Typenet

Un framework Node.js pentru aplicații scalabile

Cuprins

[Introducere](#_o9d7n9yjrxxu) 2

[Context](#_689b3ff7kcb4) 2

[Motivație](#_fyc2d4fjvuw3) 4

[Vocabular](#_xj5ddfz77at9) 6

[Contribuție](#_qr0v21kele14) 7

[Capitolul 1. Stadiul framework-urilor Node.js](#_3wfcfr8fc82r) 8

[Capitolul 2. Descrierea soluției](#_4vjmj0myt3wh) 10

[Tehnologii folosite](#_c85wtsizhfgz) 10

[Node.js](#_hsp5v9x4iw5m) 10

[TypeScript](#_tny6duyyx2c6) 12

[Capitolul 3. Arhitectura typenet](#_izap9o2n1wcp) 14

[Arhitectura micro-kernel](#_d3ki0eu1kzup) 14

[Pachetul „core”](#_az9f6hrtqdz0) 16

[Command line interface](#_hq1ivxxywuit) 23

[Pachete de tip modul](#_66q5omhxf94h) 26

[Pachetul „web-socket”](#_p0z7kuljsbvv) 26

[Pachetul „Identity”](#_lfig8tlm8ke4) 28

[Alte pachete](#_blmyt4qx8i1i) 29

[Beneficii](#_ntvuglvulabm) 30

[Concluzii și direcții viitoare](#_9bcu6het2h4a) 31

[Bibliografie](#_3gkt1b7whh05) 32

[Anexe](#_n3y7lg5wrkmc) 33

[Anexa 1](#_40vw293cclc3) 33

### 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.

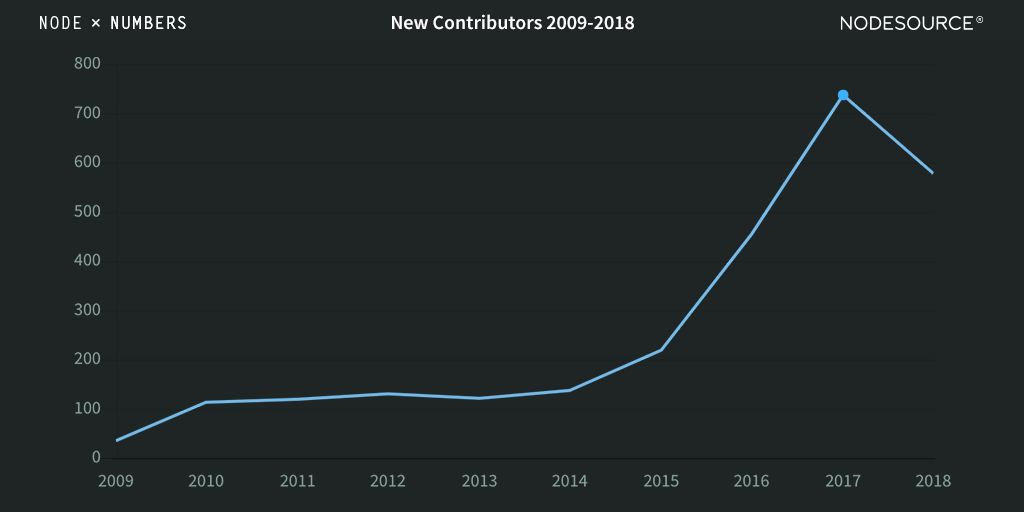


Figura 1 - Creșterea avută de Node.js din 2009 până în 2018 a numărului de contribuitori noi[[1]](#footnote-0)

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.

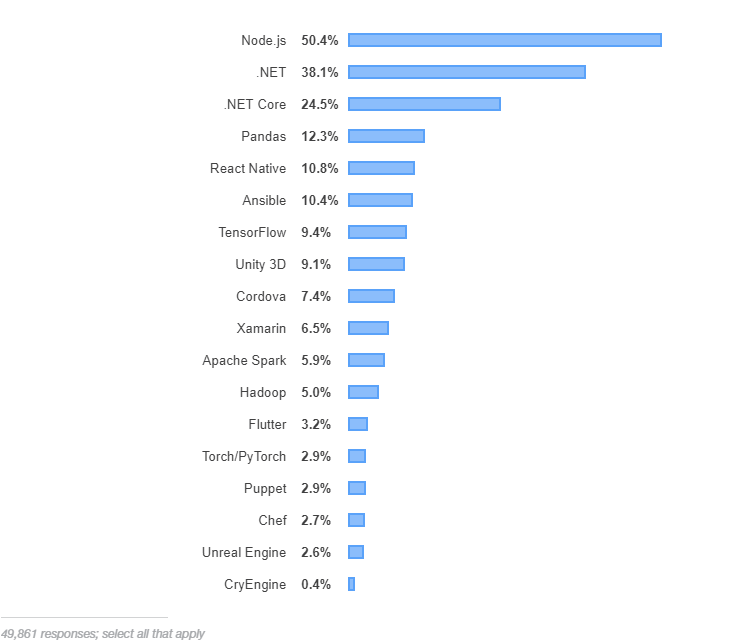


Figura 2 - Statistică făcută de către Stackoverflow referitoare la tehnologiile folosite de către programatori în anul 2019 (49,861 participanți la statistică)[[2]](#footnote-1)

Node.js oferă o suită de pachete pentru a realiza aplicații scrise JavaScript destinate aplicațiilor server-side, iar comunitatea se folosește de aceste pachete pentru a crea diferite framework-uri și 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 și ușor și care sa fie viabile, rapide și scalabile.

