

Service Registry

Theodor Bogdan Vrâncean

Iunie 2018

Cuprins

1	Introducere	3
1.1	Arhitectură orientată pe microservicii	3
1.2	Serviciu de înregistrare	6
1.3	Extensibilitate	6
1.4	Load Balancer	7
2	Tehnologii	9
2.1	.NET Framework	9
2.2	C#	9
2.3	Asp.NET	10
2.3.1	Asp.NET MVC	11
2.3.2	MVC pattern	11
2.3.3	Asp.NET WebApi	12
2.4	JavaScript	13
2.5	HTML	14
2.6	CSS	14
2.7	Entity Framework	15
2.8	Microsoft SQL Server	16
3	Implementare	18
3.1	Arhitectura	18
3.1.1	Stratul de stocare a datelor	19
3.1.2	Stratul de accesare a datelor	22
3.1.3	Startul de logică de business	24

3.1.4	Stratul de prezentare	27
3.2	Injectarea dependențelor	27
3.3	Curățarea automată	28
3.4	Autentificarea și autorizarea	30
3.5	Aspectul comun	33
3.6	Managementul serviciilor	37
3.7	Managementul cluster-elor	37
3.8	Monitorizarea unui serviciu	38
3.9	Introducerea serviciilor	39
3.10	Tratarea erorilor	41
3.11	Componente	42
3.11.1	NuGet	42
3.11.2	Registry Connector	42
3.11.3	Registry Manager	43
3.12	Testare	43
3.12.1	Test Driven Development	45
4	Utilizare	48
4.1	Serviciul de înregistrare	49
4.2	Interfața grafică a serviciului de înregistrare	50
5	Planuri de viitor	53
5.1	Securitatea	53
5.2	Testarea	54
5.3	Adaptarea	55
6	Concluzii	56

CAPITOLUL 1

Introducere

1.1 Arhitectură orientată pe microservicii

Arhitectură orientată pe microservicii este o abordare relativ nouă în dezvoltarea de software. Microserviciile reprezintă aplicații mici și autonome care lucrează împreună¹. Ele sunt considerate mici relativ la un sistem monolitic care ar oferi toate funcționalitățile de care aplicația are nevoie. Cu toate acestea un microserviciu poate oferi orice fel de funcționalități, începând cu ceva simplu precum descărcarea de fișiere, până la complexe precum analiza imaginilor. Această abordare arhitecturală a venit ca o alternativă la arhitectura monolitică, în care există un singur server care satisface toate necesitățile unei aplicații. Limitările acestei abordări ies la iveală odată cu creșterea aplicației. Pentru a înțelege de ce arhitectura de microservicii începe să înlocuiască arhitectura monolitică trebuie să cunoaștem următoarele beneficii:

- Din cauza dimensiunii și complexității unui proiect monolitic, acesta este dificil de înțeles, motiv pentru care schimbările sunt mai dificil de făcut și există un risc mai mare ca acestea să producă efecte ne-

¹Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2014, p. 1.

dorite. Aceste probleme pot fi atenuate printr-un cod de calitate, dar acest lucru se întâmplă de prea puține ori. Schimbările aduse unui microserviciu nu afectează alte module și, datorită dimensiunii reduse a acestora, ele sunt și mai ușor de înțeles pentru programatori, astfel scade probabilitatea erorilor. Se poate spune că microserviciile duc un pas mai departe principiul singurei responsabilități, definit de Robert C. Martin.

- Pentru dezvoltarea unei aplicații monolitice trebuie să alegem tehnologii standardizate care să poată realiza toate cerințele aplicației. Pe de altă parte, dacă avem mai multe microservicii care colaborează, nu există această limitare, ceea ce ne permite să alegem una alta cea mai potrivită pentru fiecare serviciu. Să luăm spre exemplu un site al unei pizzerii care folosește microservicii (Fig 1). Partea de front-end doar apelează serviciile când are nevoie. Putem avea un server scris în C# care folosește un sistem de gestiune a bazei de date Microsoft Sql Server, unul scris în php cu MySql și unul scris în Node.js cu mongoDb [1.1]. Deoarece toate comunică prin protocolul http, acestea pot lucra împreună, fiecare având responsabilitatea sa.

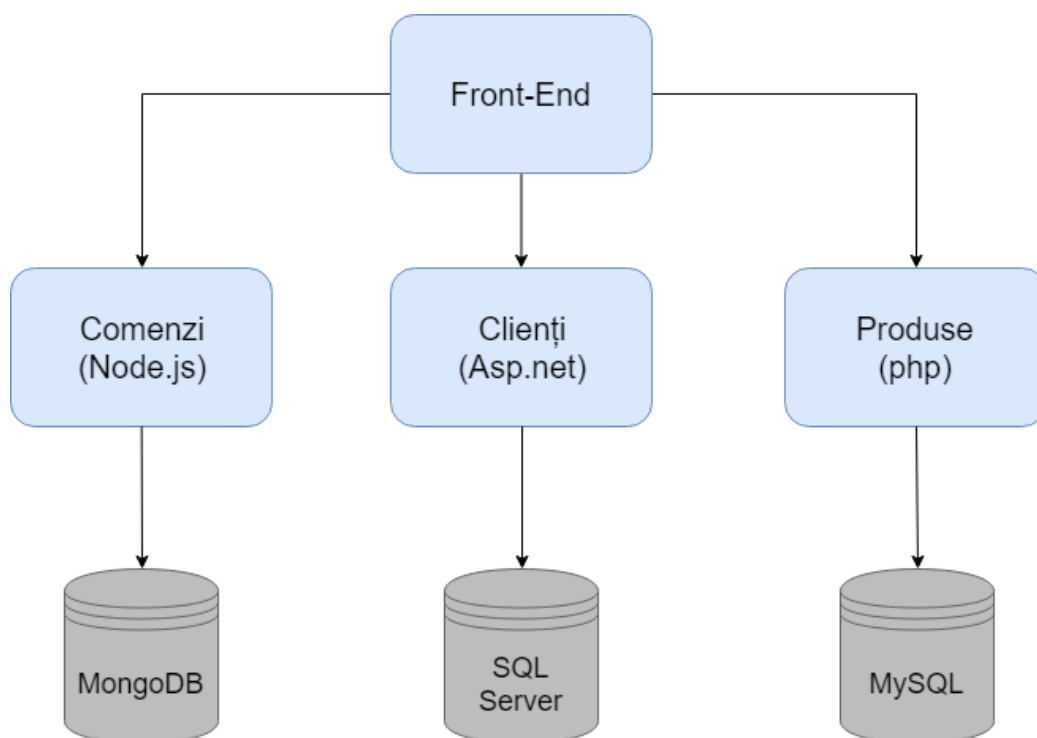


Figura 1.1: Arhitectură cu microservicii pentru un site de pizzerie

- O aplicație a cărei componente sunt distribuite este mai rezistentă, în sensul că dacă un serviciu este compromis, funcționalitățile care nu depind de acel serviciu vor continua să funcționeze. În cazul unei aplicații clasice, tot sistemul va fi compromis din cauza unei singure componente.
- Cu un serviciu mare, monolitic, trebuie să scalăm totul împreună. O parte mică a întregului nostru sistem, este constrânsă în performanță, dar dacă acest comportament este blocat într-o aplicație monolitică gigantă, trebuie să scalăm totul împreună ca o bucată². Acest lucru se face de obicei folosind mai multe servere și un server de tip load balancer care să distribuie cererile către unul din acele servere pentru a reduce munca depusă de un singur server o depune, astfel evitând supraîncărcarea și crescând performanța. În cazul arhitecturii cu microservicii, ajunge să scalăm doar acele servicii care au probleme de performanță.
- Microserviciile oferă și reutilizabilitate, ele pot fi utilizate de multiple aplicații. Dacă două aplicații au funcționalități similare, acestea nu trebuie implementate de două ori, această funcționalitate poate fi oferită de un serviciu comun, folosit de ambele aplicații. Uneori acest lucru este chiar necesar, dacă un producător de software are mai multe aplicații ce necesită autentificare, este mult mai comod pentru utilizator să aibă un singur cont cu care să se autentifice în fiecare aplicație și acest lucru este avantajos și pentru producător. Microserviciile pot fi făcute și publice, pe baza unei chei de autentificare, pe care le pot folosi pentru aplicațiile lor, persoane în afara companiei producătoare. În prezent există multe servicii de meteorologie, gratuite sau contra cost, pe care le putem folosi în aplicațiile noastre.

Bineînțeles arhitectura cu microservicii nu este o soluție miraculoasă la toate problemele, adată cu aceste beneficii, ea vine și cu dezavantaje. Fiind un sistem distribuit toate problemele acestor sisteme afectează și microservicii. O problemă este menținerea legăturii între componentele aplicației. Un serviciu își poate schimba adresa, poate deveni inactiv sau poate fi înlocuit. În aceste situații se va pierde funcționalitatea pe care acesta o oferă. Pentru a remedia această situație ar trebui să schimbăm adresele la care aplicația se așteaptă să găsească serviciul respectiv, după care ar trebui ca toți utilizatorii să facă un update la noua versiune. Această soluție simplă nu doar că nu este convenabilă, dar devine imposibilă dacă avem servicii care rulează pe

²Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2014, p. 5.

mașini virtuale sau containere de docker care sunt create dinamic în funcție de nevoie.

1.2 Serviciu de înregistrare

Soluția pe care eu am implementat-o este un sistem de tip service registry. Acesta este la bază un serviciu web care face legătura între microservicii. Astfel aplicația și serviciile trebuie să cunoască doar serviciul de înregistrare și el se va furniza datele despre celelalte servicii necesare funcționării aplicației. Serverul de înregistrare oferă trei operații de bază:

1. Înregistrarea

Serviciile se pot înregistra la server atunci când își încep funcționarea. În cazul în care serviciu folosit este făcut de un alt producător care nu cunoaște acest mecanism sau nu este dispus să îl folosească, există și posibilitatea ca administratorul aplicației să înregistreze serviciul respectiv manual prin intermediul interfeței grafice.

2. Dezînregistrarea

Serviciile au posibilitatea de a se dezînregistra, acest lucru ar trebuie să fie făcut la închiderea serviciului. În cazul în care acestea devin inactive fără a se dezînregistra, serverul le va dezînregistra automat.

3. Căutarea

Aplicația și microserviciile pot căuta un serviciu înregistrat în server.

1.3 Extensibilitate

Extensibilitatea este o problemă permanentă în dezvoltarea de software. Un program poate fi și, de obicei, trebuie dezvoltat continuu. Serviciu de înregistrare, din cauza rolului pe care îl are, nu ar trebui oprit pentru actualizări doar dacă acestea sunt absolut necesare și acest lucru trebuie să se întâmple cât mai rar posibil. În acest sens am implementat un mecanism de extensibilitate care nu necesită oprirea serverului. Aplicația poate fi extinsă prin add-inuri. Din acest motiv am expus un API pe care îl pot folosi pentru dezvoltarea acestor add-inuri. Bineînțeles, schimbările mai importante nu pot fi făcute prin add-inuri, dar acestea reprezintă un mecanism suficient de bun

pentru a adăuga funcționalități aplicației fără a întrerupe funcționarea serverului. Un alt avantaj al acestei abordări constă în posibilitatea de adăugare de funcționalități la aplicație de către utilizator. Acesta poate face un add-in folosind API-ul pe care îl pun la dispoziție. Am ales să ofer trei tipuri de add-inuri:

1. **Add-In acțiune**

Acesta poate fi declanșat de către utilizator prin interfața grafică.

2. **Add-In periodic**

Acest tip de add-in se execută la un interval de timp.

3. **Add-In pentru echilibrarea încărcăturii**

Acesta este un add-in special care conține implementarea unui algoritm de echilibrare a încărcăturii. În aplicație poate exista un singur add-in pentru echilibrarea încărcăturii. În cazul în care programul găsește mai multe astfel de add-inuri, este selectat aleator unul dintre ele.

1.4 Load Balancer

Aplicația mea, fiind un serviciu de înregistrare trebuie să ia în calcul posibilitatea de a avea mai multe servicii înregistrate cu același nume. Acest lucru poate părea că ar trebui interzis, însă eu am ales să accept, și chiar să încurajez această situație. Cu acest scop am adăugat funcționalitatea de load balancing. Load balancing se traduce mot a mot în balansarea încărcăturii și constă în distribuția cererilor către un server între mai multe servere identice. Cea mai directă metodă de scalare a unui server este multiplicarea lui. Nu este suficient să avem mai multe servere identice și să lasăm utilizatorii să decidă ce server doresc să folosească, așa că folosim un load balancer care să decidă care dintre copiile serverului să proceseze fiecare cerere în funcție de un algoritm. Doar serverul de balansare știe de existența serverelor pe care le are în grijă.



Figura 1.2: Accesarea unui server

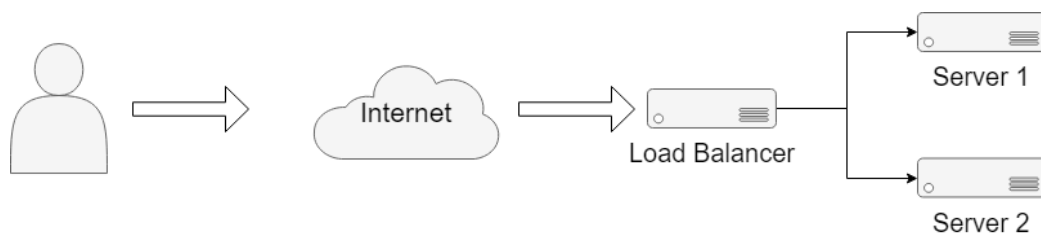


Figura 1.3: Accesarea unui server prin intermediul unei load balancer

Aș fi putut să nu accept înregistrarea mai multor servicii cu același nume și echilibrarea încărcăturii de muncă să fie făcută de fiecare serviciu în parte, iar aplicația mea să înregistreze doar acest server, dar dacă rezolv și responsabilitatea de balansare, elimin un punct de eșec. Dacă toate serviciile care au nevoie de scalare folosesc un load balancer, atunci funcționarea lor depinde de acesta. Eliminând nevoia existenței a încă unui risc crește siguranța.

CAPITOLUL 2

Tehnologii

2.1 .NET Framework

.NET este o platformă de dezvoltare a software-ului gratuită pentru diverse sisteme de operare. Platforma include o bibliotecă de mari dimensiunii numită Framework Class Library și permite interoperabilitatea mai multor limbaje de programare. Programele scrise în .NET Framework rulează într-un mediu numit Common Language Runtime (CLR), o mașină virtuală de aplicație care oferă servicii precum securitatea, managementul memoriei și tratarea excepțiilor. Biblioteca de clase și CLR formează .NET Framework.¹

2.2 C#

C# este un limbaj de programare modern, orientat pe obiecte și puternic tipizat. C# își are rădăcinile în familia de limbaje C și le este imediat familiar programatorilor C, C++ sau Java. C# are multe caracteristici care ajută la construcția aplicațiilor robuste și durabile:

¹.Net Framework. https://en.wikipedia.org/wiki/.NET_Framework.

- Garbage Collector-ul eliberează automat memoria ocupată de obiectele care nu mai sunt folosite.
- Tratarea excepțiilor oferă o abordare structurată și extensibilă pentru detectarea erorilor și recuperare
- Designul puternic tipizat care face imposibilă existența variabilelor neinițializate, indecșilor în afara limitelor șirurilor sau convertirea ne-verificată a tipurilor.

