UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Uroš Kastelic

Arhitekturni vidiki aplikacij SaaS na primeru Windows Azure

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Uroš Kastelic

Arhitekturni vidiki aplikacij SaaS na primeru Windows Azure

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: prof. dr. Matjaž Branko Jurič

Ljubljana, 2014

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Uroš Kastelic, z vpisno številko **63070267**, sem avtor magistrskega dela z naslovom:

Arhitekturni vidiki aplikacij SaaS na primeru Windows Azure

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 24. maja 2014 Podpis avtorja:

ZAHVALA

Zahvaljujem se prof. dr. Matjažu B. Juriču za vse strokovne nasvete in pomoč med pisanjem magistrske naloge.

Iskrena zahvala gre tudi domačim in najbližjim za podporo med vsemi leti študija.

Hvala tudi vsem ostalim, ki ste mi med študijem stali ob strani in me podpirali.

KAZALO

KAZALO SLIK

KAZALO TABEL

SEZNAM KRATIC IN SIMBOLOV

Kratica	Angleško	Slovensko
AJAX	Asynchronous JavaScript and XML	Asinhroni JavaScript in XML
API	Application Programming Interface	Aplikacijski programski vmesnik
AWS	Amazon Web Services	Amazonove spletne storitve
CDN	Content Delivery Network	Omrežje za dostavo vsebine
CMS	Content Management System	Sistem za upravljanje z vsebinami
CRM	Customer Relationship Management	Upravljanje odnosov s strankami
GOF	Gang Of Four	Skupina štirih avtorjev knjige o vzorcih
HTTP	Hypertext Transfer Protocol	Protokol za prenos informacij na spletu
IaaS	Infrastructure as a Service	Infrastruktura kot storitev
IIS	Internet Information Services	Spletne informacijske storitve
JSON	JavaScript Object Notation	JavaScript objektna notacija
JWT	JSON Web Token	JSON spletni žeton
MVC	Model - View – Controller	Model – Pogled – Krmilnik
NIST	National Institute of Standards and Technology	Nacionalni inštitut za standarde in tehnologijo
NoSQL	Not-Only SQL	Ne samo SQL
PaaS	Platform as a Service	Platforma kot storitev
REST	Representational State Transfer	Predstavitveni prenos stanja
SaaS	Software as a Service	Programska oprema kot storitev
SAML	Security Assertion Markup Language	Označevalni jezik za varnostne trditve
SDK	Software Development Kit	Programski razvojni paket
SLA	Service-Level Agreement	Sporazum o ravni storitev
SOA	Service-Oriented Architecture	Storitveno usmerjena arhitektura
SOAP	Simple Object Access Protocol	Enostaven dostopni objektni protokol
SQL	Structured Query Language	Strukturiran poizvedovalni jezik
SWT	Simple Web Token	Enostavni spletni žeton
W3C	World Wide Web Consortium	Konzorcij svetovnega spleta
WSDL	Web Services Description Language	Jezik za opis spletnih storitev
XML	Extensible Markup Language	Razširljiv označevalni jezik

POVZETEK

Arhitekturni in načrtovalski vzorci pomagajo reševati specifične probleme med procesom razvoja programske opreme. Pojav novih tehnologij, kot je računalništvo v oblaku, zahteva vpeljavo arhitekturnih in načrtovalskih vzorcev tudi na tem področju. Razvijalcem je potrebno ponuditi vzorce, preko katerih lahko enostavneje in hitreje implementirajo funkcionalnosti aplikacijam, ki so specifične za oblak. S pomočjo pristopa za dokumentiranje vzorcev, kot so ga definirali avtorji GoF (Gang of Four), so v magistrski nalogi definirani in implementirani trije načrtovalski vzorci. Predlagani vzorci se lotevajo področja podatkovne izoliranosti v večnajemniški aplikaciji, avtentikacije in avtorizacije uporabnika pri dostopu do spletnih storitev ter področja dvosmerne komunikacije s pomočjo sporočilnih vrst. S pomočjo podrobnega opisa implementacije vzorcev lahko razvijalci hitreje in učinkoviteje implementirajo predlagane vzorce na lastnih aplikacijah, specifičnih za oblak.

Ključne besede

Aplikacije SaaS, načrtovalski vzorci, Windows Azure, računalništvo v oblaku

ABSTRACT

Architectural and design patterns help solving specific problems during the process of software development. The advent of new technologies like cloud computing, demand the introduction of new architectural and design patterns for this type of technology. Developers need to be offered patterns, through which they could easier and quicker implement functionalities to the cloud applications. In this thesis there are defined and implemented three design patterns, using the approach of documenting patterns like it was defined by the GoF (Gang of Four). Suggested design patterns are addressing the areas like data isolation in multi-tenancy applications, authentication and authorization of the user to access web services and a two-way communication using messaging queues. Using the detailed description of the implementation of these patterns, developers can quicker and effectively implement suggested patterns on their cloud applications.

Keywords

SaaS applications, design patterns, Windows Azure, cloud computing

1. UVOD

1.1. Motivacija in cilji

Razvijalci se velikokrat pred in med razvojem programske opreme sprašujejo, kakšni so primerni načini za razvoj določene programerske naloge. Tako so začeli razmišljati o vpeljavi vzorcev, ki bi olajšali razvoj in usmerili razvijalce na pravo pot pri načrtovanju in razvoju informacijskih sistemov. Skozi daljše obdobje so se razvili načrtovalski in arhitekturni vzorci. Arhitekturni vzorci pomagajo določiti in organizirati osnovno strukturo programske opreme. Vsak nadaljnji razvoj programske opreme je namreč odvisen od te strukture. Primer arhitekturnega vzorca je model–pogled–krmilnik (angl. "Model-View-Controller"). Medtem ko arhitekturni vzorci predstavljajo predloge za določeno arhitekturo programske opreme, so načrtovalski vzorci abstraktne predloge za rešitev določenega problema v izbranem kontekstu. Primer načrtovalskega vzorca je objavi–naroči (angl. "Publish-Subscribe") [24]. Širom sveta so jih razvijalci preizkušali in nadgrajevali, dokler niso bili primerni in potrjeni za širšo uporabo. Tako lahko sedaj razvijalci uporabijo vzorce v svojem razvoju, vedoč, da bodo z njihovo pomočjo hitreje in pravilneje rešili lastne načrtovalske probleme ter omogočili hitrejši in lažji razvoj aplikacij.

S prihodom tehnologije oblaka so se odprla nova obzorja tudi v načrtovanju in razvoju programskih sistemov, zaradi česar se je pojavila potreba po načrtovalskih in arhitekturnih vzorcih na tem področju. Tehnologija oblaka je razmeroma mlada tehnologija, ki se še razvija, kljub temu pa že zajema zelo širok spekter storitev, od infrastrukture do programske opreme in aplikacije. Navkljub krajšemu obdobju na trgu in dvomljivosti uporabnikov v uporabo in varnost tehnologije, je marketinški vpliv dovolj močan, da ljudje pogledujejo proti aplikacijam za oblak. Tehnologija oblaka predstavlja novosti v uporabi sistemov in aplikacij. Marsikatera novost je dejanska prednost pred dosedanjim načinom uporabe, saj omogoča funkcionalnosti, ki prej niso bile tako lahko izvedljive. Tako se vsako leto pojavlja vedno več programskih sistemov, spletnih aplikacij in storitev, ki izkoriščajo prednosti tehnologije oblaka in ponujajo uporabnikom visoko razpoložljive storitve. Od neke točke naprej se tudi načrtovanje za oblak razlikuje od dosedanjega, zato je potrebno izdelati nove vzorce za načrtovanje in razvoj programskih sistemov. Razvoj aplikacij za oblak poteka še dovolj samosvoje, saj je vzorcev za oblak še premalo. Tako sicer že obstaja vzorec večnajemništva (angl. "Multi-Tenancy") [52], vendar se ta dotika le enega načina izoliranosti podatkov, kar postane problematično, v kolikor imamo najemnike, ki želijo drugačen način izoliranosti podatkov. Predstavili bomo vzorec, ki omogoča, da se najemnikom v večnajemniški aplikaciji nastavi način izoliranosti podatkov, kateri jim bolj ustreza. Ta vzorec rešuje problem, da bi potrebovali več istih aplikacij, katere pa bi uporabljale različni nivo izoliranosti podatkov. Poleg izolacije podatkov je prav tako pomembno področje avtorizacije uporabnika pri dostopu do spletnih storitev, saj je marsikdaj potrebno omejiti dostope do določenih metod spletnih

storitev. Glede na to, da veliko organizacij uporablja aktivne imenike, želimo ponuditi vzorec, ki za avtentikacijo in avtorizacijo uporabnikov uporablja aktivni imenik. S tem bi omogočili, da organizacije obdržijo enotno organizacijsko strukturo in razvijalce razbremenili gradnje dodatnih avtentikacijskih in avtorizacijskih mehanizmov. Zadnji vzorec se loteva področja sporočilnih vrst in obdelave uporabniških zahtev, ki ne potrebujejo takojšnjega odgovora. Pošiljanje uporabniških zahtev drugemu procesu razbremeni uporabnika in aplikacijo, tako da lahko le-ta nadaljuje s svojim delom, medtem ko se zahteva nekje drugje procesira. Korelacija zahteve in povratno sporočanje pošiljatelju je lahko problematično, v kolikor želimo v realnem času uporabniku vrniti odgovor. Tako bomo preko vzorca pokazali način dvosmerne komunikacije s pomočjo sporočilnih vrst ter komunikacijskih kanalov. S tem bomo rešili problem čakanja odgovora na zahtevo in takoj, ko bo le-ta pripravljen, o tem obvestili prvotnega pošiljatelja.

Cilj magistrske naloge je definirati nove načrtovalske vzorce na področju računalniških oblakov. Le ti omogočajo enostavnejšo, hitrejšo in bolj sistematično ponovno uporabo ter bistveno izboljšajo kakovost programske opreme, ki je načrtovana za izvajanje v okoljih oblaka. Prvi predlagani načrtovalski vzorec pomaga reševati problem izbire podatkovne izoliranosti v večnajemniški aplikaciji. Naslednji vzorec omogoča avtentikacijo in avtorizacijo uporabnikov pri dostopu do spletnih storitev preko aktivnih imenikov, ki jih večinoma uporabljajo vse večje organizacije. Zadnji predlagan vzorec pa omogoča dvosmerno sporočanje pri obdelavi uporabniških zahtev s pomočjo sporočilnih vrst. Microsoft, Amazon in Google so trenutno najbolj popularni ponudniki javnih storitev oblaka. Glede na to, da veliko organizacij uporablja programsko opremo podjetja Microsoft, obstaja veliko možnosti, da bi prenos aplikacij raje naredili na njihov oblak, kot na ostala dva. Zaradi tega se bomo posvetili gradnji vzorcev na Microsoftovem javnem oblaku Windows Azure. S pomočjo predlaganih vzorcev bodo lahko razvijalci hitreje razvili spletne aplikacije, specifične za oblak, saj so v njih prav tako podrobno opisani postopki za implementacijo.

1.2. Zgradba magistrske naloge

V drugem poglavju opredelimo koncept računalništva v oblaku, analiziramo njegove definicije ter opišemo kako se je računalništvo v oblaku razvijalo skozi zgodovino. Prav tako razložimo kategorizacijo računalništva v oblaku na storitvene in namestitvene modele. Poglavje končamo s prikazom storitev večjih ponudnikov javnih oblakov in cenovnim obračunom uporabe storitev oblaka Windows Azure.

Sledi poglavje, v katerem predstavimo načrtovalske in arhitekturne vzorce. Opišemo kako se lotiti dokumentiranja vzorcev in izpostavimo ključne arhitekturne ter načrtovalske vzorce. Vse skupaj nato prenesemo še na vzorce v oblaku.

Četrto poglavje opisuje predlagane vzorce. V poglavju opišemo tri vzorce, ki smo jih uporabili na praktičnem primeru spletne trgovine. Najprej predstavimo vzorec večnajemništva. Razložimo možnosti podatkovne arhitekture v večnajemništvu, ki je širokega pomena, saj oblak omogoča različne načine shranjevanja podatkov. Prav tako pogledamo načine, kako prepoznati najemnika v aplikaciji. Na koncu poglavja še opišemo naš vzorec večnajemništva, ki omogoča izbiro izolacije podatkov. Naslednji vzorec naslavlja področje spletnih storitev. Prikažemo načina gostovanja spletne storitve v oblaku in razlike med njima. Razložimo dve tehnologiji spletnih storitev, opišemo njune razlike, prednosti in slabosti ter način delovanja. Pomembna je tudi varnost, saj so storitve in aplikacije preko interneta izpostavljene širši množici ljudi. Zaradi tega je priporočljivo zavarovati spletne storitve pred nepooblaščenimi obiskovalci, česar se loteva naslednji vzorec. Spletne storitve morajo biti prav tako razširljive, saj je na trgu vedno več uporabnikov, ki so potencialni uporabniki naših storitev. Nadaljujemo s hranjenjem stanja spletnih storitev, ki se razlikuje od zasnove spletne storitve. Razlikuje se tudi življenjski cikel spletne vloge (angl. "Web Role") in delovne vloge (angl. "Worker Role"), ki gostujeta spletne storitve. Vse to je pomembno, da lahko razložimo naš naslednji vzorec, ki se loteva področja avtentikacije in avtorizacije spletne storitve s pomočjo Azure Active Directory in ogrodja OAuth. Zadnji vzorec naslavlja področje sporočilnih sistemov. Windows Azure podpira sporočilne vrste znotraj storitve Storage in znotraj storitvenega vodila. Poleg sporočilnih vrst omogoča storitveno vodilo še druge načine sporočanja, zato postavimo storitvi skupaj in ju primerjamo. Prav tako pogledamo oba načina komunikacije ter utemeljimo kako bi lahko to združili s sporočilnimi vrstami. V zadnjem vzorcu se lotimo dvosmerne komunikacije s pomočjo sporočilnih vrst storitvenega vodila in komunikacijskih kanalov za povratno sporočanje.

V naslednjem poglavju opisujemo prenos vzorcev na praktičnem primeru. Razvita aplikacija, ki uporablja predlagane vzorce, se izvaja na oblaku Windows Azure. Pri programiranju smo uporabili ogrodje .NET, programski jezik C# in ASP.NET MVC.

Sledi zaključek ter uporabljeni viri in literatura.

2. RAČUNALNIŠTVO V OBLAKU

Preden se lotimo samega pregleda računalništva v oblaku, poizkusimo najprej razjasniti pojem in kmalu bomo ugotovili, da si tudi računalničarji niso enotni glede tega. Kakor je na eni od konferenc povedal Andy Isherwood, takratni podpredsednik HP-ja v Evropi, je veliko ljudi, ki "skačejo na vlak za oblak", vendar pa ni slišal dveh oseb, ki bi povedali istih stvari o računalništvu v oblaku, saj je razlag pojma veliko [3]. Vseeno naštejmo nekaj definicij, s pomočjo katerih bomo poizkusili razložiti pomen računalništva v oblaku.

NIST (National Institute of Standards and Technology) opisuje računalništvo v oblaku kot model za omogočanje omrežnega dostopa do deljenega bazena nastavljivih računalniških virov (na primer omrežja, strežniki, podatkovne shrambe, aplikacij in storitev), katere je lahko hitro pripraviti in sprostiti z minimalnim upravljavskim trudom ali interakcijo s ponudnikom storitev [2].

Gartner opisuje računalništvo v oblaku kot način računalništva, v katerem so, z uporabo internetnih tehnologij, razširljive in elastične IT zmogljivosti dostavljene kot storitev [40].

IBM je opisal računalništvo v oblaku kot rešitev, v kateri so vsi računalniški viri (trda oprema, programska oprema, omrežje, shramba itd.) hitro priskrbljeni uporabniku, ko se pojavijo potrebe po njih. Ključna prednost teh rešitev je, da imajo zmožnost razširjanja in krčenja, tako da uporabniki dobijo vire kot jih potrebujejo in nič več ali manj [41].

V Univerzi Kalifornije, Berkeley, so se opredelili sledeče: Računalništvo v oblaku se nanaša tako na aplikacije, dostavljene kot storitve preko medmrežja, kot tudi na strojno opremo in sistemsko programsko opremo v podatkovnih centrih, ki zagotavljajo te storitve. Strojna oprema v podatkovnih centrih in njena programska oprema je nekaj, čemur lahko pravimo "oblak" [4].

McKinsey & Co. pravijo, da oblaki predstavljajo storitve, ki temeljijo na strojni opremi in ponujajo računske, omrežne in podatkovne kapacitete, kjer je upravljanje strojne opreme visoko abstrahirano pred kupcem. Kupci lahko infrastrukturne stroške izpostavijo kot spremenljive poslovne odhodke (angl. "OPEX"), infrastrukturne kapacitete pa so visoko elastične [17].

Kljub velikemu številu različnih definicij, pa so vsem skupne spodnje značilnosti računalništva v oblaku [1]:

- **Samostojne storitve na zahtevo** (angl. "On-demand Self Service"): kjer se uporaba storitev plačuje po porabi (angl. "Pay-Per-Use").

- **Elastičnost**: kapacitete so elastične in dajejo občutek neskončnih virov, saj je le oblak tako zelo fleksibilen, da omogoča enostavno dodajanje novih virov.
- **Samopostrežni vmesnik**: omogoča, da si nastavimo vire preko storitev, ki jih ponuja ponudnik tehnologije oblaka.
- **Viri**: so abstraktni ali virtualizirani. Viri so deljeni in so dostopni od kjerkoli in kadarkoli.
- **Omrežna povezljivost**: omogoča dostop do rešitev preko različnih naprav (pametni telefoni, tablice, računalniki), prav tako pa je poskrbljeno za varnost s pomočjo požarnih zidov.
- **Merljive**: vse storitve so merljive (od podatkovne shrambe, procesiranja, prenosa podatkov itd.) in se obračunajo po veljavnem ceniku ponudnika.

Vrnimo se še k virom, ki so lahko abstrahirani in virtualizirani in pojasnimo njuna pojma. **Abstrakcija** je pojem, ki opisuje, da so podrobnosti o sistemskih implementacijah skrite pred uporabnikom in razvijalcem. Tako so podatki o lokaciji shranjevanja podatkov neznani, upravljanje sistema pa je prepuščeno zunanjemu izvajanju (angl. "Outsourcing") [12].

Virtualizacija pa opisuje združevanje in deljenje virov. Viri so lahko razširljivi (angl. "Scalable"), omogočeno je večnajemništvo, vsi stroški skupaj pa so ocenjeni na merljivi osnovi [12].

Omenimo še definicijo storitve oblaka. **Storitev oblaka** je kakršen koli produkt, storitev in rešitev, ki je dostavljena in uporabljena v realnem času preko interneta. Računalništvo v oblaku omogoča dostavo teh rešitev v realnem času preko interneta [18]. Infrastruktura pod storitvijo je abstrahirana, sama storitev pa ima možnost razširjanja. Zelo znan primer storitve oblaka je Googlov Gmail. V kolikor želimo razumeti tehnologijo oblaka, je dobro pogledati kako je potekal njen razvoj.

2.1. Zgodovina računalništva v oblaku

Računalništvo v oblaku predstavlja popolnoma novo paradigmo, saj kombinira številne že obstoječe koncepte. Začetki računalništva v oblaku pravzaprav segajo že v petdeseta leta prejšnjega stoletja, ko se je začela uporaba velikih osrednjih računalnikov (angl. "Mainframe"). Do njih so dostopali preko terminalov, kateri niso bili zmožni lastnega procesiranja, temveč so bili namenjeni le za komunikacijo z osrednjim računalnikom. Za učinkovitejšo uporabo osrednjih računalnikov se je nato v šestdesetih letih pojavilo časovno deljenje (angl. "Time-sharing"), ki dovoljuje razvijalcem uporabo deljenih računalniških virov [42].

V začetku šestdesetih let je John McCarthy prišel na idejo služnostnega računalništva (angl. "Utility Computing"), kjer naj bi se računalniška moč oz. aplikacije prodajale skozi enak služnostni poslovni model, kot se voda in elektrika [10]. Takšen model ima nizke ali nikakršne začetne stroške za uporabnika. Kasneje je bila tudi izdana knjiga The Challenge of the Computer Utility, v kateri naj bi avtor Douglas F. Parkhill napovedal, kako se bo računalniška industrija razvila v javno korist, kjer se imajo uporabniki možnost na daljavo povezati z osrednjo računalniško enoto [43]. S tem so bili definirani prvi zametki računalništva v oblaku.

V osemdesetih letih je IBM na trg postavil osebni računalnik in začela se je rast interneta. Cenovna politika telekomunikacijskih omrežij je vplivala na to, da so organizacije podatke in aplikacije hranile in izvajale lokalno, največkrat kar v svojih prostorih. Računalniki so bili povezani točka v točko, kar je tudi zaviralo razvoj tehnologije oblaka [6]. Tako se do devetdesetih let ni zgodilo nič pretresljivega, nakar se je v devetdesetih letih začela bolj natančno oblikovati ideja o računalništvu v oblaku. Ian Foster in Carl Kesselman sta predlagala, da bi moral biti dostop do računalniških virov enak kot pri povezavi v električna omrežja. Računalnike so začeli povezovati v mreže, s čimer se je rodilo mrežno računalništvo (angl. "Grid Computing"), ki je olajšalo razširjanje virov. Zbirka računalniških virov je enotno povezana v internet, do katerih lahko dostopajo uporabniki [43].

Leta 1997 se je prvič uporabila akademska definicija računalništva v oblaku. Ramnath K. Chellappa je v pogovoru na letnem srečanju INFORMS (The Institute for Operations Research and the Management Sciences) dejal, da gre pri računalništvu v oblaku za nov računalniški vzorec, kjer bodo bile meje računalništva odločene glede na ekonomsko načelo, namesto samih tehnoloških omejitev [5]. To je postal tudi osnovni koncept računalništva v oblaku. Podjetja so se počasi začela zavedati novih možnosti in tako je konec dvajsetega stoletja podjetje Salesforce.com postalo prvo podjetje, ki je poslovne aplikacije preko interneta pripeljalo končnim uporabnikom [6].

Po letu 2000 so nato še ostala večja računalniška podjetja začela vlagati v tehnologijo oblaka. Sprva je bilo računalništvo v oblaku osredotočeno na SaaS (angl. "Software-as-a-Service"). Podjetje Amazon je začelo posodabljati svoje podatkovne centre, saj so verjeli, da lahko bolje izkoristijo svoje kapacitete. Sledil je pojav spletnih storitev (angl. "Web Services"), ki je omogočil, da so podjetja na spletu delila svoje storitve. Veliko zasluge pri računalništvu v oblaku je imel ravno Amazon. Leta 2002 so omogočili razvijalcem uporabo svojih aplikacijskih programskih vmesnikov (angl. "Application Programming Interface") [45].

Pomembni mejnik v sodobnem računalništvu je bil leta 2004, ko se je pojavil Splet 2.0. Bistvo spleta 2.0 je bila prav uporaba spletnih storitev. Podjetja so začela izdelovati aplikacije, ki so uporabljale svoje ali tuje javne spletne storitve in so delovale znotraj

spletnega brskalnika. Na spletu 2.0 temelji tudi Amazon Web Services (AWS), ki so ga pri Amazonu leta 2004 predstavili javnosti, leta 2006 pa pognali uradno različico spletnih storitev. Amazon je prav tako leta 2006 predstavil svoj Amazon Elastic Compute Cloud (EC2), ki z naborom ostalih lastnih storitev omogoča posameznikom in podjetjem najem računalniških virov, na katerih lahko poganjajo svoje aplikacije [6]. Amazonov oblak združuje tako infrastrukturo kot storitev (IaaS) kot tudi platformo kot storitev (PaaS). Oboje spada v storitvene modele računalništva v oblaku.

Google je svoj oblak Google App Engine predstavil leta 2008 [8], kateri predstavlja platformo kot storitev. Tudi Microsoft ni želel ostati brez svojega oblaka, zato je leta 2009 predstavil Windows Azure. Naj omenimo, da so tu še IBM, HP in ostali ponudniki storitev oblaka, ki se trudijo pridobiti svoj delež na področju računalništva v oblaku. Veliko podjetij redno prihaja na trg z aplikacijami SaaS. Med bolj uspešnimi podjetji z aplikacijami SaaS sta tudi Salesforce.com in LinkedIn [7]. Na področju računalništva v oblaku ima trenutno Amazon največji tržni delež, kar mu zagotavlja veliko prednost pred tekmeci. Njegov prihod kot prvi na to področje, mu je zagotovo dal prednost [11]. Novosti, ki jih Microsoft redno prinaša v obliki storitev na njihov oblak lahko povzročijo, da ga le-ta dohiti, če ne tudi tehnološko prehiti.

2.2. Modeli računalništva v oblaku

Skupno vsem definicijam računalništva v oblaku so storitve in aplikacije, ki se izvajajo na virtualiziranih virih in na modelu, plačljivem po porabi. Vendar pa je potrebno vedeti, da se storitve in aplikacije lahko namestijo na različne oblake. Za končnega uporabnika ni relevantno kakšen tip oblaka oz. storitveni model (angl. "Service Model") se uporablja in na kakšnem namestitvenem modelu (angl. "Deployment Model") temelji. Se pa s tem ukvarja podjetje, ki razvija rešitev za oblak oz. vodstvo podjetja. Za celotno delovanje oblaka ter vire mora skrbeti ponudnik oblaka. Ponudnik in podjetje, ki uporablja tehnologijo oblaka, skleneta dogovor o ravni storitev (angl. "Service Level Agreement - SLA"), v katerem so specificirana določila, katerih se mora držati ponudnik storitev oblaka.

2.2.1. Storitveni model

Računalništvo v oblaku v grobem delimo na storitvene modele in namestitvene modele. Storitveni modeli opisujejo tip storitve, ki jih ponudnik ponuja in so dosegljive preko platforme v oblaku. Delijo se v tri kategorije: IaaS, PaaS in SaaS. Vsaka oblika storitvenega modela je specificirana za svojo skupino končnih uporabnikov in namenu uporabe. Prav tako se modeli med seboj dopolnjujejo in so pravzaprav zgrajeni en na drugem [12].

2.2.1.1. Infrastruktura kot storitev (IaaS)

Ponudnik tehnologije oblaka zagotavlja polno računalniško infrastrukturo (tj. virtualne računalnike in ostale vire) preko interneta. Prav tako ponudnik preko IaaS-a ponuja strankam mehanizme oz. storitve, ki omogočajo podatkovno shrambo, požarne zidove, javno in zasebno omrežno povezljivost, porazdelitev bremena itd. Storitve, ki spadajo v IaaS, so med drugimi tudi nadzor, popravila in varnost [13].

Poglejmo na IaaS še iz praktičnega vidika. V kolikor se organizacija odloči za strežniško moč, lahko najame virtualizirane strežnike preko ponudnika IaaS. Preko vmesnika sporoči kakšen strežnik želi, nakar se preko procesov sproži zahteva na dejanski strežnik za vzpostavitev virtualnega strežnika. V roku nekaj minut se le-ta tudi zažene in je stranki na voljo za nadaljnjo uporabo. Stranka tako ne potrebuje iti v nakup fizičnega strežnika, ne potrebuje ljudi, ki bi skrbeli za fizične strežnike, niti prostora, kjer bi se strežniki nahajali v organizaciji. Postavitev in zagon strežnika na oblaku je relativno hiter in avtomatiziran proces. Stranka plača uporabo strežnikov po veljavnem ceniku ponudnika, ki pa je največkrat opredeljen po delovnih urah. Seveda pa v ceniku vsaka postavka zaseda svojo mesto, tako nam ponudnik zaračuna podatkovno shrambo, delovne ure storitev, prenos po mreži itd.

Amazon podpira model IaaS in sicer preko AWS omogoča izbiro in postavitev virtualnih računalnikov, katere lahko konfiguriramo podobno kot fizične računalnike. Amazonova storitev za model IaaS se imenuje Amazon EC2.

2.2.1.2. Platforma kot storitev (PaaS)

Model platforme kot storitev opisuje programsko okolje, v katerem programer lahko izdela lastne rešitve znotraj konteksta razvojnih orodij, ki jih ponuja platforma. To pomeni, da programer za izdelavo svojih rešitev uporablja programska orodja in knjižnice, ki jih ponuja platforma [12]. PaaS dandanes temelji na spletnih storitvah in prav tako tudi komponente, ki jih uporablja uporabnik [13]. Za gostovanje uporabniških aplikacij mora ponudnik poskrbeti za strežnike, podatkovne shrambe, omrežja in ostale storitve, preko katerih se lahko uporabniki povezujejo in izvajajo svoje aplikacije. Uporabniku tako ni potrebno skrbeti za trdo opremo ali programsko opremo sistema, temveč le za svojo interakcijo preko aplikacije do sistema. Primer modela PaaS je Google App Engine. Razvijalec napiše svojo aplikacijo v podprtem programskem jeziku in uporabi Google API oz. spletne storitve, ki so navzven odprte javnosti in namenjene uporabi. Problem modela PaaS je, da je razvijalec s svojo rešitvijo vezan na ponudnika storitve oz. platformo, saj mora biti rešitev narejena v programskem jeziku, ki ga platforma podpira [9].

2.2.1.3. Programska oprema kot storitev (SaaS)

V kolikor primerjamo omenjene storitvene modele je SaaS po lestvici najvišje, najnižje pa je IaaS, ki je najbolj odprt in ponuja veliko možnosti pri namestitvi strežnika. Pri modelu SaaS uporabljamo programsko rešitev, ki jo ponuja ponudnik in temelji na trdi opremi, kakor tudi programski opremi. S tem zajema celotno infrastrukturo. Gre torej za abstraktno uporabo računalništva v oblaku. Uporabnik preko interneta dostopa do aplikacije oz. programske opreme, ki je naložena na gostujočih storitvah. Veliko aplikacij SaaS ima navzven odprte aplikacijske programske vmesnike (API-je), preko katerih ponudi razvijalcem možnost, da v svojih lastnih aplikacijah dostopajo do prej omenjenih storitev. Primer modela SaaS je Google Gmail [12].

Vsem aplikacijam SaaS je skupno, da so globalno dosegljive preko interneta, tipično je uporaba vezana na naročnino ali se obračuna po porabi. Aplikacije SaaS pogosto podpirajo večnajemništvo. Programska oprema in storitve so merjene s strani ponudnika, kateri skrbi tudi za avtomatske nadgradnje in posodobitve. V primerjavi z lokalno nameščenimi različicami so za uporabnika največkrat nižji začetni stroški. Vsi uporabniki načeloma uporabljajo enako različico orodja [12].

2.2.2. Namestitveni model

Namestitveni modeli se ločijo na: zasebni, javni, hibridni in t.i. skupnostni (angl. "Community Cloud"). Največjo marketinško pozornost so dobili javni oblaki, v zadnjem času pa se pozornost namenja tudi hibridnim in zasebnim oblakom. Večji ponudniki tehnologije oblaka so začeli pripravljati sisteme za izdelavo zasebnih oblakov oz. storitve za uporabo pri hibridnih oblakih.

2.2.2.1. Zasebni oblak

Veliko podjetij ne želi uporabljati javnega oblaka, saj ne želi izgubiti fizične kontrole nad podatki, zaradi česar so zanje bolj primerni zasebni oblaki.

Zasebni oblaki so narejeni posebej in samo za zasebno rabo v neki organizaciji. Takšni oblaki se izvajajo s pomočjo strežnikov, katere organizacija gostuje znotraj ali zunaj svojih prostorov. Strežnike upravlja lastna organizacija ali tretja oseba [12]. Zaradi tega lahko prihaja do različnih oblik zasebnega oblaka. Podjetja načeloma gradijo zasebne oblake v obliki storitvenega modela IaaS, vendar pa se pojavljajo možnosti uporabe tudi drugih storitvenih modelov, kot jih na primer ponuja Windows Azure Pack (predhodno imenovan Windows Azure Platform Appliances). Gre za manjši in okrnjen nabor Windows Azure

storitev, ki so namenjene večjim podjetjem ob uporabi programske opreme Windows Server [15].

Po besedah podjetja Gartner naj bi IT podjetja zaradi skrbi pred uporabo javnih oblakov, naraščajočo število produktov, ki pomagajo pri izgradnji zasebnih oblakov, in še drugih razlogov, v letu 2014 veliko bolj investirala v zasebno računalništvo v oblaku, kot uporabljala javne oblike. Do leta 2015 pa naj bi se pojavilo veliko število storitev zasebnega oblaka, ki bi, skupaj s storitvami javnega oblaka, tvorili hibridni model [14].

2.2.2.2. Javni oblak

Javni oblak je namenjen javni uporabi in je v lasti ponudnika tehnologije oblaka. Temelji na naročnini in plačilu po uporabi, možen pa je tudi zakup želene količine. Viri so deljeni in virtualizirani, oblak pa temelji na večnajemniškem modelu. Storitve so dosegljive uporabniku le preko interneta. Prednosti javnega oblaka so tudi nizek začetni investicijski strošek za novega uporabnika, saj za vse vire poskrbi ponudnik. Prav tako je omogočena razširljivost (skalabilnost).

Trenutno najbolj znani primeri javnega oblaka so Amazon AWS, Google AppEngine in Windows Azure.

2.2.2.3. Hibridni oblak

Hibridni oblak združuje več oblakov, ki so lahko javni, privatni in skupnostni, vendar so med seboj vezani kot enolična enota [12]. Omogoča razširitev kapacitet ali zmožnosti storitev enega oblaka z integracijo storitev drugega oblaka. Prav tako omogoča, da še zmeraj ohranimo zasebnost, medtem ko dodamo kapacitete in nove zmožnosti.