#### Motivație

Am simțit o lipsă de maturitate în ceea ce privește framework-urile existente în lumea Node.js, comparând cu altele mai mature precum cele de la Microsoft și anume .NET, în principal ASP.NET Core. În general, framework-urile rezolvă o problemă specifică și oferă o serie de funcționalități, doar că neglijează partea de cum anume își 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 menționate 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 și de a lua decizii de structurare a proiectului.

Deoarece Node.js este un runtime environment pentru JavasScript, acesta vine atât cu avantajele, cât și 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, crește productivitatea, doar că odată cu creșterea în dimensiune a proiectului scade productivitatea, fiind foarte greu de întreținut codul. Lipsa unui instrument care sa le ofere posibilitatea de a analiza posibile probleme ale codului precum nerespectarea signaturii unei metode sau accesarea unei proprietăți inexistente de pe un obiect crește numărul bugurilor introduse in aplicație din cauza erorilor umane. Dar odată cu apariția supersetului TypeScript toate aceste probleme sunt rezolvate. Despre TypeScript și cum anume rezolvă problemele existente în JavaScript voi discuta mai multe în Capitolul 2. D[escrierea soluției](#_4vjmj0myt3wh).

Datorită dificultăților 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 își arhitectura aplicația într-un mod ușor și rapid, care va conduce la o creștere a productivității atunci când se începe un proiect nou, accelerând prima iterație a proiectului. Totodată, framework-ul nu ar trebui să afecteze performanța cu care vine Node.js și să fie rapid, viabil și ușor de scalat.

Încă de la începutul dezvoltării framework-ului am decis ca pachetele pe care le voi crea nu trebuie să aibă multe dependințe care să îngreuneze procesul de dezvoltare. Ca inspirație în ceea ce privește arhitecturarea framework-ului am urmărit același șablon pe care îl are Angular 8, deoarece este modularizat bine și gândit pentru aplicații scalabile. De asemenea, făcând comparație între framework-uri precum ASP.NET Core, am decis să preiau ideile cele mai bune care ar aduce valoare și ar îmbunătăți la fel de mult productivitatea programatorilor și să le adaptez framework-ului și mediului Node.js.

Lucrarea este împărțită în trei capitole, în care în primul capitol: Capitolul 1. [Stadiul framework-urilor Node.js](#_3wfcfr8fc82r) voi descrie problemele și abordările anterioare ale framework-urilor existente în lumea Node.js pentru aplicații server-side. Al doilea capitol: [Capitolul 2. Descrierea soluției](#_4vjmj0myt3wh) voi prezenta sumar soluția mea pentru probleme discutate în capitolul întâi; de asemenea, voi prezenta în detaliu tehnologiile folosite și cum anume influențează și îmbunătățesc experienta programatorilor. În al treilea capitol: [Capitolul 3. Arhitectura typenet](#_q56p3eowopw5) voi discuta arhitectura aleasă pentru a implementa soluția descrisă în capitolul anterior și motivarea alegerii ei, voi prezenta pachetele importante ale framework-ului, ce probleme rezolvă și cum anume pot fi folosite într-o aplicație.

### Vocabular

Acest capitol are rolul de a defini termenii din limba engleză care vor fi folosiți pe parcursul documentației și nu au o traducere directă, sau care își pierd din înțeles odată traduși.

* Framework - reprezintă o serie de funcționalități destinate programatorilor ce au ca scop rezolvarea unor probleme des întâlnite. Aceste funcționalități 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 față de o librărie, și anume: un framework este extensibil, codul sursă nu poate fi modificat și fluxul de control este dictat de către framework, nu de către apelant (inversion of control).
* Runtime environment - este un mediu de execuție a unei aplicații în cadrul unui sistem de operare. Prin acest mediu, aplicațiile pot trimite instrucțiuni sau comenzi către procesor și 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 interacțiune cu un program în care un user emite o serie de comenzi ce reprezintă o înșiruire de cuvinte/numere.
* Superset - reprezintă un limbaj de programare care conține toate funcționalitățile unui alt limbaj de programare, pe care îl extinde sau îl îmbunătățește cu alte funcționalități.
* Open-source - reprezintă o metodologie pragmatică de distribuire a aplicații sau cod sursă fără restricții, de acces, citire sau copiere. Aplicațiile open-source permit accesul oricui să contribuie la dezvoltarea și îmbunatățirea 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 conține 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 producție după ce codul trece de verificările făcute de către o serie de teste automate, dacă este cazul.