C# are un sistem de unificat de tipuri. Toate tipurile din limbaj, incluzând primitivele precum `int`, moștenesc dintr-o singură rădăcină tipul `object`. De aceea toate tipurile au un set de operații și valori de orice tip pot fi stocate, transportate și folosite pentru operații într-un mod consistent.

Pentru a se asigura faptul că programele și bibliotecile C# pot evolua în timp într-o manieră compatibilă, s-a pus accent pe versionare în designul limbajului. Multe limbaje de programare nu acordă suficientă atenție versionării și de aceea programele scrise în acele limbaje crapă nejustificat de frecvent atunci când sunt introduse versiuni noi ale bibliotecilor de care programul depinde. Aspecte din limbajul C# care au fost influențate de această decizie includ modificatorii `virtual` și `override`, regulile de supraîncărcare ale metodelor și suportul pentru declarare explicită a interfețelor membre.²

2.3 Asp.NET

ASP.NET este un framework open-source pentru dezvoltarea web. A fost dezvoltat de Microsoft pentru a le permite programatorilor să creeze site-uri și servicii web dinamice. A fost introdus odată cu .NET framework 1.0 și ca urmare este construit pe CLR, ceea ce permite programatorilor să utilizeze cod scris în oricare limbaj suportat de platformă. La începuturile sale, dezvoltarea web site-urilor Asp.NET era asociată cu WebForms, care datorită limitărilor sale a fost înlocuit de Asp.NET MVC.

²Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154916.

2.3.1 Asp.NET MVC

Asp.NET MVC reprezintă o alternativă la WebForms pentru construcția aplicațiilor web în cadrul platformei .NET. Asp.NET MVC are o abordare diferită în ceea ce privește structura aplicației, folosind arhitectura MVC. Asp.NET MVC este mai aproape de protocolul HTTP spre deosebire de Web Forms care încearcă să ascundă natura fără stare a protocolului. Folosind arhitectura MVC și asociind o singură cerere HTTP apelării unei metode, experiența dezvoltatorilor este mai naturală.

În timp ce Web Forms are o strânsă legătură între logica aplicației și interfața grafică, Asp.NET MVC încurajează un design în care interfața grafică (view-ul) este separată de codul care o conduce (controller-ul). Atunci când acest design este implementat corect, aplicația devine mult mai ușor de menținut și dezvoltată. De asemenea, această separare face componentele aplicației să fie mult mai ușor de testat în izolare, ceea ce facilitează testarea automată.

Una din componentele centrale ale paginilor Asp.NET MVC este motorul Razor. Acesta oferă un mod concis de a combina cod C# și marcaj HTML pentru a crea pagini dinamice. Razor este esențial pentru a afișa utilizatorului rezultatul cererilor sale. Cu ajutorul Razor, aceeași pagină poate fi diferită pentru fiecare utilizator.

2.3.2 MVC pattern

Model View Controller (MVC) este un șablon arhitectural care prezintă o soluție pentru tratarea interfeței grafice. Arhitectura MVC este una dintre cele mai frecvent întâlnite arhitecturi în dezvoltarea aplicațiilor web, deoarece contribuie la scalabilitatea și extensibilitatea proiectelor. Separarea explicită a responsabilităților crește puțin complexitatea designului aplicației, dar beneficiile eclipsează efortul extra.³ MVC separă aplicația în trei componente:

1. Model

Componenta model corespunde logicii legate de date și prelucrarea lor. Asta poate însemna datele care sunt transferate între componentele view și controller sau orice alte date care țin de logica aplicației.

2. View

³Jon Galloway, David Matson, Brad Wilson, and K. Scott Allen. *Professional ASP.NET MVC 5*. Wrox, 2014, p. 2.

Componenta View este folosită pentru toată logica ce ține de logica interfeței. În cazul aplicațiilor web, view-ul este un șablon după care este generat codul HTML.

3. Controller

Componenta Controller funcționează ca o interfață între componentele view și model. Această componentă răspunde inputului utilizatorului, comunică cu modelul și decide care view trebuie folosit ca rezultat al cererii.

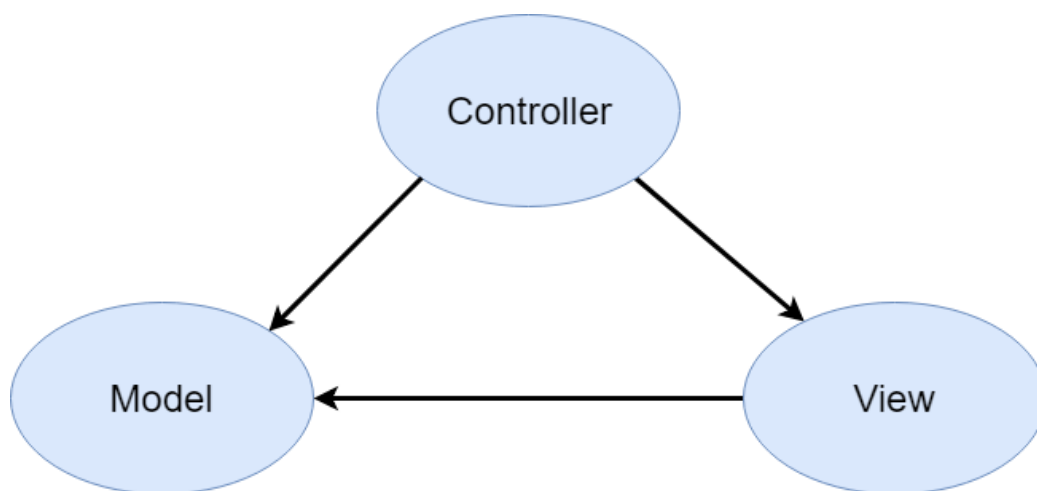


Figura 2.1: Arhitectură MVC

2.3.3 Asp.NET WebApi

Un web api este o interfață programatică la un sistem care este accesată prin metode standard HTTP. Un web api poate fi accesat de o varietate largă de clienți HTTP, incluzând browser și dispozitive mobile⁴.

Asp.net MVC este construit cu scopul de a face website-uri. Acest lucru este evident din funcționarea acestui framework, răspunde la cereri primite din browser și returnează HTML. Cu toate acestea Asp.NET MVC permite un control minuțios asupra răspunsului dat, și MVC este util în formarea unui web api. Dezvoltatorii asp.net au realizat că puteau folosi acest framework

⁴Glenn Block, Pedro Felix, Howard Dierking, Darrel Miller, and Pablo Cibraro. *Designing Evolvable Web APIs with ASP.NET: Harnessing the Power of the Web*. O'Reilly Media, 2014, p. 23.

pentru dezvoltarea de servicii web și aceasta improvizație era preferabilă alternativelor.⁵

Din acest motiv, odată cu Asp.NET MVC 4, a fost introdus și Asp.net Web Api, un framework care oferă stilul de lucru din Asp.NET MVC adaptat pentru a scrie servicii web.

2.4 JavaScript

JavaScript este un limbaj de programare interpretat cu capacități de orientare pe obiect. JavaScript este sintactic similar cu C, Java și C#, încă similaritățile se termina la sintaxă. JavaScript este un limbaj slab tipizat, ceea ce înseamnă că variabilele nu au un tip specificat. Obiectele asociază nume de proprietăți unor valori arbitrare și se aseamănă mai mult cu hashtable-urile din C# decât cu obiectele. Mecanismul de moștenire din JavaScript este bazat pe prototip și nu se aseamănă cu cea din C++ sau C#. Tipurile primitive din JavaScript sunt numere, stringuri și valori booleene dar limbajul include și date, șiruri și expresii regulate.⁶

Motivul principal pentru care am les să folosesc acest limbaj în proiectul meu este faptul că JavaScript este limbajul cunoscut de toate browser-ele și în acel context este extins cu obiecte care permit interacțiunea cu utilizatorul, alterarea conținutului paginii care este încărcată în browser și controlul browser-ului. Această versiune a limbajului este numită și JavaScript **client-side** pentru a sublinia faptul că scripturile funcționează în browser-ul clientului și nu pe server.

Deși JavaScript este utilizat predominant în dezvoltarea paginilor web, datorită popularității acestuia, s-au dezvoltat numeroase framework-uri dedicate acestui limbaj care permit dezvoltarea pe diferite platforme. Cu Node.js JavaScript poate fi folosit pentru a face servere web. Electron.js permite ca JavaScript să fie folosit pentru aplicații desktop cross-platform. JavaScript poate fi folosit și pentru aplicații pentru dispozitivele mobile prin intermediul frameworkurilor Ionic sau React.js.

⁵Jon Galloway, David Matson, Brad Wilson, and K. Scott Allen. *Professional ASP.NET MVC 5*. Wrox, 2014, p. 7.

⁶David Flanagan. *JavaScript: The Definitive Guide Activate Your Web Pages*. 6th. O'Reilly Media, Inc., 2011. ISBN: 0596805527, 9780596805524.

2.5 HTML

Hypertext Markup Language (HTML) este limbajul de marcaj standard pentru crearea paginilor web și aplicațiilor web.⁷ Împreună cu CSS și JavaScript formează cele trei limbaje esențiale care definesc interfața grafică a unei aplicații web. Navigatoarele web primesc documente HTML de la un server și pe baza acestuia afișează o pagină web. HTML descrie structura semantică a unei pagini.

Elementele HTML sunt blocuri de construcție ale paginilor HTML. Folosind HTML se pot introduce în pagini obiecte precum imaginile sau form interactive se pot introduce într-o pagină web. De asemenea, HTML permite crearea unor documente structurate, denotând structural semantice pentru text precum paragrafe, liste, linkuri, gurile și altele. Elementele HTML sunt delimitate de taguri scrise folosind paranteze unghiulare. Există taguri care introduc direct elemente, precum tagul ``. Pe de altă parte există și taguri care dau informații despre subelementele acestora, precum tagul ``. Navigatoarele nu afișează direct documentele HTML, ci știu să le interpreteze. Despre elementele HTML se pot da informații suplimentare prin atributele acestora. Majoritatea atributelor sunt perechi de tip cheie valoare separate prin semnul `=`. De asemenea valoarea atributelor trebuie specificată între ghilimele. Atributele se găsesc mereu în tagul de început al elementelor HTML.

2.6 CSS

Cascading Style Sheets (CSS) este un limbaj de stil folosit pentru descrierea prezentării documentelor scrise în HTML. CSS a fost conceput pentru a permite separarea între prezentare și conținut, incluzând forme, culori și fonturi. Această separare poate îmbunătăți accesibilitatea conținutului. De asemenea separarea permite mai multă flexibilitate și control în specificarea caracteristicilor prezentării. Un alt beneficiu pe care îl aduce CSS este faptul că mai multe pagini web pot folosi aceleași stiluri, dintr-un fișier .css, ceea ce reduce complexitatea și repetabilitatea. Mai mult, folosind același fișier css putem da un aspect similar paginilor care fac parte din aceeași aplicație, creând o unitate între acestea. Numele cascading din CSS provine de la metoda pe prioritizare a stilurilor care afectează un element. CSS folosește prioritate de tip cascade pentru a determina regula de stil aplicată unui element atunci când acesta este afectat de mai multe reguli. CSS alege regula care va fi folosită

⁷HTML. <https://en.wikipedia.org/wiki/HTML>.

parcurgând în cascadă toate regulile aplicate elementului pornind de la cea mai generală până la cea mai specifică.

Înainte de CSS, HTML era folosit și pentru partea prezentare, motiv pentru care au apărut foarte multe taguri HTML cu rol de prezentare. HTML a fost conceput pentru a avea un rol structural în pagină dar a devenit din ce în ce mai aglomerat cu taguri de prezentare⁸. Această poluare a limbajului a fost sesizată și de către World Wide Web Consortium (W3C) care au căutat o soluție la problemă. Soluția propusă de W3C a fost CSS care a devenit o recomandare cu aceeași greutate ca HTML. De atunci CSS a evoluat odată cu dezvoltarea web, până la starea în care se află astăzi.

2.7 Entity Framework

Până la .NET 3.5, programatorii obișnuiau să folosească ADO.NET pentru a alina sau prelua date din baza de date. Trebuiau deschise conexiuni, create DataSeturi pentru a citi sau scrie date care mai apoi trebuiau convertite în obiect sau invers. Acesta era un proces greoi și predispus la erori. Odată cu .NET 3.5 Microsoft a introdus EntityFramework cu scopul de automatiza lucrul cu baze de date.

EntityFramework este un ORM (Object Relational Mapper) open-source creat de Microsoft pentru platforma .NET. Framework-ul permite programatorilor să utilizeze datele prin intermediul unor obiecte specifice domeniului fără a se concentra pe tabelele și coloanele în care datele sunt stocate. Acest lucru permite dezvoltatorilor să lucreze la un nivel mai înalt de abstractizare atunci când lucrează cu date persistente.

EntityFramework oferă două modalități de lucru:

- **Code First**

Această abordare necesită ca programatorul să scrie clasele corespunzătoare datelor care trebuie să persiste în baza de date și o clasă specială numită context. Pe baza acestora el poate genera o bază de date. Baza de date generată se poate actualiza prin migrații. Migrație oferă un mod incremental de a aplica schimbări unei baze de date generate cu EntityFramework pentru a o sincroniza pe aceasta cu clasele care constituie modelul. O migrație conține toate schimbările modelului care trebuie

⁸Eric A. Meyer. *Cascading Style Sheets: The Definitive Guide*. O'Reilly Media, 2004.

reflectate în baza de date de la ultima actualizare. Împreună toate migrațiile formează istoria bazei de date.

- **Database First**

În această abordare EntityFramework crează clasele model pe baza unei baze de date deja existente printr-un proces numit scaffolding. Termenul scaffolding provine din domeniul construcțiilor și înseamnă procesul de montare a schelelor unei clădiri. Precum schelele fac posibilă construcția unei clădiri, așa modelul face posibilă prelucrarea datelor din baza de date. Această este abordarea pe care am decis să o folosesc în realizarea acestui proiect.

- **Model First** Abordarea Model first presupune crearea entităților și relațiilor dintre acestea într-o schemă, utilizând un designer grafic. Pe baza schemei sunt generate clasele ce constituie modelul și baza de date.

Din experiența mea cu variantele prezentate, am ajuns să prefer varianta database first, deoarece consider că astfel am mai mult control asupra datelor și operațiilor.

2.8 Microsoft SQL Server

Microsoft SQL Server este un sistem de management al bazelor de date relaționale, care susține o varietate largă de aplicații. Precum alte sisteme de management al bazelor de date relaționale, SQL Server este construit peste SQL, un limbaj standardizat pe care administratorii de baze de date și profesioniștii IT îl folosesc pentru a manageria baze de date și pentru a interoga datele pe care acestea le conțin. SQL Server este legat de Transact-SQL, o Implementare a SQL de la Microsoft care adaugă un set de extensii de programare limbajului standard. Componenta centrală a SQL Server este motorul bazei de date, care controlează stocarea datelor, procesarea lor și securitatea. Acesta include un motor relațional care procesează comenzi și interogări și un motor de stocare ce manageriază fișierele, tabele, paginile, indecși, buffer-ele de date și tranzacțiile. Procedurile stocate, trigger-ele, view-urile și celelalte obiecte ale bazei de date sunt de asemenea create și executate de motorul bazei de date.⁹ Mai jos decât motorul bazei de date se află sistemul de operare al SQL Server (SQLOS) care se ocupă de operații de nivel

⁹Microsoft SQL Server. <https://searchsqlserver.techtarget.com/definition/SQL-Server>.

scăzut precum managementul memoriei și intrărilor și ieșirilor, programarea joburilor și blocarea datelor pentru a evita conflictele la actualizare. Deasupra motorului bazei de date se afla un strat de rețea, care folosește protocolul Tabular Data Stream al Microsoft pentru a facilita interacțiunile de tip cerere și răspuns cu serverul.