Hibridni oblaki so lahko po svoji kompoziciji [16]:

- statični (zgrajeni za uporabo več storitev),
- sestavljeni ob namestitvi oz. uporabi (menjajoča uporaba storitev enega ponudnika ali drugega oz. združevanje glede na politiko) ter
- dinamično sestavljeni.

Dinamično sestavljeni hibridni oblaki omogočajo organizacijam, da izkoristijo razširitev oblaka (angl. "Cloud Bursting"). Razširitev oblaka pomeni, da oblak, ki je sicer zasebni, ob visokih konicah, kadar je potrebna dodatna zmogljivostna moč, naredi prehod na javni oblak in izkoristi njegove vire. Prednost razširitev oblaka je v tem, da podjetja plačujejo le za dejansko porabljene vire v javnem oblaku. Po besedah organizacije Gartner naj zasebni oblaki ne bi za dolgo ostali zasebni, ampak bodo postali hibridni [16].

2.2.2.4. Skupnostni oblak

Pri skupnostnem oblaku imamo večnajemniško infrastrukturo, ki je deljena med več organizacij. Te si delijo podobne potrebe oz. cilje glede varnosti, zasebnosti, zahtev in izvedbe. Uporabljene so prvine zasebnega oblaka, saj je bolj poskrbljeno za varnost in zasebnost, kakor tudi javnega oblaka, od koder pride večnajemniški model in plačilo po porabi. Vsekakor pa se s skupnim deljenjem oblaka porazdelijo stroški uporabe [12].

Glede varnosti je potrebno še omeniti, da si pri uporabi skupnostnega oblaka vse organizacije med seboj delijo skupno hrambo podatkov, zato je nujno, da se tega posamezniki v organizaciji zavedajo in v kolikor je to potrebno, pazijo katere podatke shranjujejo v oblak, saj se lahko zgodi, da te podatke vidijo v drugi organizaciji.

2.3. Ponudniki storitev oblaka

2.3.1. Amazon

Amazon je trenutno vodilno podjetje na področju računalništva v oblaku. Leta 2006 je bil Amazon prvi, ki je javnosti predstavil in ponudil model IaaS. Najbolj prepoznavne so njegove spletne storitve oz. AWS. AWS temelji na storitveno usmerjeni arhitekturi (SOA) in je zelo dober primer, kaj vse se da narediti s takšno tehnologijo. Vsa dokumentacija je objavljena na spletni strani podjetja Amazon, kjer imajo podrobno opisano in s primeri prikazano uporabo njihovih spletnih storitev [20].

AWS vključuje veliko storitev, ki so enostavno dosegljive uporabniku preko spletnega brskalnika ali API-jev. Nekaj izmed ključnih je opisanih v nadaljevanju [12]:

- **Elastic Compute Cloud** (EC2): namenjen je izgradnji, uporabi in upravljanju virtualnih strežnikov. Uporabnik lahko enostavno in hitro namesti virtualni strežnik iz že vnaprej pripravljenih slik. V okrilje te storitve spadata **Simple Queue Service** (SQS), ki predstavlja sporočilne vrste in transakcijski sistem za porazdeljene spletne aplikacije ter **Simple Notification Service** (SNS), kateri omogoča objavljanje sporočil iz aplikacije in dostavo sporočil drugim aplikacijam in naročnikom. EC2 lahko nadziramo in spremljamo s pomočjo storitve **CloudWatch**. Z njeno pomočjo lahko vidimo uporabo procesorske moči, omrežja, diskovnih enot ipd. S pomočjo storitve **Elastic Load Balancing** pa lahko skrbimo za porazdelitev bremena.
- **Simple Storage System** (S3): sistem za hrambo podatkov in rezervne hrambe podatkov.
- **Elastic Block Store** (EBS): uporablja se za izdelavo navideznih diskov ali shrambo zvezkov (angl. "Volumes") na blokovni ravni. Ti zvezki se lahko, z nekaterimi omejitvami, kadarkoli pripnejo delujočim instancam.

- **SimpleDB**: strukturirana hramba podatkov, ki podpira indeksiranje in podatkovne poizvedbe na EC2 in S3.
- **Relational Database Service** (RDS): dovoli izgradnjo relacijske podatkovne baze.
- **CloudFront**: sistem, ki skrbi za hranjenje in dostavo podatkov na različnih fizičnih lokacijah, tako da lahko uporabniki hitreje dobijo vrnjene podatke. Podatki na teh lokacijah so predpomnjeni in se morajo osveževati.

2.3.2. Google

Google je najbolj znan po svojem spletnem iskalniku. Poleg iskalnika ima Google veliko spletnih storitev in programskih vmesnikov, katere je v veliki meri naredil odprte za javnost. Leta 2008 je Google javnosti ponudil Google App Engine (GAE). Gre za storitveni model PaaS, kjer lahko kdorkoli na Googlovo infrastrukturo namesti spletno aplikacijo, pri čemer mora ustrezati njihovim standardom [12]. Prav tako trenutno podpira le aplikacije narejene s programskim jezikom Java, Python, PHP ali Go [8].

Google na svoji spletni strani zagotavlja, da se aplikacije na njihovem oblaku izvajajo zanesljivo tudi pod velikimi obremenitvami in z velikimi količinami podatkov. Prav tako poudarjajo značilnosti njihovega oblaka, ki naj bi bile [8]:

- Shramba podatkov, vključno s poizvedbami, razvrščanjem in transakcijami.
- Samodejno razširjanje in porazdelitev bremena.
- Asinhrone vrste z nalogami.
- Načrtovana opravila, ki sprožijo dogodke ob določenem času oz. intervalih.
- Integracija z ostalimi Googlovimi storitvami oblaka in vmesniki.

2.3.3. Microsoft

Microsoft je šele konec leta 2009 javnosti ponudil svojo tehnologijo oblaka, imenovano Windows Azure. Storitve oblaka Windows Azure so tako narave storitvenega modela IaaS kot tudi PaaS. Storitve znotraj oblaka Windows Azure se redno dodajajo, zato ni čudno, da je še do nedavnega Windows Azure veljal le za storitveni model PaaS.

Microsoft opisuje Windows Azure kot vsestransko množico storitev, ki omogočajo uporabniku hitro izgradnjo, namestitev in upravljanje aplikacij čez globalno omrežje Microsoftovih podatkovnih centrov. Microsoft ima trenutno 8 večjih podatkovnih centrov in 24 robnih lokacij Content Delivery Network (CDN) po svetu. Nam najbližji večji podatkovni centri se nahajajo na Nizozemskem ter na Irskem.

Prav tako so, zaradi bežanja uporabnikov h konkurenci, pri Microsoftu naredili velik korak naprej in omogočili uporabo drugih programskih jezikov za izgradnjo spletnih aplikacij. Tako

sedaj Windows Azure omogoča, poleg programskih jezikov znotraj ogrodja .NET, tudi izgradnjo aplikacij s programskim jezikom Java, PHP, Node.JS in Python. Odprtost do prostokodnih projektov je velik korak naprej za podjetje Microsoft, s čimer želijo privabiti nove uporabnike. Prav tako lahko uporabniki izbirajo strežnike z operacijskim sistemom Windows Server ali katerim drugim operacijskim sistemom, ki temelji na Linux-u. Za gradnjo aplikacij v ogrodju .NET potrebujemo programsko orodje Visual Studio 2010 ali novejši ter Windows Azure SDK za .NET [21]. SDK je možno namestiti tudi preko aplikacije Microsoft Web Platform Installer, ki poskrbi, da se zraven namestijo še vsi potrebni paketi.

V nadaljevanju si bomo pogledali storitve, ki jih ponuja Windows Azure. Te se delijo na računske (angl. "Compute"), podatkovne in aplikacijske storitve ter omrežje. **Virtual Machines** spada pod storitve Compute in omogoča uporabniku razširljivo infrastrukturo. Na voljo je tako Windows Server kot tudi operacijski sistemi Linux. Windows Azure dovoljuje uporabniku spremljanje razpoložljivosti, nastavljanje opozoril za določene dogodke in avtomatsko razširjanje ob potrebi [21].

Web Sites storitev prav tako spada pod računske storitve in je namenjena hitri namestitvi spletnih strani v oblak. Uporabniku se ni potrebno ukvarjati z infrastrukturo. Spletne strani dobijo svojo domeno, preko katere lahko dostopamo do strani, prav tako pa jo lahko povežemo z lastno domeno preko zapisa A ali CNAME. Spletna stran je razširljiva tako vertikalno (angl. "Scale-Up") kot tudi horizontalno (angl. "Scale-Out"). Windows Azure ponuja nabor odprtokodnih aplikacij in ogrodij, iz katerih si lahko uporabnik izbere želeno aplikacijo ter jo enostavno namesti na oblak. Med drugimi lahko izbira tudi med sistemi CMS, kot sta Wordpress, Drupal ipd. Možna je tudi izbira podatkovne baze, kjer lahko izbiramo med uporabo podatkovne baze Azure SQL ali MySQL [21].

Mobile Services spadajo pod storitve Compute in omogoča pospešitev izgradnje mobilnih aplikacij z enostavnimi načini za shranjevanje v podatkovne shrambe, overovitvijo uporabnikov in pošiljanje potisnih sporočil. S pomočjo razvijalskih orodij in API-jev lahko razvijemo aplikacije za Windows, Android, iOS. Prav tako lahko uporabimo že uveljavljene pristope za avtenticiranje uporabnika s pomočjo storitev omrežij Facebook, Twitter, Microsoft ali Google, poskrbljeno pa je tudi za dostop do lastnih podatkovnih strežnikov [21].

Cloud Services so glavni del storitev Compute in so namenjene za namestitev in upravljanje zmogljivih aplikacij, ki se namestijo na virtualne strežnike. Ob izgradnji rešitve določimo projektu spletno ali delovno vlogo. Spletna vloga je virtualni strežnik, ki poganja Microsoftov spletni strežnik IIS, kateri se odziva na spletne zahteve. Delovna vloga načeloma nima spletnega strežnika IIS, ga pa lahko uporabnik ročno vklopi, v kolikor oceni, da je to potrebno. Do virtualnih strežnikov, ki poganjajo različne vloge, je možen dostop na daljavo, nastavitve pa se lahko izvedejo tudi preko upravljavskega portala Windows Azure [21].

Azure SQL Database omogoča storitve relacijske podatkovne baze v oblaku. Temelji na Microsoftovemu strežniku SQL, vendar je prilagojen za oblak. Trenutno se ponujata dve vrsti storitev: spletna in poslovna različica. Velikost podatkovne baze poslovne različice je trenutno lahko do 150 GB. Povezovanje na podatkovno bazo je omogočeno preko upravljavskega portala Windows Azure, preko programa SQL Server Management Studio, preko REST API-jev in PowerShell-a. Omogočeno je tudi razširjanje podatkovne baze navzven, kar v tem primeru pomeni, da se ob konicah delovnega bremena (angl. "Peak Workloads") dodajajo nova vozlišča (porazdeljene podatkovne baze) obstoječim [21].

Azure Storage predstavlja dodatno shrambo, katere se lahko poslužujemo pri implementaciji aplikacij za Windows Azure. Uporablja se preko spletnih storitev tipa REST, tako da jih je mogoče uporabiti kadarkoli in kjerkoli na medmrežju. Dostop do podatkov je mogoč tudi preko HTTPS. Za dostop do podatkov se potrebuje 512 bitni asimetrični ključ, ki ga lahko preko spletnega vmesnika ponovno generiramo. Dostope do podatkov lahko prav tako omejimo s pomočjo podpisa deljenega dostopa (angl. "Shared Access Signatures"). Azure Storage podpira različne oblike shranjevanja podatkov [21]:

- **Blobi** (angl. "Blobs"): predstavljajo datoteke in njihove metapodatke. Shranjeni so v zabojnikih (angl. "Container"), katere ima lahko vsak uporabnik oblaka neomejeno. Vsak zabojnik vsebuje množico blobov. Nad zabojnikom in blobi lahko nastavimo pravila za dostop. Blobi se delijo na blokovni blob (angl. "Block Blob") in na blob strani (angl. "Page Blob"). **Blokovni blob** je namenjen učinkovitejšemu pretakanju (angl. "Streaming") večjega delovnega bremena (angl. "Workloads"). Vsak blokovni blob je sestavljen iz sekvence blokov. Velikost blokovnega bloba je omejena na 200 GB. **Blobi strani** so namenjeni naključnim bralno/pisalnim operacijam. Vsak blob strani je sestavljen iz tabele strani. Vsaka stran pa je opisana z odmikom od začetka bloba. Največja velikost bloba strani je 1 TB.
- **Diski** (angl. "Drives"): temeljijo na blobih in so odporni zvezki NTFS, ki jih lahko pripnemo in uporabljamo pri Windows Azure aplikacijah.
- **Tabele** (angl. "Tables"): so strukturirana podatkovna shramba. Vsebuje množico entitet, ta pa množico lastnosti. Gre za t.i. podatkovno shrambo tipa NoSQL.
- **Vrste** (angl. "Queues"): se uporabljajo kot zanesljiva shramba in za dostavo sporočil aplikacijam.

2.4. Cenovni obračun storitev oblaka

Uporaba storitev v oblaku je v večini primerov plačljiva. Vsak ponudnik storitev oblaka ima svojo ceno storitev, v zameno za plačilo pa se mora držati dogovorov zapisanih v pogodbi o ravni storitev. Lahko se zgodi, da je potrebno za uporabo javnega oblaka plačati več, kot bi, če bi se uporabljal zasebni oblak ali lastniška programska in strojna oprema. Ravno to je razlog, da je potrebno pregledati in analizirati stroške in jih primerjati s prednostmi, ki pridejo

z uporabo storitev javnega oblaka. Rezultate nato lahko primerjamo s stroški pri uporabi alternativ. Potrebno pa je vzeti v zakup vse prednosti oblaka, na katere velikokrat niti ne pomislimo ob prvem računanju stroškov, kot so: plačilo elektrike, generatorji v primeru izpada elektrike, internetna povezava, najemnina za prostor, plačilo osebja za vzdrževanje in tako dalje. V tem poglavju se bomo držali zaračunavanja uporabe storitev na primeru Windows Azure.

Microsoft omogoča naročnino na oblak in zaračunava uporabo storitev po porabi. To omogoča naročnino brez daljših vezav, plačilo dejanske uporabe storitev in možnost prekinitve sodelovanja v vsakem trenutku. Kljub temu Microsoft omogoča predplačniške pakete, preko katerih lahko naročnik, odvisno od dolžine vezave, konec meseca privarčuje določeno vsoto. Plačilo je v glavnem možno preko kreditne kartice. Obstaja tudi možnost plačila preko položnice, vendar je za to potrebna odobritev s strani Microsofta. Kljub vsem plačljivim storitvam, Microsoft omogoča brezplačno uporabo nekaterih njihovih storitev. V času raziskave, sta bili med brezplačni tudi uporaba storitev aktivnega imenika in uporaba spletnih strani [46].

Vse storitve imajo za svojo uporabo lastne cenike. Cenik se prav tako razlikuje med regijami, kjer gostujemo oz. uporabljamo storitve v oblaku. Takšno razliko lahko hitro opazimo že pri uporabi virtualnih strežnikov v različnih regijah (npr. Severna Evropa in Zahodna Evropa), kjer se cene storitev razlikujejo od skoraj enega centa naprej [46]. Sprva majhna vsota se konec meseca lahko spremeni v razliko nekaj evrov. Pri uporabi več instanc virtualnih strežnikov ali v kombinaciji še z drugimi storitvami, je ta vsota lahko hitro vse drugo kot zanemarljiva.

Storitve se ločijo glede na velikost instance, kapaciteto shrambe, programske opreme na virtualnem strežniku ipd. Tako je več kot primerno, da si uporabniki okvirno preračunajo, koliko jih bo konec meseca stala uporabnina oblaka. Prav tako je primerno, da uporabnik, ki ima oblak le za testiranje, svoje instance ugasne ob neuporabi in ob tem privarčuje. Poleg storitev oblaka se obračuna tudi podpora, v kolikor jo imamo naročeno, ter količina podatkov (angl. "Bandwidth"), ki se prenaša ob uporabi storitev.

Microsoft ima na svoji spletni strani za obračun uporabe storitev oblaka Windows Azure [46] prav tako kalkulator, s katerim olajša izračun uporabe storitev. Tabela 1 prikazuje nekaj najbolj pogostih storitev in njihove cene, povzetih iz omenjene spletne strani. Temu sledi še izračun uporabe storitev oblaka na primeru spletne trgovine. Cene se s spreminjanjem instanc spreminjajo, saj to največkrat pomeni več jedrov, pomnilnika ali prostora za shranjevanje. Izračunane cene se nanašajo na standardne instance in so cene uporabe storitve na uro, razen če ni drugače navedeno.

Virtualni strežnik	0,068 €
Operacijski sistemom Windows, enojedrni 1.6 GHz procesor in 1.75 GB pomnilnika.	
Storitve Compute	0,06 €
Enojedrni 1.6 GHz procesor, 1.75 GB pomnilnika, 225 GB prostora za shrambo.	
Podatkovna baza SQL	7,44 € za prvi GB, nato 2,976 € naslednji
Spletna različica do 5 GB (poslovna različica do 150 GB – cena se spremeni po 10 GB).	
Azure Storage storitve	0,004 € za vsakih 100.000 transakcij
Tabele in vrste	prvi 1 TB/mesec: 0,053 € na GB.
Blobi strani in diski	do 1000 TB/mesec: 0,038 € na GB.
Blokovni blobi	do 1000 TB/mesec: 0,018 € na GB.
Transakcija je vsaka bralno pisalna operacija preko vseh tipov shrambe podatkov.	
Storitveno vodilo	0,01 €
Za vsakih 10.000 sporočil v sporočilnih vrstah in/ali temah.	
Prenosi podatkov	Obračunajo se odhodni in vhodni prenosi:
Odhodni prenosi podatkov	prvih 5 GB/mesec brezplačnih, nato 0,09 € na GB do porabljenih 10 TB na mesec.
Vhodni prenosi podatkov	Brezplačno

Tabela 1: Cenik izbranih storitev Windows Azure.

Za informativni izračun vzemimo primer spletne trgovine, ki je pripravljena na prenos v oblak. Uporablja spletno in delovno vlogo, podatkovno bazo SQL, Azure Table in blokovne blobe ter storitveno vodilo za vrste. Poleg vsega je potrebno prišteti še podatkovne prenose in transakcije. Računali bomo porabo na mesečni ravni, za mesec s 30 dnevi. Podatki so informativne narave, točne izračune je potrebno narediti za vsako aplikacijo posebej. Cene so vzete za podatkovni center Severne Evrope.

- **Storitve Compute**

Spletno in delovno vlogo bomo namestili znotraj ene instance računske storitve, kjer bosta imeli vsaka svojo vlogo. Spletna stran bo obratovala vsak dan, 24 h/dan, kar tudi uporabimo pri izračunu, kot to prikazuje naslednja tabela (tabela 2).

Št. storitev	Št. ur/mesec	Cena v €/uro	Št. instanc	Skupaj v €/mesec
1	720 ur/mesec	0,06 €/uro	2	86,4 €/mesec

Tabela 2: Primer izračuna storitev Compute.

- **Podatkovna baza SQL**

Ocenjujemo, da bo potrebna podatkovna baza od 5-10 GB velikosti. Možna je kasnejša razširitev. Tabela 3 prikazuje izračun za 10 GB velikosti.

Št. podatk. baz	Velikost v GB/mesec	Cena v € za GB/mesec	Skupaj v €/mesec
1	1 GB/mesec	7,44 €/mesec	7,44 €/mesec
	9 GB/mesec	2,976 €/mesec	26,784 €/mesec

Tabela 3: Primer izračuna storitev podatkovne baze SQL.

- **Azure Storage storitve**

Azure Table se uporablja za dodatno shranjevanje informacij, katere ne shranjujemo v podatkovno bazo SQL. Uporabljamo shrambo z lokalno redundanco. V ta namen bo dovolj 1 GB prostora v tabelah, kar tudi vzamemo za izračun, kot je razvidno iz naslednje tabele (tabela 4). Blokovni blobi so namenjeni za shranjevanje slik in ostalih dokumentov.

Ime storitve	Velikost v GB/mesec	Cena v €/GB	Skupaj v €/mesec
Blokovni blob	5 GB/mesec	0,018 €/GB	0,09 €/mesec
Tabele	1 GB/mesec	0,053 €/GB	0,053 €/mesec

Tabela 4: Primer izračuna storitev Storage.

- **Storitveno vodilo**

Aplikacija za obveščanje o naročilih uporablja storitveno vodilo. Tabela 5 prikazuje izračun uporabe storitev storitvenega vodila za primer 20 naročil na dan.

Ime storitve	Št. transakcij/mesec	Cena v €/10.000 transakcij	Skupaj v €/mesec
Vrste	600 transakcij/mesec	0,01 €/10.000 transakcij	0,01 €/mesec

Tabela 5: Primer izračuna storitev storitvenega vodila.

- **Prenosi podatkov**

Za primer vzemimo, da ima spletna trgovina v svojem naboru približno 3000 artiklov ter 200 do 300 obiskovalcev na dan. Preko medmrežja se prenese mesečno okoli 4-6 GB podatkov. Aplikacija je nameščena v evropskem podatkovnem centru, ki trenutno spada v cono 1. Tabela 6 vsebuje izračun vhodnih in izhodnih prenosov podatkov za omenjeni primer.

Ime storitve	Skupaj v GB/mesec	Cena v €/mesec	Skupaj v €/mesec
Vhodni prenosi	6 GB/mesec	Brezplačno	0 €/mesec
Izhodni prenosi	5 GB/mesec	Brezplačno	0 €/mesec
	1 GB/mesec	0,09 €	0,09 €/mesec

Tabela 6: Primer izračuna prenosa podatkov.

- **Skupaj stroški na mesec**

Vse zgoraj naštete stroške je potrebno sešteti in dobili bomo mesečne stroške za uporabo storitev oblaka, kot je to prikazano v spodnji tabeli (tabela 7). Skupaj na letni ravni znese omenjena uporaba 1.450,40 €.

Ime storitve	Skupaj v €/mesec
Storitve Compute	86,400 €
Podatkovna baza SQL	34,224 €
Azure Storage	0,143 €
Storitveno vodilo	0,010 €
Prenosi podatkov	0,090 €
SKUPAJ / mesec	**120,867 €**

Tabela 7: Primer skupnega obračuna uporabe storitev.

Kot že omenjeno, je pri načrtovanju potrebno narediti cenovni izračun porabe storitev in primerjati z ostalimi alternativami. V ceno za lokalne strežnike in lastno programsko opremo je potrebno računati še ljudi za vzdrževanje, licence za programsko opremo, elektriko in ostalo. Prav tako lahko za potrebe zmanjševanja stroškov celotno podatkovno bazo prenesemo v Azure Table, vendar pa moramo računati na spremembo logike programskih sistemov. Primernost tehnologije oblaka se pokaže ob povišanih obremenitvah aplikacij z zahtevki oz. povečanem obisku strani. Oblak omogoča, da se avtomatsko razširi kapacitete za določen čas, torej ob povišani obremenitvi strežnika. V kolikor omogočimo to možnost, je potrebno računati, da se bodo ob povišani obremenitvi vklopile nove instance strežnikov oz. storitev, kar bo prav tako vplivalo na ceno. Včasih je vseeno bolje zagotoviti boljšo uporabniško izkušnjo oz. hitrejše procesiranje, kar lahko poviša stroške, kot pa izgubo strank, prometa ipd.

2.5. Testiranje aplikacij za oblak

Aplikacije, specifične za oblak, se v določeni meri ločijo od aplikacij, ki smo jih bili vajeni do sedaj. Arhitektura oblaka namreč omogoča, da se storitve, na katerih je nameščena aplikacija, in ostale storitve, do katerih le-ta dostopa, po potrebi razširjajo in krčijo. V ta namen je potrebno izvesti testiranja aplikacije, s katerim preverimo njeno obnašanje v oblaku in pravilnost delovanja. Testiranje nam omogoči, da preverimo, ali se aplikacija obnaša tako, kot smo to predvideli. Poleg testiranja uporabniškega vmesnika aplikacije (angl. "User Interface Testing") obstajajo še drugi testi, kot so na primer testiranje enot (angl. "Unit Testing"), testiranje zmogljivosti (angl. "Performance Testing") ipd. Priporočljivo je, da se zmogljivost aplikacije preveri s testom zmogljivosti, ki omogoča testiranje primerov, kateri so aktualni tudi v oblaku. Takšni primeri testiranja so: obremenitveni testi (angl. "Load Test"), testiranje obremenitvenih konic (angl. "Peak Test"), testiranje vzdržljivosti (angl. "Endurance Test"),

stresni testi (angl. "Stress Test") in podobni [49]. Testiranje v oblaku je malo bolj specifično kot lokalno testiranje, saj se pri testiranju zmogljivosti lahko zaganjajo nove instance, poveča pretok podatkov ipd., kar tudi poveča ceno uporabe storitev. Poleg tega se lahko zgodi, da se rezultati razlikujejo med lokalnim testiranjem in testiranjem v oblaku, zaradi razlik pri delovanju komponent razvojnega orodja [22]. Na oblak lahko gledamo tudi kot popolno orodje za testiranje aplikacij. Omogoča nam neomejene vire, s katerimi lahko enostavno in temeljito preverimo delovanje lastnih aplikacij ob različnih scenarijih. Tako lahko preverimo delovanje ob povišani ali nenadnih obremenitvah, ki so še posebej aktualni v sedanjem času, ko se povečuje število uporabnikov preko pametnih mobilnih naprav. Postopek za izvedbo kvalitetnih testov ni enostaven, saj je potrebno že v začetku narediti načrte za teste. Lahko se namreč zgodi, da izvedeni test ne testira želenih funkcionalnosti in s tem zgrešimo bistvo testa.

Windows Azure omogoča lokalni posnemovalnik (angl. "Emulator"), katerega uporabimo v fazi, ko storitve in aplikacije razvijamo ter testiramo lokalno. Kljub temu, da določenih storitev ne more posnemati, omogoča možnost posnemanja storitev Compute in storitev Storage. Za simulacijo ostalih storitev lahko uporabimo podatkovno bazo SQL Server in storitveno vodilo za operacijski sistem Windows. Prav tako je sprva priporočljivo aplikacijo testirati lokalno, s čimer se izognemo nepotrebnemu povišanju stroškov pri uporabi storitev v oblaku in večkratni namestitvi programske kode v oblak. V veliko pomoč pri testiranju aplikacij in storitev nam bo razvojno orodje Visual Studio 2013 Ultimate, oz. druge različice, ki podpirajo možnost izvedbe testov. Omenjena različica razvojnega orodja omogoča izdelavo in izvedbo obremenitvenih testov, testiranje enot, test spletnih zmogljivosti in še marsikaj. Za izgradnjo testov je potrebno, poleg obstoječih projektov v naši programski rešitvi, narediti nove projekte, kateri se bodo uporabljali za testiranje. Ob izgradnji novega testnega projekta izberemo želeni tip testa in ga zgradimo s pomočjo čarovnika. Čarovnik za izgradnjo testa nam ponudi veliko možnosti nastavitev le-tega. V primeru obremenitvenega testa lahko nastavimo vzorec bremena, ki definira želeno število uporabnikov za simulacijo. Prav tako lahko nastavimo mešani simulacijski model, kateri omogoča mešano izvajanje testnih projektov. Omogoča tudi porazdelitev bremena preko različnega tipa omrežja (LAN, 3G, modem itd.) in izbiro brskalnikov. Simulaciji nastavimo tudi čas trajanja oz. število iteracij. Test poženemo preko razvojnega orodja Visual Studio, kateri nam med simulacijo in po končanem delu prikaže rezultate testiranja, katere prav tako lahko vidimo v grafični obliki. Spremljanje delovanja storitev v oblaku je možno preko upravljavskega portala Azure Management Portal. Večinoma imajo vse storitve na voljo zavihek "Monitor", v kateri je grafično prikazano delovanje storitve. Dnevnik dogodkov ter napake, ki se zabeležijo med izvajanjem storitev, pa so shranjene v tabelah storitve Azure Table Storage, pod imenom `WADWindowsEventLogsTable` in `WADLogsTable`. V kolikor želimo do diagnostičnih storitev dostopati programsko, lahko to storimo preko API-jev ali s pomočjo programskega modela storitve Windows Azure Diagnostics.

3. NAČRTOVALSKI IN ARHITEKTURNI VZORCI

Ideja o načrtovalskih vzorcih, kot jih poznamo dandanes, sega vse do arhitekta in profesorja Christopherja Alexandra, ki je razvil teorijo o arhitekturi, gradnji in planiranju, katera temelji na razvoju in uporabi vzorcev. Pokazal je, kako je mogoče prenesti vzorce na gradnjo hiš in celo na celotna mesta [24]. Vzorce je uredil in združil v zbirko, ki jo je označil za zaporedje (angl. "Sequence"). Njegovo delo daje vpogled v to, kako naj bi bili vzorci v splošnem organizirani in strukturirani. Med drugimi je v njegovem delu zapisano, da naj imajo zaporedja vzorcev dodano vrednost, saj bistvo združevanja vzorcev ni logično organiziranje podobnih vzorcev, temveč prikaz preizkušenih procesov, ki imajo lastno vrednost. Prav tako ni potrebno, da so vzorci normalizirani, saj lahko več vzorcev z različnimi rešitvami rešuje isti problem [53].

Načrtovalski vzorci so na računalniškem področju dobili veliko pozornosti po izdaji knjige Design Patterns: Element of Reusable Object-Oriented Software, ki je bila izdana leta 1994. Štirje avtorji knjige (Gamma, Helm, Johnson in Vlissides) so dobili naziv kar Gang of Four [19], krajše GoF. V knjigi so prvič opisali načrtovalske vzorce, ki so osredotočeni na objektno usmerjen načrtovalski problem. Opisano je, kdaj se vzorec uporablja ter kakšne so njegove posledice oz. kompromisi za dosego cilja. Prav tako so v knjigo vključili nekatere primere implementacij vzorcev s pomočjo programskega jezika C++ in Smalltalk [19].

Kmalu po izidu zgornje knjige, je bila leta 1994 prirejena tudi prva konferenca na temo vzorcev na področju programiranja. Naslednje leto so tudi ustanovili Portland Pattern Repository, ki služi kot hramba programerskih načrtovalskih vzorcev. Vzorci so se nato izdajali v katalogih vzorcev, primer katerih je tudi knjiga avtorjev GoF, ali pa na medmrežju. Zanimanje za načrtovalske vzorce se je v zadnjih letih začelo povečevati [47]. Zanje je veliko zanimanja v znanstvenem področju, kjer se trudijo oceniti vpliv vzorca na kvaliteto programske opreme, njihovo uporabo, primernost ipd.

Vzorci torej rešujejo načrtovalske probleme in temeljijo na dolgoletnih izkušnjah načrtovalcev in razvijalcev. Zajemajo obstoječe dobre izkušnje na področju razvoja programske opreme in pomagajo podpirati dobro razvojno prakso. Vsak vzorec rešuje specifičen problem pri izgradnji programske opreme. V zgoraj omenjeno knjigo so avtorji vključili katalog načrtovalskih vzorcev. Katalog vsebuje 23 načrtovalskih vzorcev, ti pa so razdeljeni glede na dva kriterija. Prvi kriterij je namen (angl. "Purpose") in opisuje kaj vzorec dela. Tako ima lahko vzorec ustvarjalni, strukturni ali vedenjski namen. Drugi kriterij pa je območje delovanja (angl. "Scope") in nam poda informacijo ali se vzorec nanaša na razred (angl. "Class") ali na objekt (angl. "Object") [19]. Omenimo, da je možno preko namenskih orodij iz obstoječe programske kode tudi pridobiti uporabljene načrtovalske vzorce. Gre za obratni postopek. Rezultate lahko uporabimo npr. pri gradnji ponavljajočih metod [25].

3.1. Dokumentiranje vzorcev

V splošnem ni standardizirane oblike za dokumentacijo vzorcev. Najbolj sprejeta je oblika, kot jo avtorji GoF predlagajo v svoji knjigi. V grobem so načrtovalski vzorci sestavljeni iz štirih osnovnih elementov [19]:

- **Ime vzorca**: uporabljamo ga, da na kratko opišemo kakšnega problema se lotevamo oz. kako ga rešimo. Ime vzorca prav tako povečuje načrtovalski besedni zaklad, ki nam pomaga, da se lahko o vzorcih razumevajoče pogovarjamo in pišemo.
- **Problem**: opisuje kdaj uporabiti vzorec. Opisani so lahko različni načrtovalski problemi. Lahko se zgodi, da je za rešitev problema potrebno najprej zadostiti pogojem, preden lahko uporabimo vzorec.
- **Rešitev problema**: opisuje elemente, ki naredijo načrt ter njihove medsebojne relacije in sodelovanja. Rešitev ne opiše dejanske implementacije, temveč služi bolj kot abstraktni opis oz. predloga, ki jo lahko uporabimo.
- **Posledice**: predstavlja rezultate uporabe vzorca. Posledice in kompromisi so ključni pri odločanju glede uporabe vzorca. Moramo se namreč zavedati posledic, stroškov ter koristi vzorca, preden se odločimo zanj.

