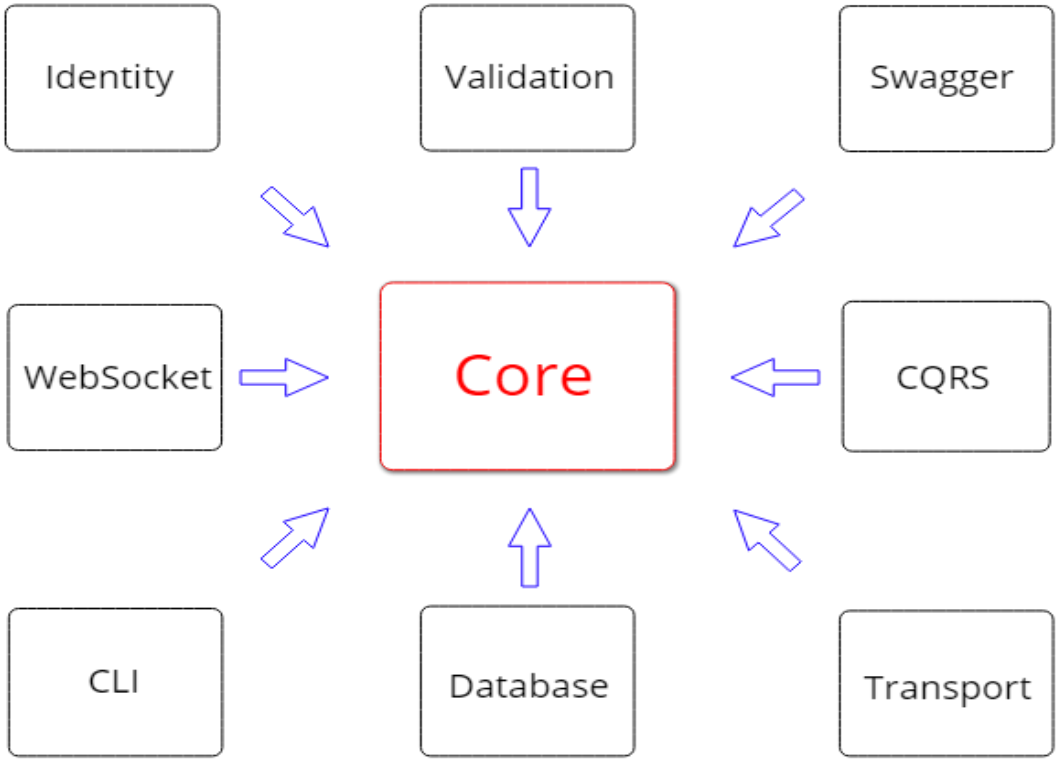


Figura 7 - Arhitectura framework-ului typenet

Această arhitectură facilitează toate nevoile iniţiale pe care le-am avut şi anume: separarea funcţionalităţilor cât mai granular, astfel încât aplicaţiile pot să depindă doar de ce pachete au nevoie pentru a funcţiona şi de a păstra calităţile Node.js intacte, precum performanţa şi scalabilitatea.

## Pachetul „core”

Acest pachet are cea mai mare responsabilitate, şi anume de a oferi minimul necesar de funcţionalităţi pentru a realiza o aplicaţie server-side. Însă nu este suficient să ofere minimul necesar pentru a îmbunătăţi experienţa programatorilor şi pentru a creste productivitatea, acesta trebuie să ofere şi capabilităţi prin care un programator să poate respecta bunele practici, să poată introduce cu uşurinţa valoare în proiect, dar şi să ofere posibilitatea programatorului de a integra pachete de tip modul prin care el să extindă aplicaţia cu capabilităţile necesare lui. De asemenea, încă de la începutul proiectării framework-ului am decis sa minimizez pe cât posibil dependinţele pe care le voi avea de pachetele externe pentru a implementa specificaţiile framework-ului. Acest fapt se poate evidenţia în Figura 8 în care

este modelat arborele de dependențe ale pachetului „@typenet/core”, din care se poate observa numărul redus de dependințe directe.