### Contribuție

În lipsa framework-urilor orientate spre a îmbunătăți experiența și productivitatea programatorilor am decis să facilitez aceste aspecte prin crearea typenet, un framework Node.js pentru aplicații scalabile. Am analizat framework-uri de succes din lumea Node.js care au ca focus crearea de aplicații client-side, dar și framework-uri pentru crearea de aplicații server-side din lumea ASP.NET. Am observat cum anume influențează și îmbunătățește Angular[[3]](#footnote-2) experiența și productivitatea programatorilor lucrând cu el de aproape doi ani, și am ajuns la concluzia că o arhitecturare corectă a framework-ului are un impact mare asupra codului scris de către programatori, dar și limbajul TypeScript, fiind un limbaj cu tipuri de date, crește productivitatea odată cu creșterea în dimensiuni a proiectului. Angular fiind concentrat asupra problemelor ce țin de aplicații client-side, am fost nevoit să analizez un framework destinat aplicațiilor server-side, și anume ASP.NET Core[[4]](#footnote-3). Acesta vine cu multe idei bune precum middleware system, despre care voi discuta în [Capitolul 3. Arhitectura typenet](#_q56p3eowopw5), subcapitolul [Pachetul „Core”](#_az9f6hrtqdz0).

### Capitolul 1. Stadiul framework-urilor Node.js

Din ce în ce mai mulți programatori se orientează spre a face aplicații folosind JavaScript, în special pentru aplicații client-side. Acest lucru a condus la o creștere mare a numărului de framework-uri făcute pentru a crea aplicații web într-un mod rapid și cu minimul de cod necesar. Folosirea limbajului JavaScript atât pentru aplicații web, cât și pentru servicii web, a condus la o adopție rapidă a Node.js și, totodată, la apariția de framework-uri. Dar se simte o diferență 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 aplicații client-side. Companii precum Google și Facebook, împreună cu comunitatea, au creat framework-uri precum Angular și respectiv React.js. Aceste framework-uri vin cu îndrumări de cum să se implementeze anumite soluții de probleme, respectă practicile bune din programare și 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]](#footnote-4). Sunt câteva framework-uri Node.js care sunt construite în jurul acestui framework și pe care îl extind sau care cel puțin se inspiră din abordările lui, unul dintre ele ar fi Fastify[[6]](#footnote-5), gândit la fel ca și Express.js, doar că pune mai mult accentul pe eficiență. Acesta oferă funcționalități minimaliste pentru a crea servicii web. Acesta pune la dispoziție un sistem de rutare prin care programatorii își pot crea endpoint-uri corect semantice din punct de vedere REST. În Figura 3 se poate evidenția simplitatea creării unei aplicații 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 aplicație folosind Express.js care expune un endpoint „hello” la care va răspunde cu mesajul „Hello World!”

Odată cu creșterea în dimensiuni a proiectului lucrurile se vor complica, întrucât Express nu pune la dispoziție un sistem de separare a endpoint-urilor pe controller care să aibă aceeași întrebuințare, și anume să grupeze operațiile posibile pentru o anumită resursă într-un singur fișier, aceste lucruri îi revin programatorilor să își creeze singuri mecanisme asemănătoare sau să păstreze într-un singur fișier toate endpoint-urile expuse. De asemenea, lipsa unui sistem de injectare a dependințelor nu facilitează realizarea și separarea codului pe module granulare care să aibă o singură întrebuințare în toată aplicația.

Odată cu adopția micro-serviciilor ca șablon arhitectural, serviciile web de tip REST devin mici ca dimensiuni, devin foarte decuplate de celelalte servicii și în general au responsabilitatea de a rezolva o singură problemă de business. Dar datorită numărului mare de dependințe pe care îl are express-ul cu alte pachete Node.js, acesta nu este foarte eficient pentru micro-servicii, el având aproape 50 de dependințe pe arborele dependințelor,[[7]](#footnote-6) ceea ce îngreunează procesul de dezvoltare și afectează și performanța.

Așadar, 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 funcționalități care să ajute la realizarea de bune practici, precum inversarea dependințelorprintr-un sistem de injectare a dependințelor. Dar un framework care să crească productivitatea și experiența nu ar trebui să ofere doar un sistem de rutarea requesturilor și un sistem de injectare a dependințelor, el ar trebui să faciliteze și realizarea de arhitecturi complexe precum onion sau micro-servicii și, în același timp, să fie performant și ușor de scalat.

### Capitolul 2. Descrierea soluției

Pentru a rezolva problemele existente în framework-urile Node.js am decis să realizez un framework împărțit cât mai granular, astfel încât fiecare proiect să fie nevoit să instaleze doar de ce anume are nevoie și chiar folosește. Datorită experienței avute de către Express.js am optat pentru a crea pachete Node.js care să aibă dependințe directe cât mai puține.

Pentru a crește productivitatea și a îmbunătăți experiența programatorilor am decis să dezvolt framework-ul folosind TypeScript, care este folosit și de către Angular. De asemenea, pentru a face cât mai ușoară trecerea programatorilor care sunt familiarizați cu Angular, am adaptat structurarea și unele concepte folosite, care se pretează și într-un framework server-side precum „Module” sau „Injectable”. TypeScript oferă posibilitatea de a adăuga metadata peste codul scris folosind decoratori, lucru întâlnit și în C# prin atribute, dar și în JAVA prin adnotări, ceea ce a făcut posibilă împrumutarea modului de a declara endpointuri, controllere și module în framework-urile mature.