Za kakovosten opis vzorca pa je potrebno podati še druge informacije. Kljub temu, da bi v določenih primerih k boljšemu razumevanju zadostovala že slika vzorca, je le-tega vseeno potrebno opisati z besedami. Slike so namreč namenjene grafom za prikazovanje relacij med objekti in razredi. Za opis vzorca se je dobro držati uveljavljenega formata [19]:

- **Ime vzorca in klasifikacija**: ime opisuje vzorec ter njegov namen. Poleg tega tudi bogati besedni zaklad. Klasifikacija se uporablja za razvrščanje vzorca, s čimer določimo prav tako namen vzorca.
- **Namen**: krajši sestavek o namenu vzorca, kaj počne in kakšen problem rešuje.
- **Prav tako poznano kot**: v kolikor obstajajo še kakšna druga poznana imena za isti vzorec, se jih zabeleži pod to točko.
- **Motivacija**: za razliko od abstraktnega opisa vzorca, gre tu za bolj razumljivo opisan dogodek, lahko tudi na primeru, ki pojasnjuje načrtovalski problem in kako vzorec s svojo strukturo rešuje omenjeni problem.
- **Uporaba**: kdaj oz. v kakšnih primerih se uporablja načrtovalski vzorec.
- **Struktura**: grafična predstavitev razredov v vzorcu.
- **Udeleženci**: razredi in/ali objekti, ki se uporabljajo pri vzorcu ter njihove obveznosti.
- **Sodelovanja**: kako udeleženci sodelujejo med sabo in izvajajo svoje obveznosti.
- **Posledice**: razložimo rezultate in kompromise uporabe vzorca.
- **Implementacija**: opiše se nasvete, pasti in pomagala, katerih se je dobro zavedati pri uporabi vzorca.
- **Vzorčna koda**: deli programske kode, ki prikazuje, kako se vzorec uporabi.

- **Znani primeri uporabe**: primeri uporabe vzorca v realnem življenju. Vključi se vsaj dva primera različnih domen.
- **Sorodni vzorci**: opišemo kateri vzorci so povezani z dotičnim, kakšne so njune razlike ter s katerimi vzorci se lahko dotičnega uporablja.

Obstajajo tudi drugi [24] načini opisovanja načrtovalskih vzorcev, npr. samo s pomočjo imena vzorca, konteksta, kjer so razložene razmere, ki so privedle do problema, opisa problema in rešitve. Konec koncev pa sta si oba koncepta zelo podobna. Pri obeh gre za opis možnosti reševanja ponavljajočega problema v bolj abstraktni obliki, kar uporabimo pri dejanski implementaciji programske opreme v določenem programskem jeziku. Preden se lotimo bolj podrobnega opisovanja definicij vzorcev in jih naštejemo, je primerno, da razložimo še nekatere druge pojme.

Arhitektura programske opreme je končni izdelek aktivnosti načrtovanja programske opreme. Gre za opis podsistemov in gradnikov programske opreme ter povezav med njimi. Podsistemi in gradniki so največkrat prikazani v različnih pogledih, da se lahko ločijo funkcionalne lastnosti sistema programske opreme [26]. **Komponenta** oz. gradnik je enkapsuliran del sistema programske opreme in ima svoj vmesnik. To so lahko moduli, razredi, objekti in množica povezanih funkcij [26]. **Podsistemi** so množica sodelujočih komponent, ki izvajajo določeno nalogo. Podsistem je ločena entiteta znotraj arhitekture programske opreme, ki izvaja določeno nalogo s pomočjo interakcije z ostalimi podsistemi in komponentami [26].

Kljub različnim notacijam, pa se je potrebno zavedati, da obstajajo različne kategorije vzorcev. Podsistemi arhitekture programske opreme (angl. "Subsystems Software Architecture"), kamor spadajo tudi relacije med njimi, so sestavljeni iz manjših arhitekturnih enot. Te enote opišemo s pomočjo **načrtovalskih vzorcev**. Poleg načrtovalskih vzorcev poznamo še arhitekturne vzorce. Načrtovalski vzorci so po obsegu manjši kot arhitekturni vzorci, vendar se trudijo biti neodvisni od programskega jezika. Uporaba vzorca pri tem ne vpliva na arhitekturo sistema [24].

Arhitekturni vzorci so vzorci na najvišjem nivoju sistema vzorcev in izražajo osnovno strukturno organizacijsko shemo za sisteme programske opreme (angl. "Software System"). Zagotavljajo množico vnaprej nastavljenih podsistemov, definirajo njihove odgovornosti, vključujejo pravila in smernice za organizacijo povezav med njimi. Drugače povedano, arhitekturni vzorci so predloge za določeno arhitekturo programske opreme. Določajo strukturne lastnosti programske opreme na področju sistema in so osnovna načrtovalska odločitev pri razvijanju sistema za programsko opremo [24]. Najbolj znan arhitekturni vzorec je model-pogled-krmilnik.

Poleg načrtovalskih in arhitekturnih vzorcev posebno mesto zasedajo tudi idiomi. **Idiomi** so nizkonivojski vzorci, ki so specifični za programski jezik. Opisujejo kako s pomočjo programskega jezika implementirati nek pogled komponent in povezav med njimi. Ravno zaradi specifičnosti določenega programskega jezika se lahko idiomi med seboj razlikujejo [24].

3.2. Arhitekturni vzorci

Arhitekturni vzorci so vzorci na najvišjem nivoju. Pomagajo določiti osnovno strukturo aplikacije. Dobra struktura sistema je pogoj za nadaljnji razvoj aplikacije, saj bodo vse nadaljnje odločitve slonele na tej arhitekturi. Poznamo več arhitekturnih vzorcev, ki jih lahko razdelimo v štiri kategorije [24]:

- **From Mud to Structure**: v njej se nahajajo vzorci "Layers", "Pipes and Filters" ter "Blackboard". Ti nam pomagajo, da se izognemo prekomerni uporabi komponent in objektov.
- **Distributed Systems**: ta kategorija vsebuje samo en vzorec, dveh iz druge kategorije pa se samo dotika. Vzorec "Broker" je edini v tej kategoriji, ponuja pa vso infrastrukturo za porazdeljene aplikacije.
- **Interactive Systems**: vsebuje vzorca "Model-View-Controller" in "Presentation-Abstraction-Control". Oba vzorca sta namenjena strukturiranju sistemov programske opreme, ki temeljijo na uporabniškem vmesniku za interakcijo človek – računalnik.
- **Adaptable Systems**: vzorca "Reflection" in "Microkernel" podpirata razširitev aplikacije za spreminjanje funkcionalnih zahtev.

3.3. Načrtovalski vzorci

Načrtovalski vzorci opisujejo splošne ponavljajoče strukture komponent, katere rešujejo splošen načrtovalski problem v določenem kontekstu. Kot že omenjeno, so GoF v svojo knjigo vključili katalog načrtovalskih vzorcev. Teh vzorcev je 23 in se delijo v tri kategorije, glede na njihov namen. Za lažjo predstavo so spodaj na kratko predstavljeni [19].

Ustvarjalni vzorci (angl. "Creational Patterns") pomagajo zgraditi sistem neodvisno od tega kako so objekti zgrajeni, sestavljeni in predstavljeni [19].

- **Abstraktna tovarna** (angl. "Abstract Factory"): ponuja vmesnik za izgradnjo množic sorodnih ali odvisnih objektov, brez določanja konkretnih razredov.
- **Graditelj** (angl. "Builder"): ločuje konstrukcijo kompleksnih objektov od njihove predstavitve, tako da lahko isti konstrukcijski procesi izdelajo različne predstavitve.

- **Tovarniška metoda** (angl. "Factory Method"): definira vmesnik za izgradnjo objekta, vendar preloži odgovornost podrazredom, da definirajo kateri razred je potrebno narediti.
- **Prototip** (angl. "Prototype"): s pomočjo prototipnih instanc določi vrsto objektov za izgradnjo in naredi nove objekte s kopiranjem prototipov.
- **Edinec** (angl. "Singleton"): poskrbi, da ima razred samo eno instanco in ponudi globalno točko dostopa do nje.

Strukturni vzorci (angl. "Structural Patterns") so osredotočeni na to, kako so razredi in objekti sestavljeni, da lahko tvorijo večjo celoto. Podajajo načine kako tvoriti objekte, da realiziramo nove funkcionalnosti. Do večje fleksibilnosti pripomore spreminjanje kompozicije v času izvajanja [19].

- **Adapter** (angl. "Adapter"): pretvori vmesnik razreda v vmesnik, ki ga odjemalec pričakuje. Pomaga razredom, da sodelujejo skupaj, kar drugače ne bi morali zaradi nezdružljivega vmesnika.
- **Most** (angl. "Bridge"): loči abstrakcijo od njene implementacije, tako da se lahko spreminjata neodvisno.
- **Kompozit** (angl. "Composite"): sestavi objekte v drevesno strukturo, s čimer predstavimo celotno hierarhijo po delih. Uporabimo ga, ko želimo, da se prezrejo razlike med kompozicijo objektov in posameznimi objekti ter da se jih obravnava enakovredno.
- **Dekorator** (angl. "Decorator"): dinamično pripne dodatne odgovornosti objektu. Ponuja prilagodljivo alternativo izgradnji podrazredov za razširjanje funkcionalnosti.
- **Fasada** (angl. "Facade"): množici vmesnikov v podsistemu ponuja enoten vmesnik. Definira vmesnik na višjem nivoju, kar omogoči lažjo uporabo podsistema.
- **Zrno** (angl. "Flyweight"): gre za deljen objekt, ki za učinkovito podporo velikemu številu manjših objektov omogoča souporabo.
- **Namestnik** (angl. "Proxy"): ponuja nadomestek ali okvir za drugi objekt, nad katerim lahko vrši nadzor.

Vedenjski vzorci (angl. "Behavioral Patterns") se uporabljajo za opisovanje komunikacije med razredi in objekti [19].

- **Veriga odgovornosti** (angl. "Chain of Responsibility"): uporablja se za združevanje prejetih objektov in podajanje teh zahtev naprej po verigi, dokler jih en objekt ne obdela. Ni zaželjeno, da je pošiljatelj zahtevka sklopljen z njegovim prejemnikom, zato je omogočeno, da zahtevek procesira več objektov.
- **Ukaz** (angl. "Command"): ovije zahtevek v objekt, posledično dovoli lastno parametrizacijo odjemalca z različnimi zahtevami, vrstami ali drugim zahtevami, saj je včasih potrebno poslati objektu zahtevo, brez da bi vedeli, kakšna operacija se zahteva oz. kdo je poslal zahtevo.

- **Tolmač** (angl. "Interpreter"): za dani jezik naredi predstavitev slovnice skupaj s tolmačem, ki uporablja predstavitev za tolmačenje stavkov v dotičnem jeziku.
- **Iterator** (angl. "Iterator"): ponudi način za zaporedni dostop do elementov združenega objekta, brez da bi razkril osnovno predstavitev.
- **Posredovalec** (angl. "Mediator"): definira objekt, ki ovije postopek sodelovanja množice objektov. Mediator promovira šibko sklopljenost na način, da objektom ne pusti eksplicitnega sklicevanja med seboj, vseeno pa pusti uporabniku neodvisno spreminjanje njihove interakcije.
- **Spomin** (angl. "Memento"): dovoljuje, da povrnemo objekt na neko stanje, brez da bi škodovali njegovemu notranjemu stanju.
- **Opazovalec** (angl. "Observer"): med objekti definira odvisnosti tipa ena–več, tako da so, ko en objekt spremeni svoje stanje, vsi podrejeni objekti avtomatsko obveščeni.
- **Stanje** (angl. "State"): dovoli, da objekt spremeni svoje obnašanje, ko se njegovo notranje stanje spremeni.
- **Strategija** (angl. "Strategy"): definira družino algoritmov, vsakega ovije in jih naredi med seboj zamenljive. Strategija dovoljuje algoritmom, da se spreminjajo neodvisno od odjemalca, ki jih uporablja.
- **Metoda predloge** (angl. "Template Method"): definira ogrodje algoritma in preloži nekaj korakov algoritma podrazredom. Na takšen način pusti podrazredom, da brez spreminjanja strukture algoritma ponovno določijo korake algoritma.
- **Obiskovalec** (angl. "Visitor"): predstavlja operacijo, ki bo izvedena nad elementi objektne strukture. Vzorec pusti, da definiramo nove operacije, brez spreminjanja razredov elementov, nad katerimi operira.

3.4. Arhitekturni in načrtovalski vzorci v računalništvu v oblaku

Z razvojem računalništva v oblaku in večjemu številu ponudnikov tehnologije oblaka, so se razvili tudi načrtovalski in arhitekturni vzorci za to področje. Predvsem slednji igrajo veliko vlogo pri računalništvu v oblaku, saj vse storitve in aplikacije slonijo na arhitekturi in ta mora biti zgrajena učinkovito in stabilno. V ta namen, so se pojavili nekateri novi arhitekturni in načrtovalski vzorci, namenjeni prav za računalništvo v oblaku. Načrtovalski vzorci, kot so na primer elastične vrste, večnajemništvo, porazdelitev bremena med strežniškimi instancami in merjenje porabe, ponujajo abstraktne rešitve za ponavljajoče probleme v računalništvu v oblaku. Prav tako niso vezani le na enega ponudnika tehnologije oblaka, temveč so s svojo abstraktnostjo dostopni tudi ostalim. Vzorci v računalništvu v oblaku se prav tako delijo glede na storitvene in namestitvene modele oblaka in glede na tip storitve (računski, shrambeni, povezovalni), v kateri se vzorec uporablja [27]. Zelo znana načrtovalska vzorca aplikacije sta večnajemništvo in elastičnost (angl. "Elasticity"). V kolikor želimo omogočiti elastičnost je potrebno poskrbeti tudi za šibko sklopljenost aplikacijskih komponent. Tako vidimo, da se je pri razvoju aplikacij in storitev za oblak potrebno držati smernic, ki jih določa storitveno

usmerjena arhitektura. Aplikacijo je potrebno razvijati v obliki ponovno uporabljivih modulov, kar nam tudi omogoči enostavnejši prenos na oblak ter enostavnejšo razširljivost. Takšen način izgradnje omogoča, da imamo področja (dostop do podatkovne baze, poslovna logika, uporabniški vmesnik) ločen med seboj, kar prinaša visoko fleksibilnost [44].

4. UPORABLJENI VZORCI

4.1. Večnajemniški podatkovni modeli

Večnajemništvo opisuje takšno arhitekturo programske opreme, kjer si več najemnikov (strank) deli eno instanco programske opreme [23]. Deljenje virov (angl. "Resource Pooling"), ki je ključna lastnost računalništva v oblaku, omogoča izvajanje večnajemniških aplikacij. Bistvo deljenja virov je abstrakcija, ki uporabnikom prikrije dejansko lokacijo virov.

Aplikacija mora biti zasnovana tako, da zajema želje najemnika in ponudnika. Ponudnik se mora pogovarjati s svojimi najemniki in ugoditi njihovim željam in ciljem. Prav tako mora spremljati izvajanje aplikacije in ukrepati v primeru napak, saj napaka ne prizadene samo enega najemnika, ampak vpliva na delovanje aplikacije pri vseh najemnikih. Ne nazadnje pa mora ponudnik najemniku tudi zaračunati uporabo aplikacije, kar lahko naredi s spremljanjem uporabe vsakega najemnika ali pa zaračuna fiksno ceno. Na drugi strani si najemnik želi, da lahko do določene mere prilagodi uporabniški vmesnik. Omogočimo mu nastavitve, kot so zamenjavo logotipa, barve ter dodajanje in odstranjevanje funkcij aplikacije. Naprednejše večnajemniške aplikacije celo omogočajo dodajanje svojih stilov, skript, pravic in do določene mere delovne tokove. Za dosego teh sprememb je potrebno vpeljati metapodatke ali hraniti podatke o nastavitvah v namestitveni datoteki oz. podatkovni bazi.

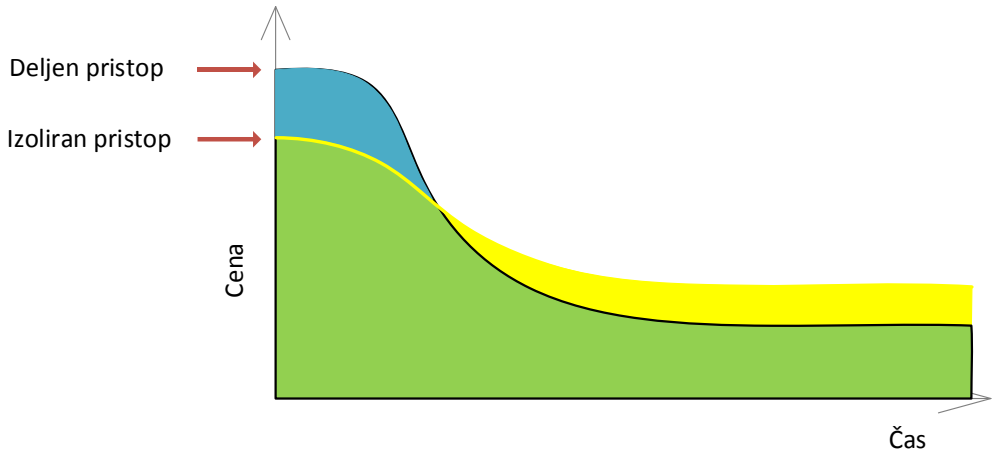
Aplikacija naj bo prav tako šibko sklopljena, s čimer omogočimo lažjo razširljivost. Razširljivost je odvisna od načrtovanja aplikacije, katera mora biti zmožna namestitve na več instanc, medtem ko še zmeraj dostopa do istih podatkov. Večnajemniške aplikacije naj bi bile zmožne tako vertikalne kot tudi horizontalne razširljivosti. V kolikor do aplikacije dostopajo najemniki iz drugih držav, je dobro razmisliti o namestitvi aplikacije na strežnike, ki so bližje njihovim lokacijam, oz. vpeljavo Azure CDN, ki je zmožen predpomnenja strani. Microsoft in drugi ponudniki oblaka imajo v ta namen postavljenih več strežnikov po različnih državah sveta, s čimer zagotovijo manjšo latenco. Vse te zahteve in dogovori so zapisani v dogovoru o ravni storitev (SLA). Z deljenjem aplikacije več najemnikom, se posledično znižajo stroški za uporabo, česar se zavedajo tudi stranke in to tudi pričakujejo. Tako je pričakovati, da cena večnajemniških aplikacij ne preseže namenskih aplikacij. Najbolj pomembno od vseh najemniških zahtev pa je, da vsakemu najemniku zagotovimo izolacijo njegovih podatkov pred drugimi najemniki.

4.1.1. Podatkovna arhitektura

Najemniki predvidevajo, da v večnajenniškem modelu, za svoje plačilo dobijo tudi varno shranjevanje in branje podatkov ter izolacijo podatkov, kljub temu da aplikacijo uporablja več

najemnikov. Podatki, ki jih najemniki shranjujejo so lahko zaupne narave, zaradi česar je varnost podatkov nujna. Ravno varnost podatkov v oblaku je razlog, zakaj se ljudje množično ne odločajo za uporabo tehnologij oblaka, saj se bojijo izgube svojih podatkov ali vdora v račun. Seveda se trend s časom spreminja, a nekateri dvomi ostajajo.

V večnajemniški arhitekturi je aplikacija deljena med uporabniki. Medtem ko podatki ne smejo biti deljeni med najemniki, je podatkovna arhitektura vseeno lahko deljena. Bolj kot je podatkovna arhitektura deljena, manj so podatki izolirani med najemniki. Izbira arhitekturnega načina je v veliki meri odvisna od ekonomskih vidikov. Začetni vložki pri deljeni strukturi podatkovne arhitekture so sprva večji kot pri izoliranem pristopu, vendar so s časom nižji (slika 1). Izbira je odvisna tudi od želene varnosti podatkov in od števila načrtovanih najemnikov, saj več kot jih imamo, bolj je primeren deljen pristop. Nenazadnje pa ne smemo pozabiti tudi na zakonsko podlago [37].



Slika 1: Spreminjanje cene pri različnih pristopih podatkovne arhitekture.

V nadaljevanju si bomo pogledali načine podatkovne arhitekture v večnajemniški arhitekturi. Zanima nas, kako so podatki izolirani oz. deljeni v vsaki od kategorij, kako to vpliva na vzdrževanje ter razmerje s ceno. Možni načini podatkovne arhitekture večnajemniške aplikacije so sledeči [28]:

4.1.1.1. Fizično ločeni strežniki z nameščeno podatkovno bazo za vsakega najemnika

Takšen način podatkovne arhitekture prinaša največjo stopnjo izolacije, saj so podatkovne baze nameščene na različnih fizičnih strežnikih. Vseeno gre za cenovno potratno možnost, kajti najemniki morajo plačevati najem strežnika (fizični strežnik, licence za programsko opremo, vzdrževanje). Zaradi visokih stroškov in težkega vzdrževanja sistema ni primerno za večje število najemnikov. Poleg tega se lahko zatakne pri razširjanju navzven, kar jih naredi manj priljubljene pri večnajemniških aplikacijah, a bolj priljubljene pri eno najemniških aplikacijah.

4.1.1.2. Ločene podatkovne baze za vsakega najemnika

Znotraj enega podatkovnega strežnika imamo narejene podatkovne baze za vsakega najemnika. Takšen pristop ohranja močno stopnjo izolacije podatkov, saj so podatki logično ločeni od ostalih. Računalniški viri so porazdeljeni med vsemi najemniki, zaradi česar se cena najema že porazdeli med najemniki. Vseeno je cena še zmeraj višja od naslednjih dveh arhitekturnih modelov, saj je potrebno vsako podatkovno bazo vzdrževati posebej. Prav tako potrebujemo bolj zmogljivo računalniško opremo, kajti strežnik omejuje možno število podatkovnih baz. Za primerjavo navedimo, da Microsoft SQL Server 2012 podpira malo več kot 32.000 podatkovnih baz na instanco strežnika, medtem ko Azure SQL Database samo 150 podatkovnih baz na strežnik.

Glede na to, da ima vsak najemnik svojo podatkovno bazo, je možno prilagoditi podatkovni model najemniku. Takšen arhitekturni model je bolj primeren za podjetja, kjer je potrebna visoka stopnja izolacije in si podjetja lahko privoščijo dražje naročnine. Pri tem modelu je tudi enostavnejše varnostno kopiranje podatkov in obnova v primeru napake.

4.1.1.3. Ločene podatkovne sheme za vsakega najemnika

Model uporablja eno podatkovno bazo, znotraj katere imajo najemniki množico podatkovnih tabel. Tabele so za vsakega najemnika skupaj združene v shemo. Najemniku so dane pravice vpogleda samo v tabele, katere spadajo v njegovo shemo. Izolacija podatkov je tu že manjša, saj so podatki v isti podatkovni bazi. Primerna je za aplikacije, ki imajo največ do 100 tabel na najemnika.

Kljub temu, da imajo vsi najemniki svoje sheme, lahko programsko, s pomočjo nastavljanja privzete sheme, ohranimo iste stavke SQL za vse najemnike. Tako nam ni potrebno spreminjati ostale programske rešitve. V kolikor je potrebno, lahko strukturo tabel prilagodimo najemniku, saj ima vsak svoje tabele. Problem se pojavi z izdelavo varnostnih kopij in obnovitvijo, kjer v primeru obnavljanja podatkov povozimo podatke ostalih najemnikov. Postopka se je potrebno lotiti drugače in najprej obnoviti podatke na začasni strežnik, od tam pa na produkcijski strežnik.

Cena takšne rešitve je nižja kot pri predhodnih modelih, kajti s tem modelom lahko sprejmemo več najemnikov v en podatkovni strežnik. Microsoft SQL Server 2012 ima omejitev več kot 2 milijardi objektov na podatkovno bazo, za Azure SQL Database tega podatka ni bilo moč izslediti iz dokumentacije. V objekte so vštete vse tabele, pogledi, procedure itd.

4.1.1.4. Razširitev obstoječih tabel s ključem najemnika

Gre za model, kjer vsi najemniki uporabljajo isto podatkovno bazo z istimi tabelami, v kateri so shranjeni vsi podatki od vseh najemnikov. Med najemniki ločimo podatke v tabelah tako, da vsaki tabeli dodamo ključ najemnika (angl. "Tenant ID"), kateri enolično označuje najemnika. Omenjeni model je med vsemi najbolj deljen in ponuja najmanjšo stopnjo izolacije podatkov. Zaradi tega je tu potrebno aplikativno poskrbeti za varno izolacijo podatkov, saj morajo bralne in pisalne operacije nujno upoštevati ključ najemnika. Na ta način preprečimo, da bi najemniki lahko dostopali do podatkov ostalih najemnikov. Glede na to, da najemniki med seboj delijo podatkovne tabele, zaradi česar imamo tudi nizke stroške z vzdrževanjem, je ta model najcenejši. Varnostno kopiranje in obnavljanje je podobno, kot pri prejšnjem primeru, kjer v primeru obnove podatkov povozimo vse podatke vseh najemnikov. Zaradi tega je potrebno uvoziti zapise iz začasne tabele.

Uporaba iste podatkovne baze z ločenimi shemami ali istimi tabelami, vendar s ključem najemnika, je med bolj uporabljenimi različicami podatkovne arhitekture večnajemništva in tudi cena takšnega tipa izolacije podatkov je nižja. Odvisno od aplikacije in ponudnika tehnologije oblaka imamo lahko poleg relacijske podatkovne baze še ne-relacijske podatkovne baze, shrambo blobov in sporočilne vrste. V kolikor se odločimo za uporabo teh storitev, je potrebno poskrbeti tudi za izolacijo na tem nivoju. Privzeto imajo namreč vsi najemniki omogočen bralni dostop do podatkov v tabelah in blobih. To lahko spremenimo z uporabo ključev računa, ki ga generiramo za vsak račun za shrambo. Račune za shrambo lahko načeloma odpremo za vsakega najemnika posebej, vendar je to pogojeno z naročnino do Windows Azure. V operacijah aplikacije pa je vseeno priporočljivo hraniti in uporabljati identifikacijski ključ najemnika.

4.1.2. Identifikacija najemnikov na nivoju aplikacije

Spletna aplikacija mora znati ločevati med najemniki, saj v primeru eno instančne večnajemniške aplikacije vsi najemniki dostopajo do iste aplikacije. V ta namen obstaja več načinov kako lahko zagotovimo prepoznavanje najemnikov [28]:

- **Prijava v sistem**: v kolikor se mora uporabnik pri dostopanju do strani prijaviti, potem lahko njegove poverilnice izkoristimo tudi za prepoznavanje najemnika. Prijava v sistem poteka preko avtentikacijskih in avtorizacijskih mehanizmov. Windows Azure omogoča prijavo preko že znanih ponudnikov identitet (Facebook, Google, …), kakor tudi z uporabo aktivnega imenika ali lastnih avtentikacijskih mehanizmov.
- **Identifikacija najemnika na podlagi naslova URL**: aplikacija ima isto domeno za vse najemnike. Za potrebe ločevanja med najemniki lahko ponudimo pot, iz katere razberemo identifikacijski ključ ali naziv najemnika. Kot primer lahko vzamemo: `http://www.glavna-domena.si/{ime-najemnika}/domov`.

- **Uporaba poddomen ali lastnih domen**: najemniki imajo lahko tudi lastne poddomene ali domene in skupaj s preostalim naslovom URL najemnike prav tako ločimo med sabo. Primer:
  `http://{najemniska-domena}.glavna-domena.si/{ime-najemnika}.`

Pri spletnih storitvah se je prav tako potrebno lotiti identifikacije najemnikov. Največkrat se uporabljajo iste spletne storitve za vse najemnike, tako da je potrebno zagotoviti, da se pri uporabi, poleg vseh parametrov, prenaša še identifikacijski ključ najemnika. Spletne storitve lahko zasnujemo tudi tako, da se razlikujejo glede na zakupljeno naročnino. Zgraditi je mogoče storitve, ki so namenjeni zastonjski uporabi in plačljivi uporabi. Tu pošiljamo, poleg identifikacijskega ključa najemnika, tudi tip naročnine oz. tip najemnika.

4.1.3. Vzorec: izbira večnajemniškega podatkovnega modela

4.1.3.1. Ime vzorca

Izbira večnajemniškega podatkovnega modela.

4.1.3.2. Namen

Zgraditi in ponuditi sistem, ki omogoča najemnikom izbiro načina podatkovne izolacije znotraj večnajemniške aplikacije, tako da večji del logike aplikacije ohranimo nespremenjene.

4.1.3.3. Prav tako poznano kot

Podatkovno večnajemništvo.

4.1.3.4. Motivacija

Glede na to, da imajo nekateri najemniki želje po večji izoliranosti podatkov, saj jim to predvideva zakon ali lastna politika, želimo zgraditi večnajemniško aplikacijo, kjer bo možno izbirati med deljenimi tabelami, ločenimi z identifikacijskim ključem najemnika, ali med shemami, ki združujejo ločene tabele za najemnike. Vseeno pa želimo, da to ne bi pretirano vplivalo na logiko aplikacije in podatkovne poizvedbe oz. da spremembe programske kode ne bi bile pretirane. S pomočjo sodobnih programskih jezikov je to možno doseči.

4.1.3.5. Uporaba

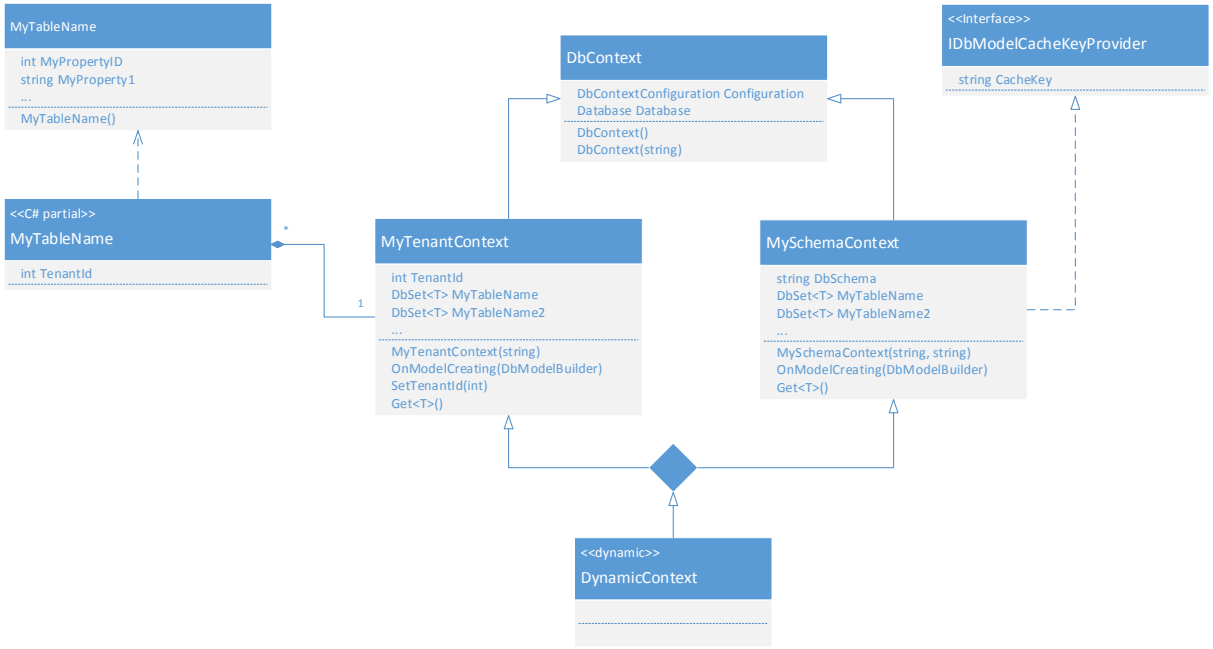
Uporabite ta vzorec, v kolikor želite:

- Preoblikovati eno instančno aplikacijo, da bo lahko uporabljala dva podatkovna modela.
- Podatkovno arhitekturo ločiti s shemami in identifikacijskim ključem najemnika.

- Ohraniti strukturo podatkovnega modela v aplikaciji.

4.1.3.6. Struktura

Slika 2 prikazuje strukturo načrtovalskega vzorca za izbiro večnajemniškega podatkovnega modela. S pomočjo dinamičnega objekta zgradimo podatkovni kontekst, ki razširja razred DbContext, kateri se uporablja za povezavo do podatkovne baze. Podatkovni kontekst, ki se bo uporabljal za podatkovne tabele s shemami, je potrebno razširiti še z vmesnikom IDbModelCacheKeyProvider. Prav tako je potrebna preslikava tabel v delne razrede, ki jih razširimo z ustreznim identifikacijskim ključem najemnika.



Slika 2: Struktura vzorca za izbiro večnajemniškega podatkovnega modela.

4.1.3.7. Udeleženci

- **DbContext**: je razred za poizvedbe po podatkovni bazi in za shranjevanje sprememb narejenih na entitetah, katere zapišemo v bazo.
- **IDbModelCacheKeyProvider**: vmesnik za ključ, ki omogoča več modelov oz. več različnih kontekstov za isti model.
- **MyTenantContext** in **MySchemaContext**: izpeljava razreda DbContext za uporabo na enem modelu, s spremembami za uporabo shem in najemniškega ključa. Uporablja se za interakcijo z bazo.
- **MyTableName** in **partial MyTableName**: več entitet iz podatkovne baze, ki so izpeljane kot razredi. `Partial` razred razširja svojo entiteto s ključem najemnika.
- **DynamicContext**: je atribut dinamičnega tipa, ki se pretvori v enega od kontekstov v času izvajanja.

4.1.3.8. Sodelovanja

- Imamo dva konteksta podatkovne baze, ki sta izpeljana iz DbContext. Za lažjo spremembo povezovalnega niza do podatkovne baze imata spremenjene konstruktorje.
- Kontekst, ki se uporablja s podatkovnimi shemami, razširja vmesnik na način, da lahko dinamično spreminjamo sheme.
- Drugi kontekst v sebi vsebuje delne razrede entitet, ki jih razširjamo s ključem najemnika.
- Vsi delni razredi razširjajo dejanske razrede entitet, ki jih oblikujemo iz dejanske podatkovne baze, katero uporablja aplikacija.
- Dinamični atribut ima možnost spreminjanja svojega tipa med časom izvajanja.