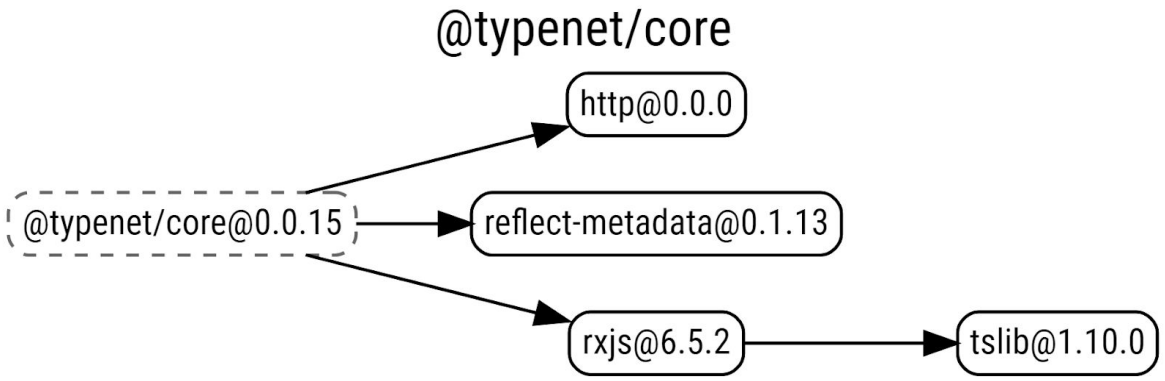


Figura 8 - arborele de dependințe a pachetului „@typenet/core”

Din analiza framework-ului ASP.NET Core am decis să împart framework-ul în două etape de executare. O etapă de inițializare și o etapă de rezolvare a requesturilor.

În etapa de inițializare, framework-ul are responsabilitatea de a colecta cât mai multe informații referitoare la structura aplicației și la capabilitățile framework-ului folosite. Etapa de rezolvare a requesturilor are ca scop să gestioneze cererile venite din partea consumatorilor aplicației și oferirea unui răspuns corect fiecărui request. Prin această abordare, a doua etapă se folosește de toate informațiile colectate în prima etapă. Astfel, framework-ul este mai rapid în comparație cu o abordare în care accentul se pune pe încărcarea leneșă a structurii aplicației. În Figura 9 și Figura 10 este exemplificată structura de proiect care poate fi generată de la linia de comandă (voi discuta mai multe despre acest aspect în subcapitolul următor: Command line interface) și respectiv cea mai simplă aplicație în care se expune un endpoint „api/hello” și pentru fiecare request către acest endpoint se răspunde cu mesajul „Hello World!”.

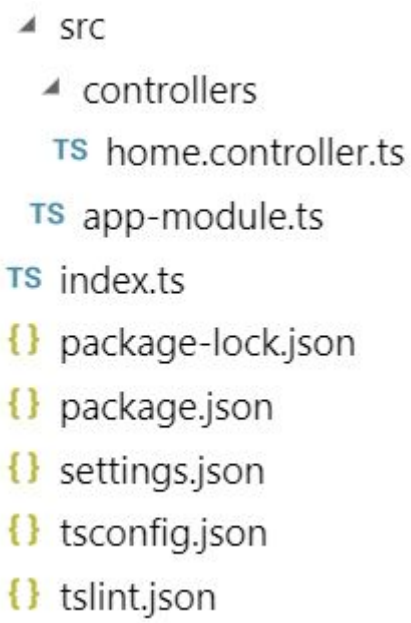


Figura 9 - structura de baza a proiectului folosind typenet

```
import { Controller, HttpGet, ActionResult, Ok } from '@typenet/core';

@Controller('api/hello')
export class HelloController {

   @HttpGet('')
   public helloWorld(): ActionResult {
       return new Ok('Hello World!');
   }
}
```

Figura 10 - exemplu de controller ce expune o resursă „hello” ce la metoda GET va returna mesajul „Hello World!”

După cum se poate vedea, comparând cu exemplele de cod din Figura 3 şi Figura 4, există o diferenţă mare între modul cum se declară o aplicaţie de tip „Hello World” folosind Node.js sau Express.js şi typenet. În typenet codul arată mai curat şi împărţit pe responsabilităţi. Nu există nici o referire la vreun server sau aplicaţie creat/ă de către Node.js şi respectiv Express.js. Aceste lucruri se fac în etapa de iniţializare a aplicaţie, lucru ce este evidenţiat în Figura 11, unde este prezentat cel mai simplu mod de a înregistra o aplicaţie ce foloseşte framework-ul typenet. Vom vedea, pe parcursul următoarelor subcapitole, că în această zonă de cod se vor înregistra capabilităţi noi oferite de către pachete de tip modul.

```
import { ApplicationFactory } from "@typenet/core";
import { AppModule } from './src/app-module';
import * as settings from './settings.json';

async function bootstrap() {
   const app = ApplicationFactory.create(AppModule);
   app.useSettings(settings);
   await app.run();
}

bootstrap();
```