CAPITOLUL 3

Implementare

3.1 Arhitectura

Aplicația este construită urmând o arhitectură pe straturi. În arhitectura pe straturi componentele aplicației sunt organizate în straturi orizontale, fiecare îndeplinind un rol specific în cadrul aplicației. Fiecare strat este o abstractizare a în jurul cerințelor aplicației. Una din trăsăturile definitorii ale acestei arhitecturi este separarea responsabilităților. Componentele unui strat al aplicației se ocupă doar cu logica specifică stratului și nu se intersectează cu logica altor straturi. Nu există un număr impus de straturi, în unele cazuri anumite straturi pot lipsi sau pot apărea noi straturi.

Serviciul meu de înregistrare este compus din 4 straturi, și anume:

1. Un strat de stocare al datelor
2. Un strat de accesare a date
3. Un strat de logică de business
4. Un strat de prezentare

3.1.1 Stratul de stocare a datelor

Acest strat constă în baza de date. Pentru că am folosit EntityFramework în modul Database First am început proiectul cu arhitecturii bazei de date. Dacă în viitor vor apărea probleme de performanță voi putea optimiza accesarea datelor scriind propriile proceduri stocate mai eficiente decât cele generate de EntityFramework pe baza codului. Pentru a putea interacționa cu aplicația prin interfața grafică într-un mod securizat este nevoie de un sistem de autentificare deci trebuie stocate date despre utilizator. Pentru autentificare fiecare utilizator trebuie să poată fi indentificat unic și din motive de securitate evidente el are nevoie de o parolă a cărei valoare hash este stocată în baza de date. Pentru partea de autorizare este nevoie ca fiecărui utilizator să îi fie asociat unul sau mai multe roluri. În funcție de rolul unui utilizator, aplicația decide dacă acestui îi este permis accesul la o anumită funcționalitate. Administratorul serviciului va avea toate rolurile și ca urmare va avea acces nerestricționat. În prezent mai există un singur alt tip de utilizator, al cărui singur rol îi permite să facă modificări pe server. Acest rol este intenționat pentru programatorii care lucrează cu unul sau mai multe din serviciile înregistrate și le pune la dispoziție informații legate de starea serviciilor. Sistemul de roluri este gândit în așa fel încât să fi ușor de adăugat și integrat un rol nou. Introducerea unui rol nou nu va afecta cu nimic rolurile existente și nici utilizatorii existenți. Rolurile pot fi asgnate și utilizatorilor vechi dându-le astfel noi permisiuni.

Pentru ca utilizatorii să aibă mai multe informații, am implementat un mecanism de mesagerie. Acesta nu este intenționat ca un mijloc de comunicare între utilizatori, ci este un mod ca ei să fie informați de sistem. Deoarece add-inurile au acces la mesaje, nu există o regulă cu privire trebuie să conțină un mesaj. Acestea pot conține orice de la informări cu privire la servicii care nu au mai fost folosite de mult, până la mesaje de înămpinare a unui nou utilizator. După cum se vede și în Figura [3.1], un mesaj are un destinatar și un expiditor. Acest lucru poate fi folosit dacă pe viitor utilizatorii vor avea nevoie să își poată trimite mesaje unul altuia, dar în prezența expeditorii mesajelor sunt add-inurile. Fiecare add-in are asociat un utilizator special fără nici un rol.

Pentru un serviciu de înregistrare cele mai importante date sunt cele referitoare la serviciile înregistrate. Un serviciu este identificat prin numele de cluster și numele de aplicație. Cluster-ul definește grupul din care face parte un serviciu. Nu există o restricție de unicitate pentru combinația nume de cluster și nume de aplicație, deoarece funcția de load balancing se

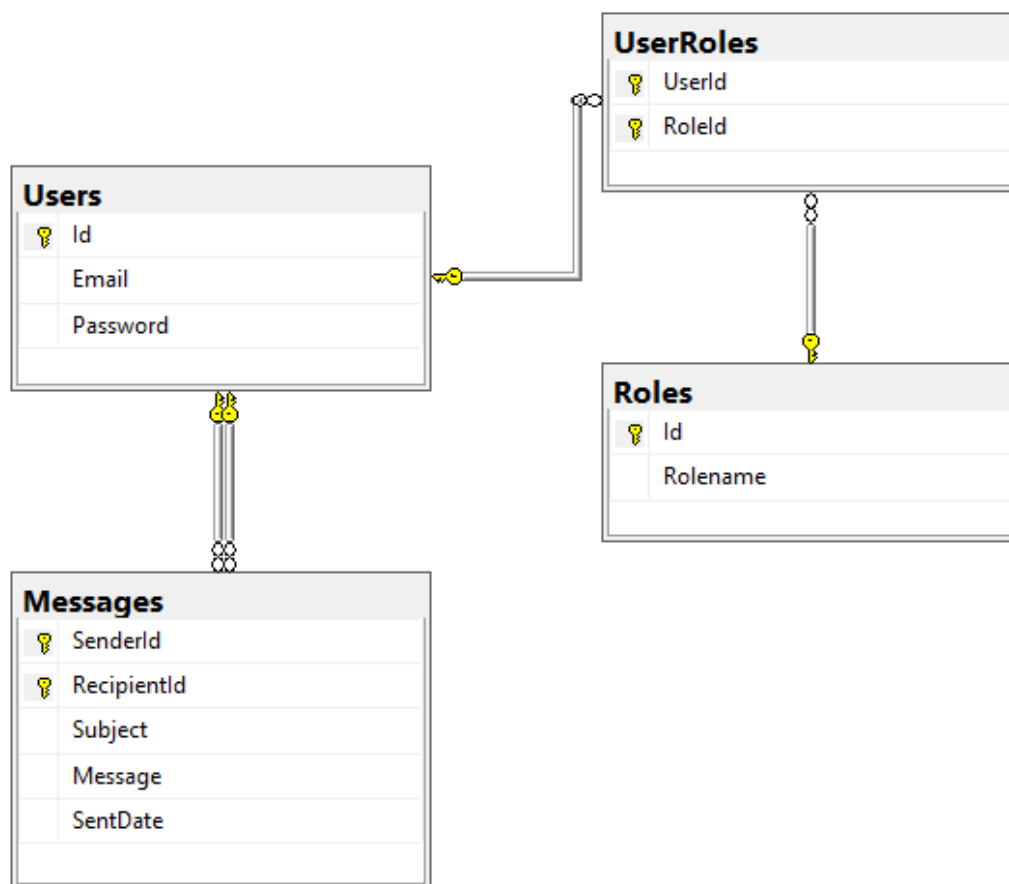


Figura 3.1: Diagrama relațiilor utilizatorului

funcționează atunci când există mai multe servere identice. Pentru a putea fi folosit, un serviciu are nevoie și de un URL. URL-ul trebuie să fie unic în cadrul unui cluster. Nu are sens ca același serviciu să fie înregistrat de două ori, însă nu ar trebui să fie oprit din a fi folosit în mai multe clustere, astfel URL-ul nu trebuie să fie unic, ci combinația cluster, URL. După cum se poate observa și în figura [3.2], am adăugat un tabel destinat monitorizării accesării serviciilor. În etapa inițială a proiectului acest tabel era înlocuit de către o coloană în tabelul Services care conținea data ultimei accesări a serviciului. Deși data ultimei accesări este o informație importantă, nu este suficientă. Stocând fiecare accesare pot realiza statistici relevante pentru utilizatori.

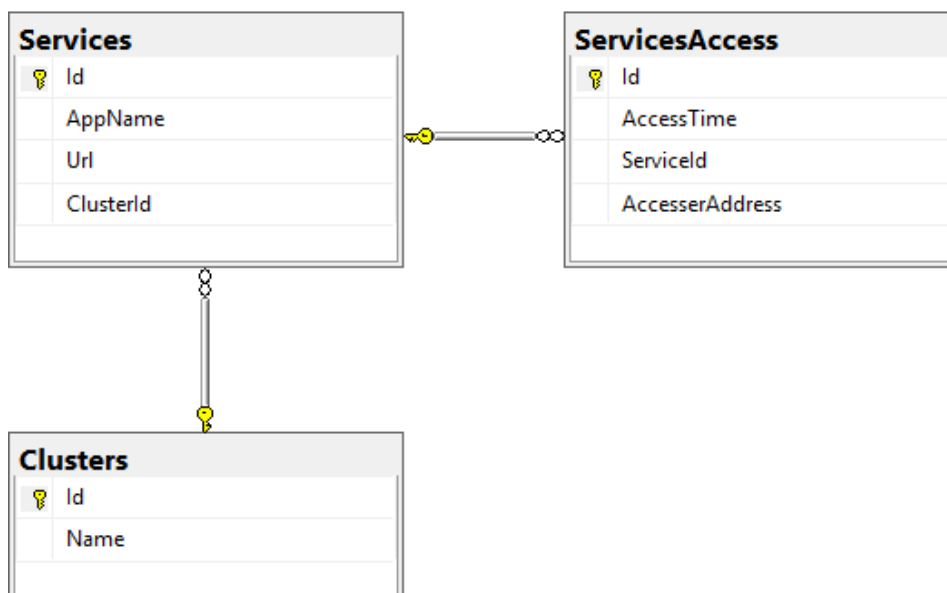


Figura 3.2: Diagrama serviciilor

3.1.2 Stratul de accesare a datelor

Stratul de accesare a datelor constă într-un proiect separat ce conține interfețe și clase care au responsabilitatea de a prelua sau prelucra datele din baza de date. Această separare aduce numeroase beneficii:

- Am eliminat duplicarea codului de acces la date.
- Separarea responsabilităților crește mentenabilitatea și citibilitatea codului
- Datorită decuplării am făcut codul testabil în izolare. Un alt avantaj
- Am decuplat aplicația de framework-ul de persistență, astfel este mai ușoară schimbarea acestuia dacă în viitor va apărea nevoia.

Accesarea datelor se face prin interfețe, a căror implementare este obținută prin injectare de dependențe. Am definit câte o interfață pentru fiecare entitate persistentă majoră.

Interfața `IUserRepository` conține metode pentru accesarea datelor despre utilizatori, sau trimiterea mesajelor către utilizatori. Un mesaj poate fi trimis către un singur utilizator sau către toți utilizatorii care îndeplinesc o anumită condiție. Pentru a stabili o condiție de trimitere către mai mulți utilizatori am folosit un delegat pentru o metodă ce primește ca parametru un obiect de tip `User` și returnează un răspuns boolean.

Interfața `IClusterRepository` definește metodele clasice ale unui repository, și anume operațiile CRUD și o metodă de filtrare folosind un delegat.

Interfața `IServicesRepository` este probabil cea mai importantă, deoarece corespunde funcționalității principale ale aplicației. Este evident că această trebuie să aibă metode pentru înregistrarea, dezînregistrarea și găsirea serviciilor. De asemenea este nevoie și de o metodă pentru a obține toate serviciile, astfel încât acestea să poată fi afișate în interfața grafică. Tot pentru partea vizuală a aplicației am avut nevoie și de un mod de a aduce din baza de date toate accesările unui serviciu dintr-un interval de timp. Pe baza datelor de acces pot realiza statistici referitoare la accesul la servicii.

Tot în stratul de accesare a datelor se află și Interfața `ILoadBalancer` [3.1]. Clasa `ServicesRepository` care implementează interfața `IServicesRepository` se folosește de un obiect ce implementează interfața `ILoadBalancer`, ca o strategie. Strategia este un șablon de proiectare comportamental care permite schimbarea algoritmului în timpul rulării aplicației. Acest lucru se face trimițând un obiect ce implementează `ILoadBalancer` prin constructorul clasei `Servi-`

cesRepository. Pentru interfața ILoadBalancer am făcut și o implementare implicită, care este folosită în cazul în care nu există o implementare sub formă de add-in. Această implementare va returna url-ul serverului care a fost folosit în urma cu cea mai îndelungată perioadă și se potrivește serviciului căutat. Parametrii clusterName și appName ai metodei GetService sunt folosiți pentru a identifica serviciul, iar parametrul ipAddress reprezintă adresa ip a celui care solicitat serviciul respectiv. Adresa ip trebuie stocată odată cu data accesării serviciului.

Listing 3.1: Interfața ILoadBalancer

```
/// <summary>
///     Strategy for load balancing algorithm
/// </summary>
public interface ILoadBalancer
{
    /// <summary>
    ///     Get a service using a load balancing
    ///     algortihm
    /// </summary>
    /// <param name="clusterName"></param>
    /// <param name="appName"></param>
    /// <param name="ipAddress"></param>
    /// <returns></returns>
    Service GetService(string clusterName, string
        appName, string ipAddress);
}
```


3.1.3 Startul de logică de business

Startul de logică de business conține 2 proiect. Un proiect dedicat Add-inurilor și unul dedicat utilizării datelor venite din stratul de acces al datelor. În acest proiect se găsesc trei interfețe și implementările lor. Interfețelor `IServicesRepository`, `IUserRepository` și `IClusterRepository` din stratul precedent, le corespunde o interfață în acest strat. Pentru `IServicesRepository` există `IServicesManager`, pentru `IClusterRepository` există `IClusterManager` și pentru `IUserRepository` există `IUserManager`. Implementările interfețelor de tip manager se folosesc de obiecte de ce implementează interfețele repository corespunzătoare pentru a accesa datele. Aceste obiecte sunt furnizate prin procedeul injectării dependențelor. Astfel `ServicesManager` care implementează interfața `IServicesManager`, primește o implementare a `IServicesRepository` prin care realizează accesarea și prelucrarea datelor. Folosind interfețe, managerii nu depind de modul în care datele sunt stocate. Metodele managerilor au rolul de a face validării asupra parametrilor cu care sunt apelate pentru a preveni căutări inutile printre datele stocate sau inserarea unor date care ar putea afecta consistența datelor stocate. Cu toate acestea managerii nu se limitează la validări, clasele repository doar aduc datele, tot rezultatul logicii se petrece în manageri. Se poate spune că un repository este o unealtă care îi dă acces la date unui manager.

În celălalt proiect din acest strat se află logica ce guvernează add-inurile. Ca un add-in să fie încărcat în aplicație acesta trebuie să fie cunoscut aplicației. Acest lucru se face printr-un fișier de tip `*.sAddIn` care definește add-inul. Fișierele `*.sAddIn` sunt fișiere în format JSON care conțin informații despre add-in. După cum se vede în [3.2], definiția unui add-in trebuie să conțină numele add-inului, o scurtă descriere, tipul de add-in, intervalul la care acesta trebuie executat și calea către add-in. Intervalul este exprimat în ore și este luat în considerare doar pentru add-inurile periodice. Locația de pe disc a add-inului.