#### Tehnologii folosite

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

##### Node.js

Node.js este un runtime environment pentru cod JavaScript construit în engine-ul V8 de la chrome.[[8]](#footnote-7) Node.js oferă posibilitatea programatorilor de a-și crea command line tools și aplicații server-side, fie servicii web de tip RESTful[[9]](#footnote-8), fie aplicații real-time folosind protocolul WebSocket.

V8 este un mediu de execuție de cod JavaScript sau WebAssembly care a fost inițial construit de către Google pentru Google Chrome[[10]](#footnote-9), iar mai apoi a devenit open source. Acesta este scris în C++, de aceea este rapid și performant când vine vorba de viteza de execuție a codului, întrucât compilează codul JavaScript direct în cod mașină înainte de a-l executa. Acesta se ocupă de alocarea obiectelor în memorie și de colectarea obiectelor care nu sunt folosite. V8 implementează atât ECMAScript, cât și WebAssembly, și poate fi executat pe mașini 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 operații ne- blocante de I/O, de aceea toate apelurile de tip I/O folosesc un callback pentru a gestiona răspunsul operației.

Node.js pune la dispoziție o serie de pachete de bază cu care se pot realiza diferite interacțiuni 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 execuție, dar a cărei răspuns nu afectează aplicația. Cea mai deasă întrebuințare este de a trimite email-uri, deoarece în majoritatea aplicațiilor trimiterea de emailuri nu ar trebui să blocheze sau să ofere un feedback despre starea trimiterii.
* Pachetul „http” pune la dispoziție o serie de funcționalități 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 soluției.

|  |
| --- |
| 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 și răspunde cu „Hello World” pentru fiecare request

* Pachetul „fs” (File system) ușurează interacțiunea cu fișierele/directoarele de pe server. Toate operațiile au implementări atât blocante cât și ne-blocante.

Node.js pune la dispoziție 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 îmbunatățirea performanței și a scalabilității ci doar la menținerea 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 soluție open-source dezvoltată și întreținută 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 crește productivitatea și. de asemenea, scade numărul de bug-uri, după cum arată un studiu care evidențiază faptul că numărul de bug-uri scade cu 15%[[11]](#footnote-10) pe proiectele care au adoptat TypeScript.

TypeScriptul oferă plug-in-uri pentru diferite IDE-uri prin care facilitează sugestii de autocompletare a codului, navigare ușoară prin cod pentru a vedea implementarea unor clase sau definirea unor interfețe, toate aceste aspecte cresc productivitatea programatorilor. TypeScript include un set de funcționalități de programare orientată pe obiecte, printre care declararea de interfețe, clase, metode și clase generice, moștenire, încapsulare, polimorfism și separarea codului sau a conceptelor prin module și namespace-uri. De asemenea, oferă funcționalități de adăugare de metadată peste codul scris ceea ce ajută la declarare de comportament pentru clase, metode și parametrii funcțiilor. În Figura 5 se poate vedea un exemplu prin care un decorator marchează o clasă ca fiind un controller, prin astfel de abordări sunt nevoiți cei ce vor folosi framework-ul să marcheze diferite concepte ale framework-ului. Mai multe despre cum anume vor fi folosiți respectivii decoratori voi vorbi în [Capitolul 3. Arhitectura typenet](#_izap9o2n1wcp).

|  |
| --- |
| 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, motivația alegerii și beneficiile cu care vine arhitectura aleasă, și anume arhitectura de tip micro-kernel, mai multe detalii despre aceasta voi discuta în subcapitolul următor: [Arhitectura micro-kernel](#_d3ki0eu1kzup).

Încă din fazele incipiente ale framework-ului am decis să separ concepte, soluții și funcționalitățile într-un mod cât mai granular, astfel încât un proiect să nu fie nevoit să își 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 și pachete ce sunt folosite doar pe partea de dezvoltare ce au ca scop îmbunătățirea scrierii de teste.

De asemenea, pentru a îmbunătăți construirea de proiecte am concluzionat că un command line interface ar spori interacțiunea pe care o are un programator cu framework-ul oferindu-i funcționalități care să îi ușureze munca și, 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 funcționalități voi discuta în subcapitolul: [Command line interface](#_hq1ivxxywuit).

#### Arhitectura micro-kernel

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

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

Sistemul central de obicei conține minimul de funcționalități necesare pentru a face aplicația funcțională, dar și funcționalități prin care modulele externe se pot integra, pentru a extinde funcționalitățile deja prezente în core. În Figura 6 se evidențiază separarea care o presupune această arhitectură dar și importanța pe care o are sistemul central.