4.1.3.9. Posledice

Prednosti uporabe tega vzorca so naslednje:

- Vzorec omogoča najemnikom, ob registraciji v večnajemniško aplikacijo, možnost izbire izolacije podatkov.
- Z možnostjo izbire izolacije ohranimo eno aplikacijo za obe skupini uporabnikov, posledično so manjši stroški z vzdrževanjem.
- Uporaba dveh kontekstnih modelov, a z dinamičnim atributom, omogoča enostavno preklapljanje konteksta v programski kodi.
- Enostavno in hitro spreminjanje med shemami najemnikov.
- Večnajemništvo omogoči najemnikom nižje stroške, saj si aplikacijo delijo še z ostalimi najemniki.
- Lažja razširljivost tako aplikacije kot podatkovne baze, saj je aplikacija šibko sklopljena.

Slabosti tega vzorca pa so:

- Potrebno je dodati vse delne razrede entitet za uporabo v enem kontekstu in vklopiti ignoriranje tega atributa v drugem kontekstu. Pri velikem številu tabel je to delo zamudno.
- Uporabljamo eno podatkovno bazo tako za tiste uporabnike, ki imajo ločene podatkovne sheme, kot za tiste, ki imajo v tabelah ločene vnose z najemniškim ključem.
- Potrebno je ločiti poizvedbe in jih prilagoditi za uporabo s ključem najemnika, kar je zamudno.

4.1.3.10. Implementacija

- **Shramba najemnikov**: potrebno je narediti podatkovno tabelo, kamor bomo hranili podatke o najemniku. V primeru vzorca smo se odločili za Azure Table Storage, kamor moramo med drugim hraniti najemniško ime, ki ga bomo uporabili pri izdelavi shem, v kolikor se je najemnik za to odločil. Prav tako hranimo način izolacije podatkov, tj. podatek ali je najemnik želel izolacijo s shemami ali najemniškim ključem. Nato sledijo še ostali podatki, kot so uporabniško ime, geslo, datum vnosa itd. Ti podatki morajo biti zaščiteni in varni, saj nosijo podatke o vseh naših najemnikih. Gesla morajo biti kriptirana, npr. v obliki izvlečka (angl. "Hash").
- **Preverjanje najemnika**: ob prihodu uporabnika na aplikacijo moramo razbrati za kakšen tip najemnika gre – torej ali ima podatke shranjene v svoji shemi ali v skupni podatkovni bazi. Iz URL-ja aplikacije dobimo ime najemnika. Postopek lahko preoblikujemo tudi v prijavno okno, iz katerega dobimo podatke o uporabniku.
    - **Sheme**: uporabimo kontekst za sheme. Preverimo ali ima uporabnik svojo shemo, ali je uporabnik nov (torej gre za registracijo), ali shema obstaja itd.
    - **Najemniški ključ**: uporabimo kontekst za skupno podatkovno bazo. Pridobimo podatek o najemniku – med drugim tudi njegov najemniški ključ, ki nam pomaga za pridobivanje podatkov iz podatkovne baze.
- **Povezava na podatkovno bazo**: na mestih, kjer imamo izgradnjo konteksta za dostop do podatkovne baze, tega zamenjamo z dinamičnim in nato s preverjanjem ali gre za najemnika, ki ima svojo shemo ali skupne tabele, izberemo in nastavimo pravilni kontekst in po potrebi tudi shemo.
- **Poizvedbe in spremembe**: poizvedbe so narejene preko ogrodja Entity Framework. Trenutne poizvedbe je potrebno razdeliti na tiste s shemo in tiste s ključem najemnika. Poizvedbe z bazo, ki so vezane na shemo, ostanejo nespremenjene. Dodati moramo poizvedbe z najemniškim ključem. Spremembe poizvedb so potrebne povsod, kjer iščemo ali zapisujemo na podlagi ključa najemnika.

4.1.3.11. Vzorčna koda

Vzorčna koda za omenjeni vzorec je predstavljena v poglavju 5.1.

4.1.3.12. Znani primeri uporabe

Salesforce.com je podjetje, ki ima aplikacije za CRM. Te aplikacije gostujejo na oblaku in so grajene prav na način večnajemniške aplikacije. Exact Online je primer večnajemniške poslovne aplikacije, specifične za oblak, ki uporablja skupno podatkovno bazo in eno instančno aplikacijo.

4.1.3.13. Sorodni vzorci

Vzorec elastičnosti, vzorec model–pogled–krmilnik.

4.2. Spletne storitve v oblaku

W3C, ki je mednarodna organizacija za standarde na področju spleta, opisuje spletne storitve kot programski sistem, zgrajen za podporo medsebojni interakciji računalnik–računalnik preko omrežja [29]. Spletna storitev ima vmesnik, kateri je definiran v obliki WSDL, s pomočjo katerega opisuje funkcionalnosti spletne storitve. WSDL opiše spletno storitev tako, da definira metode spletne storitve, kaj te metode vračajo, njihove vhodne parametre in podobno. Za komunikacijo s spletno storitvijo se uporablja protokol SOAP, ki skrbi za izmenjavo informacij o spletni storitvi preko omrežij. Sporočila SOAP so v obliki XML in se največkrat prenašajo preko internetnega protokola HTTP. Prenos sporočil je možen tudi preko protokola SMTP, FTP, HTTPS in ostalih.

S pomočjo spletnih storitev lahko dosežemo storitveno usmerjeno arhitekturo (SOA), ki pa je tudi ključni del tehnologije oblaka. SOA je načrtovalski vzorec, kjer so funkcionalnosti aplikacije izražene kot storitve, do katerih imajo dostop tudi druge aplikacije. Koncept spletnih storitev v oblaku je povsem primerljiv običajnim spletnim storitvam.

Spletne storitve se v grobem delijo na storitve skladne z REST-om in poljubne spletne storitve, ki imajo izpostavljeno poljubno množico operacij [29]. Druga skupina spletnih storitev največkrat za komuniciranje med odjemalcem in ponudnikom uporablja protokol SOAP. Ta je tako kot mnogo drugih protokolov izpeljan iz protokola **RPC (Remote Procedure Call)**. RPC je mehanizem, ki omogoča prenos podatkov in izvajanje storitev v drugem procesu. Proces se lahko nahaja na istem računalniku, v istem omrežju ali kje drugje v svetovnem spletu. Klic procedure se bo izvedel tudi če ne poznamo podrobnosti lokacije storitve.

SOAP (Simple Object Access Protocol) [50] je, tako kot RPC, komunikacijski protokol za izmenjavo podatkov. Sporočila, ki se prenašajo, so napisana s pomočjo jezika XML, za prenos pa v osnovi uporablja protokol HTTP. Možna je tudi uporaba drugih protokolov, kot so SMTP, ki se uporablja za pošiljanje elektronske pošte ali FTP, ki je namenjen dostopu do datotečnih sistemov. SOAP je v kombinaciji s protokoloma XML in HTTP odlična kombinacija za enostavno komunikacijo preko omrežja s procesi, ki lahko tečejo tudi znotraj različnih operacijskih sistemov. SOAP je v glavnem osredotočen na operacije. Sporočilo SOAP je po strukturi predstavljeno s spodnjimi elementi:

- **Ovojnica** (angl. "Envelope"): določa, da je dokument XML dejansko sporočilo SOAP. Ovojnica je obvezni element sporočila. Znotraj ovojnice so nato še elementi glava, telo in napaka.
- **Glava** (angl. "Header"): vsebuje informacije o samem sporočilu in ni obvezni element.
- **Telo** (angl. "Body"): je osrednji del sporočila in vsebuje informacije o klicu in odgovoru. Telo je obvezen del sporočila.
- **Napaka** (angl. "Fault"): je na koncu sporočila in je opcijski element. Vsebuje podatke o napakah, ki so se zgodile med izvajanjem sporočila.

REST (Representational State Transfer) [50] je arhitekturni stil, tako da spletno storitev le oblikujemo po njegovih načelih. REST opisuje množico arhitekturnih načel in se osredotoča na načrtovalska pravila, saj želi doseči storitev brez beleženja stanja (angl. "Stateless"), s katero lahko dosežemo visoko razširljivost. Osredotočen je na uporabo virov. Podatki se prenašajo preko protokola HTTP, dostop do virov pa je mogoč preko URI-ja. URI naj bo logično sestavljen. Za predstavitev virov se uporablja JSON ali oblika XML. Standardne metode HTTP, ki jih lahko izvajamo nad viri so GET, PUT, POST, DELETE.

- **GET**: zahteva določen vir iz strežnika.
- **PUT**: izdela ali posodobi vir na strežniku ob avtenticiranem uporabniku.
- **POST**: uporablja se za pošiljanje podatkov oddaljenemu strežniku, kateri podatke uporabi za nadaljnjo procesiranje.
- **DELETE**: posreduje strežniku zahtevo za brisanje vira, vendar se mora uporabnik najprej avtenticirati.

Poleg omenjenih štirih obstaja še metoda HEAD, katera pošlje zahtevek za informacijo o glavi vira. Operacija, ki jo izvede metoda je vseeno odvisna od zasnove, ki pa je prikrita uporabniku. Zaradi tega je URI zelo pomemben za končnega uporabnika, saj na edinstveni način opisuje postopek ob klicu metode.

REST je prav tako bolj primeren za spletne storitve za mobilne naprave (pametne tablice, mobilni telefoni), saj največkrat ne potrebujemo vseh elementov, ki jih uporablja protokol SOAP. Na ta način zmanjšamo porabo virov. V kolikor primerjamo implementacijo spletnih storitev REST in SOAP, je REST enostavnejši za implementacijo. Vseeno pa ima nekatere REST arhitekturne omejitve. Z arhitekturnimi lastnostmi in omejitvami tako REST poizkuša minimizirati latenco in povečati neodvisnost ter razširljivost komponent [51]:

- **Odjemalec-strežnik**: vse dokler se vmesnik med odjemalcem in strežnikom ne spremeni, sta le-ta ločena med seboj, tako da ju lahko neodvisno dopolnjujemo in zamenjujemo.
- **Brez hrambe stanja**: komunikacija med odjemalcem in strežnikom se ne hrani na strežniku, temveč se stanje seje hrani na strani odjemalca.
- **Predpomnjenje**: odgovori se lahko shranijo v predpomnilnik, kar omogoči boljšo zmogljivost in razširljivost, vendar odjemalci ne smejo dostopajo do starih podatkov.

- **Nivojski sistem**: uporabimo lahko vmesne strežnike, kateri delujejo kot izenačevalniki bremena (angl. "Load Balancer"), kateri vsebujejo deljene predpomnilnike.
- **Koda na zahtevo** (opcijsko): strežnik odjemalcu prenese izvršilno kodo, katera lahko začasno spremeni določeno funkcionalnost storitve.
- **Enotni vmesnik**: določa enotni vmesnik med odjemalcem in strežnikom. S pomočjo te omejitve poenostavimo in razdvojimo arhitekturo storitve. Gre za omejitev, ki je osnova načrtovanja storitev REST.

V kolikor se storitev drži vseh zgornjih omejitev, potem jo lahko označimo kot "RESTful" storitev. Prednosti in slabosti enega ali drugega načina spletnih storitev so do določene mere zelo subjektivne, odvisne od želje implementacije in potrebe projekta, ki ga gradimo. Vseeno lahko trdimo nekatere prednosti in slabosti spletnih storitev REST (tabela 8) in SOAP (tabela 9).

Spletne storitve REST	
Prednosti	**Slabosti**
Enostavno vzdrževanje in razširljivost.	Shema in operacije niso same po sebi definirane.
Možna enostavna in dobra uporaba predpomnjenja podatkov.	Pošiljanje podatkov preko URI-ja je lahko tudi slabost, saj mora odjemalec vse podatke hraniti lokalno pred pošiljanjem storitvi.
Ne hranijo stanja seje, kar pomeni manj porabljenih virov.	
Vhodni parametri se pošiljajo preko URI-ja, kar poudari enostavnost.	
Neodvisen od programskega jezika in platforme.	

Tabela 8: Prednosti in slabosti spletnih storitev REST.

Spletne storitve SOAP	
Prednosti	**Slabosti**
Generira datoteko WSDL, v kateri je definicija operacij in tipov.	Oblika XML zmanjša hitrost procesiranja.
Zmožen delovati tudi pod drugimi aplikacijskimi protokoli.	Ne moremo uporabiti predpomnenja, saj uporablja HTTP POST.
Neodvisen od programskega jezika in platforme.	

Tabela 9: Prednosti in slabosti spletnih storitev SOAP.

Skozi poglavje se bomo osredotočili na spletne storitve REST, saj med drugim omogočajo večjo razširljivost, so enostavnejše, ponujajo boljšo interakcijo z ostalimi platformami in napravami ter predpomnjenje. Prav tako je od leta 2005 priljubljenost spletnih storitev REST naraščala. Od konca leta 2008 pa se je njihova priljubljenost bliskovito povzpela v višave, saj so se množično začele pojavljati storitve, ki so temeljile na arhitekturnem stilu REST. Število spletnih aplikacij, ki ponujajo vmesnike API, je prav tako rastla v razmerju s priljubljenostjo storitev REST [32].

Pomembna tema pri spletnih storitvah je tudi dekompozicija spletnih storitev. Skozi načrtovalske vzorce storitveno usmerjene arhitekture se lahko naučimo, da je potrebno paziti na življenjski cikel spletnih storitev med načrtovanjem in tudi po uspešni izgradnji storitve. Največkrat se zgodi, da v fazi načrtovanja zasnujemo vse metode v eno storitev, katera hitro postane zelo obremenjena. Bistvo dekompozicije je, da vzamemo obstoječo t.i. grobozrnato storitev in jo razbijemo v logično povezane množice drobnozrnatih storitev, ki skupaj predstavljajo originalne storitve, vendar so ločene v svojo funkcijsko zvezo [53].

4.2.1. Spletne in delovne vloge

Windows Azure ponuja v okviru storitev oblaka izdelavo spletnih in delovnih vlog. Storitev oblaka je lahko sestavljena iz ene ali več spletnih in/ali delovnih vlog. Spletne in delovne vloge spadajo v računske storitve oblaka Windows Azure in lahko gostujejo eno ali več instanc vlog. Instanca vloge je dejansko navidezni strežnik, ki poganja gostujoči operacijski sistem (največkrat je to Windows Server, lahko pa tudi Linux oz. ostali možni sistemi), ter ostale storitvene datoteke (binarne datoteke, …). V kolikor imamo več instanc vlog (torej več navideznih strežnikov), so le-te med seboj povezane z izenačevalnikom obremenitve. Ta navzven odjemalcem ponudi javni naslov IP in številko vrat za dostop do storitev, medtem ko znotraj strežnike poveže s privatnimi naslovi IP in številko vrat, ki je lahko drugačna od zunanje.

Spletna vloga je namenjena aplikacijam, katere za svoje delovanje potrebujejo IIS (Internet Information Services). Primer takšnih aplikacij so spletne aplikacije z uporabniškim vmesnikom ali REST API-ji. Spletna vloga ponuja vse funkcionalnosti delovne vloge, vendar ima zraven še IIS.

Delovna vloga je namenjena izgradnji aplikacij, katere so lahko asinhrone, imajo daljši čas izvajanja, predvsem pa ne potrebujejo človeške interakcije. Delovno vlogo lahko primerjamo z Windows Service. Prav tako do delovne vloge ne moremo dostopati direktno s pomočjo protokolov HTTP ali HTTPS, temveč v ta namen potrebujemo namensko aplikacijo. Do storitev v delovnih vlogah dostopamo preko nastavljenih končnih točk (angl. "Endpoint").

Prav tako lahko izdelamo hibrida med spletno in delovno vlogo, tako da spletni vlogi dodamo implementacijo `RoleEntryPoint` razreda. `RoleEntryPoint` vsebuje tri metode, katere moramo razširiti, to so `OnStart()`, `Run()` in `OnStop()`. V metodi `Run()` imamo glavno logiko, medtem ko se ostali dve izvedeta ali ob zagonu ali ob izklopu storitve. Lastne spletne storitve lahko zgradimo in namestimo v okviru spletnih ali delovnih vlog. V naslednjih poglavjih si bomo pogledali štiri različne zasnove spletnih storitev katere bodo v pomoč pri implementaciji vzorca. Pri izgradnji spletnih storitev za Windows Azure se najprej odločiti za katero vlogo bomo izdelali spletno storitev ter ali v tehnologiji SOAP ali REST.

4.2.2. Avtentikacija in avtorizacija

Lahko bi rekli, da je varnost ena od najtežjih nalog pri načrtovanju in izgradnji spletnih aplikacij in storitev, sploh ko imamo aplikacijo javno dostopno na internetu. Način avtentikacije uporabnika preko spletnih storitev je različna glede na tip spletne storitve. V ta namen je bolj primerna uporaba spletnih storitev SOAP, saj že vsebuje standardizirane mehanizme za varnost. Tako REST kot SOAP lahko uporabljata protokol HTTPS, ki nudi vse ugodnosti protokola HTTP, le da na transportnem nivoju uporablja kriptografske protokole. Spletne storitve SOAP omogočajo uporabo varnostnega standarda WS-Security, katerega je predstavila organizacija OASIS. Poleg WS-Security v skupino varnostnih standardov spadata še razširitvi WS-Trust in WS-SecureConversation, ki skrbita za varno komunikacijo.

WS-Security specificira tri mehanizme za varnost spletnih storitev. S podpisanimi sporočili SOAP zagotovimo celovitost, kriptirana sporočila zagotavljajo zaupnost, pripeti varnostni žetoni pa preverjajo pošiljateljevo identiteto. Standard WS-Security omogoča uporabo različnih varnostnih protokolov, kot so na primer X.509, Kerberos, uporabniško ime in geslo ter lastni žetoni. Vse varnostne značilnosti se pripnejo glavi sporočila SOAP. Potrebno je opozoriti, da pri uporabi WS-Security standarda, le-ta doda varnostne značilnosti v glavo sporočila SOAP, s čimer poveča velikost sporočila. S tem se prav tako poveča tudi čas za procesiranje samega sporočila in prenos podatkov.

Znani razširitvi SOAP, ki se uporabljata kot avtentikacijska mehanizma [31]:
Osnovna avtentikacija (angl. "Basic Authentication") je namenjena avtentikaciji uporabnikov z uporabo njihovih poverilnic (angl. "Credentials"), ki so sestavljene iz uporabniškega imena in gesla. Ta način ponuja relativno nizko varnost, saj v zahtevku pošiljamo uporabniško ime in geslo, ki sicer sta kodirana s pomočjo kodirne sheme Base64, vendar to ne predstavlja varnosti. Priporočljivo je, da se osnovna avtentikacija uporablja v kombinaciji s protokolom HTTPS, ki s pomočjo varnostnih protokolov poskrbi za kriptiranje sporočila. V nasprotnem primeru je možno kodirani niz dekodirati. Celoten postopek temelji na protokolu izziv – odgovor. Postopek je sledeč: kadar želi odjemalec dostopati do zavarovanega vira in pri tem ne poda uporabniškega imena ter gesla, prejme odziv od

strežnika, da avtentikacija ni uspela.. Strežnik vrne izziv, v katerem zahteva uporabniške poverilnice, prav tako pa v telo ovojnice SOAP pripne napako, o neuspeli avtentikaciji. V glavo sporočila SOAP prav tako vključi element `BasicChallenge`, kateri vsebuje domeno/področje (angl. "Realm"). `Realm` je element tipa niz in vsebuje ime, kateri služi uporabniku kot pomoč pri identifikaciji, v kateri prostor se bo avtoriziral. Za tem odjemalec vnese svoje uporabniško ime in geslo in ponovno pošlje zahtevo strežniku. Tokrat se glavi SOAP doda element `BasicAuth`, v katerem sta elementa `Name` in `Password`, katera hranita uporabniško ime in geslo. Strežnik sedaj le še preveri poverilnico in v primeru, da je odjemalec avtenticiran, nadaljuje s procesiranjem zahteve.

Avtentikacija z izvlečkom (angl. "Digest Authentication") je sprva razširjala osnovno avtentikacijo, kasneje pa so jo dopolnili in tako sedaj predstavlja svoj avtentikacijski mehanizem. Mehanizem prav tako deluje po protokolu izziv – odgovor, kjer pa za dodatno varnost strežnik posreduje niz, katerega imenujemo "nonce" in se uporablja kot dodatni parameter pri izračunu izvlečka. Za izračun izvlečka mehanizem uporablja algoritem MD5. V izračun izvlečka se vzame uporabniško geslo ter nonce. Ta izvleček se nato pošilja v sporočilu in ne tako kot pri osnovni avtentikaciji, kjer se je pošiljalo kar uporabniško ime in geslo. Geslo uporabnika hrani tudi strežnik, kateri lahko s pomočjo niza nonce sam izračuna izvleček ter ga primerja z izvlečkom, ki se nahaja v sporočilu. V kolikor se izvlečka ujemata, se sklepa, da je uporabnik pravi, saj ima isti nonce in geslo. Tokrat v sporočilo SOAP ob izzivu posreduje strežnik znotraj elementa `Challenge` elemente `Status`, `Nonce` in `Realm`. V telo sporočila SOAP pa element `Fault`, tako kot pri osnovni avtentikaciji. Odjemalec nato kot odgovor v zahtevku posreduje element `ClientAuth`, ki se nahaja v glavi sporočila SOAP in vsebuje `Nonce`, `Auth`, `UserID`, `Realm` in `ClientNonce`. Vsi elementi so obvezni, razen zadnji, ki je opcijski. Element `Auth` je tipa hexBinary in vsebuje izračunani izvleček MD5. Opcijski element `ClientNonce` je niz, ki ga odjemalec posreduje strežniku, kadar se zahteva obojestranska avtentikacija. V kolikor je uporabnik avtenticiran, strežnik odgovori s sporočilom SOAP, v katerem ima v glavi element `NextChallenge`. V njem so elementi `Status`, `Nonce`, ter opcijska `ClientNonce` in `ServerAuth`. Element `ServerAuth` je odgovor na zahtevo po avtentikaciji strežnika, v kolikor imamo dodan opcijski element `ClientNonce` [31].

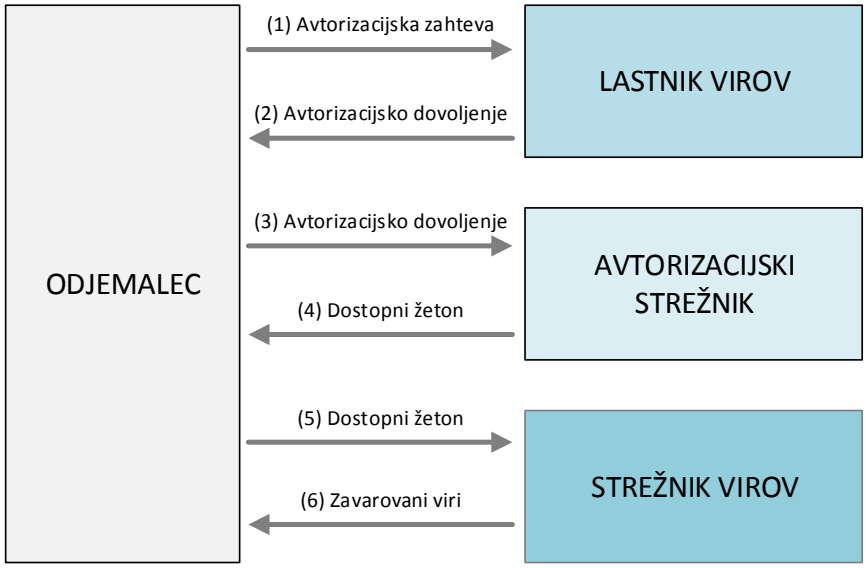
Spletne storitve REST privzeto uporabljajo HTTP, za večjo varnost pa lahko nastavimo uporabo protokola HTTPS ter uporabimo enega od načinov avtentikacije uporabnika. Prav tako kot pri načinu SOAP imamo tudi pri načinu REST možnost osnovne avtentikacije ter avtentikacije z izvlečkom. Na voljo pa so nam tudi odprti standard za avtorizacijo OAuth, pristop avtentikacije s piškotki (angl. "Cookie-Based") in podobno. Obstajajo prednosti in slabosti vsakega mehanizma, načrtovalec se mora odločiti kateri varnostni mehanizem bo uporabil na podlagi problema in nevarnostim, ki lahko pretijo aplikaciji.

V zadnjem času so veliko pozornosti dobile spletne storitve REST. Veliko zaslug gre njihovi enostavnosti, še večji pa velikemu pojavu mobilnih naprav, katerim je spletna storitev REST bolj pisana na kožo. Tudi spletne storitve v oblaku povečini temeljijo na spletnih storitvah REST, ki pa za avtorizacijo z drugimi vmesniki in aplikacijami velikokrat uporabljajo ogrodje OAuth. **OAuth** je odprti standard za avtorizacijo. Trenutna se uporablja različica OAuth 2.0. Gre za ogrodje za avtorizacijo, ki omogoča aplikacijam tretjih strank (angl. "Third-Party") dostop do omejenih virov. OAuth opisuje tok za avtentikacijo uporabnika in nato avtorizacijo uporabnika za dostop do želenih virov. Deluje drugače kot klasični avtentikacijski modeli, saj ne ponudi istih poverilnic, kot jih ima strežnik ali aplikacija. Odjemalec preko avtentikacijskih storitev pridobi dostopni žeton (angl. "Access Token"), ki vsebuje informacije za pridobitev dostopa. Ta žeton ne vsebuje uporabniškega imena in gesla. OAuth deluje preko protokola HTTP in specificira štiri vloge: lastnik virov, strežnik virov, odjemalec in avtorizacijski strežnik. Postopek, kot je prikazan na naslednji sliki (slika 3), poteka tako, da odjemalec posreduje avtorizacijsko zahtevo lastniku virov (1), ta pa preko postopkov odgovori z avtorizacijskim dovoljenjem (2). Odjemalec nato zahteva dostopni žeton tako, da posreduje dovoljenje avtorizacijskemu strežniku (3), kateri pa preveri dovoljenje in avtenticira uporabnika. V pozitivnem primeru mu vrne dostopni žeton (4). Ta žeton in zahtevo po viru nato odjemalec posreduje strežniku z viri (5), ki žeton preveri in v pozitivnem primeru vrne zahtevan vir (6) [35].

Windows Azure ponuja možnost aktivnega imenika v oblaku. Gre za storitev **Azure Active Directory**. Gre za podobno storitev, kot smo jo vajeni iz dosedanjih aktivnih imenikov, le da je ta gostovan v oblaku in ponuja visoko razširljivost in visoko dosegljivost. Tako lahko v aktivni imenik (angl. "Active Directory") v oblaku vnesemo celotno organizacijo ali samo določene uporabnike. Pri gradnji aplikacije poskrbimo za povezavo z aktivnim imenikom in tako dobimo visoko razpoložljivo storitev. Poleg Azure Active Directory imamo tudi možnosti uporabe klasičnega aktivnega imenika. Tega namestimo v Azure Virtual Machine in ga na podoben način povežemo z aplikacijo. Azure Active Directory omogoča tudi sinhronizacijo z lokalnim aktivnim imenikom, če tega tudi uporabljamo. V kolikor želimo uporabiti Azure Active Directory preko naše aplikacije, je potrebno aplikacijo najprej registrirati v aktivni imenik s pomočjo upravljavskega portala. Prav tako je možno tudi programsko dostopati do aktivnega imenika preko REST API-jev storitve Azure AD Graph, ki se tudi uporablja za integracijo aktivnega imenika z aplikacijami. Šele tako lahko v aplikacijo prenesemo podatke o uporabniku.

Microsoft s pomočjo sodobnih tehnologij oblaka in spletnih storitev ponuja avtentikacijske storitve preko katerih lahko avtenticiramo uporabnika. Tako se programerju ni potrebno lotevati implementacije svojih mehanizmov, ki bi bili drugače kompleksni in časovno potratni. Microsoft je prenovil storitve oblaka in ponuja razvijalcem storitve za omogočanje varnosti v svojih aplikacijah. V ospredje prinaša spremenjeno različico storitve Access

Control Service (ACS) imenovane Azure AD Access Control. Poleg omenjenega se uporablja še Windows Identity Foundation (WIF), ki močno olajša uveljavitev varnostnih mehanizmov. Omogočena je tudi storitev Azure Shared Access Signatures (SAS), ki pa je namenjen omejevanju dostopa do virov, shranjenih v Azure Blob Storage. Storitev oblaka **Azure AD Access Control** (bivši ACS) omogoča preverjanje identitete uporabnika ter pridobitev njegove identitete preko ponudnikov identitet (angl. "Identity Providers"), kot so to Microsoft, Yahoo, Google, Faceboook. Omogoča tudi identifikacijo uporabnika s pomočjo aktivnih imenikov Azure Active Directory ter Windows Server. S pomočjo omenjenih tehnologij lahko v aplikacijah vklopimo enotno prijavo (angl. "Single Sign-On") [34].



Slika 3: Ogrodje mehanizma OAuth.

Access Control omogoča pridobitev žetona ponudnika identitet. Ti žetoni niso standardizirane oblike in se razlikujejo od ponudnika do ponudnika. Azure AD Access Control vsebuje mehanizem za preoblikovanje teh žetonov v svoj žeton, ki je tudi enotne oblike za vse aplikacije Windows Azure. Ta žeton se nato posreduje aplikacijam oz. storitvam, katere iz njega izvlečejo identifikacijske podatke o uporabnika. V kolikor uporabljamo storitve Windows Azure za avtentikacijo uporabnika, se uporabniku ob avtentikaciji odpre posebno okno, ki je del storitev Windows Azure in ponudi možnost enotnega prijavnega mesta. Mehanizem, ki ga uporablja Windows Azure za pridobitev žetona od ponudnikov identitet in nato preoblikovanje tega žetona v svojega, je na strani Microsofta in povsem neodvisen od nas. S tem so šibko sklopili našo aplikacijo z njihovim mehanizmom in nas razbremenili vzdrževanja le-tega. Nas tako zanima le žeton, ki ga v aplikacijo pripelje storitev Access Control in s katerim mi naprej operiramo. Žeton je sestavljen iz trditev (angl. "Claims"), katere vsebujejo podatke o avtenticiranem uporabniku. Med drugim vsebujejo ime uporabnika, e-naslov, identifikacijski niz, kdo ga je ustvaril itd. Žetoni so lahko različnih oblik oz. načinov zapisa. Trenutno najbolj praktičen je JWT (JSON Web Token), ki je še posebej razširjen zaradi svoje enostavnosti in majhnosti. Zapisan je kot objekt JSON, s

katerim tudi lažje manipulirajo mobilne naprave. Drugi bolj razširjen žeton je SAML, ki pa je zapisan v formatu XML. Ima bolj strogo obliko, saj mu protokoli narekujejo kako mora biti strukturiran. Je neodvisen od platforme. Prav tako je večji in kompleksnejši kot JWT. Zaradi načina zapisa je manj primeren za mobilne naprave [30]. Poleg omenjenih, Windows Azure podpira še format SWT oz. Simple Web Token. V njem so informacije o uporabniku shranjene v zbirki ključ-vrednost. Žetoni v grobem uporabljajo algoritem HMAC SHA256, s ključem, ki ga poznata tako pošiljatelj zahteve kot prejemnik. Izračunane vrednosti se nato zakodira s pomočjo kodiranja Base64. Kot vidimo, obstaja velik nabor različnih tipov dostopnih žetonov. Odvisno od izbranega tipa žetona in protokola oz. mehanizma za avtentikacijo in avtorizacijo, obstaja tudi različno število načrtovalskih vzorcev za njihovo implementacijo. Primer takšnega vzorca je vzorec dostopnega posrednika [30].

4.2.3. Razširljivost

Razširljivost (angl. "Scalability") aplikacije je sposobnost učinkovitega izvajanja aplikacije ne glede na spremembo števila zahtev ali uporabnikov, ki uporabljajo aplikacijo. V kolikor se število zahtev poveča, se izvede tudi potrebna razširitev in obratno. Aplikaciji se torej dodeli nove vire in poskrbi za učinkovito izrabo le-teh. Največkrat gre za dodeljevanje virov v obliki trde opreme (dodeljevanje nove procesorske moči, pomnilnik, podatkovnega prostora, …), lahko pa je razširitev tudi programske narave. Poznamo dva načina razširjanja aplikacije:

- **Horizontalno** (angl. "Horizontal Scaling"): razširjanje lahko imenujemo tudi razširjanje navzven, kjer dodajamo nova vozlišča (angl. "Node") oz. vire sistemu. Tukaj ne razširjamo vsakega vozlišča posebej, temveč dodajamo zraven nova vozlišča, na primer dodamo nov strežnik obstoječim. Gre za bolj kompliciran obliki razširjanja.
- **Vertikalno** (angl. "Vertical Scaling"): razširjanje lahko imenujemo tudi razširjanje navzgor, s čimer je mišljeno da dodajamo vire vsakemu delu obstoječega sistema. Tako na primer dodajamo pomnilnik določenemu strežniku. Takšno razširjanje je enostavnejše, vendar ima svoje omejitve, saj npr. ne moremo strežniku v nedogled dodajati pomnilnike.

Razširjanje v oblaku se bolj osredotoča na horizontalno razširjanje, kjer dodajamo nova vozlišča, ki so dejansko navidezni sistemi (angl. "Virtual Machines"). Oblak poskrbi za to, da so vsa nova vozlišča opremljena z isto sliko sistema (angl. "Virtual Machine Image") ter da ponujajo iste storitve. Razširljivost posledično pomeni višje stroške.