Listing 3.2: Definiție a unui `AddIn`

```
{
    "Name": "UnusedServiceNotifier",
    "ShortDescription": "Sends a notification to the
                        administrators regarding unused services",
    "Type": "Periodic",
    "RunInterval": 4,
    "Path": "/AddIns/Periodic/USN.dll"
}
```

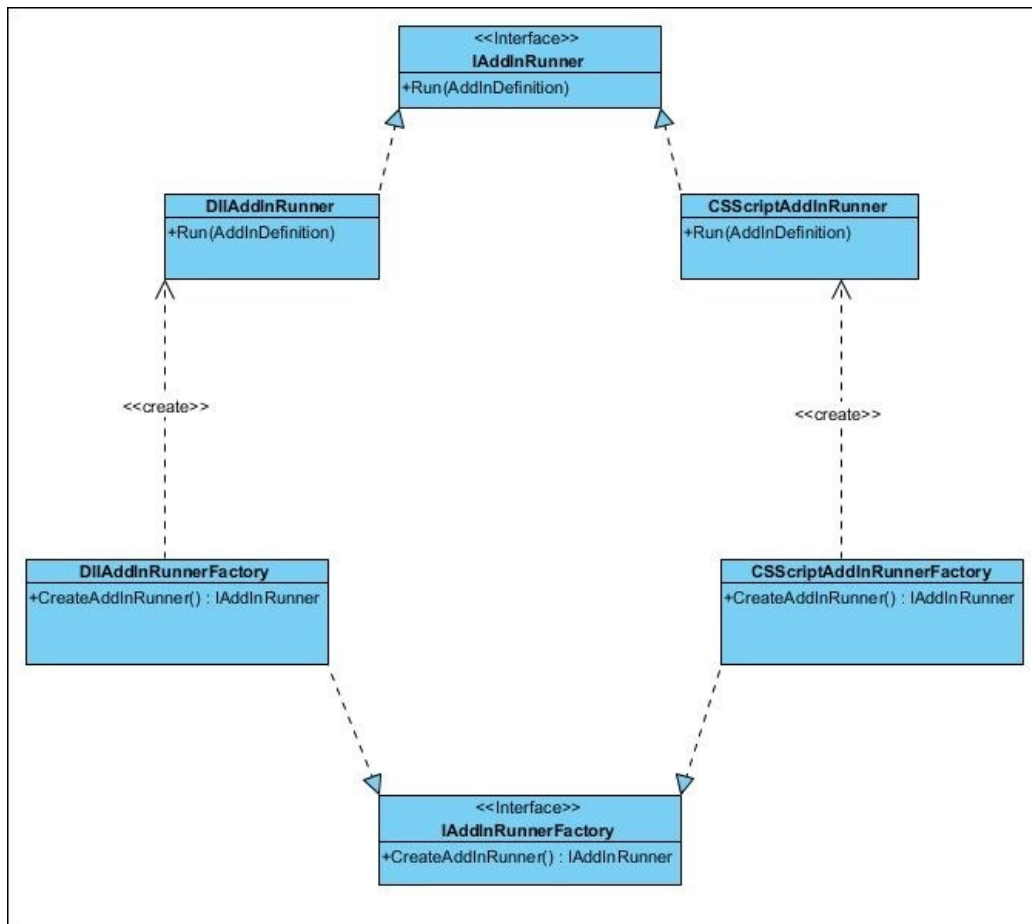
Pe lângă un fișier îl definește, pentru a putea fi executat un add-in trebuie să conțină o clasă ce implementează Interfața IAddin. Această clasă este punctul de intrare în add-in.

În prezent din punct de vedere al limbajului de programare folosit, add-inurile se împart în două categorii, și anume add-inuri scrise în C# și add-inuri scrise în CSScript. Add-inurile C# sunt sub formă de dll iar cele scrise în CSScript sunt fișiere text cu extensia .csscript. Problema în această situație este că ambele tipuri de add-inuri trebuie tratate la fel însă este evident că execuția acestor se face diferit. Bibliotecile trebuie încărcate în aplicație apoi prin mecanismul reflection trebuie găsită clasa ce implementează interfața IAddin și apoi executată metoda Run a acesteia. Pe de altă parte fișiere CSScript sunt fișiere text și acestea trebuie interpretate de un interpretor CSScript. Interpretorul caută punctul de intrare în add-in și apoi execută metoda Run. Cu siguranță nu pot. În ciuda diferenței dintre cele două tipuri de add-inuri, ele trebuie tratate similar de către aplicație. Am rezolvat o parte din problemă definind o interfață IAddInRunner care să se ocupe de lansarea în execuție a extensivelor. Prin intermediul acesteia, procesul de adăugare al unui nou tip de add-in se reduce la crearea unei noi implementări a interfeței, potrivită pentru tipul dorit. Nu doar că am făcut posibilă implementarea add-inurilor sub forma de bibliotecă cu legătură dinamică și sub formă de scrip CSScript, dar am și pregătit aplicația pentru dezvoltare viitoare. Singura problemă rămasă este faptul că aplicația are nevoie de un mod de a ști ce fel de add-inuri să folosească. Decizia cu privire la tipul de add-in se face dintr-un fișier de configurare. O posibilă soluție ar fi ca la oricând este nevoie ca aplicația să știe ce fel de extensii trebuie să folosească, această să facă o verificare. Această abordare nu doar că va polua codul cu interogări și va cauza duplicare de cod, dar este și foarte greu de întreținut și dezvoltat. Soluția mea a fost să folosesc o implementare de tip abstract factory [3.1.3]. Abstract Factory este un șablon de proiectare care oferă o interfață pentru crearea unei familii de obiecte înrudite sau dependente fără specificarea claselor lor concrete.¹

În funcție de tipul de addIn specificat în fișierul de configurare al serverului, se va alege care implementare a interfeței IAddInRunnerFactory va fi folosită de aplicație. Implementarea aleasă va fi folosită în întreaga aplicație și astfel cu o singură verificare programul va folosi tipul de add-inuri dorit.

Există situații în care un add-in devine irelevant sau nedorit de utilizator, din acest motiv, Add-Inurile trebuie să poată fi șterse de utilizator. Pentru

¹Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly & Associates, Inc., 2004. ISBN: 0596007124.



a adăuga această funcționalitate, am creat și implementat o interfață dedicată stării de pe disc a add-inurilor. Această interfață poate regăsi add-inurile de pe disc și le poate șterge.

După cum am menționat mai devreme există un tip de add-inuri care sunt executate periodic. Fiecare add-in de acest tip are un alt interval orar la care trebuie executat. Cu scopul de a planifica și lansa în execuție aceste add-inuri am făcut clasa `AddInScheduler`. `AddInScheduler` operează într-un fir de execuție separat datorită naturii asincrone a acestuia. `AddInScheduler` se ocupă de lansarea în execuție a tuturor add-inurilor ce trebuie executate periodic. Clasa oferă posibilitatea adăugării unui add-in nou oricând și la dispariția unui add-in de pe disc, `AddInScheduler` îl va scoate automat din lista de execuție. Din nou intervine problema lansării în execuție a add-inurilor, dar de data aceasta este în cadrul `AddInScheduler`. Deși în cazul unei implementări naive, execuția add-inurilor de către `AddInScheduler` ar putea părea

o problemă, de fapt execuția nu este treaba acestei clase. Tot ce trebuie să facă clasa `AddInScheduler` este să folosească o implementare a interfeței `IAddInRunner` care la rândul ei să se ocupe de execuția add-inurilor. Și după cum am stabilit anterior, `AddInScheduler` va obține o implementare corectă a `IAddInRunner` printr-un factory.

3.1.4 Stratul de prezentare

La acest strat aplicația se bifurcă formând două servere. Primul server este serviciul de înregistrare care este componentă principală a proiectului meu. Acesta este un server de tip Web Api astfel interfața grafică întâlnită în mode obișnuit în stratul de prezentare este înlocuită de o interfață de comunicare cu serverul prin intermediul HTTP. Stratul de prezentare din webapi are o arhitectură MVC în care lipsește partea de View.

Primul server este un serviciu web de tip RESTful API, o interfață de programare a aplicațiilor care se folosește de protocolul HTTP. Cu rest, componentele din rețea sunt tratate ca niște resurse la care se cere acces, și a căror implementare este necunoscută. Designul RESTful pleacă de la prezumția că toate apelurile sunt fără stare și nimic nu se păstrează între cereri. Faptul că apelurile sunt fără stare fac ca aplicațiile REST să se potrivească perfect aplicațiilor în cloud.

Al doilea server este interfața grafică a aplicației și permite administrarea aplicației. În cazul acestui server stratul de prezentare se regăsește în sensul clasic. Arhitectura stratului este MVC complet spre deosebire de cazul anterior.

Toate starturile precedente sunt comune în cele două servere. Separarea aplicației inițiale în două servere distincte nu doar reduce munca fiecărui server, dar și facilitează dezvoltarea. Serviciile RESTful sunt ușor de scalat deoarece oricare cerere poate fi trimisă către oricare instanță a unei astfel de componente, datorită lipsei unei stări.

3.2 Injectarea dependențelor

Intenția implementării pe straturi este o cuplare mai slabă între modulele aplicației. Acest lucru este realizat prin folosirea interfețelor în locul implementărilor concrete și prin injectarea dependențelor. În Asp.NET MVC și Web Api toate cererile utilizatorilor sunt trimise către un Controller care se

ocupă de răspunsul la acestea. Controller-ele sunt instanțiate de către framework ceea ce mă împiedică să fac o injectare de dependențe clasică. Asta nu înesamnă că trebuie să deteriorez arhitectura, ci trebuie să fac injectarea dependențelor printr-un alt mod. Abordarea alternativă este oferită de către Autofac. Autofac este un container pentru inversarea controlului. Cu autofac pot înregistra într-un container implementările interfețelor de care am nevoie și atunci când va fi nevoie de o implementare, se va folosi tipul înregistrat în container. De exemplu, când se va crea o instanță a clasei AccountController, care necesită implementarea interfeței IUserManger, acesta va primi o nouă instanță a clasei UserManager și pentru ca aceasta necesită o implementare a IUserRepository, și UserManager va primi implementarea IUserRepository înregistrată în container. Cu autofac pot înregistra și tipuri în timpul rulării, cum am făcut în cazul interfeței ILoadBalancer, a cărei implementare se caută într-un add-in.

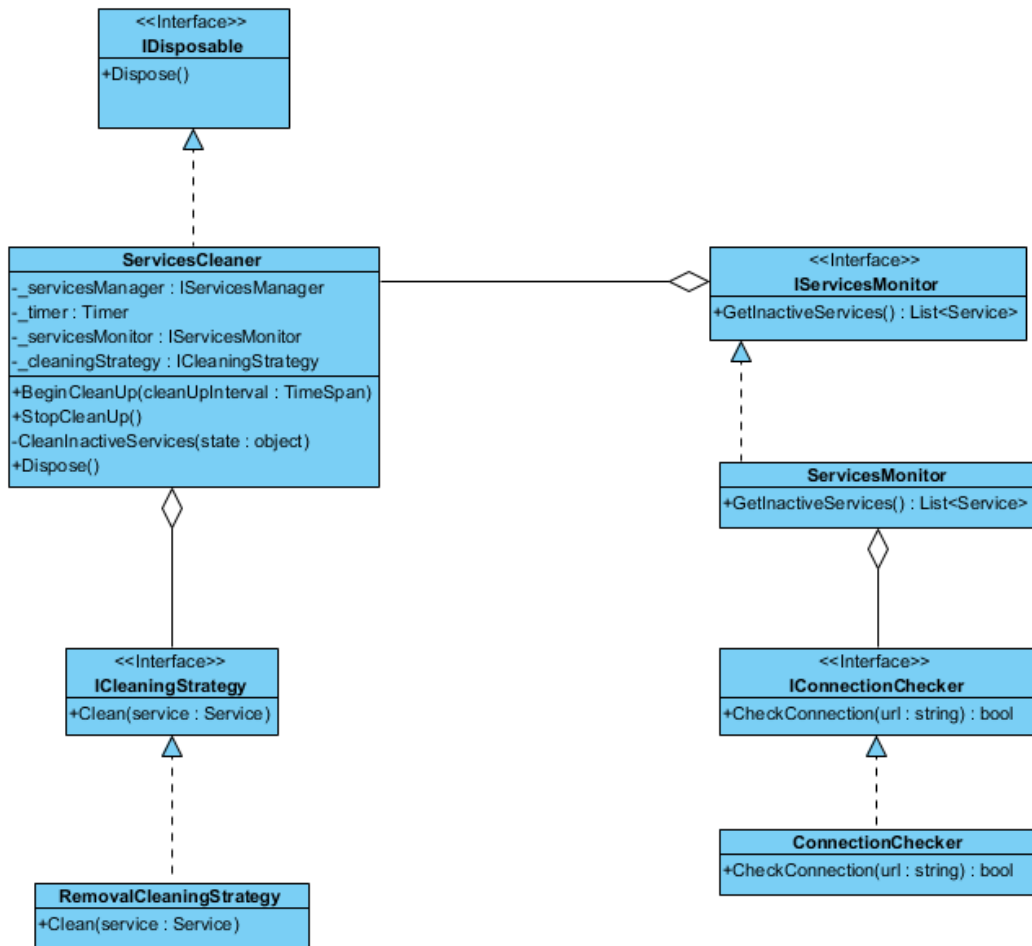
3.3 Curățarea automată

Una dintre situațiile care pot cauza erori este situația în care un serviciu este inactiv. Dacă acest serviciu este solicitat și serverul furnizează acest serviciu aplicație care a făcut cererea, cel mai probabil vor apărea erori în aplicația respectivă. Serviciul d înregistrare trebuie să fie cât mai de încredere. Acesta trebuie să rezolve problemele de conexiune între aplicații, nu să le cauzeze.

Pentru a evita situația în care există servicii inactive, toate serviciile trebuie monitorizate. Serviciu de înregistrare trebuie să știe mereu starea serviciilor pentru a evita trimiterea lor solicitanților. Cu acest scop creat clasele din figura [3.3]. Clasa ServicesCleaner are rolul de curățare a serviciilor inactive. Această curățare presupune că serviciile inactive nu vor putea fi percepute ca servicii funcționale de către aplicație. Curățarea propriu-zisă este făcută printr-o strategie. Strategia folosită în prezent este RemovalCelaningStrategy, o implementare a ICleaningStrategy care elimină din sistem serviciile inactive. Pentru a avea acces la servicii, RemovalCelaningStrategy se folosește de o implementare a IServicesManager despre care am discutat într-o secțiune anterioară. ServicesCleaner folosește interfața IServicesMonitor pentru a căuta serviciile inactive. Interfața are o singură metodă și anume GetInactiveServices care returnează o listă de servicii. Implementarea acestei interfețe se numește ServicesMonitor. Se ServicesMonitor se folosește de o implementare a interfeței IServicesManager pentru a extrage servi-

ile înregistrate din baza de date. Serviciile înregistrare sunt mai apoi verificate prin intermediul interfeței `ICheckConnectionChecker.ConnectionChecker` care implementează `ICheckConnectionChecker`, și unica sa metodă, `CheckConnection`, încearcă să trimită un ping către un server și în funcție de răspunsul primit stabilește dacă conexiunea la server se poate realiza.

`ServicesCleaner` are două metode publice, `BeginCleanUp` și `StopCleanUp`. `BeginCleanUp` semnalează faptul că obiectul trebuie să înceapă verificarea și curățarea serviciilor. Această verificare este realizată la un interval fix de timp, specificat prin parametrul metodei `BeginCleanUp`. Intervalul de timp este transmis către obiectul `Timer` din al obiectului de tip `ServicesCleaner`.