Figura 11 - cod folosit pentru a iniţializa framework-ul typenet

În Figura 10 se evidenţiază folosirea câtorva funcţionalităţi oferite de către TypeScript discutate în Capitolul 2. Descrierea soluţiei. Am folosit decoratori cu scopul de a marca comportamente diferite pe care îl poate are codul scris de către un programator dar şi de a colecta informaţiile necesare comportamentului respectiv. Spre exemplu, decoratorul „Controller” are nevoie, pe lângă o clasă la care să îi atribuie proprietăţi specifice, şi de un endpoint la care să expună funcţionalităţile oferite de către controller.

De asemenea, pentru a specifica ce metode ale clasei sunt legate de verbe HTTP[13], programatorii se pot folosi următorii decoratori: „HttpGet”, „HttpPost”, „HttpPut”, „HttpDelete” şi „HttpPatch”. Pe lângă aceşti decoratori este nevoie şi de o sub-rută la care este expusă metoda. Această sub-rută poate conţine atât parametri cât şi un query string. Un astfel de exemplu se poate vedea în Figura 12.

[13] https://www.restapitutorial.com/lessons/httpmethods.html

```
import {
   Controller, HttpGet, ActionResult, Ok, HttpPost, Created,
   FromRoute, FromQuery, HttpPut, FromBody, NoContent, HttpDelete
} from '@typenet/core';

@Controller('api/values')
export class ValuesController {

   @HttpGet('')
   public get(@FromQuery() filter: any): ActionResult {
       return new Ok([1, 2, 3]);
   }

   @HttpGet(':id')
   public getById(@FromRoute(':id') id: string): ActionResult {
       return new Ok(id);
   }

   @HttpPost('')
   public create(@FromBody() resource: any): ActionResult {
       return new Created(resource);
   }

   @HttpPut(':id')
   public update(@FromRoute(':id') id: string): ActionResult {
       return new NoContent();
   }

   @HttpDelete(':id')
   public delete(@FromRoute(':id') id: string): ActionResult {
       return new NoContent();
   }
}
```

Figura 12 - exemplu de controller în care sunt folosiţi toate metodele HTTP

Parametrii unei rute sunt prefixaţi cu două puncte. Pentru a îi putea folosi în cadrul aplicației se poate adăuga un parametru la metoda respectivă şi decorat cu decoratorul „FromRoute”, specificând şi numele parametrului. De asemenea, requesturile de tip POST, PUT sau PATCH au şi un conţinut ataşat lor, acest conţinut poate fi încărcat folosind decoratorul „FromBody”. Pentru a interpreta query stringul unui request se poate folosi decoratorul „FromQuery”.

Pentru a ajuta programatorii să îşi structureze aplicația pe module am introdus un concept numit „Module” care colectează informații despre capabilitățile pe care le are respectivul modul. Un exemplu de astfel de modul se poate vedea în Figura 13.

```
import { Module } from "@typenet/core";
import { StudentsController } from "./controllers/students.controller";
import { AuthorizationModule } from './authorization-module';
import { StudentsRepository } from './students.repository';

@Module({
   controllers: [
       StudentsController
   ],
   imports: [
       AuthorizationModule
   ],
   providers: [
       StudentsRepository
   ]
})
export class AppModule { }
```

Figura 13 - exemplu de declarare de modul folosind framework-ul typenet

Un modul trebuie să ştie despre existenţa tuturor controllerelor din acel modul, despre alte module de care se foloseşte, dar şi de toate componentele care pot fi injectate.

Pentru a ajuta programatorii în a realiza bune practici din programare precum inversarea dependinţelor, framework-ul oferă funcţionalităţi de injectare a dependinţelor. Clasele care pot fi injectate trebuie marcate cu decoratorul „Injectable” şi specificat în lista de „providers” din modulul în care face parte. De asemenea, atunci când se marchează o clasă ca fiind injectabilă se poate specifica şi o perioadă de viaţă a instanţelor create de către framework a respectivelor clase. Exista trei tipuri de perioadă de viaţă: „singleInstance”, „scopedInstance” şi „transientInstance”. Perioada de viaţă „singleInstance” specifică framework-ului să creeze o singură instanţă în întreaga aplicaţie. Marcarea unei clase injectabile cu perioada de viaţa „scopedInstance” va specifica framework-ului să creeze câte o instanţă de clasă pentru fiecare request în parte. Iar „transientInstance” va specifica framework-ului să creeze cate o instanţă de fiecare dată când este nevoie de una.