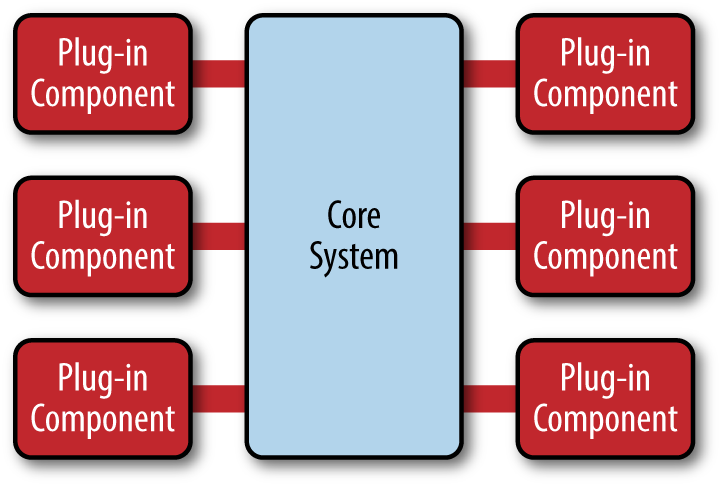


Figura 6 - Arhitecura micro-kernel[[12]](#footnote-11)

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

De obicei sistemul central știe despre existența modulelor disponibile, expunând o funcționalitate de a înregistra fiecare modul.

Această arhitectură, ca orice ce altă arhitectură, vine cu avantaje și dezavantaje. Un avantaj, dar în același timp și un dezavantaj al acestei arhitecturi este separarea capabilităților într-un mod cât mai granular. Lucru ce îmbunătățește performanța, abilitatea de a scrie teste, ușor de publicat și distribuit. Dar, în același, timp nu este ușor de gestionat proiectarea modulelor și a sistemului central, complicând procesul de dezvoltare cautând mereu soluții pentru a integra cat mai ușor o funcționalitate nouă fără a afecta codul deja existent.

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

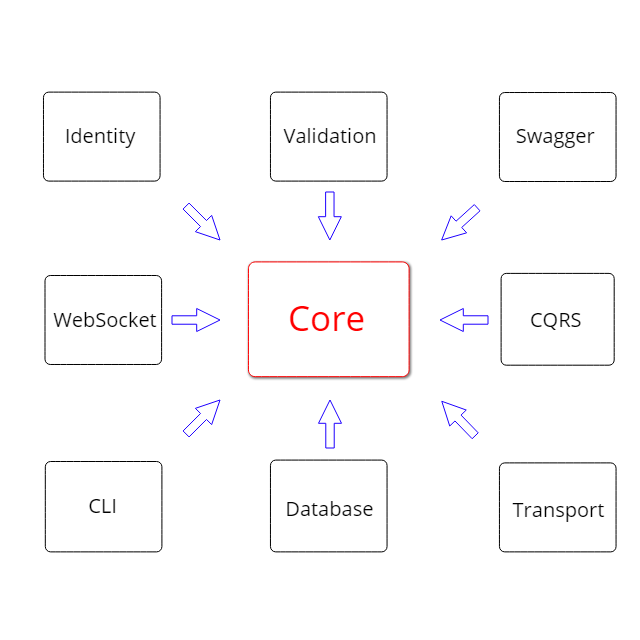


Figura 7 - Arhitectura framework-ului typenet

Această arhitectură facilitează toate nevoile inițiale pe care le-am avut și anume: separarea funcționalităților cât mai granular, astfel încât aplicațiile pot să depindă doar de ce pachete au nevoie pentru a funcționa și de a păstra calitățile Node.js intacte, precum performanța și scalabilitatea.

#### Pachetul „core”

Acest pachet are cea mai mare responsabilitate, și anume de a oferi minimul necesar de funcționalități pentru a realiza o aplicație server-side. Însă nu este suficient să ofere minimul necesar pentru a îmbunătăți experiența programatorilor și pentru a creste productivitatea, acesta trebuie să ofere și capabilități prin care un programator să poate respecta bunele practici, să poată introduce cu ușurința valoare în proiect, dar și să ofere posibilitatea programatorului de a integra pachete de tip modul prin care el să extindă aplicația cu capabilitățile necesare lui. De asemenea, încă de la începutul proiectării framework-ului am decis sa minimizez pe cât posibil dependințele pe care le voi avea de pachetele externe pentru a implementa specificațiile framework-ului. Lucru ce se poate evidenția în Figura 8 în care este modelat arborele de dependențe ale pachetului „@typenet/core”, din care se poate observa numărul redus de dependințe directe.