Pentru a putea folosi

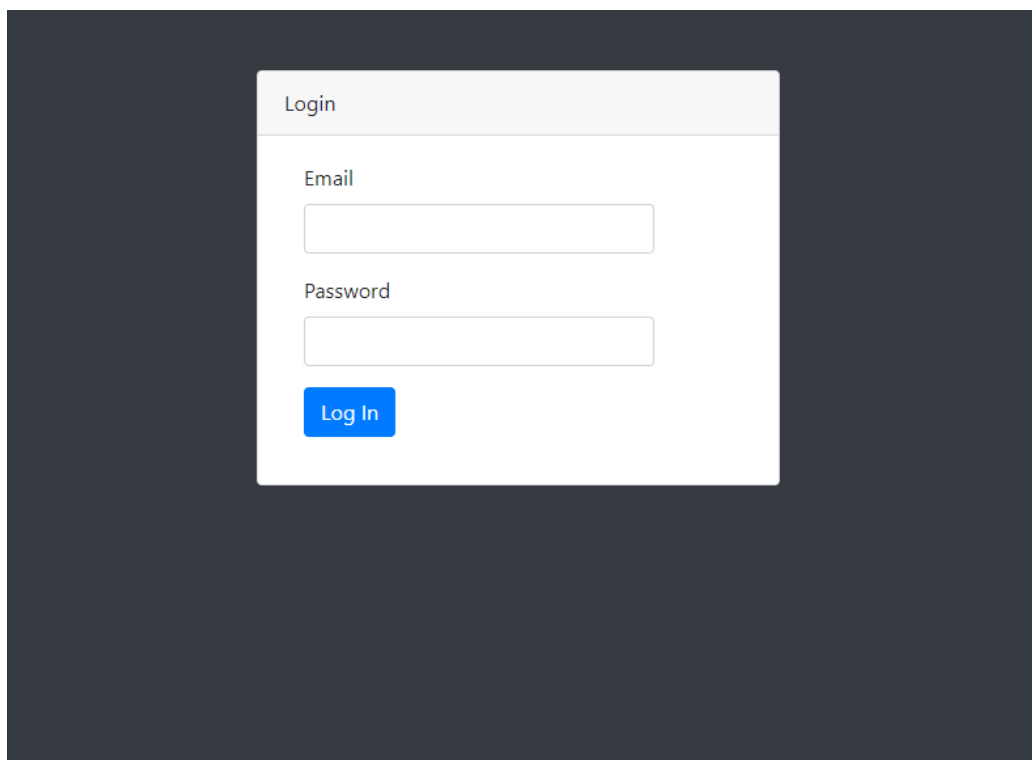
3.4 Autentificarea și autorizarea

În asp.net există un framework numit IdentityModel care se ocupă de o mare parte din autentificarea și autorizarea utilizatorilor. Acesta se integrează bine cu framework-ul ASP.NET MVC și este suficient pentru securitatea majorității aplicațiilor. IdentityModel este de asemenea ușor de folosit și cu un efort minim o aplicație poate avea partea de autentificare și autorizare funcțională. Din aceste motive este foarte comun în aplicațiile ASP.NET.

Cu toate aceste beneficii pe care le aduce, IdentityModel este un framework destul de restrictiv. Utilizarea acestui framework ar fi presupus compromisuri în arhitectura pe care o doresc. IdentityModel se pretează cel mai bine pentru o arhitectură mai simplă. IdentityModel necesită acces direct la baza de date, ceea ce nu este valabil într-o arhitectură pe straturi precum cea a aplicației mele. De asemenea framework-ul este gândit pentru a fi folosit cu EntityFramework într-o abordare Code First. IdentityModel aduce cu el niște clase model și o clasă context, ceea ce în abordarea Code First simplifică munca dezvoltatorului. Cum eu am ales abordarea Database First, acest lucru este un impediment. Acest obstacol poate fi ocolit dacă generez baza de date într-o altă aplicație, apoi copiez exact tabelele create de IdentityModel în baza de date a aplicației mele. Totuși chiar cu această metodă de ocolire rămâne problema accesului direct la baza de date. Din cauza restricțiilor impuse de acest framework am ales să renunț la acesta și să îmi fac propriul sistem de autentificare și autorizare pe care să îl integrez cu ASP.NET.

Pentru a putea folosi aplicația, utilizatorul trebuie să se autentifice. Dacă acesta încercă să intre pe o pagină a site-ului de administrare fără să fie autentificat, va fi redirecționat către pagina de autentificare[3.4]. Autentificarea se face prin intermediul adresei de email și o parolă. Un utilizator nu își poate crea un cont nou deoarece, serviciul de înregistrare nu este un serviciu public, acesta destinat utilizării în cadrul unei companii, doar de persoane autorizate. Din acest punct de vedere, serviciul este similar cu o rețea intranet. Pentru confortul utilizatorului, am ales să folosesc pentru autentificare cookie-uri așa încât utilizatorul să nu fie nevoit să se autentifice de fiecare dată când dorește să intre în aplicație. Termenul de expirare al cookie-ului am decis să fie 30 de minute. Cookie-ul este creat imediat după ce utilizatorul s-a autentificat cu succes.

Mecanismul de autorizare din ASP.NET se bazează pe atributul `AuthorizeAttribute`. Acesta se poate aplica unei metode sau unui controller, caz în care va afecta toate metodele acestuia. Defapt există două attribute `AuthorizeAttribute`, unul pentru ASP.NET MVC și unul pentru Web Api. Diferențele



între ele sunt destul de mici și din punct de vedere al utilizării sunt identice. Aceste atribute se folosesc de o interfață numită `IPrincipal`[3.3], care reprezintă utilizatorul. Am creat o interfață nouă care moștenește interfața `IPrincipal`, care să îmi ofere o metodă de a accesa rolurile utilizatorului. Implementarea noii interfețe implementează și interfața `IPrincipal`, deci poate fi folosită pentru autorizare de către atributul `AuthorizeAttribute`. Pentru controllerele de Web API pot folosi `AuthorizeAttribute` și nu mai este nevoie să fac o clasă derivată a acestuia. Pentru controllerele ASP.NET MVC, deoarece atributul este diferit deși se numește la fel ca cel din Web API, am ales să fac un nou atribut care să moștenească `AuthorizeAttribute`. Noul atribut știe să trateze situațiile în care utilizatorul nu este autorizat și să îl redirecționeze pe acesta către o pagină potrivită situației în care se află. Dacă utilizatorul nu este autentificat, atunci va fi redirecționat către pagina de autentificare. Dacă este autentificat dar nu are permisiunea necesară accesării paginii respective va fi redirecționat către o pagină de eroare corespunzătoare.

În momentul în care un utilizator se încearcă să se autentifice, din motive de securitate, se calculează o valoare hash a parolei introduse. Apoi se caută un cont în baza de date cu același email ca cel introdus de utilizator, dacă un

asemenea cont este găsit, se compară hash-ul nou cu hash-ul salvat în baza de date. Dacă în urma comparației, se decide că parola utilizatorului este greșită, el va fi întors către pagina de autentificare. Dacă parola introdusă este corectă se crează un principal de care este criptat și introdus într-un cookie. După ce este setată durata până la expirarea cookie-ului, acesta este trimis către utilizator.

Pentru ca atributele de autorizare să aibă un `IPrincipal` corect, acesta trebuie extras din cookie. În ASP.NET punctul de intrare în aplicație este un fișier numit `Global.asax.cs` în care se află clasa `MvcApplication` din care pornește aplicația. Această clasă este derivată din `HttpApplication` și din acest motiv se pot schimba comportamente ale aplicației prin suprascrierea unor metode moștenite. Un dintre aceste metode este `Application.PostAuthenticateRequest`, pe care am suprascris-o cu scopul de a extrage datele pentru `IPrincipal`ul meu. Aici se caută cookie-ul și dacă este găsit, se decriptează conținutul acestuia și se crează principalul corespunzător utilizatorului autentificat.

Listing 3.3: Interfața `IPrincipal`

```
//
// Summary:
//     Defines the basic functionality of a
//     principal object.
[ComVisible(true)]
public interface IPrincipal
{
    //
    // Summary:
    //     Gets the identity of the current
    //     principal.
    //
    // Returns:
    //     The System.Security.Principal.IIdentity
    //     object associated with the current principal
    //
    IIdentity Identity { get; }

    //
    // Summary:
    //     Determines whether the current principal
    //     belongs to the specified role.
    //
}
```

```

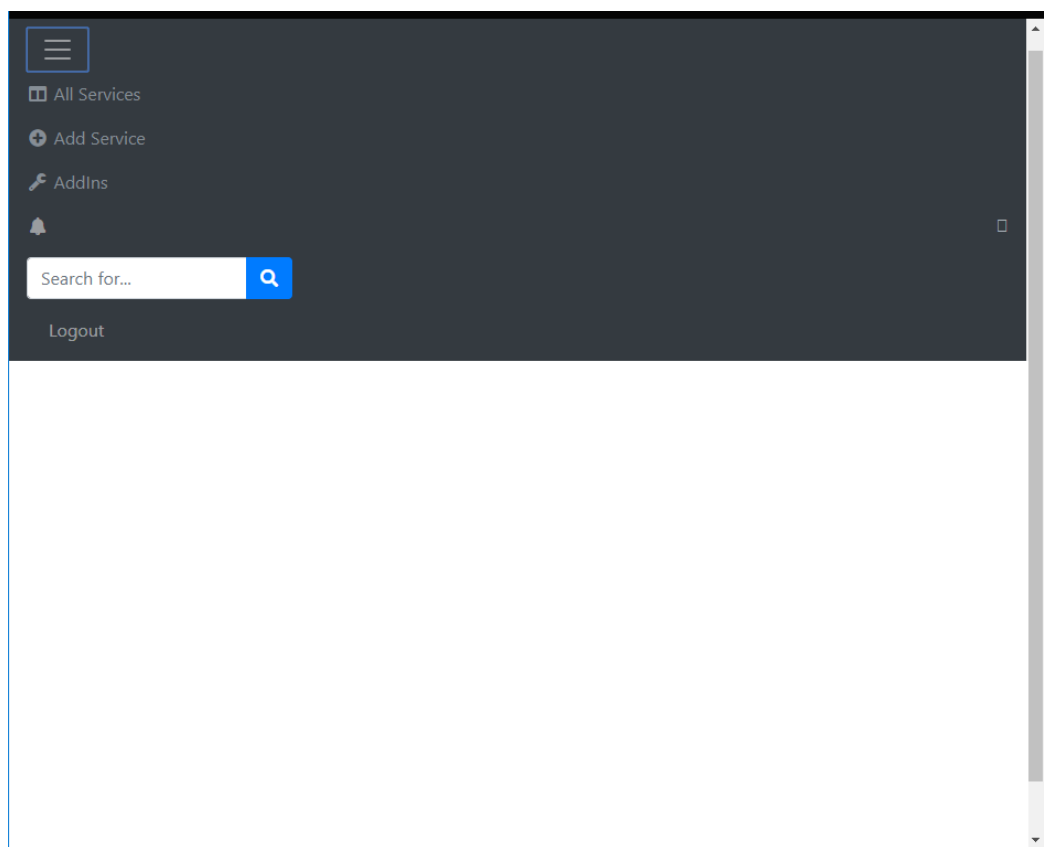
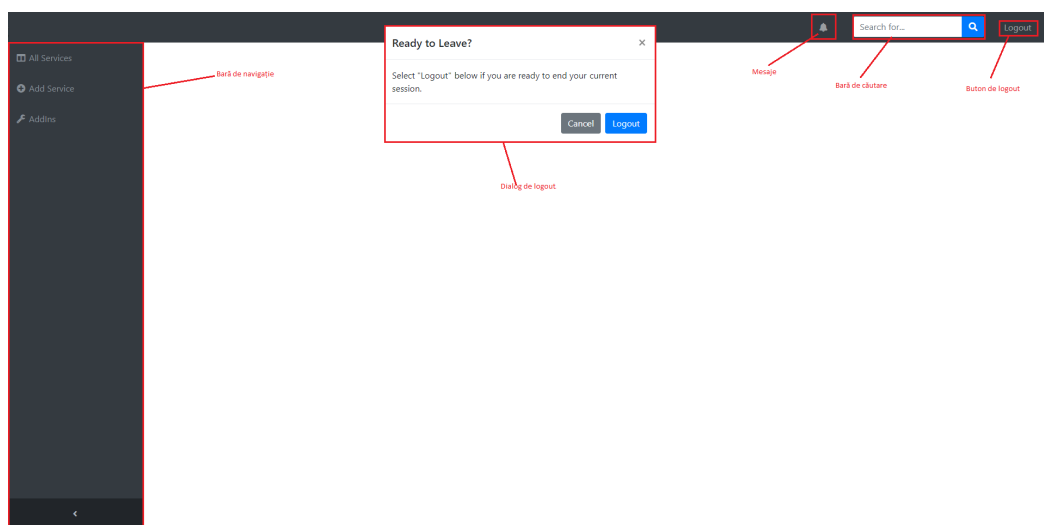
// Parameters:
//   role:
//       The name of the role for which to check
//       membership.
//
// Returns:
//       true if the current principal is a
//       member of the specified role; otherwise,
//       false.
bool IsInRole(string role);
}

```

3.5 Aspectul comun

În aplicația mea există câteva elemente comune care trebuie să apară în fiecare pagină. Bara de navigație trebuie să facă parte din fiecare pagină întrucât utilizatorul trebuie să poată schimba paginile site-ului cu ușurință. Utilizatorul trebuie să se poate deautentifica și ar fi absurd să poate face asta doar dintr-o anumită pagină. Este mult mai convenabil pentru utilizator să poată vedea dacă are mesaje noi din orice pagină, nu doar din pagina dedicată. Pe lângă acestea și funcționarea de căutare ar fi bine să fie accesibilă de oriunde în aplicație. Având aceste trăsături comune pentru fiecare pagină am nevoie de o metodă de implementare mai bună decât să copiez partea comună în fiecare pagină. Din fericire în ASP.NET există un tip de pagini numite pagini layout. Aceste pagini permit crearea unei pagini care are scopul de a aduna toate trăsăturile comune ale paginilor. Pagina layout nu este o pagină de sine stătătoare, ea trebuie folosită împreună cu o pagină care să conțină partea specifică a paginii ce va fi prezentată utilizatorului. În pagina layout se poate specifica exact locul în care pagina ce folosește această pagină de aspect va apărea. Numele paginii de aspect trebuie să înceapă cu `_`. Caracterul `_` este foarte important deoarece el înștiințează framework-ul să nu trimită niciodată direct pagina layout.

În cardul aplicației mele, am introdus în pagina de aspect, elementele comune ale tuturor paginilor [3.5]. Pagina are o bară de navigație, mimizabilă care poate trimite utilizatorul la pagina de management a serviciilor, la pagina de management a add-inurilor sau la pagina de introducere a unui nou serviciu. Bara de navigație se află în partea din stânga a paginii. Butonul din josul barei reduce dimensiunea acesteia. Interfața grafică este



receptivă, incluzând bara de navigație. Când ecranul este prea mic bara va fi înlocuită de un buton. La apăsarea butonului va apărea conținutul barei de navigație ca în imaginea [3.5].

Butonul Logout deschide o fereastră modală care cere confirmarea utilizatorului pentru a preveni, ieșirea din aplicație neintenționată. Dacă utilizatorul confirmă deautentificarea, sesiunea sa va fi distrusă și data de expirare a cookie-ului său va fi redusă în așa fel încât cookie-ul să fie expirat la ieșirea din aplicație.

Dupaăcum se vede și în figura [3.5] lângă butonul Logout se află o bară de căutare. Această bară est folosită pentru a căuta servicii sau clustere întregistrare în aplicație. După ce utilizatorul a scris numele sau o parte a numelui serviciului sau cluster-ului căutat, când va apăsa butonul de căutare, va fi trimis către o pagină ce conține rezultatul căutării sale. Dacă nu există nimic care să se potrivească cu ce a căutat el, în pagină va fi afișat un mesaj de informare.

În stânga barei de căutare se află un buton cu semnul unui colpot. La click pe buton va apărea o fereastră în care apar ultimele mesaje primite de utilizator și un buton. Mesajele sunt caracterizate prin subiect, conținut, expeditor și data expedierii. Butonul de sub mesaje duce utilizatorul către o pagină în care acesta poate vedea toate mesajele pe care le-a primit.