Sistem je homogen, kadar so vsa vozlišča, ki podpirajo določeno funkcijo enako nastavljena. Tak sistem olajša horizontalno razširjanje, saj lahko izenačevalnik obremenitve enostavno preusmerja zahtevke. V kolikor sistem ni homogen, je to delo bolj kompliciran. Prav tako vozlišča istega tipa ne potrebujejo komunicirati med seboj, da bi opravila svoje delo. Delo izvajajo neodvisno od drugega vozlišča, s čimer se zagotovi avtonomnost sistema. Pri zasnovi

je potrebno v sistemu paziti na ozka grla (angl. "Bottleneck"), ki lahko resno ogrozijo naše zmogljivosti. Primer ozkega grla je sistem, ki je sicer sposoben razširjanja, a ima počasno internetno povezavo. Takšne stvari ogrozijo zmogljivosti sistema, zato je potrebno poskrbeti za enotnost – ali razširiti vse dele sistema ali pa zmanjšati hitrost delovanja ostalim virom. Poiskati moramo kje so naša ozka grla, katere dele sistema moramo prav tako narediti razširljive in ustrezno spremljati izvajanje ter ukrepati. Pri razširjanju je dobro vedeti kolikšen del novih vozlišč bomo dodali oz. t.i. razširitveno enoto (angl. "Scaling Unit"). Ta nam pove, pri koliko novih uporabnikih / zahtevah po virih bomo dodali nova vozlišča. V oblaku za dodajanje in odstranjevanje vozlišč skrbi avtomatsko razširjanje.

Izenačevanje obremenitve (angl. "Load Balancing") je način porazdelitve bremena med računalniškimi viri (strežniki, vrstami itd.). Namen je porazdeliti uporabniške zahteve, s čimer se optimizira poraba virov in zmanjša odzivni čas. Prav tako se s pomočjo izenačevalnika izognemo preobremenitvi samo enega strežnika. Izenačevalnik obremenitve v Windows Azure se imenuje Traffic Manager. Trenutno ima izenačevalnik obremenitve tri metode po katerih lahko deluje [33]:

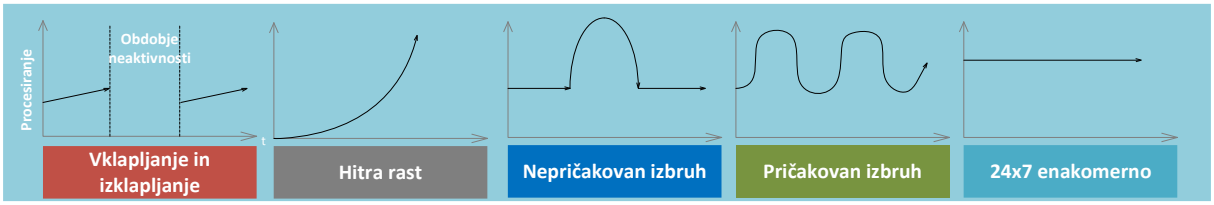
- **Performance**: to metodo izberemo, kadar imamo končne točke na različnih geografskih lokacijah in želimo uporabnikom dodeliti njim najbližjo končno točko, s čimer zmanjšamo latenco.
- **Round-Robin**: porazdeli breme enakomerno med vsemi določenimi končnimi točkami.
- **Failover**: izberemo, v kolikor imamo glavno končno točko, vseeno pa določimo drugo končno točko, v kolikor glavna točka ni dosegljiva.

Z različnimi načini preusmerjanja uporabnika strežnikom je potrebno dobro načrtovati kako bomo upravljali s stanjem seje uporabnikov. Vozlišča je tako potrebno odrešiti naloge hranjenja stanja seje in hrambo stanja raje prenesti uporabniku v piškotek (angl. "Cookie"). V kolikor je piškotek premajhen za vse podatke lahko uporabimo hrambo podatkov kot je Azure Table Storage oz. kakšno drugo podatkovno bazo NoSQL. Tam lahko hranimo podatke o stanju seje, uporabniku pa v piškotek zapišemo le identifikacijski ključ za dostop do teh podatkov. Ob vsaki zahtevi lahko s tem ključem pridobimo podatke o stanju seje. Tako vidimo, da metode lepljivih sej ne potrebujemo in da je metoda Round-Robin zadovoljiva.

Aplikacije in storitve, ki so v oblaku, se s pomočjo izenačevalnika obremenitve lažje odzovejo na različne obremenitve. Najbolj standardne obremenitve aplikacije so sledeče (Slika 4):

- **Vklapljanje in izklapljanje** (angl. "On and Off"): kjer storitev vklopimo za določen čas, da se neka operacija izvede, nato ugasnemo. Takšen postopek ponavljamo. Primer so aplikacije za procesiranje velike količine podatkov v kratkem času.
- **Hitra rast** (angl. "Growing Fast"): kjer se zelo hitro povečuje število zahtevkov. Primer so "start-up" aplikacije.

- **Nepričakovan izbruh** (angl. "Unpredictable Bursting"): kadar imamo enakomeren potek izvajanja, nato pride do visokega in nepričakovanega povišanja prometa. Primer takšne obremenitve so spletne igrice, katerim po določenem času upade promet.
- **Pričakovan izbruh** (angl. "Predictable Bursting"): kadar smo pripravljeni, da bo število zahtevkov ob določenih dnevih zrastlo. Primer so božični prazniki ali kampanjske akcije.
- **24x7 enakomerno** (angl. "24x7 Steady"): enakomerno obremenjeni viri.



Slika 4: Obremenitve aplikacije.

Spremljanje in merjenje izvajanja aplikacije in posledično virov je poglavitni del razvoja v oblaku, saj je potrebno spremljati stanje in delovanje vozlišč, instanc aplikacije itd. V upravljavskem portalu tako dobimo vse informacije tudi v grafični obliki o delovanju aplikacije in ostalih storitvah Windows Azure. Vsaka storitev se nahaja v svojem zavihku, znotraj njega pa imamo zavihek Monitor, s čimer dobimo vpogled v delovanje aplikacije. Graf se osvežuje na pet minut, prikaz pa je mogoč tudi za 7 oz. 14 dni nazaj, odvisno od storitve. Vsaki storitvi oz. aplikaciji lahko dodamo tudi metrike za spremljanje delovanja. Metrike imajo tako grafično kot tudi tekstovno opisano podatke o minimalnem in maksimalnem dosežku metrike, njenem povprečju in skupno. Metrikam lahko nastavimo tudi pravila za obveščanje. Nastavimo lahko pravilo, da nas Windows Azure obvesti, v kolikor je določena metrika presegla/padla nad/pod določenim številom. Obvestilo se lahko pošlje po e-pošti izbranemu upravljalcu. Do podatkov o izvajanju lahko prav tako dostopamo preko programske kode s pomočjo Windows Azure Diagnostics. Ta nam omogoča, da pridobimo diagnostične podatke iz spletne ali delovne vloge in spremljamo zmogljivosti, porabo virov, pretok prometa ipd. Še posebej učinkoviti so diagnostični podatki pri testiranju aplikacije.

Veliko vlogo pri razširjanju v Windows Azure oblaku ima tudi storitev **Azure Content Delivery Network** (Azure CDN). Gre za rešitev, specifično za oblak, kjer lahko predpomnimo statično vsebino in blobe v fizična vozlišča po različnih lokacijah po svetu [12]. S tem omogočimo boljšo odzivnost v primeru nenadnega povišanja prometa ter boljšo uporabniško izkušnjo za odjemalce, ki so daleč od Microsoftovih podatkovnih centrov, saj promet ne potuje daljše razdalje. Trenutno ima Microsoft skoraj 30 vozlišč CDN postavljenih po različnih lokacij po svetu, ta številka pa se bo še povečala. CDN deluje tako, da vsaka zahteva po viru, ki je narejena s pomočjo sintakse CDN, vzame predpomnjen vir (blob ali stran) iz najbližje lokacije CDN. V kolikor je ta zahteva po viru prvič narejena in/ali vira ni

shranjenega na tej lokaciji, se vir naloži in predpomni s pomočjo ustrezne storitve, npr. storitve za blob. Prav tako se nastavi metapodatek, ki skrbi, da vir po določenem času zastari. Razširljive lahko naredimo tako spletne kot delovne vloge, tako storitve REST kot SOAP. Res pa je, da je REST v osnovi bolj nagnjen k enostavnosti in razširljivosti, zato je bolj primeren za razširljive storitve.

4.2.4. Hranjenja stanja

Spletne storitve lahko hranijo podatke o stanju med odjemalcem in strežnikom ali pa tega stanja ne hranijo. Tukaj govorimo o spletnih storitvah s hranjenjem stanja (angl. "Stateful") in brez hranjenjem stanja. Kljub temu, da govorimo o hranjenju stanja v spletnih storitvah, se je potrebno zavedati, da se dejansko stanje hrani na strežniku. SOAP je privzeto storitev s hranjenjem stanja, REST je po teoriji storitev brez hranjenja stanja, s čimer omogoča veliko mero razširljivosti storitve. Oba tipa spletnih storitev lahko z nekaj programske kode pripravimo v obraten način delovanja. Tudi protokol HTTP, na katerem temeljijo spletne storitve, ne hrani stanja, saj se vsaka zahteva obravnava individualno in neodvisno od prejšnje.

Oblak Windows Azure nima nikakršnih problemov z delovanjem storitev s hranjenjem stanja ali brez hranjenja stanja. Storitev preprosto namestimo v eno od vlog. Vprašanje je, kaj potrebujemo pri svoji storitvi ter kakšno jo želimo narediti. Oblak ima možnost razširjanja vozlišč, v kolikor pride do povišanja zahtevkov oz. prometa. V ta namen ima tudi izenačevalnik obremenitve. Spletne storitve morajo biti zmožne delovati normalno, v kolikor se doda novo vozlišče. V primeru, da se določeno vozlišče ugasne, morajo storitve normalno delovati naprej. Podatki o uporabnikih se ne smejo izgubiti. Zaradi tega je večji poudarek na storitvah brez hranjenja stanj.

Storitve brez hranjenja stanj so bolj razširljive, saj izenačevalnik obremenitve ne potrebuje posebne metode za preusmerjanje odjemalca pravemu vozlišču. Prav tako vsakič, ko zaženemo novo vozlišče oz. instanco, ni potrebno skrbeti za začetna stanja. Storitve brez hranjenja stanja so tudi bolj enostavne, saj se nam ni potrebno ukvarjati s stanji. Posledično imamo več prostega pomnilnika na ta račun, kajti nanj ne potrebujemo shranjevati stanja odjemalcev. Enostavnejša so tudi z vidika programiranja in lažjega predpomnenja. Hranjenje stanja naj se prestavi na druge nivoje spletne aplikacije, npr. v predstavitveni nivo aplikacije, natančneje v uporabniške piškotke. Večje zaledne storitve, ki uporabljajo vrste ali veliko procesiranja naj vseeno hranijo stanja, saj se s tem zmanjša število zahtevkov med sistemi in manj trošijo viri.

4.2.5. Življenjski cikel vlog

Razložili smo že, da sta v oblaku Windows Azure dve vlogi, ki skrbita za izvajanje naših storitev. To sta spletna in delovna vloga. Spletna se od delovne razlikuje v tem, da ima nameščen IIS, kateri poganja naše storitve. Delovna vloga tega nima. Obe vlogi imata svoj življenjski cikel, kateri je definiran s pomočjo razreda `RoleEntryPoint`. Ta razred je za spletno vlogo neobvezen, medtem ko ga delovna vloga obvezno ima. Spletno vlogo lahko razširimo s tem razredom in dobimo nekakšno hibridno vlogo. V spletni vlogi strežnik IIS poskrbi za ASP.NET življenjski cikel.
`RoleEntryPoint` je razred, ki je sestavljen iz treh metod, ki definirajo življenjski cikel vlog:

- **OnStart**: izvede programsko kodo, ki vzpostavi instanco vloge. Vrača `true`, če je inicializacija uspela, drugače `false`.
- **Run**: izvede programsko kodo, ki je namenjena izvajanju v času žive instance vloge. Metoda ne vrača ničesar.
- **OnStop**: izvede programsko kodo, ki je namenjena ob ugasnitvi instance vloge. Metoda ne vrača ničesar.

Življenjski cikel spletnih vlog je malo drugačen od delovnih vlog. Tu se najprej požene IIS konfigurator (angl. "IIS Configurator"), ki nastavi IIS. Nato se izvedejo naše zagonske naloge in za njimi ponovno IIS konfigurator. Tokrat se do konca nastavijo aplikacijski bazeni ter naša spletna aplikacija. Za tem se pokliče razred `RoleEntryPoint`, ki je opcijski v spletnih vlogah. Nazadnje se zažene IIS strežnik in naša aplikacija. V delovni vlogi moramo ob ugašanju zagotoviti dovolj časa, da se programska koda pravilno zaključi. V ta namen v metodo `OnStop()` vstavimo zamik oz. čakanje. Spletna vloga lahko svojo programsko kodo, ki se bo izvedla ob ugasnitvi instance vloge vključi tudi v ASP.NET življenjski cikel `Application_End`, katera se pokliče še preden se pokliče metodo `OnStop()`. Na ta način lahko po potrebi pravočasno ugasnemo ali shranimo stvari.

V okviru življenjskega cikla vloge lahko dodamo tudi zagonsko nalogo (angl. "Startup Task"). Zagonske naloge se izvršijo še pred `OnStart()` metodo oz. zagonom instance vloge. Takšne naloge so primerne za namestitev komponent, zagon programov na navideznem strežniku, zagon dlje trajajočih procesov ipd. Znotraj vloge imamo lahko več zagonskih nalog, ki se bodo izvedle zaporedno. Zagonske naloge so lahko različnega tipa:

- **Simple**: izvaja se zaporedno, s čimer zaustavi zagon instance vloge.
- **Foreground** in **Background**: se zaženeta vzporedno od `OnStart()` metode, ki poskrbi za zagon instance vloge. Na ta način ne blokiramo zagona instance vloge.

4.2.6. Vzorec: varnost razširljivih spletnih storitev

4.2.6.1. Ime vzorca

Varnost razširljivih spletnih storitev.

4.2.6.2. Namen

Hitro spreminjanje tehnologije in porast mobilnih naprav zahteva tudi integracijo le-teh s storitvami, ki so razširljive, varno dostopne, predvsem pa enostavnejše za izgradnjo. Spletne storitve v oblaku so javno dostopne, vseeno pa določene ne želimo ponuditi odprti javnosti, zato jih zavarujemo pred nepooblaščenimi dostopi. V kombinaciji z drugimi storitvami želimo izdelati REST spletno storitev, ki za delovanje v oblaku uporablja spletno vlogo ter avtorizacijske in avtentikacijske mehanizme.

4.2.6.3. Prav tako poznano kot

Varnost spletnih aplikacijskih programskih vmesnikov.

4.2.6.4. Motivacija

Šibko sklopljene spletne aplikacije v veliki meri uporabljajo za svoje delo spletne storitve. Vendar pa spletne storitve z objavo v oblaku naredimo ranljive pred nepooblaščeno uporabo. Veliko podjetij že uporablja aktivni imenik za hranjenje informacij o zaposlenih, kar omogoči tudi osrednjo točko za avtentikacijo in avtorizacijo. Poslovne aplikacije lahko sedaj izkoristijo aktivni imenik v oblaku. Zgradili bomo spletno storitev REST na način, da bomo uporabnika avtenticirali s pomočjo avtentikacijskih storitev oblaka povezanih z Azure Active Directory. S pomočjo ogrodja OAuth 2, bomo zgradili postopek avtentikacije in avtorizacije.

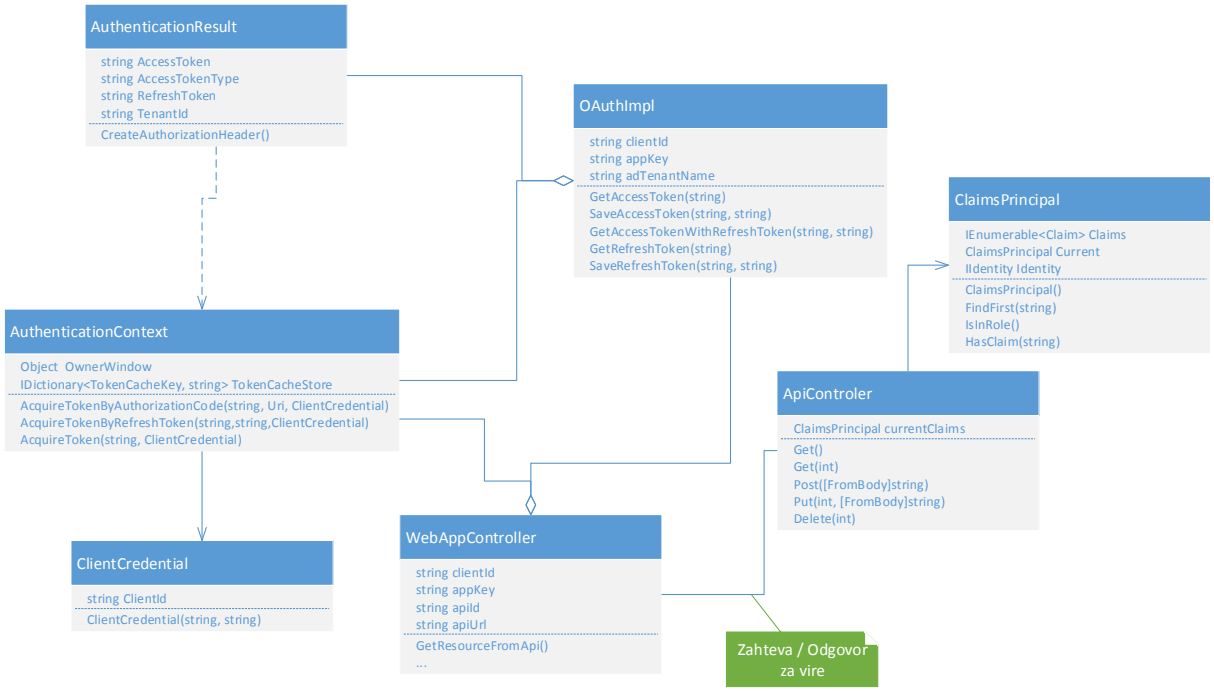
4.2.6.5. Uporaba

Uporabite ta vzorec, v kolikor želite:

- Zgraditi spletno storitev, ki bo razširljiva, šibko sklopljena in ne bo hranila stanja.
- Zgraditi storitev, ki bo omogočala avtentikacijo preko avtentikacijskih storitev Azure Active Directory.
- Avtenticirati uporabnika še preden izvede zahtevek za storitev.
- Zagotoviti avtoriziran dostop do spletnih storitev.

4.2.6.6. Struktura

Slika 5 prikazuje strukturo načrtovalskega vzorca za avtentikacijo in avtorizacijo uporabniškega dostopa do spletnih storitev. Krmilnik v spletni aplikaciji razširimo z ustreznimi spremenljivkami, katere so potrebne za nadaljnjo delo. WebAppController za avtentikacijo uporablja OAuthImpl razred, ki omogoča shranjevanje in posodobitev dostopnih žetonov ter AuthenticationContext, ki se uporablja za avtentikacijo uporabnika preko spletnih storitev oblaka Windows Azure. Za delovanje potrebuje razred ClientCredential, v katere se shranijo uporabniške poverilnice. AuthenticationContext vrne rezultat avtentikacije v obliki razreda AuthenticationResult. Pri dostopu do spletnih storitev se v glavi zahtevka posreduje dostopni žeton, katerega ApiControler pregleda in na podlagi le-tega dovoli oz. zavrne dostop. Podatke o uporabniku lahko pridobimo s pomočjo razreda ClaimPrincipal.



Slika 5: Struktura vzorca za avtentikacijo in avtorizacijo.

4.2.6.7. Udeleženci

Glavni udeleženci vzorca so:

- **WebAppControler**: izpeljuje razred Controller in je del aplikacije MVC, ki pošlje zahtevo po viru do spletne storitve.
- **AuthenticationContext**: predstavlja razred AuthenticationContext in vsebuje glavne metode za avtentikacijo uporabnika. Implementira prijavno okno za avtentikacijske storitve Windows Azure.
- **AuthenticationResult**: predstavlja razred AuthenticationResult in vsebuje rezultat avtentikacije prej omenjenega konteksta.

- **OAuthImpl**: je lasten razred, ki omogoča hranjenje žetonov in pošiljanje ponovne zahteve po žetonih.
- **ApiController**: izpeljuje razred ApiController in predstavlja našo spletna storitev, ki zahteva dostopni žeton za avtorizacijo uporabnika.
- **ClaimsPrincipal**: podpira več identitet tipa Claims.

4.2.6.8. Sodelovanja

WebAppController deklarira in inicializira AuthenticationContext, kateremu poda naslov v obliki niza. Ta omogoči prijavno okno za avtentikacijo uporabnika in kot rezultat vrne AuthenticationResult, ki vsebuje med drugim tudi dostopni žeton. Razred OAuthImpl skrbi za shranjevanje teh žetonov v sejo uporabnika. Omogoča tudi pridobivanje žetonov iz seje oz. posreduje ponovno zahtevo za žeton, v kolikor je to potrebno. Dostopni žeton se uporablja za dostop do spletne storitve ApiController, katera avtorizira uporabnika in shrani njegove poverilnice v razred ClaimsPrincipal.

4.2.6.9. Posledice

Prednosti uporabe tega vzorca so naslednje:

- Enostavna izgradnja mehanizmov za avtentikacijo in avtorizacijo za potrebe spletne storitve.
- Uporablja JWT žetonov, ki so primernejši za mobilne naprave.
- Enkratna prijava do izteka življenjske dobe žetona.

Slabosti tega vzorca pa so:

- Uporablja le avtentikacijo uporabnika preko Azure Active Directory.
- Žetone hrani v seji uporabnika, kar bi se dalo izboljšati s shranjevanjem v bazo.
- Vseeno obstajajo varnostni pomisleki in grožnje, kot so napadi z ribarjenjem (angl. "Phishing") ali medspletno ponarejanje zahtev (angl. "Cross-site Request Forgery").

4.2.6.10. Implementacija

Za uspešno implementacijo je potrebno imeti Windows Azure naročnino ali vsaj naročnino na Azure Active Directory. Testiranje implementacije je možno lokalno, tako da vsaj sprva ni potrebno projekt oz. spletno storitev prenesti v oblak. V Azure Active Directory je potrebno registrirati uporabnike in aplikacijo. Tako spletna aplikacija kot tudi spletna storitev lahko komunicirata z aktivnim imenikom samo, kadar sta tudi registrirani na portalu Windows Azure. Aplikacije imajo lahko različen dostop do aktivnega imenika. Za zagotavljanje komunikacije med aplikacijami, pa je potrebno spletni storitvi omogočiti deljene pravice z ostalimi aplikacijami preko manifest datoteke.

4.2.6.11. Vzorčna koda

Primer vzorčne kode bo prikazan v poglavju 5.2.

4.2.6.12. Znani primeri uporabe

Najbolj odmevni primeri uporabe tega vzorca so storitve Office 365, ki uporabljajo aktivni imenik ter organizacije z internimi poslovnimi aplikacijami, katere za avtentikacijo in avtorizacijo uporabnika pri dostopu do virov uporabljajo aktivni imenik.

4.2.6.13. Sorodni vzorci

Vzorec avtentikacijskega posrednika, vzorec upravljanja z viri.

4.3. Obdelovanje uporabniških zahtev z uporabo sporočilnih vrst

Sporočilne vrste so v računalništvu poznane že daljše obdobje in vsekakor niso novost v tehnologiji oblaka. Omogočajo asinhrono komunikacijo, saj pošiljatelj ne potrebuje biti v interakciji s prejemnikov v času pošiljanja sporočila. V ta namen se poslano sporočilo shrani v sporočilno vrsto in čaka prejemnika, da ga prevzame.

Windows Azure pozna dva tipa sporočilnih vrst. Prvi je predstavljen v okviru Azure Storage Services (Windows Azure Queues), drug pa v okviru storitvenega vodila (Azure Service Bus Queues). Medtem ko obe storitvi ponujata sporočilne vrste, pa se razlikujeta v delovanju.

4.3.1. Windows Azure Queues

Windows Azure Queues je Microsoft predstavil kot prvo rešitev za sporočilne vrste za oblak. Gre za mehanizem sporočilnih vrst v okviru storitev Windows Azure Storage. Omogoča zanesljivo in obstojno sporočanje med storitvami. Obstojno sporočanje (angl. "Persistent Messaging") pomeni, da se sporočila nekam shranjujejo, največkrat v podatkovno bazo. Tako tudi, če se posrednik (angl. "Broker") izklopi, sporočil ne izgubimo.

Azure Queues ponuja vmesnik, kateri temelji na metodah REST. Tako lahko do sporočil dostopamo od koderkoli preko protokola HTTP ali HTTPS z avtenticiranim zahtevkom. Največja velikost sporočil v Azure Queue je 64KB. V kolikor imamo podatek ali vir, ki je večji kot omenjena velikost, potem je priporočljivo, da se za shranjevanje uporabi Blob Storage ali Table Storage in v sporočilo vstavimo povezavo do tega vira. Sporočila se v Azure Queue dodajajo na konec, medtem ko se berejo iz začetka vrste. Takšno delovanje je značilno za razvrščanje FIFO (First In, First Out). Vseeno pa pri Azure Queue razvrščanje FIFO ni

popolnoma zagotovljeno, saj se lahko zgodi, da se strežnik, ki procesira sporočilo, ugasne. Časovni okvir zaklenjenega sporočila nato poteče, nakar se sporočilo vrne na konec vrste in tako izgubimo razvrščanje FIFO [36].

4.3.2. Storitveno vodilo Windows Azure

Po nekaj spremembah in dodajanju novih funkcionalnosti storitev oblaka, je Microsoft izdal storitveno vodilo Windows Azure Service Bus. To storitveno vodila je podobno ostalim sporočilno usmerjenim vmesnim slojem (angl. "Message-Oriented-Middleware"). Omogoča sporočanje preko vrst, tem ali relejev. Prav tako storitveno vodilo omogoča pošiljanje potisnih sporočil preko storitve Notification Hub. Spodaj so razložene sporočilne entitete, ki so na voljo v okviru storitvenega vodila Windows Azure [48].

- **Vrste** (angl. "Queue"): omogočajo enosmerno komunikacijo med pošiljateljem in prejemnikom. Vsaka vrsta se obnaša kot posrednik, kamor je shranjeno sporočilo, dokler ga prejemnik ne prevzame. Deluje po metodi FIFO, torej se bodo sporočila prevzela po istem vrstnem redu, kot so prišla v vrsto. Z uporabo vrst omogočimo šibko sklopljenost, saj pošiljatelj in prejemnik ne komunicirata direktno, temveč preko sporočilne vrste. Zaradi tega lahko brez problemov enega ali drugega spremenimo ali nadgradimo, saj to ne bo vplivalo na drugega. Sporočilne vrste imajo dva možna načina prebiranja sporočil. Prvi je "Poglej in zakleni" (angl. "Peek & Lock"), kjer prejemnik posreduje zahtevo po sporočilu. Storitveno vodilo sporočilo zaklene, tako da ga drugi ne morejo brati in posreduje prvotnemu prejemniku. Prejemnik ga prebere in ob zaključku pokliče metodo `Complete`, katera omogoči, da storitveno vodilo označi sporočilo kot uporabljeno. Ta model je primernejši za sisteme, kjer se sporočila ne smejo izgubiti ali zbrisati, brez da bi jih zares obdelali. Drugi način prebiranja je "Prejmi in zbriši" (angl. "Receive & Delete"), kjer storitveno vodilo po prejeti zahtevi za branje označi sporočilo kot uporabljeno in ga posreduje prejemniku. Sporočilo se takrat odstrani tudi iz vrste. Gre za enostavnejši model, kateri je primeren za sisteme, kjer izguba kakšnega sporočila ne pomeni drastičnih problemov. Namreč, v kolikor se zgodi napaka pred ali med procesiranjem sporočila, le-tega ne moremo več dobiti nazaj, saj se je odstranil iz vrste.
- **Teme** (angl. "Topic"): so uporabne pri pošiljanju sporočil velikemu številu prejemnikov. S pomočjo naročnin (angl. "Subscriptions") in vzorca objavi – naroči se prejemniki prijavijo na temo. Tema ima lahko enega ali več prejemnikov. Prav tako ima lahko enega ali več pošiljateljev na to temo. Sporočilo, ki prispe v temo, se pošlje vsem prejemnikom, ki so naročeni na določeno temo. Tukaj se teme razlikujejo od vrst, kjer se sporočilo posreduje samo enemu prejemniku. Vseeno pa pošiljanje sporočil ne gre direktno iz teme, temveč preko naročnin. Naročnina na temo naredi navidezno vrsto, preko katere se pošiljajo sporočila. Ob naročninah lahko prav tako uporabimo pravila, s katerimi se sporočila filtrirajo in tako naročniki prejmejo le tista

sporočil, ki ustrezajo določenemu pravilu. Pravila se nanašajo na lastnosti sporočila, ki so shranjene v metapodatkih le-tega.

- **Rele** (angl. "Relay"): omogoča izgradnjo hibridnih aplikacij, ki tečejo tako v oblaku Windows Azure, kot tudi na lastni strežniški infrastrukturi. Omogoča izpostavitev lastnih storitev preko končnih točk javnega oblaka, brez da bi odpirali vrata v požarnem zidu. To doseže s pomočjo ogrodja WCF (Windows Communication Foundation) in protokola SOAP. Povezava je vzpostavljena preko protokola TCP in vrat SSL ali s pomočjo povezave HTTP.
- **Obvestila** (angl. "Notifications"): storitveno vodilo ponuja tudi možnost storitve za pošiljanje potisnih sporočil mobilnim napravam iz katerekoli zaledne aplikacije. Potisna sporočila so dostavljena mobilnim napravam na podlagi njihovega sistema za obveščanje, oz. Platform Notification Systems (PNS). Ta sistem je različen od operacijskega sistema mobilne naprave (Apple Push Notification Service, Windows Notification Service itn.). Storitev Notification Hubs, ki spada v storitveno vodilo, olajša postopek pošiljanja sporočil tako, da ponudi infrastrukturo za potisna sporočila za različne platforme (Android, iOS, Windows Phone in Windows).

4.3.3. Primerjava Azure Queue s storitvenim vodilom

Kot smo že videli, se Azure Queue in storitveno vodilo razlikujeta med seboj. Prednost storitvenega vodila pred Azure Queue je integracija z ogrodjem WCF. Prav tako poleg sporočilnih vrst omogoča še teme in rele. Storitveno vodilo je prav tako močno, v kolikor delamo šibko sklopljene aplikacije, saj se obnaša kot vmesni sloj med aplikacijo in storitvijo. V primeru neodzivne storitve sporočila počakajo in jih lahko kasneje, ko storitev popravimo, v pravilnem vrstnem redu (FIFO) storitev prevzame.

Kljub temu, da imata obe storitvi praktično isti namen, je med njima kar nekaj razlik. Naredimo še enkrat krajši pregled [36]:

- Azure Queue imajo neomejeno število vrst in neomejeno število istočasnih odjemalcev, medtem ko ima storitveno vodilo lahko 10.000 vrst in neomejeno število uporabnikov.
- Največja velikost Azure Queue je 100 TB, velikost storitvenega vodila pa le maksimalno 5 GB. Nastavitve se izbere ob izgradnji storitvenega vodila.
- Sporočilo se na Azure Queue hrani največ 7 dni, medtem ko se v storitvenem vodilu neomejeno. To je lahko tudi problematično, v kolikor sporočila ne brišemo sproti, saj lahko presežemo maksimalni prostor na vodilu.
- Storitveno vodila zagotavlja razvrščanje FIFO, medtem ko Azure Queue ne zagotavljajo nikakršnega razvrščanja. Posebej se to pokaže, ko poteče zaklop na sporočilu in je sporočilo ponovno na voljo.

- Storitveno vodilo ima dva načina prevzemanja sporočila, med drugim tudi Prejmi in zbriši, česar Azure Queue nimajo. V kolikor želimo izbrisati sporočilo, moramo pri Azure Queue za izbris sporočila narediti dodatni klic storitve.
- V Azure Queue je največja velikost sporočila 64 KB oz. 48 KB, v kolikor smo sporočilo kodirali z algoritmom Base64. Pri storitvenem vodilu je lahko sporočilo veliko do 256 KB, kar vključuje glavo in telo sporočila.
- Azure Queue uporabljajo REST preko protokola HTTP/HTTPS, medtem ko storitveno vodilo uporablja načina TCP preko TLS ali REST preko HTTPS.
- Za avtentikacijo uporablja storitveno vodilo Claims-e, Azure Queue pa simetrični ključ.
- Storitveno vodilo omogoča polno kompatibilnost z ogrodjem WCF in Windows Workflow Foundation. Azure Queue tega ne podpirajo.
- Povprečna latenca pri storitvenem vodilu je 100ms, pri Azure Queue pa 10ms.
- Oba imata pretočnost do 2.000 sporočil na sekundo.

Razlik je torej kar nekaj, tukaj so naštete le nekatere. Načrtovalec mora na podlagi želja in specifikacij izbrati primeren tip sporočilnih vrst. Vprašanje je, kakšen sistem je potrebno zgraditi, ali želimo aplikacijo razširjati, koliko hitro bodo sporočila prihajala v vrsto, njihovo velikost itd. Prav tako moramo računati tudi na življenjski čas sporočila v vrsti, v kolikor razvijamo procese z daljšim časovnim procesiranjem sporočil. V kolikor potrebujemo enostavnejšo komunikacijo in sporočanje med storitvami je bolje izbrati Azure Queue. Glede na želje in upoštevaje razlik naj razvijalec izbere želeno sporočilno vrsto.