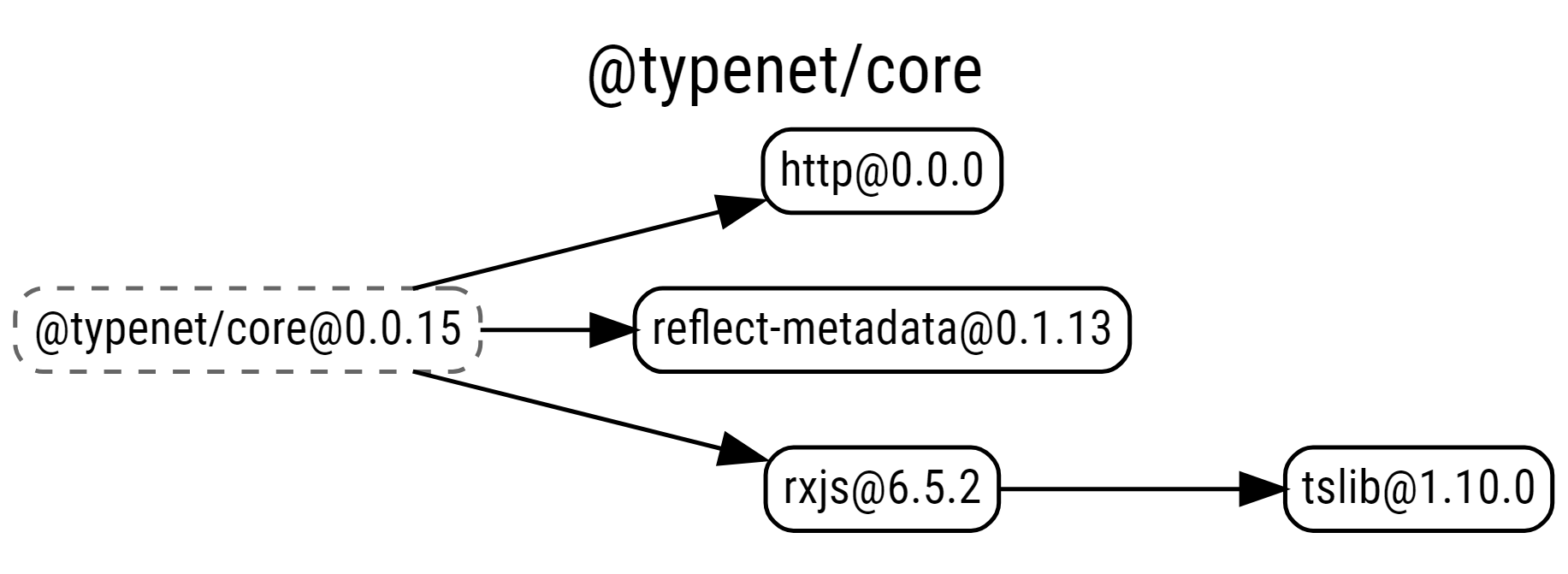


Figura 8 - arborele de dependințe a pachetului „@typenet/core”

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 gestionarea 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 comparând 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](#_hq1ivxxywuit)) 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!”.

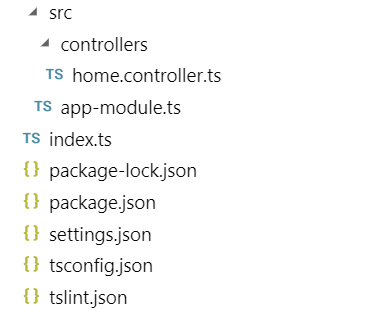


Figura 9 - structura de baza a proiectului folosind typenet

|  |
| --- |
| 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 și Figura 4, există o diferențe mare între modul cum se declară o aplicație de tip „Hello World” folosind Node.js sau Express.js și typenet. În typenet codul arată mai curat și împărțit pe responsabilități. Nu există nici o referire la vreun server sau aplicație creat/ă de către Node.js și respectiv Express.js. Aceste lucruri se fac în etapa de inițializare a aplicație, lucru ce este evidențiat în Figura 11, unde este prezentat cel mai simplu mod de a înregistra o aplicație ce folosește framework-ul typenet. Vom vedea, pe parcursul următoarelor subcapitole, că în această zonă de cod se vor înregistra capabilități 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 inițializa framework-ul typenet

În Figura 10 se evidențiază folosirea câtorva funcționalități oferite de către TypeScript discutate în [Capitolul 2. Descrierea soluției](#_4vjmj0myt3wh). Am folosit decoratori pentru cu scopul de a marca comportament diferite pe care îl are codul scris de către un programator dar și de a colecta informațiile necesare comportamentului respectiv. Spre exemplu, decoratorul „Controller” are nevoie, pe lângă o clasă la care să îi atribuie proprietăți specifice, și de un endpoint la care să expună funcționalitățile oferite de către controller.

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

|  |
| --- |
| 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 folosiți toate metodele HTTP

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

Pentru a ajuta programatorii să își 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 știe despre existența tuturor controllerelor din acel modul, despre alte module de care se folosește, dar și de toate componentele care pot fi injectate.  
Pentru a ajuta programatorii în a realiza bune practici din programare precum inversarea dependințelor, framework-ul oferă funcționalități de injectare a dependințelor. Clasele care pot fi injectate trebuie marcate cu decoratorul „Injectable” și specificat în lista de „providers” din modulul în care face parte. De asemenea, atunci când se marchează o clasa ca fiind injectabilă se poate specifica și o perioadă de viață a instanțelor create de către framework a respectivelor clase. Exista trei tipuri de perioadă de viață: „singleInstance”, „scopedInstance” și „transientInstance”. Perioada de viață „singleInstance” specifică framework-ului să creeze o singură instanță în întreaga aplicație. Marcarea unei clase injectabile cu perioada de viața „scopedInstance” va specifica framework-ului să creeze câte o instanță de clasă pentru fiecare request în parte. Iar „transientInstance” va specifica framework-ului sa creeze cate o instanța de fiecare data când este nevoie de una.