În ASP.NET MVC fiecărei pagini îi corespunde un model. Nu există nici o restricție de unicitate pe modele, mai multe pagini pot folosi același model. Dacă modelul este folosit doar pentru pagini, atunci acesta este numit viewmodel, deoarece este specific view-ului aplicației. Modelul unei pagini este folosit prin intermediul Razor pentru a genera pagina html care este trimisă în final utilizatorului site-ului.

Pentru că sunt date care sunt comune pentru toate paginile, acestea ar fi bine să facă parte dintr-un viewmodel. Soluția mea la această problemă a fost crearea unei clase BaseViewModel care conține informațiile comune. Toate celelalte viewmodel-e derivă din clasa BaseViewModel și pagina de aspect se folosește de clasa de bază pentru a-și lua datele necesare.

În ASP.NET fișierele de stiluri CSS și fișierele de scripturi JavaScript pot fi grupate în niște pachete numite bundle. Această grupare se face într-io clasă BundleConfig, care este apelată din clasa MVCApplication care reprezintă punctul de intrare în aplicație. Motivul pentru care este mai bine ca aceste fișiere să fie trimise împreună este că este mai rapid să fi trimise unitar către client decât separat. În pagina layout a aplicației mele am setat un bundle de fișiere CSS și unul de fișiere JavaScript care sunt folosite de către

toate paginile site-ului.

Fișierul CSS conține definiții pentru mai multe clase folosite în aplicație. Am ales să organizez partea de prezentare în clase CSS deoarece mi se pare cel mai flexibil mod de prezentare a elementelor HTML. Un element poate avea multiple clase ceea și fiecare poate descrie un aspect diferit al elementului. Spre exemplu un element poate avea În același timp clasa `dark` și clasa `sm-margin`. Clasa `dark` setează culoare fundalului și culoarea textului din element. Clasa `sm-margin` adaugă o margine mică elementului. Elementul nu este limitat la numai două clase și acesta va lua proprietăți din toate clasele sale. Consider că organizarea stilurilor pe clase este superioară setării proprietăților unui element după id, deoarece clasele sunt reutilizabile. De asemenea cred că utilizarea claselor este superioară setării proprietăților tuturor elementelor de un anumit tip deoarece există numeroase stituiții în care nu dorim ca toate elementele de același tip să fie tratate la fel. În pagina de management al serviciilor din Figura [3.6], atât cluster-ele cât și serviciile sunt elemente de tip `li`, dar acestea arată foarte diferit. De asemenea clasele pot afecta mai multe tipuri de elemente HTML. Folosind clasa `button` putem prezenta și un `button` simplu, și un `submit` al unui form și un `link`. Deși acestea sunt elemente diferite, se poate să aibă roluri similar și astfel să dorim ca acestea să arate la fel.

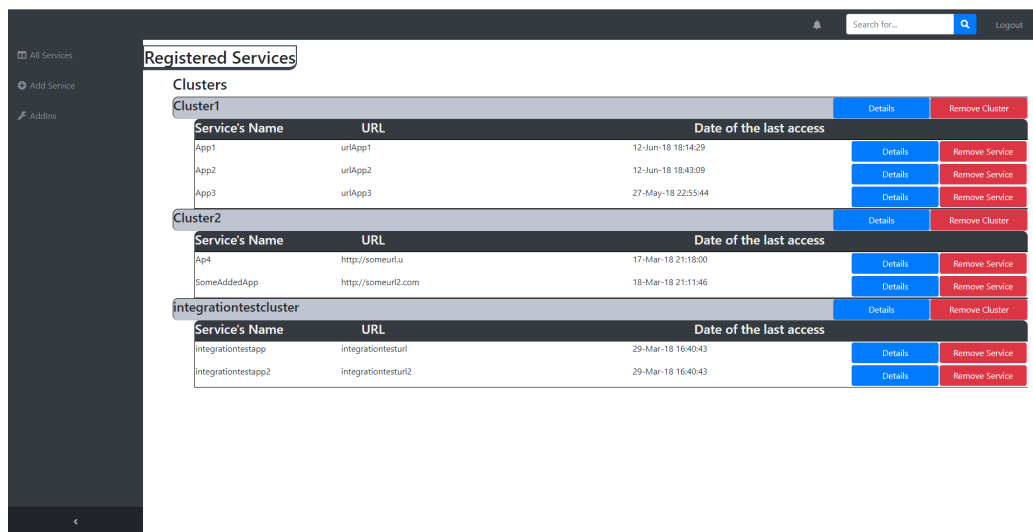
Tot pe baza claselor pot schimba comportamentul elementelor prin cod JavaScript. De exemplu, atunci când documentul este pregătit, toate elementele cu clasa `btn-run-addin` vor avea evenimentul de click setat. Atunci când utilizatorul va face click pe aceste elemente, printr-o cerere de tip `GET` la controller-ul `AddInsApiController`, se va lansa în execuție un `add-in`.

Fișierele mele JavaScript sunt făcute după un șablon de proiectare numit `Module Pattern`. `Module Pattern` este probabil cel mai comun șablon de proiectare din JavaScript. Acesta încearcă să imite clasele din limbaje de programare orientate pe obiecte precum `C#` sau `C++`. Deși nu putem avea toate beneficiile programării orientate pe obiecte, folosind acest șablon putem avea o încapsulare, returnând din modul doar ceea ce este public. Funcțiile care adaugă comportament la elementele HTML, fac parte din niște module care se adaugă la un obiect principal. Acest obiect conține o metodă pentru fiecare fișier și acestea sunt apelate prin intermediul acestuia doar atunci când documentul HTML este complet încărcat, împreună cu toate fișierele JavaScript.

Singura pagină din site care nu folosește aspectul comun este pagina de autentificare, deoarece toate funcționalitățile comune, necesită ca utilizatorul să fie autentificat.

3.6 Managementul serviciilor

Una dintre cele mai importante pagini din aplicație este pagina de management a serviciilor, care se poate vedea în figura [3.6]. În această pagină administratorul serviciului de înregistrare poate observa toate serviciile înregistrate, organizate în clusterele din care acestea fac parte. Inițial toate clusterurile sunt expandate, însă acestea pot fi minimizate. Dacă utilizatorul va da click pe numele sau în zona gri din jurul numelui cluster-ului, serviciile care aparțin clusterului se vor îndrepta către acesta până ce vor dispărea. Utilizatorul poate accesa cluster-ele pentru a le vedea activitatea mai în detaliu, sau poate accesa un serviciu pentru a-i monitoriza activitatea. De asemenea el poate șterge servicii din server, sau chiar clusteruri întregi. Pentru a efectua o operație de ștergere, utilizatorului i se cere confirmarea printr-un alert al browser-ului. Ștergerea se face printr-un apel ajax către o metodă a cotroller-ului `ServicesApiController`. În momentul în se primește un răspuns pozitiv de la server serviciul respectiv este șters și din pagină prin javascript. În cazul în care răspunsul primit este negativ se afișează un mesaj de tip alertă.



Registered Services		
Clusters		
Cluster1 Details Remove Cluster		
Service's Name	URL	Date of the last access
App1	urlApp1	12-Jun-18 18:14:29
App2	urlApp2	12-Jun-18 18:43:09
App3	urlApp3	27-May-18 22:55:44
Cluster2 Details Remove Cluster		
Service's Name	URL	Date of the last access
Ap4	http://someurl.u	17-Mar-18 21:18:00
SomeAddedApp	http://someurl2.com	18-Mar-18 21:11:46
integrationtestcluster Details Remove Cluster		
Service's Name	URL	Date of the last access
integrationtestapp	integrationtesturl	29-Mar-18 16:40:43
integrationtestapp2	integrationtesturl2	29-Mar-18 16:40:43

3.7 Managementul cluster-elor

Aplicația include și o pagină dedicată cluster-elor. În această pagină se află un grafic al activității serviciilor membre al cluster-ului ca în figura

[3.7]. Sub grafic se află lista serviciilor care fac parte din cluster, caracterizate prin numele de aplicație, numele cluster-ului, adresa serviciului și data ultimei sale accesări. În dreptul fiecărui serviciu din cluster se află 2 butoane. Un buton îndreaptă utilizatorul spre pagina dedicată serviciului respectiv și un buton șterge serviciul. În realitate butonul de navigare către pagina serviciului, nu este un element HTML de tip buton ci este un element de tip link. Diferența nu se observă deoarece acest buton și butonul de ștergere folosesc clase CSS similare.

Graficul este făcut în JavaScript, folosind biblioteca Chart.js. Prin controller poate returna o pagină care folosind Razor, transformă datele din viewmodel-ul paginii în structuri de date care pot fi folosite din cod JavaScript. Pentru ca un obiect C# să fie folosibil în cod JavaScript, acesta trebuie serializat în formatul JSON. Prin Razor folosesc cod C# pentru a serializa obiectul din viewmodel care corespunde datelor care vor defini graficul. Motivul pentru care pot folosi obiectul serializat este faptul că codul C# este executat înainte ca pagina să fie trimisă către utilizator. Codul JavaScript este interpretat de către browser-ul utilizatorului, deci se execută după ce s-a făcut serializare. În browser va apărea rezultatul serializării care fiind în format JSON, va fi interpretat exact ca crearea unui obiect JavaScript.

Pentru repartizarea culorilor corespunzătoare serviciilor reprezentate pe grafic am făcut un algoritm, care generează un număr dorit de culori distincte. Culorile rezultate nu sunt doar distincte ca ton, ci sunt și vizibil diferite. Pentru un număr mic de culori diferența între acestea va fi foarte mare dar se va reduce odată cu creșterea numărului de culori necesare. Algoritmul este scris în JavaScript deoarece consider că atâta timp cât nu reprezintă un risc de securitate este preferabil ca mașina clientului să efectueze cât mai mult din munca necesară. Cu cât mai multă procesare este efectuată de către client, cu atât mai mult pot reduce încărcătura server-ului.

3.8 Monitorizarea unui serviciu

Am considerat ca fiind de interes activitatea unui serviciu înregistrat în program, ceea ce m-a determinat să fac o pagină dedicată monitorizării unui serviciu. După cum se vede și în exemplul din Figura [3.3], pagina conține un grafic al activității serviciului într-un interval de timp precum și toate accesările serviciului. Graficul poate fi minimizat ca în pagina de management a cluster-elor, întrucât nu am dorit să ocup prea mult din ecranul utilizatorului dacă acesta este interesat la momentul respectiv de cererile



facute. Sub reprezentarea grafică s află o listă cu fiecare accesare a serviciului din intervalul de timp selectat. Pentru fiecare accesare se cunoaște adresa ip de proveniență a cererii și data la care aceasta a fost făcută.

3.9 Introducerea serviciilor

Deși serviciile ar trebui să se înregistreze singure, în cazul în care se dorește utilizarea unui serviciu de proveniență externă, se pot introduce servicii manual prin intermediul interfeței grafice. După cum se vede în figura [3.5], pentru a adăuga un serviciu este nevoie să se introducă numele de aplicație cu care acesta va fi înregistrat, numele cluster-ului din care acesta face parte și adresa serviciului. Numele cluster-ului se poate alege dintr-o

Figura 3.3: Pagina de management a unui serviciu

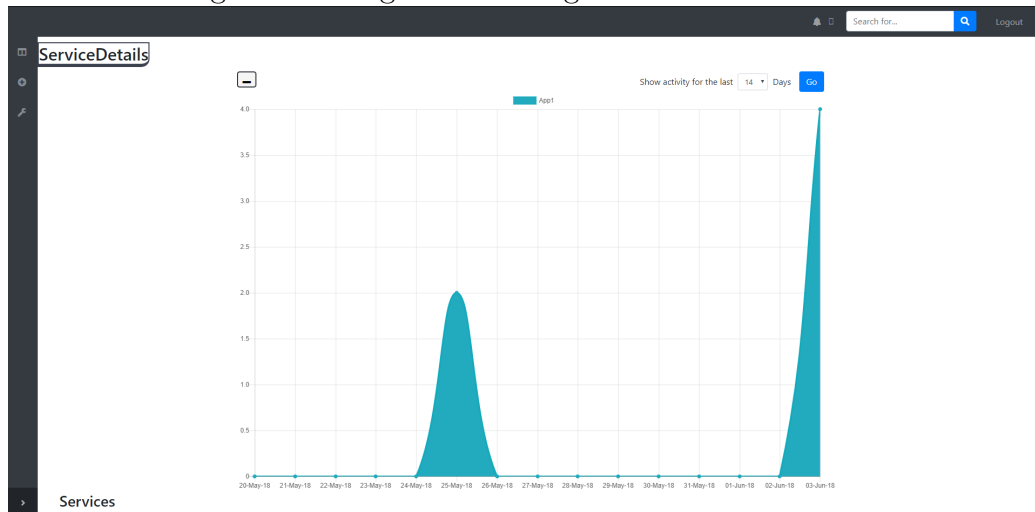


Figura 3.4: Pagina de management a unui serviciu în care graficul este minimizat

The screenshot shows the 'ServiceDetails' page with the graph minimized. Below the 'Services' header, there is a table with the following data:

Accesser's Address	Access Time
1	03.06.2018 22:52:25
1	03.06.2018 22:52:39
1	03.06.2018 22:52:41
1	03.06.2018 22:55:42
1	03.06.2018 22:58:51

listă ce conține cluster-ele existente sau se poate introduce un cluster nou. La apăsarea butonului "Congirm" cererea este trimisă către server și serviciul este înregistrat. Dacă înregistrarea eșuează, utilizatorul va fi înștiințat.

Figura 3.5: Pagina adăugare a unui serviciu

The screenshot shows a web application interface. On the left is a dark sidebar with three menu items: 'All Services', 'Add Service', and 'Addons'. The 'Add Service' item is highlighted. The main content area has a title 'Add Service' and a form. The form has two fields: 'Service's name' with a text input, and 'SelectedCluster' with a dropdown menu. The dropdown menu is open, showing three options: 'Cluster1', 'Cluster2', and 'Integrationtestcluster'. Below the dropdown is a blue 'Confirm' button. At the top right of the page, there is a search bar and a 'Logout' link.

3.10 Tratarea erorilor

Funcționarea serviciului de înregistrare este critică deoarece mai multe aplicații se bazează pe acesta. Din cauza rolului important pe care serviciul îl are, acesta trebuie să fie cât mai robust. Prima problemă care poate apărea constă în cererile primite cu parametri greșiți sau cu scop malițios. Fiecare acțiune a aplicației trebuie să știe să returneze un răspuns adecvat, fie el mesaj de eroare, o pagină web sau niște date. Decizia răspunsului în cazul în care o cerere eșuează se bazează pe tipul de excepție primit în metodele controllerului. Dacă pe parcursul procesării cererii venite prin controller nu apare nici o excepție sau apar doar excepții din care aplicația își revine cu succes, cererea va primi răspunsul dorit. În caz contrar depinde de tipul de cerere făcută. Dacă utilizatorul încearcă să acceseze o pagină a site-ului fără a fi autentificat, acesta va fi redirectionat către pagina de autentificare. Dacă se încearcă accesarea unei pagini la care utilizatorul nu are acces, acesta va fi redirectionat la o pagină de eroare care îi va explica situația în care se află. În ceea ce privește parte de Web API, unde răspunsul nu este sub formă de pagină web, utilizatorul care a făcut o cerere eronată va primi un cod de răspuns de eroare, Bad Request sau Not Found. Am ales să nu dau mai multe detalii despre eroare, din cauză problemelor de securitate care pot fi cauzate de aceste informații. O persoană rău intenționată ar putea profita de informațiile suplimentare pentru a produce daune serverului. Bineînțeles toate excepțiile care apar pe parcursul derulării aplicației, sunt salvate într-un

fișier de tip log.