Pentru a facilita integrarea rapidă şi uşor de implementat a pachetelor de tip modul din cadrul framework-ului, am împrumutat sistemul de „middleware” din ASP.NET Core. Acest sistem este compus dintr-o mulţime de componente ce lucrează în serie una după alta, ce pot trata un request. Fiecare componentă poate decide dacă pasează requestul către următoarea componentă sau care pot să manipuleze requestul sau să facă alte lucruri înainte sau după ce au dat requestul către următoarele componente. Pipelineul de astfel de componente începe cu un „EntryMiddleware” care are rolul de a încheia requestul şi se termină cu

„DefaultRequestHandler”, ce are rol de a delega requestul către o metodă din controller dacă requestul este valid, sau să invalideze requestul dacă acesta nu este valid. În Figura 14 este reprezentat vizual o serie de componente de tip „middleware” şi cum anume trece requestul prin fiecare componenta în parte.

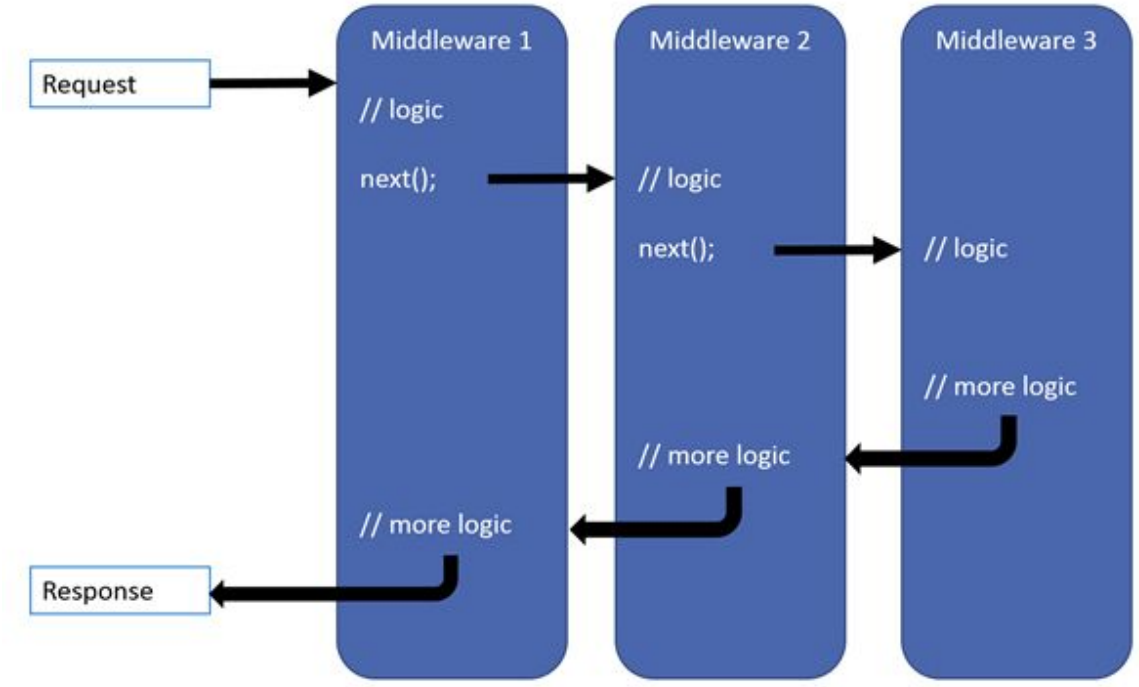


Figura 14 - un sistem de trei middleware-uri[14]

Componente de tip middleware nu sunt destinate doar pachetelor de tip modul, ci programatorilor care creează aplicaţii. Aceştia pot folosi acest mecanism pentru a înregistra activitatea aplicaţie, pentru a trata excepţiile aruncate prin aplicaţie ş.a.m.d. Pentru a marca o clasă ca fiind o componentă de tip middleware se foloseşte decoratorul „Middleware”, iar clasa respectivă este nevoită să implementeze interfaţa „PipelineMiddleware” corespunzătoare.