Pentru a facilita integrarea rapidă și ușor 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 mulțime 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 și se termină cu „DefaultRequestHandler” ce are rol de a delega requestul către o metoda 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” și cum anume trece requestul prin fiecare componenta în parte.

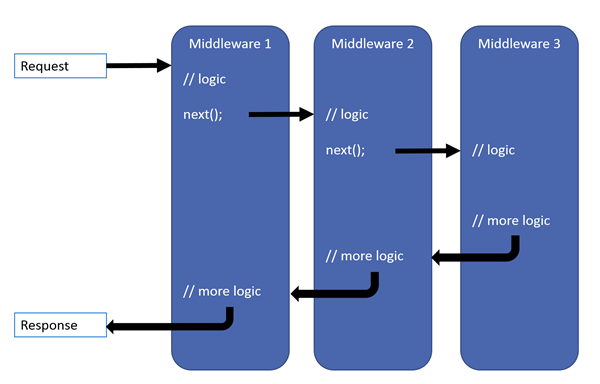


Figura 14 - un sistem de trei middleware-uri[[14]](#footnote-13)

Componente de tip middleware nu sunt desintate doar pachetelor de tip modul ci programatorilor care creează aplicații. Aceștia pot folosi acest mecanism pentru a înregistra activitatea aplicație, pentru a trata excepțiile aruncate prin aplicație ș.a.m.d. Pentru a marca o clasa ca fiind o componenta de tip middleware se folosește decoratorul „Middleware” iar clasa respectiva este nevoita sa implementeze interfața „PipelineMiddleware” corespunzător.

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

|  |
| --- |
| 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 funcționalitate programatorii își pot crea fișiere de configurare în care să specifice portul la care să se deschidă aplicația, diferite conexiuni catre baze de date sau alte tipuri de configurări pe care mai apoi să le poată modifica într-un pipeline de continuous deployment și ajusta cerințelor fiecărui mediu de publicare a codului.

Așadar, acest pachet oferă funcționalități destinate atât programatorilor, prin sistemul de inversie a dependințelor, prin rutarea requesturilor folosind controllere și metode a lor, prim modularizarea codului folosind module, prin sistemul de componente de tip middleware; cât și lansării aplicație pe diferite medii prin folosirea fișierelor de configurare.

#### Command line interface

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

Prin acest pachet programatorii își pot genera:

* Proiecte de la zero
* Controller
* Module

Toate aceste funcționalități 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 in loc de al treilea parametru care poate sa nu existe pe disc, acesta va fi folosit și ca nume al aplicației. Această comandă va genera minimul de fișiere pentru a crea o aplicație, va instala dependințele proiectului și va configura proiectul să lucreze cu TypeScript stabilind și un set de reguli de redactare a codului. Pentru a vedea structura proiectului generat vezi Figura 9.

Pentru a genera controller sau module se poate folosi aceeași comandă dar oferind parametri corespunzători. Comandă care generează astfel de fișiere este: „typenet generate <template> <path>”, unde al doilea parametru poate fi „controller” sau „c” pentru a genera controllere și „module” sau „m” pentru a genera module, iar al treilea parametru reprezintă calea unde să fie salvat fișierul creat. Numele clasei controllerului sau a clasei modulului este preluat din calea unde trebuie salvat respectivul fișier. 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 și fișierul din Figura 17.

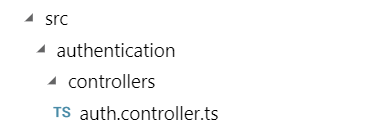


Figura 16 - structura folderelor create de comanda „typenet g c src/authentication/controllers/auth”

|  |
| --- |
| 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 fișierului 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](#_d3ki0eu1kzup). Voi discuta despre capabilitățile introduse, motivația 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]](#footnote-14), 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” împreuna 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 aplicație.

|  |
| --- |
| 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

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 și să reacționeze la ele în mod corespunzător. În Figura 19 este un exemplu de cod în care este implementat o funcționalitate 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 și socket.io

Așadar, acest pachet aduce funcționalități pentru comunicare în timp real folosind minimul necesar de cod pentru a se putea integra într-un proiect deja existent și ușor de folosit pentru a implementa cerințe noi ale aplicației.

##### Pachetul „Identity”

Acest pachet a fost creat datorită necesității de a introduce concepte de securitate într-o aplicație. În majoritatea serviciilor web de tip REST există și un sistem de autentificare bazat pe tokenuri ce ar trebui să poată facilita limitarea accesului asupra anumitor resurse de catre entități ce nu au drepturile necesare.

Pentru a furniza o soluție pentru această problemă am creat pachetul „@typenet/identity”. Pachetul are aceleași caracteristici ca pachetul @typenet/web-socket și anume este ușor de folosit și integrat într-o aplicație deja existentă.