3.11 Componente

3.11.1 NuGet

NuGet est un manager de pachete gratuit și open-source creat pentru platforma .NET. Acesta a fost creat în 2010 și de atunci a devenit din ce în ce mai popular. Datorită ușurinței cu care se pot aduce în proiect pachete externe, și posibilității de publicare a acelor pachete, NuGet a devenit omniprezent în dezvoltarea aplicațiilor în cadrul platformei .NET. NuGet facilitează dezvoltarea atât .NET Framework cât și în .NET Core. NuGet este și mecanismul de împărțit cod sprijinit de Microsoft. NuGet este distribuit ca o extensie de Visual Studio care începând cu Visual Studio 2012 vine preinstalată în IDE. Popularitatea NuGet a condus la integrarea sa completă în Visual Studio, în Visual Studio 2017 pe lângă managementul pachetelor, NuGet poate fi folosit direct din IDE pentru împachetarea bibliotecilor de clase.

3.11.2 Registry Connector

Am dorit ca serviciu de înregistrare să fie cât mai ușor de utilizat. Cu acest scop am creat două biblioteci care simplifică interacțiunea cu serverul. Mai mult, ca utilizatorii să poată beneficia de biblioteci cât mai ușor, am decis să distribui aceste biblioteci sub forma unor pachete NuGet. NuGet permite setarea surselor de pachete. NuGet se poate configura să folosească mai multe servere pe post de surse. Pachetele mele ar trebui să se afle pe un server privat, pentru a fi consumat intern în compania care folosește serviciul de înregistrare.

Prima bibliotecă, `IRegistryConnector` oferă un mod simplu de utilizare al serviciului. Serviciul expune o interfață prin intermediul căreia utilizatorul poate cere url-ul pentru un anumit serviciu. Interfața are două metode, una pentru obținerea adresei unui serviciu și una pentru reîmprospătarea cache-ului. Deoarece implementarea `RegistryConnector` folosește un mecanism de caching bazat pe un fișier, această clasă trebuie să fie sigură din punct de vedere al lucrului cu fire de execuție. Dacă același fișier este folosit din două fire de execuție diferite pot apărea erori. Problemele apar atunci când sunt

scrise date în fișier, iar citirea simultană nu este o problemă. Implementarea mea blochează operațiile ce folosesc fișierul, dacă un alt fir de execuție scrie în fișier în acel moment. După ce se termină scrierea în fișier, se poate opera din nou pe fișier. Mulțumită sistemului de caching crește viteza de execuție a programului pentru că nu mai este nevoie de o cerere către server. Mai important este faptul că având un cache local scade dependența aplicație de serviciul de înregistrare. Un alt beneficiu, de data aceasta pentru server, este că serverul nu mai este atât de solicitat.

3.11.3 Registry Manager

A doua componentă pe care am făcut-o este destinată serverelor care vor să se înregistreze în serviciul de înregistrare. Componenta conține o interfață, care permite înregistrarea și dezînregistrarea serviciilor.

3.12 Testare

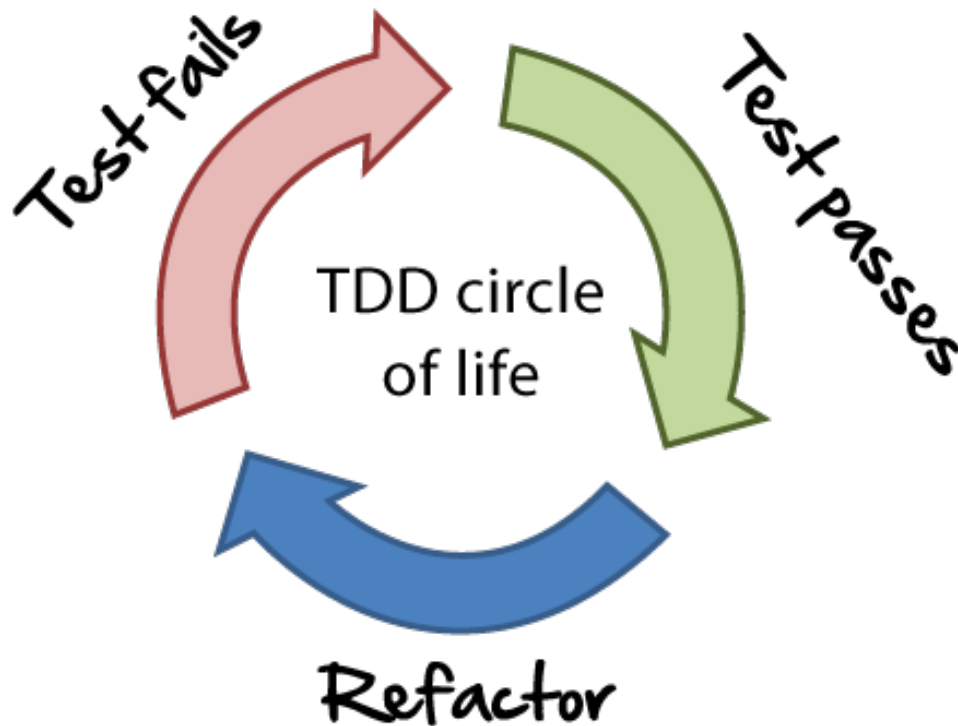
Este evident că cel mai important lucru la un program este ca acesta să fie funcțional. Singurul mod în care funcționarea acestuia poate fi asigurată este testarea programului. Testarea trebuie făcută la fiecare modificare a aplicație. Deși modificarea nu afectează direct alte funcționalități ale aplicație, există totuși riscul ca indirect să afecteze alte module. Genul acesta de schimbări neașteptate sunt unul din motivele pentru care o aplicație trebuie testată frecvent. Cu cât calitatea codului este mai ridicată cu atât mai puțin se vor petrece schimbări neașteptate, însă indiferent cât de sigur aș putea fi de modificările pe care le fac și de consecințele acestora, lipsa testării este un risc care ar trebui evitat. Testarea manuală este necesară și uneori nu poate fi evitată, dar aceasta consumă mult timp. Din acest motiv am ales să automatizez procesul de testare scriind teste automate.

Am ales să folosesc framework-ul NUnit pentru testarea automată din cadrul proiectului meu. NUnit este un framework de testare open source pentru limbajele platformei .NET. NUnit a pornit ca o portare a JUnit, însă versiunea curentă, NUnit 3, este rescrisă complet. Odată cu această rescriere, au fost adăugate și caracteristici noi și suport pentru mai multe platforme .NET. Folosind NUnit am putut să scriu teste automate cu care să testez componentele aplicației în izolare.

Tot pentru testele automate am folosit biblioteca Moq. Această bibliotecă îți permite crearea unor implementări de tip mock a interfețelor necesare testării unei clase. De multe ori clasele testate depind de alte clase sau interfețe dar când testăm o clasă în izolare este de preferat ca clasa respectivă să nu aibă dependențe. Cu acest scop se folosesc implementările mock ale interfețelor de care clasele depind. Cuvântul mock provine din engleză și se poate traduce în imitație. Implementările mock ale interfețelor sunt practic niște imitații utile doar pentru testare. Acestea sunt mai ușor de implementat și au funcționalități ajutătoare. De exemplu într-o implementare mock se poate verifica de câte ori a fost apelată o metodă sau se pot executa acțiuni după apelarea unei metode astfel implementate. Tot cu Moq putem face ca implementările unor să se comporte diferit în funcție de parametrii primiți sau să se comporte indentic indiferent de parametrii.

Testele automate încurajează un design cu o cuplare cât mai slabă între componente deoarece în caz contrar acestea nu mai pot fi testate în izolare. De exemplu, dacă în loc să structurez aplicația pe straturi așa fi avut toată logica aplicației într-un singur strat deja codul ar deveni mai dificil de testat. Mai mult, dacă fiecare cerere pe care serverul poate primi ar fi tratată în metoda corespunzătoare din controller, atunci pentru o metodă din controller ar trebuie făcut câte un test pentru fiecare scenariu. Astfel s-ar ajunge la un număr mare de teste din ce în ce mai complexe în funcție de cât e departe în execuția metodei va ajunge testul. Pe lângă această problemă, pentru multe din testele făcute va fi nevoie ca testele să facă o conexiune cu baza de date ceea ce nu doar încetinește viteza de execuție a testelor dar și crește posibilitatea de eșec a unui test din cauze externe. Cu cât testele depind mai mult de alte resurse este mai probabil ca un test să pice din alte motive decât dintr-o problemă codului testat ceea ce scade relevanța testelor. Testarea individuală în izolare, ajută la identificarea problemelor în cod, deoarece astfel se știe exact componenta în care se află o problemă. Având la dispoziție suita de teste pe care am scris-o, oricând doresc pot detecta dacă modificările pe care le fac funcționează corect. Pentru o astfel de verificare am nevoie de mai puțin de un minut. Bineînțeles această siguranță pe care o am legată de funcționarea aplicației este limitată de scenariile acoperite de teste. Într-o situație ideală în care tot codul ar fi acoperit complet de teste, acestea ar fi suficiente pentru a asigura funcționarea corectă a întregii aplicații.

Figura 3.6: Procesul de dezvoltare TDD



3.12.1 Test Driven Development

Pentru o parte a dezvoltării am lucrat urmând un proces numit test driven development (TDD). Acest proces se bazează pe repetarea unui scurt ciclu de dezvoltare. Primul pas al ciclului presupune scrierea unui test automat până în punctul în care codul testat pică testul respectiv. Dacă codul testului nu poate fi compilat, atunci se consideră că testul este picat. Acest lucru se întâmplă de exemplu atunci când într-un scenariu de test se folosește o metodă care încă nu există. În pasul al doilea se scrie suficient din codul testat încât testul să fie trecut, dar nu mai mult de atât. Mai există un pas în care se face o refactorizare a codului pentru eliminarea redundanțelor. Robert C. Martin a definit 3 legi ale TDD² menite să-i mențină pe practicanții

²Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 0137081073, 9780137081073.

aceste discipline într-un ciclu de aproximativ 30 de secunde:

1. Nu este permisă scrierea codului de producție înainte scrierea unui test pe care codul să îl pice.
2. Nu este permisă scrierea a mai mult dintr-un test decât este suficient pentru ca testul să eșueze.
3. Nu este permisă scrierea a mai mult cod de producție decât este necesar pentru ca testul corespunzător să fie trecut.

Una dintre clasele pe care le-am făcut practicând TDD este clasa `ServicesManager`. Spre exemplu, am creat metoda `GetService` din clasa `ServicesManager` începând cu testul `GetServiceFoundTest` [3.4] care testează metoda `GetUrl` din `ServicesManager`. Știind că clasa manager se folosește de o interfață `IServicesRepository`, a trebuit să încep testul făcând un mock pentru interfața respectivă. Implementarea mock va returna un obiect de tip `Service` specific atunci când metoda `GetService` va fi apelată cu parametrii specificați. Mai apoi instanțiez un obiect de tip `ServicesManager` căruia i-am pasat prin constructor instanța implementării mock a interfeței `IServicesRepository`. Compilarea codului eșuează deoarece nu există clasa `ServicesManager`. Din acest moment am trecut la partea următoare a ciclului, în care am început să scriu clasa. Din momentul în care am făcut constructorul cu parametru m-am întors la codul testului. Am început cu verificarea rezultatului metodei pe care o doresc și din nou compilarea se va încheia cu eșec. Atunci a fost momentul să creez metoda `GetUrl` și să implementez funcționalitatea care returnează url-ul corect din repository. În final am verificat ca metoda `GetService` să fi fost apelată.

Listing 3.4: Testul pentru metoda

```
[ Test ]
[ Category("GetService") ]
public void GetServiceFoundTest()
{
    const string LOCALHOST =
        "127.0.0.1";
    Mock<IServicesRepository> mockRepository =
        new Mock<IServicesRepository>();
    mockRepository.Setup(t => t.GetService("app",
        "cluster", LOCALHOST)).Returns(new
        Service() { AppName = "app", Cluster =
        new Cluster { Name="cluster" }, Url = "
        url" });
}
```

```
        . Verifiable ();  
    ServicesManager manager = new  
        ServicesManager (mockRepository . Object );  
    Assert . AreEqual ( " url " , manager . GetUrl ( " app  
        " , " cluster " , LOCALHOST ) );  
    mockRepository . Verify ();  
}
```


CAPITOLUL 4

Utilizare

Serviciul de înregistrare și site-ul de management al acestuia sunt menite să simplifice dezvoltarea și utilizarea microserviciilor. Din acest motiv ele trebuie să fie disponibile dezvoltatorilor acestor microservicii. Această aplicație nu este destinată unui public larg, ci est gândită pentru a fi folosită intern într-o companie. Doar membrii autorizați ai acestei companii au acces la site-ul de management, dar probabil toți cei care sunt implicați în dezvoltarea microserviciilor au acces la serviciu de înregistrare.

Primul lucru de care este nevoie pentru funcționarea serviciului de înregistrare și a site-ului este baza de date. Pentru asta este nevoie în primul rând e un sistem de gestiune al bazelor de date Microsoft SQL Server. Cel mai probabil acesta va fi pe un server separat de aplicații, deci trebuie ca cel ce face instalarea să se asigure că serverul pe care se află baza de date poate fi accesat de către celelalte servere. Creare efectivă a bazei de date pe server se poate face cu ușurință folosind scripturile SQL pe care le-am făcut cu acest scop. După ce baza de date a fost creată și este funcțională, trebuie create conturile utilizatorilor serverului. Acestea trebuie create printr-un script de către o persoană autorizată. Din moment ce serviciul de înregistrare este privat, conturile nu trebuie să poată fi create de oricine. Dacă conturile ar putea fi făcute de către oricine, securitatea serverului ar putea fi compromisă.

După ce baza de date funcționează și conturile au fost create, trebuie

instalat serviciul de înregistrare. Acesta trebuie instalat primul deoarece fără acesta, interfața de management este inutilă, ne având obiectul muncii sale. După ce serviciul de înregistrare funcționează, se poate face instalarea site-ului de management al serviciului.

Nu este important unde este instalată aplicația. Aceasta se poate afla pe Azure, pe un server intern sau se poate folosi orice serviciu de găzduire. Cu toate acestea poate ar fi mai bine ca serverul de management să fie într-o rețea de tip intranet. Site-ul nu este nevoie să fie accesibil din afara companiei și acest lucru ar putea îmbunătăți securitatea.

Dacă utilizatorii serviciului de înregistrare plănuiesc să dezvolte aplicații în C#, atunci ar fi cel mai bine să se folosească componentele de NuGet pe care le-am făcut. Acestea ar trebui distribuite folosind un server NuGet privat care să aparțină companiei. Folosind un server NuGet este mai ușor pentru utilizatori să actualizeze componentele în momentul în care apar noi versiuni. Astfel de modificări pot apărea atunci când se schimbă modul de utilizare al serverului. Dacă vor apărea noi măsuri de securitate în contactarea serviciului de înregistrare, componentele va trebui actualizate pentru a suporta schimbările de pe server. Dacă nu se poate folosi un Server de NuGet, există două soluții alternative. A soluție este crearea unui director în care să se aște toate versiunile componentelor și utilizarea directorului ca un depozit NuGet local. O altă soluție, mai primitivă este utilizarea directă fișierului dll din pachetul NuGet.