O altă capabilitate a pachetului „core” este de a putea specifica ce aplicaţii, metode HTTP şi conţinut pot avea requesturile făcute către aplicaţie. Acest mecanism este cunoscut drept „Cross-Origin Resource Sharing”, abreviat CORS. În Figura 15 este exemplificată o aplicaţie care permite doar requesturi de la „http://localhost:4200”, cu metode HTTP de tip GET şi POST.

[14] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2

```
import { ApplicationFactory } from "@typenet/core";
import { AppModule } from './src/app-module';

async function bootstrap() {
   const app = ApplicationFactory.create(AppModule);
   app.useCorsPolicy((builder) =>
       builder.withOrigins('http://localhost:4200').withMethods('GET', 'POST'));
   await app.run();
}

bootstrap();
```

Figura 15 - exemplu de înregistrare a unei politici de „Cross-Origin Resource Sharing”

În figura Figura 11 se observa o înregistrare a unor setări prin apelul: „app.useSettings(settings);”. Prin această funcţionalitate programatorii îşi pot crea fişiere de configurare în care să specifice portul la care să se deschidă aplicaţia, diferite conexiuni către baze de date, sau alte tipuri de configurări pe care mai apoi să le poată modifica într-un pipeline de continuous deployment, şi ajusta cerinţelor fiecărui mediu de publicare a codului.

Aşadar, acest pachet oferă funcţionalităţi destinate atât programatorilor, prin sistemul de inversie a dependinţelor, prin rutarea requesturilor folosind controllere şi metode ce aparţin lor, prim modularizarea codului folosind module, prin sistemul de componente de tip middleware; cât şi lansării aplicaţie pe diferite medii prin folosirea fişierelor de configurare.

## Command line interface

Un aspect important care m-a motivat să creez acest framework a fost lipsa instrumentelor ajutătoare pentru dezvoltarea aplicaţiilor folosind framework-uri Node.js. De aceea, am creat acest pachet Node: „@typenet/cli” ce trebuie instalat global pentru a facilita diferite operaţii de auto-generare de cod.

Prin acest pachet programatorii îşi pot genera:

- Proiecte de la zero
- Controller
- Module

Toate aceste funcţionalităţi se pot realiza printr-o serie de comenzi.

Pentru a genera proiecte de la zero se poate folosi comanda: „typenet new <dir>” sau „typenet n <dir>”. Această comandă trebuie sa primească ca parametru un director în loc de al treilea

parametru, care poate să nu existe pe disc, acesta va fi folosit şi ca nume al aplicaţiei. Această comandă va genera minimul de fişiere pentru a crea o aplicaţie, va instala dependinţele proiectului şi va configura proiectul să lucreze cu TypeScript, stabilind şi un set de reguli de redactare a codului. Pentru a vedea structura proiectului generat vezi Figura 9.

Pentru a genera controllere sau module se poate folosi aceeaşi comandă dar oferind parametri corespunzători. Comandă care generează astfel de fişiere este: „typenet generate <template> <path>”, unde al doilea parametru poate fi „controller” sau „c” pentru a genera controllere şi „module” sau „m” pentru a genera module, iar al treilea parametru reprezintă calea unde să fie salvat fişierul creat. Numele clasei controllerului sau a clasei modulului este preluat din calea unde trebuie salvat respectivul fişier. Ultimul segment, de după caracterul slash („/”), va fi luat în considerare ca nume. Exemplu: comanda „typenet g c src/authentication/controllers/auth” va crea structura de foldere din Figura 16 şi fişierul din Figura 17.


Figura 16 - structura folderelor create de comanda „typenet g c src/authentication/controllers/auth”

```
import {
   Controller, HttpGet,
   ActionResult, Ok,
   HttpPost, Created,
   FromRoute, HttpPut,
   FromBody, NoContent,
   HttpDelete
} from '@typenet/core';

@Controller('api/auth')
export class AuthController {

   @HttpGet('')
   public get(): ActionResult {
       return new Ok([1, 2, 3]);
   }

   @HttpGet(':id')
   public getById(@FromRoute(':id') id: string): ActionResult {
       return new Ok(id);
   }

   @HttpPost('')
   public create(@FromBody() resource: any): ActionResult {
       return new Created(resource);
   }

   @HttpPut(':id')
   public update(@FromRoute(':id') id: string): ActionResult {
       return new NoContent();
   }

   @HttpDelete(':id')
   public delete(@FromRoute(':id') id: string): ActionResult {
       return new NoContent();
   }
}
```

Figura 17 - controller generat de catre @typenet/cli

Pentru generat module comportamentul comenzii este asemănător, schimbându-se doar structura fişierului creat, acesta fiind unul asemănător cu cel din Figura 13.