Acest pachet lucrează cu JSON Web Token, abreviat JWT, mai multe detalii despre ce este și cum funcționează vezi Anexa 1. Pentru a valida tokenurile am folosit un pachet scris de catre comunitatea „auth0”[[16]](#footnote-15), și anume: „jsonwebtoken”[[17]](#footnote-16). Aceștia au o serie largă de pachete atât pentru JavaScript, cât și pentru C#, Ruby și Java.

Pentru a securiza anumite endpointuri expuse din aplicație, programatorii sunt nevoiți să instaleze cele două pachete: „@typenet/identity” și „jsonwebtoken” și să înregistreze modulul de autentificare în inițializarea aplicației, 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 funcționalităților pachetului @typenet/identity

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

|  |
| --- |
| 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 proiecția inițială a framework-ului am optat în a separa capacitățile 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 funcționalități ar trebui să expună.

* Pachetul „validation” are ca scop principal validarea conținutului unui request pe baza unei scheme de validare, acestă funcționalitate este des întâlnită în aplicații ce folosesc ASP.NET Core. Acest pachet ar trebui să ofere un strat intermediar între controllerele aplicației și 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 își documenteze endpointurile expuse prin standardul OpenAPI[[18]](#footnote-17). Acest lucru ajută la înțelegerea capabilităților serviciului creat fără a avea acces la codul sursă. O definire riguroasă a documentației OpenAPI poate conduce la o integrare mai ușoară a unor servicii externe cu serviciile expuse folosind framework-ul typenet. Există instrumente care pot genera cod sursă client-side ce consumă serviciile ce au o documentație riguroasă a capabilităților, ceea ce poate fi integrat in procesul de testare a aplicației.
* Pachetul „CQRS” a fost proiectat pentru facilita realizarea șablonului arhitectural cu același nume[[19]](#footnote-18). Acesta se poate extinde cu implementări pentru a realiza și „event sourcing”[[20]](#footnote-19), o modalitate de a reprezenta starea aplicației printr-o serie de evenimente cronologice care au alterat starea inițiala a aplicației.
* Pachetul „transport” a gândit pentru a oferi o metodă de interacționare cu sisteme de mesagerie între aplicații.
* Pachetul „database” are ca scop îmbunătățirea interacțiunii cu diferite baze de date. Interacțiunea cu bazele da date este posibilă și fără existența lui, framework-ul îl lasă pe programator să își aleagă singur pachetul care sa îi faciliteze interacțiunea 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.

### Beneficii

### Concluzii și direcții viitoare

### 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 conțin o serie de drepturi. Acesta este structurat în trei părți: antet, încărcătura și semnătură care sunt separate prin punct.

Antetul are rolul de a identifica ce algoritm a fost folosit în generarea semnăturii, această informație este folosită ulterior pentru a valida dacă un token a fost creat de aceeași entitate sau dacă a fost corupt.

Încărcătura sau payload-ul conține toate proprietățile și drepturile pe care le are deținătorul tokenului. Exista și o serie de șapte proprietăți standard precum identitatea care a creat tokenul, cui ii este adresat tokenul, când expira ș.a.m.d.

Semnătura este calculată folosindu un algoritm de criptare specificat și în antet ce criptează conținutul 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 ușurință în requesturile HTTP cu scopul de a autoriza requestul. De asemenea, server-ul nu trebuie sa memoreze toate token-urile generate deoarece poate mereu sa valideze dacă un token a fost generat de el sau dacă a expirat.

1. https://nodesource.com/node-by-numbers [↑](#footnote-ref-0)
2. https://insights.stackoverflow.com/survey/2019#technology-\_-other-frameworks-libraries-and-tools [↑](#footnote-ref-1)
3. https://angular.io/docs [↑](#footnote-ref-2)
4. https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2 [↑](#footnote-ref-3)
5. https://expressjs.com [↑](#footnote-ref-4)
6. https://www.fastify.io [↑](#footnote-ref-5)
7. https://npm.anvaka.com/#/view/2d/express [↑](#footnote-ref-6)
8. https://nodejs.org/en/ [↑](#footnote-ref-7)
9. https://restfulapi.net [↑](#footnote-ref-8)
10. https://v8.dev/docs [↑](#footnote-ref-9)
11. https://blog.acolyer.org/2017/09/19/to-type-or-not-to-type-quantifying-detectable-bugs-in-javascript/ [↑](#footnote-ref-10)
12. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html [↑](#footnote-ref-11)
13. https://www.restapitutorial.com/lessons/httpmethods.html [↑](#footnote-ref-12)
14. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.2 [↑](#footnote-ref-13)
15. https://socket.io [↑](#footnote-ref-14)
16. https://github.com/auth0 [↑](#footnote-ref-15)
17. https://github.com/auth0/node-jsonwebtoken [↑](#footnote-ref-16)
18. https://swagger.io/specification/ [↑](#footnote-ref-17)
19. https://martinfowler.com/bliki/CQRS.html [↑](#footnote-ref-18)
20. https://martinfowler.com/eaaDev/EventSourcing.html [↑](#footnote-ref-19)