4.3.4. Enosmerna komunikacija

Windows Azure ponuja različne tehnologije za sporočanje med storitvami, med drugim tudi storitveno vodilo in Azure Queue. Sporočilne vrste lahko uporabimo na različne način za komunikacijo med prejemnikom in pošiljateljem.

Enosmerno sporočanje (angl. "One-Way Messaging") je v sporočilnih sistemih najbolj pogost način komunikacije in prav tako tudi najenostavnejši. Deluje tako, da pošiljatelj pripravi sporočilo in ga pošlje v vrsto. Sporočilo čaka v vrsti, dokler ga prejemnik ne prevzame in obdela oz. mu poteče življenjski čas (angl. "Time-To-Live").

Sporočanje vsem (angl. "Broadcast Messaging") je sporočanje, kjer načeloma uporabljamo sporočilne teme, vendar se je potrebno zavedati, da vsaka naročnina na temo dejansko pomeni vrsto z nastavljenimi atributi, s katerimi se filtrirajo sporočila. Pošiljatelj posreduje sporočilo v temo, katera ima več naročnin. Na naročnino je lahko prijavljen eden ali več uporabnikov, kateri imajo svojo vrsto za prejem sporočila iz teme.

4.3.5. Dvosmerna komunikacija

Za vzpostavitev dvosmerne komunikacije (zahtevek / odgovor) je možnih več načinov.

- **Ločene vrste za pošiljatelje**: za vsakega pošiljatelja sporočila lahko vzpostavimo njegovo lastno povratno vrsto, v katero se bodo hranila povratna sporočila. Takšna možnost pride v poštev, ko imamo manj kot 10.000 pošiljateljev oz. manj kot 10.000 sočasnih uporabnikov, saj je toliko trenutna omejitev vrst na en imenski prostor storitvenega vodila. Od te vsote moramo odšteti tudi vse teme in releje, ki jih uporabljamo.
- **Uporaba sporočilnih sej**: možno je narediti sporočilno sejo (angl. "Message Session"), katera omogoča, da preko identifikacijskega ključa seje, prejemnik vrne eno ali več sporočil pošiljatelju. Ključ seje se pripne sporočilu kot njegova lastnost. Uporabljamo dve ločeni vrsti, v eno se pošiljajo vse zahteve, v drugo pa odgovori.
- **Hranjenje odgovora v bazi**: pošiljatelj pošlje sporočilo v skupno vrsto, iz katere ga vzame prejemnik in obdela. Rezultat procesiranja shrani v podatkovno bazo ali tabelo. Pošiljatelj redno pregleduje tabelo, v kolikor se tam nahaja njemu namenjen rezultat. Sporočilo v tabeli je opremljeno s korelacijsko identifikacijsko številko, ki jo je že v začetku pošiljatelj dodal poslanemu sporočilu.
- **Uporaba tem**: teme omogočajo dodajanje več pošiljateljev in več prejemnikov, ki lahko berejo sporočila iz iste naročnine. Prav tako lahko več prejemnikov prebere isto sporočilo. V kolikor sporočilo opremimo z metapodatki oz. še s korelacijsko identifikacijsko številko, omogočimo povratno komunikacijo in pravilno korelacijo s pošiljateljem. Pošiljatelj se lahko na temo naroči s korelacijskim filtrom, kateri omogoči prikaz njemu namenjenih sporočil. Za uspešno implementacijo takšnega načina komunikacije potrebujemo dve temi, eno za poslana sporočila in eno za prejeta. Prejemnik iz prve prevzame sporočilo in ga obdela, nato pa rezultat pošlje na drugo temo. Poslanemu sporočilu mora pripeti isti korelacijski ključ, kot ga je poslal pošiljatelj. Pošiljatelj čaka na sporočila z vklopljenim filtrom.
- **Sporočanje odgovora**: odgovor lahko sporočimo pošiljatelju tudi preko spletne vtičnice (angl. "Web Socket") oz. naredimo klic na spletno storitev z zahtevo HTTP. Zavedati se moramo, da takšen način povratne komunikacije ni najbolj zanesljiv, saj se lahko zgodi, da se na poti od prejemnika do pošiljatelja kakšno sporočilo izgubi. V ta namen je priporočljivo odgovor ločeno shraniti še v tabeli.

Izbira načina dvosmernega sporočanja je odvisna tudi od okolja. V kolikor imamo dve delovni vlogi, ki si med seboj pošiljata zahteve in odgovore, potem bodo teme ali sporočilne seje primerno opravljale potrebno delo. Drugačnega pristopa se moramo lotiti v primeru, da želimo odgovor sporočiti uporabniku, ki sedi za spletnim brskalnikom, saj so operacije čakanja na sporočilo precej potratne, tako časovno kot z viri. Glede na preizkušeno, se je v takšnem primeru najbolje lotiti procesiranja na strani prejemnika, ki je največkrat delovna

vloga in nato shraniti odgovor v tabelo. Iz tabele nato pošiljatelj lahko kadarkoli prebere rezultat procesiranja.

4.3.6. Spletna vtičnica

Spletna vtičnica je protokol, ki omogoča dvosmerno komunikacijo preko ene povezave TCP. Namenjen je izgradnji povezave med spletnim brskalnikom in strežnikom, kjer si lahko obe strani izmenjujeta podatke. Poleg protokola TCP spletna vtičnica za delovanje uporablja tudi protokol HTTP, katerega potrebuje za vzpostavitev povezave. Med rokovanjem (angl. "Handshake") odjemalec posreduje zahtevo, kateri v glavo pripne atribut `Upgrade`. S tem sporoči strežniku, da želi povezavo nadgraditi v WebSocket povezavo. Strežnik nato povezavo nadgradi, odjemalcu vrne odgovor in povezava je vzpostavljena. Nadaljnja komunikacija poteka preko povezave TCP in vrat 80.

SignalR ni del storitev javnega oblaka Windows Azure, vendar pa ponuja knjižnice, katere lahko olajšajo komunikacijo med strežnikom in odjemalcem [39]. SignalR je skupek knjižnic, ki so jih razvijalci združili in s tem omogočili enostavno in v realnem času, strežniško pošiljanje vsebine povezanim odjemalcem. SignalR podpira spletne vtičnice pri brskalnikih, ki podpirajo tehnologijo HTML5, pri starejših pa uporablja skupek programske kode JavaScript, s katerimi nadomesti tehnologijo. Prav tako omogoča več API-jev, preko katerih lahko upravljamo povezavo med odjemalci in strežnikom. Pomembno je predvsem to, da deluje v realnem času, kar pomeni, da strežnik takoj, ko je neka vsebina pripravljena, le-to potisne odjemalcem. Takšno sporočanje je primerno za primere povratne komunikacije, kot jo imamo v naslednjem vzorcu.

4.3.7. Vzorec: sporočilno obdelovanje zahtev

4.3.7.1. Ime vzorca

Sporočilno obdelovanje zahtev.

4.3.7.2. Namen

Oblak omogoča izgradnjo razširljivih aplikacij. V kolikor razdelimo aplikacijo na spletne in delovne vloge ter vmes vstavimo še sporočilne vrste in teme, omogočimo šibko sklopljen sistem. Glavni namen vzorca je omogočiti asinhrono obdelovanje uporabniških zahtev z uporabo koncepta sporočilnih vrst ali tem, korelacijo sporočila s pošiljateljem in obveščanje pošiljatelja preko komunikacijskih kanalov.

4.3.7.3. Prav tako poznano kot

- Sporočilni vzorec.
- Vzorec zahtevek / odgovor.

4.3.7.4. Motivacija

Velikokrat pridemo do potrebe po šibko sklopljenem sistemu, kjer lahko za vmesni sloj uporabimo sporočilne vrste in/ali teme, s čimer omogočimo procesiranje zahtev tudi v primeru izpada spletne ali delovne vloge. Za shranjevanje in neodvisno procesiranje uporabniških zahtev lahko uporabimo sporočilne vrste. S tem uporabniku omogočimo nadaljnje delo, ne da bi čakal na odgovor. Prav tako želimo, da se sporočila obdelajo v istem vrstnem redu, kot pridejo v vrsto. S pomočjo identifikatorja v poslani zahtevi poskrbimo za korelacijo sporočila s pošiljateljem, tako da pošiljatelj prejme svoj odgovor. Odgovori naj se shranijo v tabelo, saj tako spletna aplikacija ne potrebuje čakati na odgovor, temveč ga prevzame, kadar je želeno. Po procesiranju zahteve naj delovna vloga preko komunikacijskih kanalov obvesti spletno aplikacijo o končanem delu.

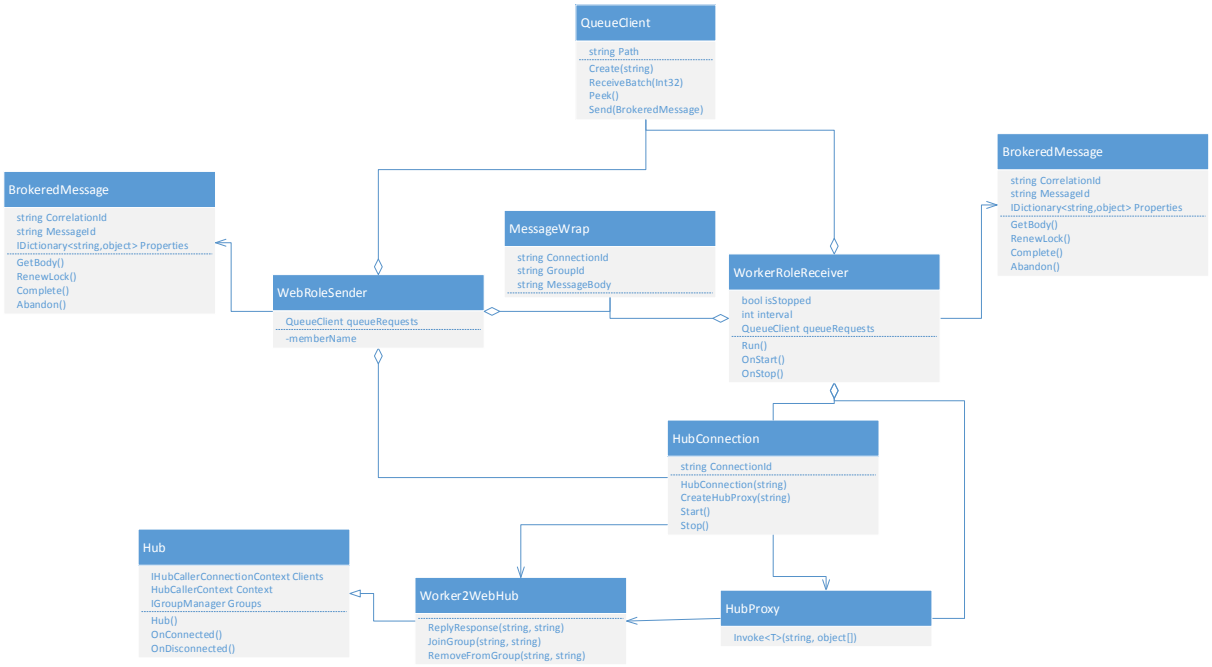
4.3.7.5. Uporaba

Uporabite ta vzorec, v kolikor hočete:

- Zgraditi vmesni sloj med spletno in delovno vlogo za sprejemanje sporočil.
- Uporabiti korelacijske identifikatorje za dvosmerno komunikacijo.
- Shraniti odgovor v tabelo za zagotovo sporočanje procesiranega odgovora.
- Med spletno aplikacijo in delovno vlogo zgraditi vmesni sloj za obveščanje o poteku procesiranja.

4.3.7.6. Struktura

Slika 6 prikazuje strukturo načrtovalskega vzorca za dvosmerno sporočilno komunikacijo. Spletna vloga WebRoleSender za oblikovanje sporočila uporablja razred BrokeredMessage. Telo sporočila je sestavljeno iz razreda MessageWrap. Sporočilo posreduje preko razreda QueueClient, ki se uporablja za pošiljanje in prejemanje sporočila. Prav tako ima narejeno dostopno vozlišča HubConnection, kateri zgradi dostopno vozlišče Worker2WebHub, ki razširja razred Hub. Delovna vloga WorkerRoleReceiver sporočila tipa BrokeredMessage prejme preko razreda QueueClient, jih obdela, nato pa preko povezave HubConnection in HubProxy posreduje odgovor uporabniku.



Slika 6: Struktura sporočilnega obdelovanja zahtev preko vrst.

4.3.7.7. Udeleženci

Glavni akterji v tem vzorcu so:

- **Worker2WebHub**: je razred, ki razširja razred `Hub` in vsebuje metode za komunikacijo s povezanimi odjemalci na dostopno vozlišče (angl. "Hub"), v tem primeru na Worker2WebHub.
- **HubProxy**: gre za proxy na strani odjemalca `IHubProxy`,
- **HubConnection**: je razred, ki ustvari povezavo z dostopnimi vozlišči.
- **WebRoleSender**: predstavlja spletno vlogo, ki pošlje sporočilo in ima ustvarjeno povezavo za komunikacijo.
- **WorkerRoleReceiver**: predstavlja delovno vlogo, ki procesira sporočila in posreduje odgovor preko proxyja.
- **BrokeredMessage**: predstavlja razred za izgradnjo sporočila.
- **QueueClient**: predstavlja razred za sporočilno vrsto, vsebuje metode za pošiljanje sporočil v vrsto in prejemanje sporočil iz vrste.
- **MessageWrap**: predstavlja razred, katerega vstavimo v telo poslanega sporočila in vsebuje podatke za korelacijo sporočila s pošiljateljem.

4.3.7.8. Sodelovanja

WebRoleSender in WorkerRoleReceiver ustvarita medsebojno komunikacijo s pomočjo razredov Worker2WebHub, HubConnection in HubProxy. Worker2WebHub razširja `Hub` in vsebuje metode za povezavo odjemalcev v skupino ter metodo za korelacijo in pošiljanje odgovora pravemu odjemalcu. WebRoleSender pripravi sporočilo tipa BrokeredMessage, v

katerega v telo vstavi razred tipa MessageWrap. S pomočjo razreda QueueClient se naredi povezavo do sporočilne vrste in vanjo pošlje sporočilo. WorkerRoleReceiver prav tako preko razreda QueueClient naredi povezavo do sporočilne vrste, kjer čaka na nova sporočila. Sporočilo prevzame in ga s pomočjo razreda BrokeredMessage razdela in izlušči telo, ki je tipa MessageWrap. Sporočilo obdela, nato pa rezultat obdelave preko ustvarjene povezave do odjemalca posreduje s pomočjo `Invoke` metode razreda `HubProxy`.

4.3.7.9. Posledice

Uporaba tega vzorca ima prednosti in slabosti:

- Asinhrono in neodvisno procesiranje uporabniških zahtevkov.
- Izgradnja šibko sklopljene strukture aplikacije.
- Omogočena višja razširljivost aplikacije.
- Omogočena korelacija procesiranega sporočila s pošiljateljem.
- Slaba stvar je, da sporočanje odgovora preko komunikacijskih kanalov poteka preko HTTP protokola, ki ne zagotavlja, da bodo vsi paketi zagotovo prispeli, zato je potrebno vpeljati shranjevanje odgovorov v tabelo.

4.3.7.10. Implementacija

Spodaj so podani nasveti za pomoč pri implementaciji vzorca.

- **Povratno sporočanje**: deluje v realnem času, zato se po končanem procesiranju takoj obvesti pošiljatelja. V kolikor ta nima odprtega brskalnika, odgovora ne bo videl. Zaradi tega je priporočljivo shraniti odgovor v tabelo, kamor lahko kasneje odjemalec pogleda procesirane odgovore.
- **Dodajanje več naročnikov**: kljub temu, da je možno narediti dvosmerno komunikacijo preko vrst, je v določenih primerih priporočljivo uporabiti teme. Na temo se lahko prijavi več naročnikov, vsak s svojo naročnino. Prav tako se lahko na eno naročnino prijavi več naročnikov. To pomeni, da lahko več prejemnikov prebere eno sporočilo, kar je primerno kadar želimo preusmeriti odgovor o procesirani zahtevi tudi administratorju ali tehnični pomoči, v kolikor odgovor vsebuje podrobnosti napake ali kaj podobnega. Iz vrste pa lahko sporočilo prebere samo ena oseba.
- **Testiranje storitvenih vodil**: v času pisanja magistrske naloge ni bilo mogoče lokalno testirati storitvenih vodil, brez naročnine na Windows Azure. Zadeva deluje le, če imate naročnino na Windows Azure in vzpostavljeno storitveno vodilo. V kolikor želite testirati lokalno potrebujete "Service Bus for Windows Server", ki pa naj bi se izvajal tudi na operacijskem sistemu Windows 7 ali novejši. V tem primeru ne gre za Windows Azure storitev, zato se lahko izvajanje storitve razlikuje.
- **Uporaba korelacije**: v kolikor implementirate dvosmerno komunikacijo je nujno potreben nekakšen korelacijski ključ. Pri uporabi `SessionId` iz ASP.NET MVC

spletne strani je potrebno biti pozoren, saj se le-ta, v kolikor ni predhodno drugače implementirano, ob osveževanju strani spreminja. Priporočljivo je narediti svoj mehanizem za statičen ID seje. To lahko naredimo s pomočjo metode `Session_Start()`, katero dodamo v `Global.asax` datoteko.

4.3.7.11. Vzorčna koda

Primer vzorčne kode je predstavljen v poglavju 5.3.

4.3.7.12. Znani primeri uporabe

Primeri spletnih aplikacij, kjer spletna vloga posreduje zahtevke v vrsto in jih lahko delovna vloga obdela neodvisno. Primer takšne uporabe je:

- oddaja naročil preko spletne trgovine,
- dodajanje in brisanje vnosov iz podatkovne baze.

4.3.7.13. Sorodni vzorci

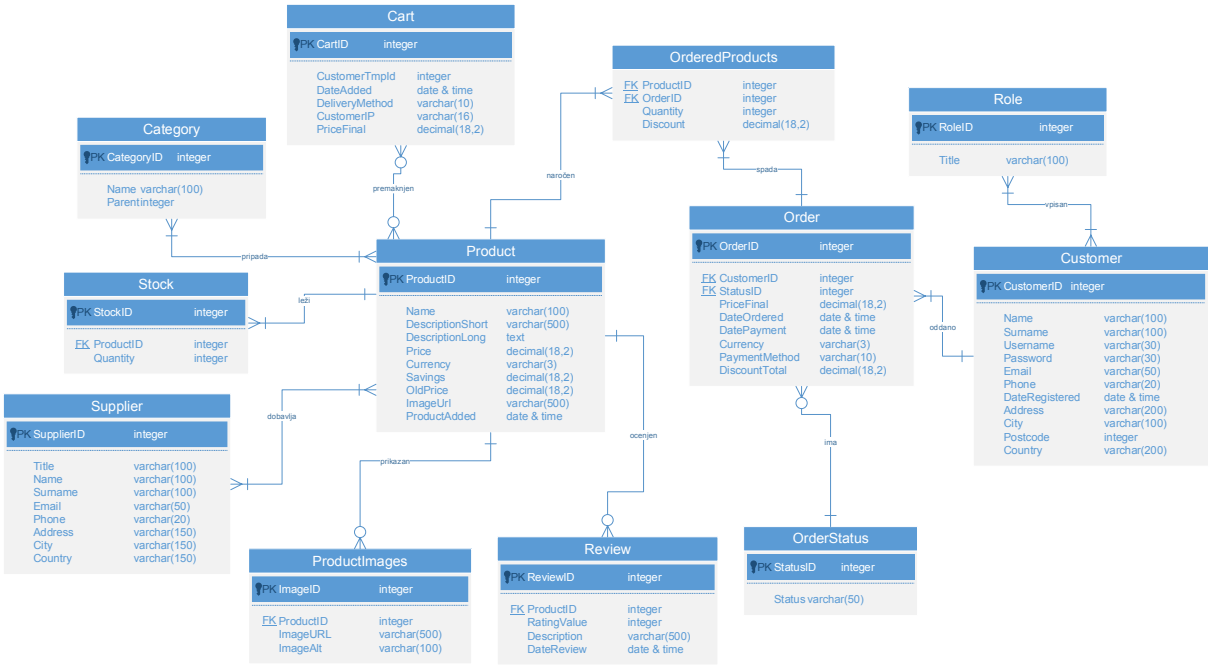
Vzorec sporočanja objavi-naroči, vzorec enosmernega sporočanja.

5. PRIKAZ UPORABE VZORCEV NA PRIMERU WINDOWS AZURE

Namen tega poglavja je pokazati, kako lahko vzamemo obstoječo spletno aplikacijo in nanjo apliciramo želene vzorce ter na takšen način razširimo delovanje in uporabo aplikacije. Za potrebe implementacije vzorcev bomo vzeli lastno aplikacijo, ki je bila zgrajena za to nalogo. Gre za aplikacijo tipa spletne trgovine. Zgrajena je s pomočjo ogrodja .NET ter na podlagi arhitekturnega vzorca MVC. Programski jeziki, ki jih bomo uporabljali skozi primere so C#, ASP.NET, HTML5 in JavaScript. Vse skupaj je grajeno za okolje Windows Azure. Postopno razširjanje aplikacije skozi vsak vzorec bo pripeljalo do razširitve aplikacije in implementacije vseh vzorcev, kar je tudi cilj te naloge.

Spletna aplikacija je poimenovana MagShop in je razdeljena na uporabniški in administrativni del. Uporabniški del omogoča pregledovanje izdelkov, registracijo in prijavo uporabnika, nakupovanje izdelkov ter pregledovanje svojih nakupov. Upravljavski del pa omogoča prav tako prijavo uporabnika, spreminjanje podatkov o produktih, dodajanje slik produktom, spremljanje naročila in spreminjanje faze naročila. Aplikacija v začetni fazi uporablja podatkovno bazo SQL Server 2012, katero kasneje nadomestimo z uporabo Windows Azure SQL Database. Zasnova podatkovne baze je prikazana na spodnjem konceptualnem diagramu (slika 7). Diagram je začeten, iz kakršnega je izhajala aplikacija spletne trgovine pred uporabo vzorcev in prenovo v večnajemništvo. Kasneje bomo za izgradnjo vzorca večnajemništva, v potrebne tabele dodali še identifikacijski ključ najemnika, ki bo enolično določal najemnika. Za shranjevanje datotek smo v začetni fazi uporabili datotečni sistem znotraj aplikacijskega strežnika, kamor smo shranjevali datoteke, kot so slike, PDF datoteke in ostale datoteke. V kasnejši fazi, ko bomo naredili prenos na Windows Azure, bomo v ta namen uporabili Windows Azure Blob Storage, kamor bomo datoteke shranjevali v obliki blobov.

Dostop do podatkovne baze Azure SQL in ostalih storitev Windows Azure je mogoč preko portala Windows Azure Management Portal ter preko aplikacijskih programskih vmesnikov. Razvijalcem je prav tako olajšan dostop do podatkovne baze Azure SQL, saj lahko do nje dostopajo kar preko programa SQL Server Management Studio. Do storitev oblaka Windows Azure lahko dostopamo tudi preko razvojnega orodja Visual Studio 2013. Potrebni so le podatki za povezavo, katere najdemo na portalu Azure Management, ki je trenutno dostopen preko naslova https://manage.windowsazure.com.



Slika 7: Diagram podatkovne baze aplikacije.

5.1. Večnajemništvo

Vzorec opisuje preoblikovanje aplikacije v večnajemniško, z možnostjo izbire izoliranosti podatkovnega modela. Podatkovna izoliranost je eden od bolj pomembnih in odločilnih faktorjev, ko se podjetje odloča med enonajemniškimi ali večnajemniškimi aplikacijami. Z vpeljano spremembo lahko preko ene aplikacije omogočimo izbiro izoliranosti podatkov na podlagi shem ali najemniškega ključa. Pri slednji so podatki vseh najemnikov združeni v skupne tabele, za izolacijo pa je potrebno poskrbeti preko aplikacije in poizvedb. Druga možnost, ki se jo loteva vzorec, pa je izoliranost s pomočjo shem, kjer ima vsak najemnik svoje tabele. Omogočene ima pravice za vpogled samo v svoje tabele, ki so združene s podatkovno shemo. Ta možnost je bolj primerna za najemnike, ki želijo večjo stopnjo varnosti ali pa jim tako veleva politika zasebnosti. Preko vzorca tako preoblikujemo obstoječo aplikacijo, da bo omogočala dva načina izoliranosti, s čimer jo lahko ponudimo večjemu spektru najemnikov. Posledično se s tem tudi zmanjšajo stroški za vzdrževanje, saj nam ni potrebno vzdrževati dve ločeni aplikaciji za vsak način izolacije podatkov.

5.1.1. Podatkovna izolacija

Kot smo že omenili, pri izgradnji aplikacije in vzorcev uporabljamo ogrodje .NET, kateri ponuja ogrodje Entity Framework (EF). Entity Framework je ogrodje za objektno-relacijsko preslikavo (angl. "Object Relational Mapping" – ORM), ki, kot že ime samo pove, omogoča preslikavo objektov v relacijsko podatkovno bazo in obratno. Za nas so zanimivi trije razredi:

- Razred, ki predstavlja entitete iz podatkovnih baz. Vsebuje atribute in relacije z drugimi entitetami. Razred z imenom entitete je shranjen v istoimensko `.cs` datoteko.

- Razred, ki predstavlja nastavitve za preslikavo entitete v podatkovno bazo. Razred vsebuje tudi ostale nastavitve entitete, kot so podatki o ključih, zahtevanih lastnostih, omejitve glede velikosti, v kateri stolpec se lastnost preslika v tabelo itd. Takšni razredi so izpeljani iz razreda `EntityTypeConfiguration<TEntityType>`, kjer je `TEntityType` želeni razred.
- Razred, ki predstavlja podatkovni kontekst. Ta razred je izpeljan iz razreda `DbContext` in se uporablja kot glavni objekt za interakcijo z bazo preko določenega podatkovnega modela. Preko njega lahko delamo poizvedbe iz baze, prav tako pa združuje narejene spremembe na entitetah, ki jih nato shrani v bazo. Vsebuje `DbSet<TEntity>` razrede, ki povežejo prej generirane razrede entitet s podatkovno bazo.

Vzorec za izolacijo podatkov bomo uporabili na obstoječi podatkovni bazi in projektu ASP.NET MVC. Projekt že ima iz podatkovne baze zgrajene modele, tj. razrede in preslikave. Za izgradnjo modelov obstajata dva pristopa. Najbolj znan je pristop "Code-First", kjer sami izdelamo razrede, ki predstavljajo podatkovne modele ter podatkovni kontekst. Nato preko razvojnega okolja Visual Studio dodamo povezavo do podatkovne baze, tako da povezavo do podatkovne baze zgradimo z modelom, ki smo ga predhodno implementirali. S pomočjo razvojnega okolja se nam ustvari povezava do podatkovne baze, na kateri se tudi generirajo potrebne tabele. Orodje Entity Framework Power Tools omogoča obratni inženiring pristopa "Code-First", kjer iz že obstoječe podatkovne baze in ustvarjenih tabel generiramo razrede, podatkovni kontekst ter preslikave. Ta pristop je zelo priročen, v kolikor imamo že narejeno podatkovno bazo ter večje število tabel in želimo hitro ter učinkovito izdelati razrede za dostop do podatkovnih tabel. Predvidevamo, da imate pred uporabo vzorca zgrajeno enonajemniško ali večnajemniško aplikacijo, katera gostuje v oblaku Windows Azure. V kolikor aplikacija še ne gostuje v oblaku, lahko vzorec prav tako uporabite na lokalni različici, kasneje pa vse prenesete na oblak. Poleg omenjenega morate imeti podatkovno bazo s tabelami postavljeno ali v oblaku ali na lokalnem strežniku Microsoft SQL Server. Za dostop do podatkovne baze iz aplikacije uporabljate ogrodje Entity Framework, različica 6 ali novejša. Vzorec implementira izbiro načina izolacije podatkov glede na najemnikovo izbiro ob registraciji in temelji na omenjenih tehnologijah. Pogledali si bomo dva načina izolacije, enega s shemami in drugega na podlagi najemniškega ključa.

5.1.1.1. Izolacija s podatkovnimi shemami

Izolacijo podatkovnih tabel na podlagi podatkovnih shem omogoča višjo stopnjo izolacije podatkov, saj ima najemnik lastne tabele, ki so združene v shemo. Varnostno kopiranje in obnavljanje podatkov v takšnem primeru je lažje. Število najemnikov, ki lahko uporabljajo aplikacijo je zaradi možnosti razširjana odprto navzgor. Vzorec omogoča dinamično gradnjo podatkovnih tabel in shem glede na najemnika. Obstoječe podatkovne tabele bomo preuredili za uporabo s ključem najemnika, a več o tem v naslednjem poglavju.

Za vsakega novega najemnika, ki bo izbral izolacijo s shemami, moramo narediti nove tabele in jih združiti v shemo. To bomo naredili z uporabo shranjenih procedur v podatkovni bazi. Preko aplikacije bomo klicali procedure, ki bodo storile omenjeno delo. Najprej bomo izdelali procedure za izdelavo novega uporabnika v podatkovni bazi in njegove sheme, kasneje pa še procedure za izdelavo podatkovnih tabel.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
      -- Vhodni parametri za uporabo v proceduri
      <@Param1, sysname, @p1> <Datatype_For_Param1, , int> =
<Default_Value_For_Param1, , 0>
AS
BEGIN
    -- Stavki SQL za izvedbo v proceduri
END
GO
```

Programska koda 1: Okvir za izgradnjo procedur.

Programska koda 1 prikazuje okvir, katerega uporabimo pri izgradnji procedur. Procedure imajo lahko vhodne parametre, ki jih bomo v našem primeru uporabili pri klicanju procedure za vnos podatkov o najemniku. Stavke SQL lahko združimo in vstavimo v eno proceduro ali pa uporabimo vsak stavek posebej, v kolikor nam je to ljubše. Proceduram je potrebno nastaviti ime in vhodne parametre, nato pa skripto poženemo v podatkovni bazi Azure SQL. Tako bomo imeli v podatkovni bazi narejene procedure za ustvarjanje novega najemnika in sheme. Prvi stavek SQL v spodnji programski kodi (programska koda 2) je namenjen izgradnji uporabniškega imena za povezavo do sheme. Vnesti moramo uporabniško ime, njegovo geslo ter privzeto podatkovno bazo. `CHECK_EXPIRATION` smo v našem primeru izklopili in namenoma nismo dodali argument `MUST_CHANGE`, saj bomo aplikativno poskrbeli za spreminjanje gesla najemniku. SQL server bo sam naredil izvleček gesla in ga v tej obliki tudi shranil. Naslednji stavek SQL prikazuje izgradnjo uporabnika na podlagi prijavnega imena iz glavne podatkovne baze `master`. Nato izdelamo shemo ter ga povežemo z novo narejenim uporabnikom. Za konec pa nastavimo uporabniku še privzeto shemo, katero smo naredili v prejšnjem koraku.

```
CREATE PROCEDURE [dbo].[createUserAndSchema]
      @databaseName varchar(100), @tenantSchema varchar(100)
      @userName varchar(100), @loginName varchar(100), @pass varchar(100),
AS
BEGIN TRANSACTION;
BEGIN TRY
EXEC('CREATE LOGIN '+@loginName+' WITH PASSWORD='''+@pass+''',
DEFAULT_DATABASE = '+@databaseName+', CHECK_EXPIRATION=OFF,
CHECK_POLICY=ON')
EXEC('CREATE USER '+@userName+' FOR LOGIN '+@loginName)
EXEC('CREATE SCHEMA '+@tenantSchema +' AUTHORIZATION '+@userName)
```

```
EXEC('ALTER USER ['+@userName + '] WITH DEFAULT_SCHEMA =
['+@tenantSchema+']')
COMMIT TRANSACTION;
END TRY
BEGIN CATCH
      ROLLBACK TRANSACTION;
END CATCH;
GO
```

Programska koda 2: Primer procedure za novega najemnika in shemo.

S tem smo zaključili postopek izgradnje novega uporabnika. Velikosti parametrov naj programer nastavi tako, da bodo ustrezali njihovi aplikaciji. Ne pozabimo, da je procedure in vse stavke SQL, ki spreminjajo vrednosti v bazi, primerno oviti s `try-catch` bloki in transakcijami ter povratnimi operacijami. Vsakokratno izdelovanje novega najemnika, v kolikor ta že obstaja, ni primerno, zato je dobro narediti proceduro, ki bo vrnila pozitivno vrednost, v kolikor shema oz. uporabnik že obstajata in negativno v nasprotnem primeru. Šele v pozitivnem primeru se lotimo izgradnje novega uporabnika, sheme ter tabel.

Po uspešno ustvarjenem uporabniku je potrebno narediti tabele. V ta namen se lahko prav tako uporabi procedure, saj imamo veliko večjo preglednost in enostavno možnost kasnejšega popravljanja, saj lahko stavke SQL popravljamo ločeno od aplikacije. V kolikor nove spremembe ne vplivajo na podatkovni model v aplikaciji, le-te ni potrebno po nepotrebnem prevajati in ponovno nameščati na oblak. Predvidevamo, da smo podatkovne tabele zgradili s pomočjo SQL stavkov in jih imamo nekje shranjene. V kolikor jih nimamo, podatkovne tabele pa imamo vseeno narejene, se lahko preko Microsoft SQL Server Management Studio povežete do podatkovne baze in jih generirate. Z desnim klikom kliknete na želeno tabelo ter poiščite možnost *Script Table As* in pa *CREATE To*. To nam ustvari skripte za izdelavo podatkovnih tabel. Te skripte nato preoblikujemo, da se bodo nove tabele povezale s shemo, kot je to narejeno spodaj (programska koda 3).