# Pachete de tip modul

În acest subcapitol voi discuta despre câteva dintre pachetele propuse inițial în subcapitolul Arhitectura micro-kernel. Voi discuta despre capabilitățile introduse, motivarea creării lor și cum anume se integrează și se folosește pachetul într-o aplicație.

## Pachetul „web-socket”

Acest pachet are ca scop de a introduce capabilități specifice aplicațiilor ce comunică în timp real. Pachetul trebuie să fie ușor de integrat într-o aplicație deja existentă și să nu afecteze logica deja implementată.

Pentru a satisface aceste nevoi m-am folosit de pachetul „socket.io”[15], o soluție Node.js pentru a realiza comunicare în timp real folosind socketuri web. Am decis să depind de acest pachet deoarece este o soluție matură și performantă, cu o comunitate mare care ajută la dezvoltarea lui, ce oferă și capabilități de schimb de fișiere în format binar.

Pentru a integra funcționalitățile oferite de pachetul „@typenet/web-socket” trebuie urmați o serie de pași. Mai întâi trebuie instalat pachetul „@typenet/web-socket” împreună cu pachetul „socket.io”. Apoi trebuie înregistrat modulul extern oferit de acest pachet, în Figura 18 sunt subliniate liniile de cod ce trebuie adăugată în inițializarea aplicației.

```
import { ApplicationFactory } from "@typenet/core";
import { WsModule } from '@typenet/web-socket';
import { AppModule } from './src/app-module';
import * as settings from './settings.json';

async function bootstrap() {
   const app = ApplicationFactory.create(AppModule);
   app.useSettings(settings);
   WsModule.registerWebSockets(app);
   await app.run();
}

bootstrap();
```

Figura 18 - codul necesar înregistrării funcționalităților pachetului @typenet/web-socket într-o aplicație deja existentă

[15] https://socket.io

După integrarea pachetului, programatorii au posibilitatea de a crea clase care să fie marcate ca hub prin decoratorul „WebSocketHub” unde se poate abona la o serie de mesaje prin decoratorul „Subscribe”, ce are ca parametru numele mesajului, şi să reacţioneze la ele în mod corespunzător. În Figura 19 este un exemplu de cod în care este implementată o funcţionalitate pentru un chat web.

```
import { WebSocketHub, Subscribe } from "@typenet/web-socket";
import { Socket } from 'socket.io';
import { messages } from './messages';

@WebSocketHub()
export class ChatHub {

   @Subscribe('join')
   public connect(socket: Socket, message: any): void {
       socket.broadcast.emit('user-joined', { username: message.username });
       socket.emit('welcome', messages.welcome);
   }

   @Subscribe('message')
   public messageReceived(socket: Socket, message: any): void {
       socket.broadcast.emit('user-message', message);
   }

   @Subscribe('typing')
   public typing(socket: Socket, message: any): void {
       socket.broadcast.emit('user-typing', { username: message.username });
   }

   @Subscribe('left')
   public userLeft(socket: Socket, message: any): void {
       socket.broadcast.emit('user-left', { username: message.username });
   }
}
```

Figura 19 - exemplu de cod pentru a realiza un chat folosind @typenet/web-socket şi socket.io

Aşadar, acest pachet aduce funcţionalităţi pentru comunicare în timp real folosind minimul necesar de cod pentru a se putea integra într-un proiect deja existent şi uşor de folosit pentru a implementa cerinţe noi ale aplicaţiei.

### Pachetul „Identity”

Acest pachet a fost creat datorită necesităţii de a introduce concepte de securitate într-o aplicaţie. În majoritatea serviciilor web de tip REST există şi un sistem de autentificare bazat

pe tokenuri ce ar trebui să poată facilita limitarea accesului asupra anumitor resurse de către entităţi ce nu au drepturile necesare.

Pentru a furniza o soluţie pentru această problemă am creat pachetul „@typenet/identity”. Pachetul are aceleaşi caracteristici ca pachetul @typenet/web-socket, şi anume este uşor de folosit şi integrat într-o aplicaţie deja existentă.

Acest pachet lucrează cu JSON Web Token, abreviat JWT, mai multe detalii despre ce este şi cum funcţionează pot fi găsite în Anexa 1. Pentru a valida tokenurile am folosit un pachet scris de către comunitatea „auth0”[16], şi anume: „jsonwebtoken”[17]. Aceştia au o serie largă de pachete atât pentru JavaScript, cât şi pentru C#, Ruby şi Java.