4.1 Serviciul de înregistrare

Serviciul de înregistrare este menit pentru a fi utilizat de către alte aplicații, nu de către persoane. Acesta se utilizează prin metode HTTP. Pentru obținerea adresei unui serviciu înregistrat, și deînregistrarea se folosește metoda GET, pentru înregistrarea unui serviciu se folosește put. Înregistrarea unui serviciu prin acest mod ar trebui făcută de către serviciul care dorește să fie înregistrat. Dacă se dorește înregistrarea unui serviciu în alt mod, atunci este recomandată folosirea site-ului de management al serviciului de înregistrare. Deînregistrarea unui serviciu trebuie făcută de către serviciul care s-a înregistrat. Dacă se încearcă ștergerea unui serviciu de la altă adresă ip decât cea de la care acesta a fost înregistrat, atunci această încercare va eșua. Am ales acest comportament din motive de securitate, posibilitatea de a șterge servicii din orice loc este o vulnerabilitate care are premite unui atacator să șteargă servicii necesare și astfel să oprească

funcționarea unor aplicații care necesită serviciul respectiv. Deși un serviciu nu poate fi dezînregistrat prin metoda prezentată anterior din alte servere decât cel care a făcut înregistrarea, asta nu înseamnă că un serviciu poate să rămână veșnic înregistrat. Dacă serviciul devine inactiv acesta va fi șters automat de către serviciul de înregistrare. Dacă se dorește eliminarea unui serviciu deși acesta funcționează, acest lucru se poate realiza prin interfața grafică a site-ului de management.

Comportamentul serviciului de înregistrare poate fi modificat printr-un add-in. Acesta este un add-in diferit de restul add-inurilor pe care aplicația le acceptă. Acest add-in are rolul de a implementa un algoritm de echilibrare a încărcăturii (load balancing). Odată încărcat în aplicație add-inul va fi folosit pentru aducerea serviciilor din baza de date. Scopul acestui add-in este de a reduce încărcătura unui singur serviciu de un anumit tip, distribuind cererile între toate serviciile de același tip. Bineînțeles acest add-in nu este util în cazul în care există doar câte un singur serviciu de un anumit tip.

4.2 Interfața grafică a serviciului de înregistrare

Interfața grafică a serviciului de înregistrare se află pe un server separat de serviciu. Prin intermediul acestei interfețe grafice se poate monitoriza activitatea serviciului și se poate interacționa cu acesta. Interfața grafică a serviciului constă într-un site făcut în ASP.NET MVC.

Primul pas necesar pentru interacționarea cu serverul constă în autentificarea utilizatorului. Autentificarea utilizatorilor se face pe baza conturilor menționate în secțiunea anterioară. Autentificarea necesită adresa de email și parola utilizatorului după cum se poate observa în imaginea [3.4] de la pagina 31. Pasul de autentificare este obligatoriu și dacă utilizatorii încearcă accesarea oricărei alte pagini ale site-ului fără ca înainte să treacă prin pasul de autentificare, aceștia vor fi redirecționați către pagina de autentificare.

După ce autentificarea este reușită utilizatorul este redirecționat către pagina pe care se regăsesc toate serviciile. Dacă utilizatorul a încercat să acceseze o pagină fără să fie autentificat și a fost redirecționat către pagina de autentificare, după autentificare va fi redirecționat către pagina pe care a încercat să o acceseze.

Pagina implicită la care utilizatorul este trimis după autentificare este pagina din imaginea [3.6] de la pagina 37, prin care se poate interacționa cu toate serviciile. Am ales această pagină ca pagină deoarece aceasta este pagina

care oferă privirea cea mai generală asupra serviciilor. pe lângă prezentarea unei priviri de ansamblu asupra serviciilor, tot prin această pagină se poate interacționa cu serviciile înregistrate. Prin intermediul paginii se pot șterge atât clustere cât și servicii individuale. De asemenea această pagină oferă și un punct de acces către paginile dedicate clusterelor sau serviciilor.

Pagina dedicată unui cluster informează utilizatorul cu privire la activitatea serviciilor din cluster dintr-un interval de timp. După cum se poate observa și pe exemplul din imaginea [3.7] de la pagina 39, activitatea serviciilor este reprezentată într-un grafic. În grafic fiecare serviciu este reprezentat printr-o culoare distinctă și reprezintă activitatea serviciilor după numărul de accesări din fiecare zi. Graficul poate fi minimizat în cazul în care utilizatorul este mai interesat de lista serviciilor, afișată sub grafic. Serviciile reprezentate în listă pot fi șterse cluster prin butonul 'Remove Service'. De asemenea prin butonul vecin, 'Details' se poate trece la pagina dedicată serviciului respectiv.

Prin intermediul paginii pe care sunt afișate toate serviciile și al paginii dedicate unui cluster, se poate accesa pagina din figura [3.3] de la pagina 40. Această pagină este dedicată unui serviciu și oferă mai multe detalii despre activitatea acestora. Utilizatorul poate vedea activitatea serviciului dintr-un interval de timp selectat de el. Activitatea serviciului este reprezentată printr-un grafic. La fel ca în cazul graficului din pagina activității unui cluster, și acest grafic de pe această pagină este minimizabil. Sub grafic se află o listă cu toate datele în care serviciul a fost accesat și adresa ip de la care a fost accesat.

În stânga paginii se află bara de navigație a site-ului. Acestă bară se află în fiecare pagină a site-ului și poate fi minimizată de către utilizator dacă acesta dorește mai mult spațiu pentru pagina accesată. Bara de navigație reprezintă modul de schimbare a paginilor aplicației. Dacă ecranul utilizatorului nu este destul de încăpător, bara de navigație se schimbă într-un buton în partea din stânga sus a ecranului ca în imaginea [3.5] de la pagina 34.

Interfața grafică permite și adăugarea de servicii noi, nu doar ștergerea celor deja existente. Introducerea noilor servicii se face prin pagina din figura [3.5] de la pagina 41. Această pagină poate fi accesată prin intermediul barei de navigație. Pentru a adăuga un nou serviciu trebuie completate toate câmpurile din formularul de pe pagină, mai exact numele aplicației înregistrate, numele cluster-ului din care va face parte și adresa la care se găsește acesta. Câmpul în care trebuie completat oferă o listă cu toate clusterurile deja existente din care utilizatorul poate alege, dar permite și introducerea unui cluster nou.

A treia pagină care poate fi accesată prin intermediul barei de navigație este pagina de management a add-inurilor. Această pagină oferă utilizatorului informații despre add-inurile încărcate în aplicație și posibilitatea de a interacționa cu acestea. O astfel de interacțiune este înlăturarea acestora prin intermediul unui buton, la apăsarea căruia, utilizatorul este rugat să își confirme decizia. Pentru add-inurile active, există un buton care permite lansarea acestora în execuție.

Add-inurile sunt un mod avansat în care utilizatorii pot interacționa cu aplicația. Aceștia pot decide să extindă chiar ei aplicația prin add-inuri. Utilizatorii pot extinde aplicația fără să aibă acces la codul sursă al acesteia. Le-am pus la dispoziție utilizatorilor libertatea de a extinde aplicația după nevoia și preferințele lor. Aplicația va încărca add-inurile pe baza unor fișiere cu extensia `.sAddIn` care descriu add-inul care trebuie încărcat.

Pe lângă bara de navigație mai există și alte elemente comune tuturor paginilor site-ului. Un astfel de element constă în butonul Logout prin care utilizatorul se poate deautentifica. După apăsarea acestuia utilizatorul autentificat trebuie să confirme opțiunea sa printr-o fereastră de confirmare.

Un alt element pe care toate paginile îl conține este bara de căutare. Aici utilizatorul poate scrie numele sau prețurile din numele unui serviciu sau cluster. După apăsarea butonului de căutare, utilizatorului i se va prezenta o pagină în care poate vedea rezultatele căutării sale. Din pagina cu rezultatele utilizatorul poate naviga către pagina corespunzătoare rezultatului. Dacă nu este găsit nici un rezultat la cererea de căutare a utilizatorului, acesta va fi înștiințat.

Lângă bara de căutare se află un buton de mesaje. La apăsarea acestui buton, se va deschide o fereastră în care vor apărea mesajele pe care utilizatorul autentificat le-a primit. De asemenea în această fereastră există un buton care duce la o pagină în care sunt afișate toate mesajele primite de către utilizator. Utilizatorii nu pot trimite mesaje, acestea pot fi trimise doar de către aplicație sau de către add-inuri.

CAPITOLUL 5

Planuri de viitor

5.1 Securitatea

Poate cel mai important lucru care trebuie avut în vedere atunci când se pune problema dezvoltării viitoare a aplicației este securitatea. Serviciul de înregistrare poate fi considerat un punct sensibil în funcționarea multor aplicații, nu pentru că ar fi mai vulnerabil ca alte servicii, ci pentru că de funcționarea corectă a acestuia depind mai multe aplicații. Dacă un server specific unei aplicații își va întrerupe activitatea, acea aplicație va avea de suferit, dar dacă serviciul de înregistrare este inactiv atunci se poate ca zeci de aplicații să aibă de suferit.

De exemplu, în cazul unui magazin online care folosește o arhitectură orientată pe microservicii, dacă serviciul de plată devine inactiv, utilizatorii nu vor mai putea finaliza comenzile lor, însă accesul nu vor fi opritți din a beneficia de restul site-ului. Utilizatorii vor putea păstra produsele dorite în coși și vor da comanda la o altă dată. Nefuncționarea serviciului de plată va fi o inconveniență dar nu va doborâ întreg site-ul.

Pe de altă parte dacă magazinul ar folosi serviciul de înregistrare și acesta ar ceda, ar înceta întreaga activitate a site-ului. Această problemă poate fi atenuată în cazul în care există un sistem de caching pentru rezultatele

primate de la serviciul de înregistrare. Dependența serviciilor de serviciul de înregistrare face ca securitatea acestuia să fie o prioritate în dezvoltarea sa.

În momentul de față am implementat câteva măsuri de securitate. Una dintre aceste măsuri constă în procedurile de autentificare și autorizare ale site-ului de management al serviciului de înregistrare. Dezînregistrarea serviciilor este probabil operația care are cel mai multe nevoie de securitate deoarece această operație poate opri întreaga funcționare a serverului. Un atacator al serviciului ar putea șterge toate serviciile din server și astfel oprind funcționarea tuturor aplicațiilor care utilizează serviciul de înregistrare. Pentru a preveni această tip de atac am restricționat dezînregistrarea serviciilor. Prin verificarea adresei ip, doar serviciul care a făcut înregistrarea poate face dezînregistrarea.

O altă măsură de securitate pe care am implementat-o este utilizarea certificatelor pentru înregistrarea sau ștergerea serviciilor de pe server. Serviciile care sunt autorizate să se înregistreze ar vor avea un certificat și pe baza acestuia, serverul le va accepta sau refuza cererea. Deși implementarea pentru acest mecanism a fost făcută, în momentul de față ea nu este folosită. Această metodă de autorizare ar putea reprezenta o alternativă la restricționarea celor care pot face dezînregistrarea.

Pe lângă măsurile de securitate pe care le-am prezentat trebuie să mă asigur că nu există alte vulnerabilități exploatabile în aplicație. În sensul acesta, va trebui să folosesc un scanner de vulnerabilități. Imediat după descoperirea vulnerabilităților va trebui să încep eliminarea acestora.

5.2 Testarea

Un alt obiectiv pe care va trebui să îl am în vedere este testarea automată. Deși am început scrierea testelor automate, acestea nu sunt încă suficiente pentru a asigura funcționarea completă aplicației conform așteptărilor mele. Ca urmare este nevoie să cresc numărul testelor până în punctul în care codul meu va avea o acoperire de aproape 100% .

Deși scrierea tuturor testelor necesare se va întinde pe o perioadă îndelungată, pe termen lung voi putea economisi mult timp care ar fi consumat de testarea manuală a tuturor scenariilor de utilizare. Mai mult, testele vor reduce timpul petrecut în debug deoarece vor oferi o localizare destul de specifică a erorilor.

5.3 Adaptarea

Pe măsură ce aplicația va fi folosită de către utilizatori, vor apărea și noi cerințe. Este inevitabil ca utilizatorii să aibă propuneri referitoare la caracteristici pe care le doresc în aplicație. Astfel eu va trebuie să mă conformez cu noile cerințe pe care le voi aduna de la utilizatori. Prin arhitectura aplicației am încercat să anticipez schimbările. Deciziile pe care le-am luat pe parcursul implementării fac dezvoltarea acestora cât mai ușoară.

CAPITOLUL 6

Concluzii

Aplicația mea oferă un mod simplificat pentru dezvoltarea aplicațiilor folosind o arhitectură orientată pe microservicii. Serviciul de înregistrare oferă o metodă de a găsi adresele microserviciilor necesare aplicației într-un mod dinamic și reduce rigiditatea sistemului.

Deoarece înregistrarea pe server este responsabilitatea microserviciilor, este asigurat faptul că atâta timp cât există servicii funcționale, acestea vor fi folosite de către aplicații. Restricționarea dezînregistrării împiedică ștergerea malițioasă, dar permite ștergerea dacă serviciile o necesită.

De asemenea monitorizarea serviciului de înregistrare elimină posibilitatea utilizării unui microserviciu care nu mai este într-o stare de funcționare. Nu numai că nu vor mai exista servicii inactive, dar acestea sunt și mult mai ușor de actualizat.

Un alt beneficiu al serviciului de înregistrare constă în faptul că acesta acționează și ca un server de tip load balancer. Un serviciu de înregistrare se potrivește bine pentru funcția de echilibrare a încărcăturii serverelor pe care le gestionează. Algoritmul care face echilibrarea poate fi schimbat printr-un Add-In ceea ce facilitează îmbunătățirea acestuia.

Datorită mecanismului de add-inuri, se pot adăuga funcționalități noi chiar și în timp ce aplicația funcționează. Mai mult, chiar și utilizatorii au

libertatea de a-și crea propriile lor add-inuri.

Cu ajutorul interfeței grafice administratorii sistemului au control deplin asupra serviciilor și add-inurilor din aplicație. Utilizarea interfeței grafice a aplicației nu este limitată la utilizatorii cu drepturi de administratori, pe lângă aceștia există și utilizatori care nu au dreptul de a modifica serviciile sau add-inurile încărcate, dar pot utiliza restul aplicației.

Bibliography

- Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154916.
- Freeman, Elisabeth, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly & Associates, Inc., 2004. ISBN: 0596007124.
- Meyer, Eric A. *Cascading Style Sheets: The Definitive Guide*. O'Reilly Media, 2004.
- Flanagan, David. *JavaScript: The Definitive Guide Activate Your Web Pages*. 6th. O'Reilly Media, Inc., 2011. ISBN: 0596805527, 9780596805524.
- Martin, Robert C. *The Clean Coder: A Code of Conduct for Professional Programmers*. 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 0137081073, 9780137081073.
- Block, Glenn, Pedro Felix, Howard Dierking, Darrel Miller, and Pablo Cibraro. *Designing Evolvable Web APIs with ASP.NET: Harnessing the Power of the Web*. O'Reilly Media, 2014, p. 23.
- Galloway, Jon, David Matson, Brad Wilson, and K. Scott Allen. *Professional ASP.NET MVC 5*. Wrox, 2014, p. 2.
- *Professional ASP.NET MVC 5*. Wrox, 2014, p. 7.
- Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2014, p. 1.
- *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2014, p. 5.
- .Net Framework*. https://en.wikipedia.org/wiki/.NET_Framework.
- HTML*. <https://en.wikipedia.org/wiki/HTML>.

Microsoft SQL Server. <https://searchsqlserver.techtarget.com/definition/SQL-Server>.