```
CREATE PROCEDURE [dbo].[createTablesForSchema]
      @tenantSchema varchar(100)
AS
BEGIN
...
EXEC('CREATE TABLE ['+@tenantSchema+'].[Customer](
...
```

Programska koda 3: Primer povezave nove tabele s shemo.

To je potrebno narediti za vse tabele, ki jih uporabljamo v aplikaciji. Kasneje bomo v aplikaciji poklicali to proceduro in z njeno pomočjo generirali vse tabele za novega najemnika. Sedaj, ko smo uredili vse potrebno na podatkovni bazi, se lotimo preoblikovanja v aplikaciji. V aplikaciji naredimo metode, preko katerih bomo poklicali procedure. Najprej naredimo klice za preverjanja – ali uporabnik že obstaja, ali shema že obstaja, nato pa, v kolikor je dovoljeno, naredimo novega uporabnika, shemo in podatkovne tabele. Spodnji

izvleček (programska koda 4) prikazuje preverjanje ali shema že obstaja. Na podoben način naredimo tudi ostale klice procedur. Potrebno je najprej narediti povezavo s podatkovno bazo SQL, nato pridobimo ukaz SQL, nastavimo tip in parametre in jo izvedemo. V kolikor imamo proceduro, ki nam vrača vrednost, lahko to vrednost dobimo preko parametra. V kolikor je rezultat klica drugačen, pa lahko uporabimo objekt.

```
using (var sqlConn = new SqlConnection(
ConfigurationManager.ConnectionStrings["MyDbContext"].ConnectionString)) {
   try {
      sqlConn.Open();
      var sqlComm = new SqlCommand("dbo.checkSchemaExist", sqlConn);
      sqlComm.CommandType = CommandType.StoredProcedure;
      sqlComm.Parameters.Add("@schemaName", SqlDbType.VarChar, 100).Value =
tenantName;
      sqlComm.Parameters.Add("@ret", SqlDbType.Int).Direction =
ParameterDirection.ReturnValue;
      sqlComm.ExecuteNonQuery();
      var schemaExists = (int)sqlComm.Parameters["@ret"].Value;
      if (schemaExists < 1) {
      //nadaljevanje logike
```

Programska koda 4: Izvleček za izvedbo procedure.

Predhodno smo že omenili razred, ki razširja kontekst podatkovne baze in nam s pomočjo ogrodja Entity Framework omogoča povezovanje na podatkovno bazo in poizvedbe. Za uspešno povezovanje na podatkovno shemo vsakega najemnika moramo omogočiti dinamično spreminjanje sheme in posledično povezavo do podatkovne baze. Razred privzeto izpeljuje samo `DbContext`. Zaradi tega je potrebno narediti spremembo in poleg obstoječega implementirati še vmesnik `IDbModelCacheKeyProvider`. Ta vmesnik namreč omogoča lastno logiko za pridobitev ključa. Tako lahko znotraj iste aplikacije z uporabo enega konteksta dostopamo do različnih modelov. Uporabimo lahko tudi več kontekstov, ki uporabljajo iste modele. Mi bomo uporabili en podatkovni kontekst, kateremu bomo dinamično prilagajali ime sheme.

```
public partial class MySchemaContext : DbContext, IDbModelCacheKeyProvider
{
   private string DbSchema { get; set; }
   public string CacheKey {
      get { return this.DbSchema; }
   }
   static MySchemaContext() {
      Database.SetInitializer<MySchemaContext>(null);
   }

   public MTPatternContext(string connString, string dbSchema)
      : base(connString) {
      Database.SetInitializer< MySchemaContext >(null);
      this.DbSchema = dbSchema;
   }

   public MySchemaContext() : base("Name=MySchemaContext") { }
```

```
   public DbSet<Customer> Customers { get; set; }
   public DbSet<Order> Orders { get; set; }
   //ostali razredi
   protected override void OnModelCreating(DbModelBuilder modelBuilder) {
      if (this.DbSchema != null)
         modelBuilder.HasDefaultSchema(this.DbSchema);
         modelBuilder.Configurations.Add(new CustomerMap());
         modelBuilder.Configurations.Add(new OrderMap());
         //ostale nastavitve razredov
         modelBuilder.Entity<Customer>().Ignore(f => f.TenantId);
         modelBuilder.Entity<Order>().Ignore(f => f.TenantId);
         //nastavimo ignoriranje lastnosti TenantId za razrede (več v
naslednjem poglavju)
         base.OnModelCreating(modelBuilder);
      }
   }
   public IEnumerable<T> Get<T>() where T : class {
      var set = Set<T>();
      return set;
   }
}
```

Programska koda 5: Primer konteksta za uporabo z več shemami.

Zgornja primer (programska koda 5) prikazuje spremembo razreda za podatkovni kontekst, ki po novem implementira še en vmesnik. Zaradi tega je potrebno dodati atribut `CacheKey` in preoblikovati metodo `get`, ki bo vračala naš niz `DbSchema`. Ta se uporablja za shranjevanje izbire sheme za trenutnega najemnika. Prav tako moramo narediti nov konstruktor, ki sprejme povezovalni niz in ime sheme. Kot vidimo konstruktor razširja osnovni konstruktor iz razreda `DbContext`, kateremu pošlje povezovalni niz. Entity Framework na podlagi tega in s predpomnjenim ključem `CacheKey` naredi novo povezavo, ki vključuje tudi pravo povezavo do sheme.

```
MySchemaContext dbContext = new MySchemaContext(
   ConfigurationManager.ConnectionStrings["MyDbContext"].ConnectionString,
   tenantSchema);
```

Programska koda 6: Izdelava konteksta za sheme.

V programski kodi aplikacije je potrebno spremeniti tudi inicializacijo podatkovnega konteksta. Tako bomo uporabljali konstruktor, ki sprejme povezovalni niz in shemo, kot je to prikazano v zgornjem izseku programske kode (programska koda 6). Ime sheme lahko dobimo na dva načina: ali omogočimo uporabniku prijavo preko prijavne strani, s čimer dobimo njegove poverilnice, v katerih je zapisano ime sheme, ali pa ime sheme preberemo kar iz URL-ja, ko uporabnik dostopa do strani. V slednjem primeru je potrebno nastaviti lastno pot URL z dodatnimi parametri. Več o tem v poglavju 5.1.3.

5.1.1.2. Izolacija v skupnih tabelah z najemniškim ključem

Za uporabo skupnih tabel z najemniškim ključem bomo obstoječim tabelam, ki so v shemi `dbo`, dodali nov atribut (stolpec) z imenom `TenantId`. Ta bo vseboval enolično določen najemniški ključ. Obstoječa praksa je, da se vsem tabelam doda nov atribut, ki hrani ključ najemnika. Vseeno je dobro razmisliti, katere tabele obvezno potrebujejo dodaten atribut in katere ne. Za primer lahko vzamemo šifrant držav, ki je načeloma enak za vse najemnike in takšna tabela ne potrebuje razširitve s ključem najemnika, razen če bi bil šifrant vezan na najemnika in bi lahko določeni najemniki izbirali samo določene države. Prav tako lahko izpustimo atribut pri tabelah, kjer lahko zagotovimo, da podatek lahko pridobimo enolično preko druge tabele, katera ima ključ najemnika (tabele en nivo višje). Takšen primer sicer ni najboljša praksa, saj moramo, v kolikor želimo podatek pridobiti iz takšne tabele, nujno najprej iskati en nivo višje, šele nato lahko povežemo podatke iz prejšnje tabele s to tabelo, preko katerih pridobimo prave podatke. Takšno iskanje je zamudnejše in tudi, v primeru nekonsistentnih podatkov, lahko pride do nepravilnih poizvedb. Zato je primernejše, da razširimo vse tabele s ključem najemnika, razen tiste, ki so skupne vsem najemnikom.

V primeru vzorca bomo za shranjevanje ključa naredili tabelo v Azure Table Storage. Po potrebi lahko naredite tabelo tudi v relacijski podatkovni bazi. Pomembno je, da nekje naredite ločeno tabelo, v kateri boste hranili podatke o najemniku z identifikacijskim ključem najemnika, ki ga boste uporabili pri poizvedbah. Problem pri razširjanju obstoječih tabel je v primerih, ko že imamo podatke v tabelah. Kadar želimo dodati nov atribut, lahko tega dodamo le tako, da dovoljuje ničelne vrednosti. Glede na to, da se pri nas ne sme zgoditi, da bi v bazi imeli zapis, ki ne bi imel najemniškega ključa, moramo najprej poskrbeti za pravilno razširitev tabele. Aplikacija je pred prenovo v večnajemniško zagotovo enonajemniška, tako da podatki v podatkovni bazi, najverjetneje pripadajo le enemu najemniku oz. uporabniku. V kolikor razširimo podatkovne tabele in za privzeto vrednost vnesemo ključ prvega najemnika, na takšen način obidemo omejitev, ki prepoveduje, da bi bili vnosi brez vrednosti. Kasneje moramo še poskrbeti, da odstranimo omejitve s privzeto vrednostjo, saj bodo drugače imeli novi vnosi privzeto vrednost, kar pa ni naš namen.

```
CREATE TABLE [dbo].[Tenant] (
      [TenantId] [int] NOT NULL IDENTITY(1,1),
      [TenantName] [varchar](100) NOT NULL
      -- ostale lastnosti
 CONSTRAINT [PK_TENANT] PRIMARY KEY CLUSTERED ( [TenantId] ASC )
);

INSERT INTO [dbo].[Tenant] ([TenantName])
      VALUES ('MagShop');

ALTER TABLE [dbo].[Customer]
ADD [TenantId] [int] NOT NULL DEFAULT(1),
CONSTRAINT [FK_Customer_Tenant] FOREIGN KEY ([TenantId])
REFERENCES [Tenant] (TenantId);
```

```
SELECT * FROM sys.objects WHERE parent_object_id = object_id('Customer');

ALTER TABLE [dbo].[Customer]
DROP CONSTRAINT [DF_Customer_TenantId];
```

Programska koda 7: Ustvarjanje najemniške tabele in povezovanje s ključem.

Programska koda 7 prikazuje izdelavo nove tabele za shranjevanje informacij o najemniku, v kolikor se uporabnik odloči, da bo na takšen način implementiral hranjenje teh podatkov. V našem primeru, smo se odločili za uporabo Azure Table Storage, kar bomo prikazali v nadaljevanju. Vseeno pa imamo v zgornjem izvlečku programske kode tudi primer, kjer s ključem najemnika razširimo vse potrebne tabele. Tabelo razširimo z novim atributom, ki ima določeno privzeto vrednost, zato moramo po uspešni razširitvi to omejitev odstraniti. Ime omejitve (angl. "Constraint") najdemo s pomočjo spodnjega stavka SELECT. Na takšen način bomo najenostavnejše prišli do novega polja v tabeli, brez da bi izbrisali podatke v tabeli. Azure SQL, za razliko do navadnega Microsoft SQL strežnika, ne podpira nekaterih ukazov in lastnosti (npr. `CLUSTERED INDEX`), zato je potrebno skripte prilagoditi.

Nadaljujmo z vzorcem. Sedaj, ko smo tabele v podatkovni bazi razširili s ključem najemnika, moramo za isto poskrbeti še v aplikaciji, kar prikazuje tudi spodnja programska koda 8. Poleg konteksta za dostop do podatkovne baze s pomočjo shem, bomo naredili še en razred, ki bo razširjal `DbContext` in ga bomo uporabljali za dostop do podatkovne baze in tabel s ključem najemnika. V njem bomo za začetek dodali konstruktor za povezavo preko povezovalnega niza. Razrede, ki predstavljajo podatkovne modele, moramo nato razširiti s ključem najemnika. Razširimo torej vse tiste, ki smo jih tudi v relacijski podatkovni bazi. Zaradi razširitve razreda s ključem najemnika smo morali v nastavitvah entitet nastaviti ignoriranje tega atributa, kot je to prikazano v prejšnjem poglavju. Le tako ne bomo dobili napake pri uporabi tistega konteksta, saj tabele, ki so združene v shemo, nimajo tega atributa. Podatkovnemu kontekstu bomo dodali tudi atribut za shranjevanje ključa najemnika. Tako bomo lahko povezovali kontekst z najemnikom.

```
public class MyTenantContext : DbContext {
   public int TenantId { get; set; }

   static MyTenantContext() {
      Database.SetInitializer<MyTenantContext>(null);
   }

  public MyTenantContext(string connectionString): base(connectionString) {
      Database.SetInitializer<MyTenantContext>(null);
   }

   public MyTenantContext (): base("Name=MyTenantContext") { }
   public DbSet<Customer> Customers { get; set; }
   public DbSet<Order> Orders { get; set; }
   // ostali razredi...
   protected override void OnModelCreating(DbModelBuilder modelBuilder) {
      modelBuilder.Configurations.Add(new CustomerMap());
```

```
      modelBuilder.Configurations.Add(new OrderMap());
      // ostale nastavitve razredov...
   }

   public IEnumerable<T> Get<T>() where T : class {
      var set = Set<T>();
      var tIdProperty = typeof(T).GetProperty("TenantId");
      if (tIdProperty != null) {
         var argument = Expression.Parameter(typeof(T),"entity");
         var left = Expression.Property(argument, tIdProperty);
         var right = Expression.Constant(TenantId);
         var predicate = Expression.Lambda<Func<T, bool>>
                           (Expression.Equal(left, right), argument);
         IEnumerable<T> query = set.Where(predicate);
         return query;
      }
      return null;
   }
}

public partial class Customer { public int TenantId { get; set; } }
public partial class Order { public int TenantId { get; set; } }
// ostali razredi...
```

Programska koda 8: Primer konteksta s ključem najemnika.

Glede na to, da bi bilo potrebno veliko sprememb, če bi hoteli pri vseh poizvedbah implementirati tudi iskanje po ključu najemnika, smo se odločili za pristop, kjer v sam kontekst vgradimo metodo, katera nam iz baze vrača podatke za dano entiteto in pri tem že upošteva ključ najemnika. Metoda se imenuje `Get` in ji moramo podati tip entitete, saj gre za generično funkcijo. Metoda deluje tako, da za dostop do določene entitete najprej pridobi instanco razreda `DbSet`, katero podamo na mesto generičnega tipa. Nato s pomočjo metode `GetProperty` pridobimo lastnost tipa. V kolikor te lastnosti ni, preskočimo spodnje korake v kodi, drugače pa nadaljujemo in zgradimo predikat. Predikat je sestavljen iz argumenta, kateri opisuje po kateri entiteti iščemo, levega dela predikata, ki opisuje po kateri lastnosti iščemo, ter desnega dela predikata, ki opisuje kaj iščemo oz. lastnost najemniškega ključa. Ta predikat združimo s pogojem "je enako" in vstavimo v `Where` metodo, ki filtrira in vrne podatke iz baze na podlagi predikata. Tako bomo ob klicu funkcije za določen tip dobili podatke, vezane na določenega najemnika.

Sedaj imamo dva konteksta in potrebno je narediti preverbo, na podlagi katere se odločimo, kateri kontekst bomo uporabili. Prav tako uporabimo dinamično globalno spremenljivko, kateri ob inicializaciji nastavimo pravilni tip, kot je to prikazano spodaj (programska koda 9).

```
private static dynamic dbContext;
//...
if (tenantType.ToLower().Contains(Schema.ToLower())) {
   dbContext = new MySchemaContext(

ConfigurationManager.ConnectionStrings["MyDbContext"].ConnectionString,
      tenantName);
}
```

```
else {
    dbContext = new MyTenantContext();
}
```

Programska koda 9: Primer dinamične inicializacije konteksta.

Vsem poizvedbam je potrebno spremeniti način dostopanja do podatkov preko entitet, na način, kot kaže spodnja programska koda 10. Tip `IEnumerable` omogoča filtriranja s predikatom, kot to omogoča razred `DbSet`, ki smo ga uporabljali do sedaj. Shranjevanje poteka kot do sedaj, zato je potrebno spremeniti le t.i. klice GET. Glede na to, da ima isto metodo tudi drugi kontekst, bo spremenjena koda delovala v obeh primerih.

```
var result = ((IEnumerable<Customer>)dbContext.Get<Customer>());
//ali
IEnumerable<Customer> result2 = dbContext.Get<Customer>();
//nato lahko izvajamo filtriranje
result2.First();
//ali direktno filtriramo s predikatom
var result3 = ((IEnumerable<Customer>)dbContext.Get<Customer>()).Where(c =>
c.Country.ToLower().Equals("Ljubljana").ToLower());
```

Programska koda 10: Primer pridobivanja podatkov.

5.1.1.3. Tabela najemnikov

Vse podatke o najemniku bomo v našem primeru shranjevali v Azure Table Storage. Nujno moramo shranjevati identifikacijski ključ najemnika, katerega bomo uporabili znotraj aplikacije za dostop do podatkov v bazi. Potrebujemo še njegovo prikazno ime, katerega bomo uporabili pri generiranju shem. Ostali podatki so prav tako pomembni, vendar se lahko razlikujejo od zasnove aplikacije. Kot primer naj navedemo le nekatere: aktivnost računa, e-pošta odgovorne osebe, uporabniško ime in geslo, datum registracije itd.

Ob registraciji najemnika najprej naredimo povezavo do storitve Azure Storage, kjer preverimo ali tabela obstaja in jo po potrebi naredimo. Nato preverimo obstoj uporabnik in ga po potrebi naredimo. Pri izdelavi naj omenimo, da ima Windows Azure možnost operacije `Replace` in `InsertOrReplace`. Druga operacija se od prve razlikuje v tem, da ne bo pokazala napake, v kolikor je med našim izvajanjem operacije kdorkoli spremenil entiteto, temveč bo entiteto posodobila oz. vnesla na novo. Prav tako omenimo, da imamo med razvojem aplikacije v projektu Windows Azure dve datoteki, ki sta nastavitveni datoteki storitve. Ti datoteki imata ime `ServiceConfiguration.Cloud.cscfg` oz. `ServiceConfiguration.Local.cscfg` in se razlikujeta od okolja, v katerem se izvajata. Prva je namenjena izvajanju v oblaku, druga pa v lokalnemu okolju. V ti datoteki lahko vnašamo nastavitve, pomembne za delovanje aplikacije, katere se razlikujejo med lokalnim okoljem in oblakom. Programska koda 12 prikazuje povezavo na Azure Storage in izdelavo

tabele v storitvi Azure Table, v kolikor ta še ne obstaja. Prav tako prikazuje vnos oz. posodobitev entitete v tabeli. Omeniti moramo, da je za vnos entitete potrebno narediti razred, ki izpeljuje razred `TableEntity`, kot je to prikazano spodaj (programska koda 11). Tabela ima dva ključa – `PartitionKey` in `RowKey`, ki ju skrbno izberemo, saj nam omogočata hitrejše iskanje po zapisih. Tabele so namreč razdeljene v particije in prav ključ `PartitionKey` enolično določa, v kateri particiji se nahaja naš zapis. Ključ `RowKey` je enolični identifikator entitete znotraj določene particije. Skupaj predstavljata primarni ključ entitete. Pri pridobivanju podatkov iz entitete ju podamo kot parametra, kot to prikazuje spodnja programska koda 12.

```
internal class TenantInfo : TableEntity
    {
        public TenantInfo(string tenantName, string tenantUser, string
tenantPass, string tenantType, int tenantActive, int tenantId) {
            this.PartitionKey = tenantName;
            this.RowKey = tenantUser;
            this.TenantId = tenantId;
        //ostale lastnosti
        }
        public TenantInfo() {}
        public int TenantId { get; set; }
        //ostale lastnosti
    }
```

Programska koda 11: Razširitev razreda TableEntity.

```
CloudStorageAccount storageAccount =
   CloudStorageAccount.Parse(
ConfigurationManager.ConnectionStrings["StorageConnectionString"].Connectio
nString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
CloudTable table = tableClient.GetTableReference("TenantInfo");
table.CreateIfNotExists();


var retrieveOp = TableOperation.Retrieve<TenantInfo>(tenantName,
tenantUser);
var result = table.Execute(retrieveOp);
var update = (TenantInfo)result.Result;
var updateOp = TableOperation.InsertOrReplace(update);
table.Execute(updateOp);
```

Programska koda 12: Povezava in posodobitev entitete.

V kolikor ste sledili vsem korakom in popravili potrebne metode in klice, potem ste implementirali vzorec za izbiro izolacije podatkov. V nadaljevanju si bomo ogledali še nekaj kratkih nasvetov za implementacijo večnajemniške aplikacije.

5.1.2. Lastne najemniške nastavitve

Večnajemniške aplikacije omogočajo svojim najemnikom, do neke mere, prilagoditev aplikacije svojim željam. Te prilagoditve se lahko dotikajo sprememb podatkovnega modela, največkrat pa so zaželene grafične spremembe strani (logotipov, slik, stilov), s čimer se aplikacija čim bolj prilagodi željam najemnika oz. celostni grafični podobi podjetja. Omenili smo že, da je shranjevanje slik in ostalih datotek možno v Windows Azure Blob Storage, katera spada v storitev Windows Azure Storage. Znotraj Azure Storage storitev imamo poleg blobov še tabele in vrste. Za izolacijo podatkov moramo poskrbeti tudi na tem nivoju. Windows Azure omogoča izgradnjo več računov za shranjevanje (angl. "Storage Account"), kar je primerno v večnajemniških aplikacijah. Tako je mogoče vsakemu najemniku narediti svoj račun, s čimer bo dobil izoliran prostor za shranjevanje podatkov in datotek. Dostop do tega prostora je možen preko aplikacije le s pravim imenom in ključem za dostop do računa. Ti podatki so v aplikaciji shranjeni v povezovalnem nizu (angl. "Connection String"), ki jih lahko dinamično pridobimo na podlagi najemnika. Pomembno je opozoriti, da je izgradnja novih računov za shranjevanje odvisna od naročnine na Windows Azure. Privzeto je možno zgraditi le en račun za shranjevanje na storitvah Windows Azure, vsaka nadaljnja izgradnja je dodatno plačljiva. Če kot ponudniki aplikacije ali programerji nimamo na voljo izgradnje novih računov za shranjevanje, bodisi zaradi naše naročnine na storitve Windows Azure, bodisi uporabljamo preizkusni naročniški račun, potem takšne izgradnje novega računa za svoje najemnike ne bomo mogli narediti. V primeru le enega računa za shranjevanje je možna rešitev uporaba več zabojnikov za shranjevanje blobov. Za vsakega najemnika lahko programsko izdelamo zabojnike za blobe, ki jih poimenujemo po imenu najemnika ali njegovemu identifikacijskemu ključu. Programska koda 13 prikazuje možnost programske izdelave zabojnikov in nastavljanje javnih pravic temu zabojniku. V projektu je potrebno dodati referenco na `Microsoft.WindowsAzure.Storage`.

```
var storageAccount = CloudStorageAccount.Parse(
ConfigurationManager.ConnectionStrings["StorageConnectionString"].Connectio
nString);
var blobClient = storageAccount.CreateCloudBlobClient();
var blobContainer = blobClient.GetContainerReference("TenantName");
blobContainer.CreateIfNotExists();
blobContainer.SetPermissions(new BlobContainerPermissions(){PublicAccess =
BlobContainerPublicAccessType.Blob});
```

Programska koda 13: Izdelava najemniškega zabojnika.

Windows Azure ne podpira hierarhičnega grajenja zabojnikov, zato so vsi zabojniki zgrajeni na istem nivoju. S pravilnim poimenovanjem blobov lahko dosežemo želeno hierarhijo. To storimo tako, da v ime bloba, kjer želimo imeti mapo, logično vrinemo znak poševnice ("/"). Za lažjo ponazoritev si poglejmo primer: `najemnik1/`**`css/images/bkgrnd-big.png`**. "najemnik1" je dejanski zabojnik znotraj Azure Blob Storage, ostalo ime datoteke pa je

ločeno le s poševnicami, z namenom gradnje hierarhije. Takšno poimenovanje datotek ni nujno potrebno, vendar na takšen način lažje uredimo blobe glede na njihov namen. Omenimo še, da se morajo vsa imena zabojnikov začeti z malo začetnico.

Pri uporabi blobov, tabel in vrst v okviru storitev Windows Azure, lahko uporabimo deljeni podpis za dostop, ki se uporablja za omogočanje dostopa do podatkov in datotek, ne da bi uporabniku posredovali ključ računa. V naši rešitvi imamo ključ računa za dostop do shrambe zapisan v povezovalnem nizu, znotraj nastavitvene datoteke aplikacije. Ta ključ računa omogoča aplikaciji administrativni dostop do vseh datotek in podatkov v shrambi. Azure Storage ponuja za vsak račun primarni in sekundarni ključ, katera omogočata administrativni dostop in imata možnost ponovnega generiranja. Tako lahko drugi osebi posredujemo sekundarni ključ za dostop do shrambe in ga po določenem času ponovno regeneriramo. Pri tem je potrebno biti pazljiv, saj oseba s ključem dobi administrativni dostop. Zaradi tega se za dostop do podatkov uporablja deljeni podpis, kjer lahko generiramo pravila za dostopanje do podatkov. Vsakemu pravilu lahko določimo pravice, katere želimo ponuditi uporabniku pri dostopanju. Možno je tudi specificirati časovno obdobje veljave pravila. Pravice lahko nastavimo tako na zabojniku, kot tudi na sami datoteki (blobu).

5.1.3. Večnajemniška aplikacija

Windows Azure omogoča poleg večnajemniškega eno instančnega modela prav tako večnajemniški več instančni model, ki poskrbi za prenos večnajemniške aplikacije na več instanc. Napaka na eni instanci tako ne vpliva na napake iz druge instance. Prav tako je možno popravljanje programske kode na enem mestu, spremembe pa se prenesejo na vse strežnike. Takšen pristop pa zahteva, da že v osnovi načrtujemo aplikacijo na takšen način. V tem razdelku se bomo osredotočili na izgradnjo večnajemniške eno instančne aplikacije.

Glede na to, da je aplikacija eno instančna, bodo vsi najemniki dostopali do iste instance aplikacije. Potrebno je zagotoviti, da bodo najemniki dostopali le do svojega dela aplikacije. V ta namen bomo najprej implementirali dostop do aplikacije za vsakega najemnika preko URL-ja. V `RouteConfig.cs` datoteki je potrebno popraviti pot v URL-ju, kot je to prikazano spodaj (programska koda 14). Povezovanje preusmeritev je mogoče za več naslovov z različnimi maskami, odvisno od namena in potreb.

```
routes.MapRoute(
                name: "Default",
                url: "{tenant}/{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
            );
```

Programska koda 14: Določitev najemnikov glede na URL.

Vsaka metoda znotraj krmilnika mora imeti specificirano za vhodni parameter še spremenljivko `tenant`, ki je tipa niz. Na ta način bomo dobili v metodo niz najemnika. Niz najemnika se ujema z zapisom v bazi in tako lahko prikažemo uporabniku prilagojeno spletno stran. Vsaka prijava v stran mora sprožiti še dodaten mehanizem, kjer poleg naziva najemnika v bazi preverjamo še identifikacijski ključ najemnika in ustreznost uporabnika določenemu najemniku. Tako zagotovimo izolacijo podatkov ob uporabi skupne podatkovne baze. Uporabnika je pri prijavi v stran potrebno avtenticirati. V našem primeru smo naredili lastno metodo za avtentikacijo, katera se izvede v dogodku `Application_AuthenticateRequest` in ob klicu `Authenticate()`. Prav tako uporabimo razred `FormsAuthenticationTicket`, kateri se uporablja za dostopanje do shranjenih lastnosti in vrednosti avtenticiranega uporabnika preko obrazcev. V razred shranimo potrebne podatke skupaj z razredom `MagShopPrincipal`, ki razširja vmesnik `IPrincipal`. Razred `MagShopPrincipal` vsebuje nujne podatke o registriranem uporabniku, katere lahko nato skozi aplikacijo uporabljamo. Na koncu je potrebno vse podatke šifrirati in shraniti v piškotek. Privzeti šifrirni algoritem je SHA1, ki za izvleček uporablja algoritem HMACSHA1. Vsakemu krmilniku lahko dodamo pred metodo še atribut `Authorize`, kateremu v oklepaju lahko sledi atribut `Roles` in specificirane vloge, ki imajo dostop do krmilnika. Primer takšnega atributa je sledeč: `[Authorize(Roles="admin")]`. Seveda je potrebno, v primeru lastne avtentikacije, podatek o vlogah uporabnika shranjevati v bazo. Poleg lastne avtentikacije Windows Azure ponuja Windows Azure Access Control, kateri je namenjen avtentikaciji uporabnika na podlagi identitete oz. računa, upravljanega pri drugem ponudniku. Tako lahko avtenticiramo uporabnika na podlagi računa Windows Live ID, Google, Yahoo ali Facebook. Prav tako Windows Azure podpira aktivni imenik. V naslednjem vzorcu se bomo lotili avtentikacije uporabnika s pomočjo storitev Azure Active Direcotry.

5.2. Spletne storitve

5.2.1. Predpogoj

Za implementacijo vzorca bomo uporabili Azure Active Directory. Microsoft trenutno omogoča brezplačno uporabo osnovnih storitev Azure Active Directory, ob predpogoj, da imamo naročnino na Windows Azure. Po uspešni prijavi v upravljavski portal Windows Azure najprej zgradite aktivni imenik. Njegovo ime bomo potrebovali pri nadaljnji implementaciji vzorca. V narejen aktivni imenik vnesimo nekaj uporabnikov. Azure AD omogoča tudi vnos uporabnikov z drugimi Microsoft računi (Outlook.com in ostali). Po želji lahko generirate tudi skupino in vanjo vnesete uporabnike, kar pa za implementacijo tega vzorca ni potrebno.

Aktivni imenik omogoča tudi registracijo aplikacij, katerim omogoči dostop do njega. Izbiramo lahko iz nabora preko 500 vnaprej narejenih aplikacij ali vnesemo podatke o svoji aplikaciji. Slednje je potrebno narediti za naše nadaljnje delo. Registrirajmo spletno aplikacijo, za tem pa še spletno storitev. Potrebno je podati naslov URL do mesta za prijavo v aplikaciji (angl. "Sign-on URL") in enolično določen URL aplikacije (angl. "App ID URL"). Oba naslova lahko naknadno spreminjamo. V kolikor aplikacijo testiramo lokalno, sta naslova lahko tudi `http://localhost:st_vrat`.

5.2.2. Aplikacijske nastavitve Azure AD

Najprej potrebujemo eno registrirano aplikacijo, katera bo predstavljala vstopno točko za spletno storitev. Poleg nje potrebujemo še eno za aplikacijo, katera bo dostopala do prej omenjene spletne storitve. To je lahko domorodna aplikacija za mobilne naprave, namizna aplikacija ali spletna aplikacija. Aplikaciji, ki predstavlja spletno storitev, je potrebno popraviti manifest datoteko, kar bo omogočilo ostalim aplikacijam dostop do le-te. Gre za datoteko, ki v obliki JSON vsebuje nastavitve za aplikacijo. Preko portala lokalno shranimo manifest datoteko in popravimo vnos `appPermissions`, kot to kaže spodnji izvleček programske kode (programska koda *15*). Nato datoteko ponovno naložimo preko portala, s čimer povozimo prejšnje nastavitve [38].