Pentru a securiza anumite endpointuri expuse din aplicaţie, programatorii sunt nevoiţi să instaleze cele două pachete: „@typenet/identity” şi „jsonwebtoken” şi să înregistreze modulul de autentificare în iniţializarea aplicaţiei, vezi Figura 20.

```
import { ApplicationFactory } from "@typenet/core";
import { AppModule } from './src/app-module';
import { AuthModule } from '@typenet/identity';

async function bootstrap() {
   const app = ApplicationFactory.create(AppModule);
   AuthModule.registerAuth(app, { key: '123456' });
   AuthModule.registerRoleIdentification((token: any, role: string) => {
       return token.role === role;
   });
   await app.run();
}

bootstrap();
```

Figura 20 - codul necesar pentru a înregistrarea funcţionalităţilor pachetului @typenet/identity

Pentru a marca faptul că un controller este autorizat se poate folosi decoratorul „AuthorizeController”, acesta poate primi ca parametru şi rolul pe care trebuie să îl aibă tokenul pentru a considera requestul ca fiind unul autorizat. Pentru a marca o metodă drept autorizată se foloseşte decoratorul „AuthorizeMethod”, care are acelaşi comportament ca şi decoratorul pentru controlere. În Figura 21 este prezentat un controller ce permite doar administratorilor accesul la metoda „getById” şi la metoda „get” permite doar administratorilor ce sunt şi manageri.

[16] https://github.com/auth0
[17] https://github.com/auth0/node-jsonwebtoken

```
import { Controller, HttpGet, ActionResult, Ok } from '@typenet/core';
import { AuthorizeController, AuthorizeMethod } from '@typenet/identity';

@AuthorizeController('administrator')
@Controller('api/values')
export class ValuesController {

    @AuthorizeMethod('manager')
    @HttpGet('')
    public get(): ActionResult {
        return new Ok([1, 2, 3]);
    }

    @HttpGet(':id')
    public getById(): ActionResult {
        return new Ok();
    }
}
```

Figura 21 - controller autorizat

## Alte pachete

În proiecţia iniţială a framework-ului am optat în a separa capacităţile în 8 pachete, vezi Figura 7, multe dintre ele au rămas la un nivel de concept. De aceea, în acest subcapitol voi discuta pe scurt ce funcţionalităţi ar trebui să expună.

- Pachetul „validation” are ca scop principal validarea conţinutului unui request pe baza unei scheme de validare, acestă funcţionalitate este des întâlnită în aplicaţii ce folosesc ASP.NET Core. Acest pachet ar trebui să ofere un strat intermediar între controllerele aplicaţiei şi lumea exterioară, în care requesturile ce nu respectă un set de reguli să fie catalogate drept request eronat („bad request”).
- Pachetul „swagger” ajută programatorul să îşi documenteze endpointurile expuse prin standardul OpenAPI[18]. Acest lucru ajută la înţelegerea capabilităţilor serviciului creat fără a avea acces la codul sursă. O definire riguroasă a documentaţiei OpenAPI poate conduce la o integrare mai uşoară a unor servicii externe cu serviciile expuse ce folosesc framework-ul typenet. Există instrumente care pot genera cod sursă client-side ce consumă serviciile ce au o documentaţie riguroasă a capabilităţilor, ceea ce poate fi integrat în procesul de testare a aplicaţiei.

[18] https://swagger.io/specification/

- Pachetul „CQRS” a fost proiectat pentru a facilita realizarea şablonului arhitectural cu acelaşi nume[19]. Acesta se poate extinde cu implementări pentru a realiza şi „event sourcing”[20], o modalitate de a reprezenta starea aplicaţiei printr-o serie de evenimente cronologice care au alterat starea iniţiala a aplicaţiei.
- Pachetul „transport” e gândit pentru a oferi o metodă de interacţionare cu sisteme de mesagerie între aplicaţii.
- Pachetul „database” are ca scop îmbunătăţirea interacţiunii cu diferite baze de date. Interacţiunea cu bazele de date este posibilă şi fără existenţa lui, framework-ul îl lasă pe programator să îşi aleagă singur pachetul care sa îi faciliteze interacţiunea cu baza de date, dar typenet-ul ar putea veni în ajutorul lor, pentru a scurta timpul de integrare cu cele mai folosite dintre ele.

[19] https://martinfowler.com/bliki/CQRS.html
[20] https://martinfowler.com/eaaDev/EventSourcing.html

# Beneficii

Typenet este o soluţie pentru a realiza servicii web în Node.js folosind TypeScript. Acesta vine cu abordări mature preluate şi adaptate mediului Node.js din framework-uri precum Angular şi ASP.NET Core.

Acesta este proiectat folosind o arhitectură de tip micro-kernel ce sporeşte performanţa, ajută proiectele print-o separare riguroasă a funcţionalităţilor pe care le are framework-ul şi astfel aplicaţiile vor depinde doar de pachetele necesare lor. Optarea pentru a avea cât mai puţine dependinţe de alte pachete terţe ajută la scăderea în dimensiuni a produsului final care va ajunge sa fie livrat clienţilor, îmbunătăţind în acelaşi timp şi performanţele lui.

Framework-ul este scris în TypeScript, ceea ce a îmbunătăţit procesul de dezvoltare a lui. TypeScript-ul vine cu o serie de beneficii precum compilare şi verificare statică a codului care creşte productivitatea pe proiectele ce folosesc acest superset.

# Concluzii şi direcţii viitoare

Framework-urile existente în Node.js, în marea majoritate a cazurilor, nu se concentrează asupra codului pe care programatorii trebuie să îl scrie pentru a se folosi de funcţionalităţile oferite de catre framework. Acest lucru dăunează productivităţii şi experienţei avute cu framework-uri de acest tip.

Cred că un framework, precum cel realizat în această licenţă, aduce mult mai multă valoare unui proiect, deoarece folosirea lui influenţează programatorii să folosească bune practici de programare, să separe codul pe care îl scriu în module şi să se gândească mereu la arhitectura pe care o aleg, ajutându-i cu mecanisme care sa le faciliteze realizarea ei.

După cum este specificat şi în Capitolul 3. Arhitectura typenet, nu multe pachete de tip modul au trecut de stadiu de proiectare, implementarea lor poate continua adăugând capabilităţi diverse ce rezolvă probleme des întâlnite în servicii web. Cred că o abordare asemănătoare cu cea a framework-ului Angular, care publică şi pachete destinate pentru a scrie teste, ar creşte productivitatea semnificativ de mult încât să se motiveze alegerea făcută.

# Bibliografie


- https://nodesource.com/node-by-numbers
- https://insights.stackoverflow.com/survey/2019#technology-_-other-frameworks-libraries-and-tools
- https://angular.io/docs
- https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2
- https://expressjs.com
- https://www.fastify.io
- https://npm.anvaka.com/#/view/2d/express
- https://nodejs.org/en/
- https://restfulapi.net
- https://v8.dev/docs
- https://blog.acolyer.org/2017/09/19/to-type-or-not-to-type-quantifying-detectable-bugs-in-javascript/
- https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html
- https://www.restapitutorial.com/lessons/httpmethods.html
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2
- https://socket.io
- https://github.com/auth0
- https://github.com/auth0/node-jsonwebtoken
- https://swagger.io/specification/
- https://martinfowler.com/bliki/CQRS.html
- https://martinfowler.com/eaaDev/EventSourcing.html

# Anexe

## Anexa 1

JSON Web Token, abreviat JWT este un standard bazat pe JSON de creare de token-uri pentru acces ce conţin o serie de drepturi. Acesta este structurat în trei părţi: antet, corp şi semnătură, care sunt separate prin punct.

Antetul are rolul de a identifica ce algoritm a fost folosit în generarea semnăturii. Această informaţie este folosită ulterior pentru a valida dacă un token a fost creat de aceeaşi entitate sau dacă a fost corupt.

Corpul sau payload-ul conţine toate proprietăţile şi drepturile pe care le are deţinătorul tokenului. Există şi o serie de şapte proprietăţi standard precum identitatea care a creat tokenul, cui îi este adresat tokenul, când expiră ş.a.m.d.

Semnătura este calculată folosindu un algoritm de criptare specificat şi în antet, ce criptează conţinutul antetului reprezentat în baza 64, concatenat prin punct cu cel al payload-ul reprezentat în baza 64, folosind o cheie privată.

Tokenul poate fi folosit cu uşurinţă în requesturile HTTP cu scopul de a autoriza requestul. De asemenea, server-ul nu trebuie să memoreze toate token-urile generate, deoarece poate mereu să valideze dacă un token a fost generat de el sau dacă a expirat.