```
"appPermissions": [
    {
      "claimValue": "user_impersonation",
      "description": "Dovoli aplikaciji polni dostop do spletne storitve v
imenu prijavljenega uporabnika",
      "directAccessGrantTypes": [],
      "displayName": "Polni dostop do spletne storitve MagShopApi",
      "impersonationAccessGrantTypes": [
        {
          "impersonated": "User",
          "impersonator": "Application"
        }
      ],
      "isDisabled": false,
      "origin": "Application",
      "permissionId": "e8a47428-213d-4da2-9ffd-4488430eb9ca",
      "resourceScopeType": "Personal",
      "userConsentDescription": "Dovoli aplikaciji polni dostop do spletne
storitve MagShopApi v vašem imenu.",
      "userConsentDisplayName": "Polni dostop do spletne storitve
MagShopApi"
    }
  ],
```

Programska koda 15: Sprememba manifest datoteke.

Programska koda 15 prikazuje dodajanje novih pravic aplikaciji. Medtem ko ostale vrednosti lahko, za potrebe vzorca, pustite nespremenjene, je potrebno poskrbeti, da bo `permissionId` enoličen identifikator med vsemi pravicami, ki jih imamo. Prav tako je obvezno, da je ta vrednost tipa `GUID`. Polje `claimValue` je vrednost iz nabora trditev (claim) in dobi svoje mesto v dostopnem žetonu. Polje `Origin` je zmeraj isto, saj se ga zaenkrat še ne uporablja. Polje `isDisabled` je negativno, v kolikor izdelamo novo pravilo oz. ga posodabljamo. Za brisanje pravic moramo najprej to vrednost nastaviti na pozitivno, manifest naložiti na oblak in šele nato lahko izbrišemo pravice. Polje `resourceScopeType` ima lahko dve vrednosti – ali omogoča privolitev končnega uporabnika v uporabo informacij ali pa zahteva privolitev globalnega upravitelja. Polja `description` in `displayName` sta za prikaz oz. informacijo o pravici. Polje `impersonationAccessGrantTypes` omogoča, da se aplikacija izdaja za uporabnika.

Poleg datoteke je potrebno še omogočiti pravice aplikaciji Windows Azure Active Directory do branja imenika oz. branja in pisanja, v kolikor spletna storitev tudi to izvaja. Preko upravljavskega portala nastavimo lastnosti aplikacije pri uporabi aktivnega imenika. Najprej je potrebno nastaviti aplikacijo, ki bo dostopala do spletne storitve. Tam je potrebno registrirati ključ, ki se bo uporabljal pri avtentikaciji aplikacije. Ključ je potrebno prepisati in shraniti na varno, kajti do njega kasneje ne bo več mogoče priti. V primeru, da ga izgubimo, ga je potrebno ponovno narediti in spremeniti v aplikaciji. V kolikor smo prejšnjo manifest datoteko uspešno posodobili, se na seznamu pravic pojavi nova aplikacija za izbiro. Izberemo jo in omogočimo delegirane pravice. V kolikor aplikacija poleg dostopa do spletne storitve omogoča tudi branje oz. branje in pisanje po aktivnem imeniku, še tu izberemo primerne pravice.

5.2.3. Avtentikacija v aplikacijah

Omenili smo že postopek delovanja protokola OAuth. Odjemalec želi dostopati do nekega vira s klicem spletne storitve. Ta mu vir ne bo posredovala, v kolikor uporabnik ni avtenticiran in posledično tudi ne avtoriziran. V ta namen je potrebno najprej opraviti postopek avtorizacije. Azure AD olajša postopek avtentikacije, saj se vsa logika avtentikacije uporabnika odvija na strani Microsofta. Azure AD nam ob klicu storitve ponudi vpisno okno, kamor vnesemo uporabniško ime in geslo. Potek same avtentikacije je odvisen od izbranega avtentikacijskega protokola. Razvijalec lahko izbira med WS-Federation, OAuth 2.0, OpenID Connect in SAML-P. Za vsak postopek Azure AD ponuja razvijalcu končne točke, preko katerih dostopa do API storitev avtentikacije. Pri komunikaciji med strežnikom in aplikacijo veliko pomaga vmesnik OWIN (The Open Web Interface for .NET), ki je nekakšen vmesni sloj med spletnim strežnikom in aplikacijo ter je namenjen njuni razdvojitvi. Našemu projektu dodamo potrebne OWIN reference. To lahko naredimo s pomočjo vtičnika NuGet Packages ali preko konzole Package Manager. Novo dodane reference dodajo aplikaciji vmesni nivo za

želeno podporo avtentikacije preko protokola OAuth. V ta namen je potrebno poleg dodanih referenc ustvariti še razred `Startup.cs` in v projektu MVC, znotraj mape `App_Start`, vključiti še `Startup.Auth.cs`, ki je dejansko `partial class Startup`. Slednji datoteki dodamo metodo `ConfigureAuth`, v kateri omogočimo vse načine prijave v našo aplikacijo. Primer takšne metode je prikazan spodaj (programska koda 16). Tukaj specificiramo možne prijave preko ponudnikov identitet kot so Microsoft, Google, Yahoo in tudi Azure AD.

```
public void ConfigureAuth(IAppBuilder app) {
     app.UseWindowsAzureActiveDirectoryBearerAuthentication(
     new WindowsAzureActiveDirectoryBearerAuthenticationOptions {
          Audience = ConfigurationManager.AppSettings["Audience"],
          Tenant = ConfigurationManager.AppSettings["Tenant"]
     });
}
```

Programska koda 16: Primer nastavitve za prijavo z Azure AD.

Za komunikacijo z Azure AD moramo poznati `Application ID URI`, ki enolično določa registrirano aplikacijo v aktivnem imeniku. Prav tako je velikokrat potrebno podati `Reply URL`, kamor bo, v primeru uspešne avtentikacije, Azure AD poslal odgovor, skupaj z dostopnim in osvežilnim žetonom. Podati je potrebno še `Client ID`, ki je niz tipa `GUID` in ga ustvari Azure AD ob registriranju aplikacije. Ob avtentikaciji je potrebno posredovati še ključ, ki je pravzaprav naključno generirano geslo. V kolikor je aplikacija večnajemniška in je potebno avtenticirati uporabnike različnih najemnikov, potem je avtentikacijska končna točka za takšne aplikacije `https://login.windows.net/common`. Tja se pošlje zahteva za avtentikacijo, nakar Azure AD najde kateremu aktivnemu imeniku spada najemnik, nato pa izvede njegovo avtentikacijo.

Postopki avtentikacije in avtorizacije uporabnika se torej razlikujejo od izbranega avtentikacijskega protokola in tipa aplikacije. Vzorec implementira ogrodje OAuth 2.0. Najprej je potrebno pridobiti avtorizacijsko kodo, kar naredimo tako, da generiramo naslov URL. Ob želeni prijavi uporabnika v našo aplikacijo, le-tega preusmerimo na generirani naslov, kjer je prijavno okno, kamor se uporabnik prijavi s svojim uporabniškim imenom in geslom. Uporabniško ime je v tem primeru dejanski e-poštni naslov uporabnika, shranjen v aktivnem imeniku. Programska koda 17 predstavlja generirani naslov URL za pridobitev avtorizacijske kode:

```
https://login.windows.net/{ime-aktivnega-imenika}/oauth2/authorize
?api-version=1.0
&response_type=code
&client_id={GUID-aplikacije}
&resource={polno ime našega Azure AD računa}
&redirect_uri={preusmeritveni-naslov}
```

Programska koda 17: Naslov za pridobitev avtorizacijske kode.

Strežnik nam ob uspešni prijavi vrne kodo in nas preusmeri na želeno preusmeritveno stran. V aplikaciji MVC je to lahko krmilnik, ki vsebuje logiko za zamenjavo avtentikacijske kode za žeton dostopa in osvežitve. Programska koda 18 prikazuje, kako to naredimo programsko.

```
string clientId = ConfigurationManager.AppSettings["ClientId"];
string appKey = ConfigurationManager.AppSettings["AppKey"];

ClientCredential clientCredential = new ClientCredential(clientId, appKey);
string authority = ConfigurationManager.AppSettings["Authority"];
Uri redirectUri =
      new Uri(ConfigurationManager.AppSettings["appBaseAddress"]);
AuthenticationContext authContext = new AuthenticationContext(authority);
AuthenticationResult result =
      authContext.AcquireTokenByAuthorizationCode(code, redirectUri,
clientCredential);
SaveAccessToken(resourceId, result.AccessToken);
SaveRefreshToken(resourceId, result.RefreshToken);
```

Programska koda 18: Zamenjava pridobljene kode za žetone.

Po uspešni avtentikaciji uporabnika imamo v predpomnilniku shranjena žetona dostopa in osvežitve. Avtentikacijski kontekst vrne rezultat v obliki objekta `AuthenticationResult`, ki vsebuje oba žetona, čas poteka žetona in še nekatere druge podatke. V kolikor bi dostopni žeton potekel in bi bilo potrebno zahtevati novega, to naredimo na način, kot je prikazan spodaj (programska koda 19). Tako ni potrebno zahtevati, da se uporabnik ponovno prijavlja.

```
ClientCredential clientCredential = new ClientCredential(clientId, appKey);
string authority = ConfigurationManager.AppSettings["Authority"];
AuthenticationContext authcontext = new AuthenticationContext(authority);
AuthenticationResult result =
      authcontext.AcquireTokenByRefreshToken(refreshToken, clientId,
clientCredential);
```

Programska koda 19: Ponovna pridobitev dostopnega žetona.

Kadar nimamo žetona za osvežitve dostopa nimamo oz. z njim ne uspemo pridobiti dostopnega žetona, potem najprej pobrišemo predpomnilnik, kjer imamo shranjene žetone in naredimo novo zahtevo za žetone. Ob uspešni pridobitvi žetonov lahko naredimo klic spletne storitve in dostopni žeton dodamo v glavo zahtevka. Avtentikacijski zahtevek ima v glavi posebno mesto, kamor vstavimo vrednost avtentikacije. To naredimo na način, kot je predstavljeno v naslednjem izseku programske kode (programska koda 20).

```
string apiBaseURL = ConfigurationManager.AppSettings["ApiBaseURL"];
string apiGetNarocila = apiBaseURL + "/api/narocila"
HttpClient client = new HttpClient();
HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get,
apiGetNarocila);
request.Headers.Authorization = new AuthenticationHeaderValue("Bearer",
accessToken);
HttpResponseMessage response = await client.SendAsync(request);
```

```
if (response.IsSuccessStatusCode) { // izvedemo obdelavo odgovora }
```

Programska koda 20: Avtoriziran klic spletne storitve.

Glede na to, da so nekateri klici asinhroni, morajo biti tudi metode v krmilniku asinhrone operacije, kot je to narejeno pri primeru `public async Task<ActionResult>`. `HttpResponseMessage` je objekt, ki nato vsebuje odgovor s strani spletne storitve. V njej imamo različne metode, preko katerih preberemo glavo sporočila, status vrnjenega sporočila, vsebino itd. Vsebino sporočila nato zopet z asinhrono operacijo preberemo v niz, tabelo bajtov ali `Stream`. To lahko naredim na sledeč način: `string responseString = await response.Content.ReadAsStringAsync();`

5.2.4. Avtorizacija v spletnih storitvah

Microsoft je skozi razvoj ogrodja .NET zamenjal številne načine izgradnje spletnih storitev. Trenutno se uporabljata dve obliki izgradnje spletnih storitev. Prva je uporaba ogrodja WCF, ki omogoča izgradnjo storitveno usmerjenih aplikacij. Spletne storitve WCF privzeto uporabljajo način SOAP, možno pa je storitev tudi preoblikovati v uporabo načina REST. Druga oblika je izgradnja ASP.NET Web API, ki nam olajša izgradnjo t.i. "RESTful" API-jev. Web API je grajen na tehnologiji ASP.NET MVC, zaradi česar je omogočena velika fleksibilnost. Razširja razred `ApiController` in se v tem razlikuje od ostalih upravljalnikov. Za potrebe tega vzorca bomo uporabili ogrodje ASP.NET Web API, saj je enostavnejši za gradnjo, ima boljšo podporo mobilnim napravam in je enostavno razširljiv. Prav tako ga lahko gostujemo znotraj spletne aplikacije, zaradi česar ni potrebe po dodatnih spletnih ali delovnih vlogah. Izvajanje v spletnih vlogah omogoča uporabo funkcij storitev IIS.

Spletno storitev prav tako kot prej aplikacijo, razširimo s potrebnimi referencami za vmesni sloj, ki jih vključuje OWIN. Za vzorec smo uporabili reference `OAuth` in poleg `Microsoft.Owin.Security.ActiveDirectory` še nekatere ostale. Razvijalec lahko izbira in namesti potrebne reference iz paketa `Microsoft.Owin.Security`. Nato naredimo nov razred `Startup.cs` in `Startup.Auth.cs`. V datoteki `Startup.Auth.cs` nastavimo `IAppBuilder` v metodi `ConfigureAuth` na podoben način, kot je to narejeno v prejšnjem poglavju. To naredimo s pomočjo metode, katera zgradi vmesni sloj za uporabo Azure AD postopkov, `app.UseWindowsAzureActiveDirectoryBearerAuthentication(...)`. V kolikor želimo API krmilnik v celoti zaščititi pred nepooblaščenimi uporabniki, na vrhu razreda dodamo opisni atribut `[Authorize]`. Prav tako je možno avtorizirati samo določene metode, v kolikor se za to odločimo. V tem primeru atribut dodamo pred metodo. Primer dobre prakse je, da onemogočimo nepooblaščenim osebam dostop do celotnega krmilnika, nato pa z atributom `[AllowAnonymous]` omogočimo dostop kjer želimo. Možno je tudi avtorizirati samo določeno skupino uporabnikov ali določene vloge. To naredimo tako, da

atributu `Authorize` dodamo še lastnost `Roles` ali `Users` in v nizu opišemo vlogo oz. uporabnike. Natančnejša predstavitev je prikazana spodaj (programska koda 21).

```
[Authorize]
public class NarocilaController : ApiController {
   [Authorize(Roles = "Administrator")]
   public IEnumerable<string> Get() {
      //...
   }
   //...
}
```

Programska koda 21: Primer avtorizacije spletne storitve.

Ob pozitivni avtorizaciji žetona se uporabniku dovoli uporabo virov oz. izvedbe spletne storitve. Znotraj spletne storitve lahko nato natančneje preverimo prejeti žeton. To naredimo odvisno od želja in spletne storitve, na primer zato, da preverimo ali je bil izdan za pravo področje ipd. Trditve o uporabniku so shranjene v razredu `ClaimsPrincipal` in hranijo veliko informacij o uporabniku. Žeton omogoča dodajanje teh trditev, kar pa je tudi različno pri različnih avtorizacijskih strežnikih. Različni ponudniki identitete uporabnikov namreč ponujajo različne trditve o uporabniku. V trditvah so shranjeni podatki o imenu, priimku, e-poštnem naslovu, spolu, naslovu in še marsikaj. Podatki lahko pridobimo odvisno od tega, kaj je shranjeno v žetonu. Dostop do trenutnih trditev je možen preko URL-ja sheme ali pa direktno preko razreda `ClaimTypes`. Programska koda 22 prikazuje oba načina pridobivanja podatkov o uporabniku:

```
var lastName =
      ClaimsPrincipal.Current.FindFirst("http://schemas.xmlsoap.org/ws/2005
/05/identity/claims/surname").Value;
var firstName =
      ClaimsPrincipal.Current.FindFirst(ClaimTypes.GivenName).Value;
```

Programska koda 22: Primer pridobivanja podatkov o uporabniku.

Sedaj smo v spletno storitev prinesli podatke o uporabniku, prav tako je uporabnik avtoriziran za uporabo spletne storitve, tako da lahko izvedemo potrebno delo in mu vrnemo rezultat. Ta rezultat prejme in vsebino izvleče na načine, ki so bili omenjeni v prejšnjem poglavju.

5.3. Sporočilno obdelovanje zahtev

5.3.1. Predpogoj

Za potrebe našega vzorca bomo uporabili storitev Azure Service Bus. Za testiranje delovanja storitvenega vodila potrebujemo naročnino do Windows Azure, saj Azure Emulator takšne storitve zaenkrat še nima vgrajene v sebi. V kolikor nimamo naročnine do Windows Azure ali

želimo le lokalno testiranje, lahko namestimo "Service Bus for Windows Server". Za namestitev najprej potrebujemo "Web Platform Installer", preko katerega namestimo želena orodja. Po namestitvi sledi še konfiguracija okolja in pripravljeni smo na lokalno razvijanje in testiranje. V kolikor je želja po uporabi Windows Azure, je potrebno najprej ustvariti imenski prostor za storitveno vodilo. To je najlažje narediti v upravljavskem portalu, seveda pa se da narediti tudi preko Windows Azure REST API-ja za storitveno vodilo. Po uspešno ustvarjenem imenskem prostoru smo pripravljeni na nadaljnje programiranje.

V tej nalogi razvijamo vzorec na primeru spletne trgovine, vseeno pa se lahko vzorec uporabi na kateri koli drugi spletni aplikaciji, ki za delovanje uporablja ogrodje .NET. Spletna trgovina je spletna aplikacija tipa ASP.NET MVC, ki je postavljena v spletno vlogo na oblaku Windows Azure. Narediti moramo, da aplikacija ob oddaji naročila, generira sporočilo in ga pošlje v sporočilno vrsto. Hkrati zgradimo tudi povezavo na dostopno vozlišče, kjer bomo čakali na odgovor. Na drugi strani bomo naredili delovno vlogo, ki bo prejemala sporočila iz vrste. Po procesiranju bo delovna vloga, s pomočjo SignalR knjižnic, preko komunikacijskih kanalov poslala odgovor direktno do prvotnega pošiljatelja. Prav tako bo v Azure Table shranila svoj odgovor, katerega bo lahko spletna vloga prebrala, kadar bo uporabnik tako želel.

5.3.2. Storitveno vodilo - vrste

Vzorec narekuje, da naredimo povezavo med pošiljateljem in prejemnikom preko sporočilnih vrst, za kar uporabimo sporočilno vrsto v storitvenem vodilu. Izvleček programske kode (programska koda 24) prikazuje način pošiljanja sporočila v vrsto. Glede na to, da ima projekt Windows Azure dve nastavitveni datoteki `ServiceConfiguration`, tudi v tem primeru pišemo povezovalne nize in ostale nastavitvene lastnosti v ti dve datoteki. Nastavitve se nato samodejno, glede na okolje, vzamejo iz prave datoteke. Branje iz teh datotek je možno preko razreda `CloudConfigurationManager`. S pomočjo razreda `NamespaceManager` preverimo ali vrsta že obstaja in jo, v nasprotnem primeru, naredimo. Razred `QueueClient` je zadolžen za ustvarjanje povezave do vrste. Ta razred tudi pošlje sporočilo v vrsto. Poslano sporočilo mora biti tipa `BrokeredMessage`. `BrokeredMessage` je razred, kateri le s konstruktorjem nastavi telo sporočila. Telo je lahko razreda `Stream` ali kakšnega drugega tipa, ki omogoča serializacijo. V našem primeru bomo uporabili razred `MessageWrap`, ki je v projektu izdelan kot Class Library, nato pa smo referenco nanj dodali v ostalih projektih. Ta razred vsebuje vsebino sporočila, korelacijsko številko odjemalca in njegovo identifikacijsko številko seje, ki jo bomo uporabili za združevanje povezav do dostopnega vozlišča. Primer takšnega razreda prikazuje naslednja programska koda 23.

```
public class MessageWrap {
   public string Body { get; set; }
   public string Group { get; set; }
```

```
    public string ConnectionId { get; set; }
}
```

Programska koda 23: Primer razreda za telo sporočila.

Sporočilu lahko pripnemo še dodatne lastnosti, kot so identifikacijska št. sporočila, korelacijski identifikacijski ključ, v kolikor se bomo poslužili drugačnega načina korelacijske sporočila ter lastnost `msgType`, ki opisuje kakšnega tipa je telo sporočila. To je pomemben podatek, saj moramo vedeti natančno katerega tipa je telo sporočila, v kolikor ga želimo prebrati. V našem primeru bomo za telo sporočila uporabili razred `MessageWrap`, zato ni potrebno nastaviti tip sporočila. Takšno varovalko je priporočljivo nastaviti, kadar sporočila dinamično spreminjamo in vsa pošiljamo v isto vrsto. V nasprotnem primeru bomo kmalu dobili napako pri branju, saj ne vemo kakšnega tipa je telo sporočila. Po uspešni izdelavi sporočila vse skupaj le še pošljemo v vrsto. Glede na to, da uporabljamo aplikacijo MVC, smo programsko kodo za pošiljanje sporočila vstavili v metodo POST novega krmilnika (angl. "Controller"), kar je tudi razvidno iz spodnjega primera (programska koda 24). Metodo bomo klicali s pomočjo asinhrone zahteve AJAX.

```
[HttpPost]
public JsonResult Index(string connId, string groupId, string message) {
    try {
        var QueueRequestName =
CloudConfigurationManager.GetSetting("QueueName");
        var connectionString =
CloudConfigurationManager.GetSetting("ServiceBus.ConnectionString");
        var namespaceManager =
NamespaceManager.CreateFromConnectionString(connectionString);
        if (!namespaceManager.QueueExists(QueueRequestName)) {
            namespaceManager.DeleteQueue(QueueRequestName);
            namespaceManager.CreateQueue(QueueRequestName);
        }

        var queueClient =
QueueClient.CreateFromConnectionString(connectionString, QueueRequestName);
        var newMessage = new MessageClass.MessageWrap();
        newMessage.Body = message;
        newMessage.ConnectionId = connId;
        newMessage.Group = groupId;
        var bmsg = new BrokeredMessage(newMessage);
        bmsg.MessageId = new Guid().ToString();
        bmsg.Properties["msgType"] =
              newMessage.GetType().AssemblyQualifiedName;
        queueClient.Send(bmsg);
        return Json("Narejeno.");
    }
    catch (Exception ex) {
        return Json("Napaka: " + ex.Message);
    }
}
```

Programska koda 24: Primer pošiljanja sporočila v vrsto.

Sedaj je sporočilo v vrsti, kjer čaka na delovno vlogo, da ga prevzame in obdela. Tukaj je potrebno več dela, saj je potrebno poskrbeti za celoten proces delovne vloge. Delovna vloga razširja razred `RoleEntryPoint`, katera ima tri metode, ki skrbijo za življenjski cikel delovne vloge. V `OnStart()` metodi poskrbimo za ustvarjanje vrste. To storimo s pomočjo razreda `NamespaceManager`, tako kot je že opisano v zgornji kodi. Nato z razredom `QueueClient` naredimo odjemalca, ki bo poslušal vrsto in čakal na sporočila. Postopek je isti kot pri spletni vlogi, le da odjemalca naredimo globalno, tako da ga bomo lahko klicali tudi iz ostalih metod. V metodi `Run()` imamo zanko `while`, katera se izvaja, dokler ni poklicana metoda `OnStop()`, saj se takrat nastavi vrednost atributa za izhod iz zanke na pozitivno, s čimer se omogoči pogoj za izhod iz zanke `while`. V zanki `while` pa poskrbimo za branje in procesiranje sporočil. S pomočjo ustvarjenega odjemalca lahko pridobimo sporočila. Pridobimo jih z metodo `Receive()`, ki prejme le eno sporočilo, ali pa z metodo `ReceiveBatch(int)`, kjer dobimo toliko sporočil, kot smo specificirali v atributu, oz. manj, če jih toliko ni v vrsti. S pomočjo metode `ReceiveBatch` zmanjšamo število prenosov po mreži. Sledi preverjanje, ali smo iz vrste sploh prejeli kakšno sporočilo, tj. če je v vrsti sploh obstajalo kakšno sporočilo. V kolikor ni nobenega sporočila, nastavimo časovno zakasnitev za obdobje intervala, kot to prikazuje spodnja koda (programska koda 25). Po vsakem pretečenem intervalu ponovno pogledamo vrsto, v kolikor se v njej nahaja kakšno sporočilo. Postopek nadaljujemo do maksimalnega intervala, nato se interval ne povečuje več.

```
var messages = _queueClient.ReceiveBatch(32);
if (!messages.Any()) {
     Thread.Sleep(_interval);
     if (_interval < _maxInterval)
          _interval *= 2;
}
```

Programska koda 25: Primer intervala pri prevzemanju sporočil iz vrste.

V kolikor nadaljujemo s kodo, potem smo najverjetneje prejeli vsaj eno sporočilo, katerega moramo preveriti. S pomočjo zanke `foreach` se sprehodimo čez vsa prejeta sporočila. Za vsako sporočilo najprej preverimo ali gre za t.i. strupeno sporočilo (angl. "Poison Message"). Strupeno sporočilo je sporočilo, ki ga ne moremo obdelati zaradi pomanjkljivih informacij ali ker so podatki pokvarjeni. Zaradi tega med procesiranjem pride do izjeme in se sporočilo vrne nazaj v vrsto, kjer ga delovna vloga ponovno prevzame za procesiranje. V kolikor nimamo narejenega preverjanja, se lahko ta postopek ponavlja. Sporočilo preverimo tako, da preverimo njegovo lastnost `DeliveryCount`, katera se povečuje vsakič, ko sporočilo prejmemo in ni zaključeno. V primeru, da ima sporočilo to lastnost večjo od 3, potem ga prestavimo v vrsto mrtvih sporočil (angl. "Dead Letter Queue"), kot je to zavedeno v naslednji programski kodi (programska koda 26). Sporočilo v tej vrsti čaka in ga lahko kdaj uporabimo v druge namene ali zavržemo. V kolikor ga ne potrebujemo, življenjska doba sporočila po določenem času poteče. Sporočilo prestavimo v drugo vrsto, saj glede na to, da

smo ga pred tem že trikrat prevzeli in nikoli zaključili, to predstavlja veliko verjetnost, da sporočilo povzroči zaustavitev delovne vloge ali kakšno drugačno preobremenitev procesa.

```
if (msg.DeliveryCount > 3) {
     msg.DeadLetter();
}
```

Programska koda 26: Primer preverjanja pokvarjenega sporočila.

Kadar nič od zgornjih pogojev ni uresničeno, potem lahko sporočilo preberemo in obdelamo. V kolikor pošiljamo sporočila z različnim razredom v telesu, moramo najprej prebrati pripeto lastnost `msgType`. V njo smo zapisali tip telesa s pomočjo kvalificiranega imena zbirnika (angl. "Assembly-Qualified Name"). S pomočjo metod iz imenskega prostora `System.Reflection` nato zgradimo generične metode, preko katerih dobimo podatke iz telesa sporočila. V našem primeru bomo vedno pošiljali sporočila z istim tipom telesa, tj. `MessageWrap`, zato takšne programske kode načeloma ne potrebujemo. Do telesa sporočila dostopamo preko metode `GetBody<type>()`, kjer namesto *type* podamo dejanski tip sporočila, kar je prikazano spodaj (programska koda 27). Sedaj imamo telo sporočila v spremenljivki, do preostalih lastnosti pa dostopamo direktno preko razreda `BrokeredMessage` oz. preko `Properties`, v kolikor gre za lastno dodane lastnosti. Sporočilo obdelamo, odgovor shranimo v tabelo, tako kot smo to spoznali pri prejšnjem vzorcu in posredujemo odgovor preko proxy-ja prvotnemu pošiljatelju.

```
MessageClass.MessageWrap message = msg.GetBody<MessageClass.MessageWrap>();

/* izvedemo potrebno obdelavo sporočila. Uporabniško sporočilo se nahaja v
message.Body. Nato shranimo odgovor v Azure Table in sporočimo prvotnemu
pošiljatelju odgovor preko spletne vtičnice. */

proxy.Invoke("ReplyResponse", message.Group, "Odgovor pošiljatelju.");
msg.Complete();
```

Programska koda 27: Primer pridobivanja telesa sporočila z znanim objektom.

V metodi `OnStop()` nastavimo spremenljivko, katera skrbi za izstop iz zanke v `Run()` metodi, na pozitivno. Omenjeni postopke prikazuje spodnja programska koda 28 Nato naredimo zanko v kateri poskrbimo za časovno zakasnitev, tako da lahko `Run()` metoda zaključi s svojim procesiranjem. Zanka se bo zaključila, ko bo `Run()` metoda končala in nastavila isto spremenljivko na pozitivno.

```
public override void OnStop() {
...
_finishWorker = true;
while (!_runMethodFinished) {
     Thread.Sleep(1000);
}
_queueClient.Close();
base.OnStop();
```

```
...
}
```

Programska koda 28: Vzorčna koda iz OnStop() metode.

5.3.3. Storitveno vodilo - teme

Vzorec lahko preoblikujemo tudi tako, da uporabljamo sporočilne teme namesto vrst. To bi uporabili, v kolikor bi imeli več delovnih vlog, ki bi morale sočasno obdelati sporočilo. Kot primer lahko vzamemo, da imamo delovne vloge ločene po funkcijskih vlogah, katere morajo neodvisno od druge vloge obdelati sporočilo. Spodnja implementacija je opisana le kot alternativa, princip pa je podoben kot pri sporočilnih vrstah. V sledeči programski kodi (programska koda 29) je zavedeno, kako lahko v spletni vlogi naredimo povezavo do teme s pomočjo razreda `TopicClient`, kjer navedemo ime teme in povezovalni niz.

```
if (!namespaceManager.TopicExists(TopicName))
     namespaceManager.CreateTopic(TopicName);
_topicClient =
      TopicClient.CreateFromConnectionString(connString, TopicName);
```

Programska koda 29: Povezava do teme.

Prav tako generiramo sporočilo tipa `BrokeredMessage`, le da mu tokrat dodamo v lastnosti še `CorrelationId`, ki bo omogočal filtriranje sporočil glede na naročnino prijavljenih na temo. V spodnjem primeru (programska koda 30) je prikazan takšen način generiranja in pošiljanja sporočila.

```
var newBody = new MessageClass.MessageWrap();
var bmsg = new BrokeredMessage(newBody);
bmsg.MessageId = new Guid().ToString();
bmsg.CorrelationId = msg.CorrelationId;
bmsg.Properties["msgType"] = newBody.GetType().AssemblyQualifiedName;
_topicClient.Send(bmsg);
```

Programska koda 30: Pošiljanje sporočila v temo.

Sedaj se sporočilo nahaja v temi, od koder jo je potrebno s pomočjo delovnih vlog prebrati in procesirati. Postopek je podoben kot pri sporočilnih vrstah, le da je tu pri povezavi na temo potrebno podati še naročnino. Tako lahko različni prejemniki dobijo različna sporočila. Prav tako več delovnih vlog prejme isto sporočilo, kar pri sporočilnih vrstah ni mogoče. V metodi `OnStart()` je potrebno najprej narediti povezavo na storitveno vodilo in na naročnino. Povezava do naročnine in izgradnja odjemalca je prikazana spodaj (programska koda 31).

```
var connString =
      CloudConfigurationManager.GetSetting("ServiceBus.ConnectionString");
var namespaceManager =
      NamespaceManager.CreateFromConnectionString(connString);
```

```
if (!namespaceManager.TopicExists(TopicName))
     namespaceManager.CreateTopic(TopicName);
if (!namespaceManager.SubscriptionExists(TopicName,SubscriptionName))
     namespaceManager.CreateSubscription(TopicName,SubscriptionName);
/*     Ali spodnji način, v kolikor želimo takoj nastaviti filter: */
//namespaceManager.CreateSubscription(TopicName,SubscriptionName, new
CorrelationFilter(workerCorrFilter));

var subscripClient = SubscriptionClient.CreateFromConnectionString(
      connString, TopicName, SubscriptionName);
```

Programska koda 31: Izgradnja odjemalca za naročnino na temo.

Pri ustvarjanju naročnine na temo je dobro vedeti, da se ji lahko doda filtre, ki filtrirajo vhodna sporočila. V sporočilo lahko pripnemo lastnosti z imenom in vrednostjo. Naročnina nato, s pomočjo razreda `SqlFilter`, filtrira po teh lastnostih. Pogoj, ki ga vstavimo v filter ima lastnosti poizvedb SQL, kar olajša pisanje nizov za filter. `CorrelationFilter` je le vnaprej specializiran filter, ki upošteva lastnost `CorrelationId`. Primer dodajanja korelacijskega filtra je v spodnjem izvlečku programske kode (programska koda 32).

```
var rule = new RuleDescription(){
   Filter = new CorrelationFilter(correlationId),
};
subscripClient.AddRule(rule);
```

Programska koda 32: Dodajanje naročnini korelacijski filter.

Nato je celoten proces procesiranja sporočila enak kot pri temah, le da za dostop do naročnine na temo uporabljamo objekt `subscripClient`.

5.3.4. Povratno sporočanje s SignalR

Povratno komunikacijo bomo zagotovili s knjižnicami SignalR. Gre za dokaj mlado tehnologijo, ki uporablja tehnologijo spletne vtičnice, katera je možna v okviru standarda HTML5. Najprej je potrebno s pomočjo upravljalnika paketov NuGet namestiti knjižnice SignalR in SignalR .NET Client. To nam v projekt doda knjižnice JavaScript in ostale reference, potrebne za implementacijo in delovanje komunikacije. Najprej se bomo osredotočili na spletno vlogo. V mapo `App_Start` je potrebno dodati datoteko `Startup.Auth.cs`, v primeru, da se te datoteke te še nimamo. Vanjo, v metodo `ConfigureAuth(IAppBuilder app)`, naredimo klic metode `app.MapSignalR()`. Na takšen način bomo povezali naša dostopna vozlišča z graditeljem aplikacije (angl. "App Builder"). Lastna dostopna vozlišča nato znotraj spletne aplikacije vstavimo v novo mapo `Hubs`. Zdaj lahko začnemo z gradnjo lastnega dostopnega vozlišča, kateri bo vseboval metode za pošiljanje obvestila odjemalcem, prijavo odjemalcev v skupino in odjavo. Primer takšnega

vozlišča je prikazan v naslednji programski kodi (programska koda 33). Ime `showMessage` mora biti enako tistemu imene metode, ki jo bomo uporabili v programski kodi JavaScript.

```
public class Worker2WebHub : Hub {
   public void ReplyResponse(string groupName, string result) {
      Clients.Group(groupName).showMessage(result);
   }
   public void Join(string connectionId, string groupName) {
      Groups.Add(connectionId, groupName);
   }
   public void RemoveFromGroup(string connectionId, string groupName) {
      Groups.Remove(connectionId, groupName);
   }
}
```

Programska koda 33: Izgradnja lastnega dostopnega vozlišča.

V nadaljevanju v spletno aplikacijo dodamo spodnjo programsko kodo (programska koda 34). Najbolj primerno je kodo dodati v datoteko `cshtml`, kjer imamo skupno predlogo za spletne strani. Največkrat gre to za datoteko `_Layout` znotraj mape `Shared`. SignalR za delovanje uporablja jQuery, tako da moramo paziti, da imamo referenco na jQuery definirano pred spodnjo programsko kodo JavaScript. Prvo definirano skripto moramo imeti v projektni mapi `Scripts`, medtem ko se druga generira dinamično v času izvajanja. Nato naredimo povezavo do našega vozlišča, ki je poimenovan `Worker2WebHub`. Primer ustvarjanja povezave je prikazan v spodnji programski kodi (programska koda 34). Prav tako definiramo funkcijo, katero bomo klicali iz vozlišča. Logiko v funkciji lahko spišemo povsem samostojno, le paziti moramo, da se ime ujema s tistim v vozlišču. Ob zagonu vozlišča povežemo odjemalca s svojo skupino, ki je vezana na uporabnikovo sejo. To storimo zato, ker se vsak zavihek v brskalniku šteje za svojega odjemalca in ima svojo identifikacijsko številko. S pomočjo združevanja v skupino lahko združimo vse identifikacijske številke, ki pripadajo isti seji. Tako bomo, ko bomo poslali sporočilo tej skupini, preko dostopnega vozlišča obvestili vse odprte povezave. Nazadnje še skritemu polju (angl. "Hidden Field") v spletnemu obrazcu nastavimo vrednost, katero bomo uporabili ob generiranju sporočila ob vsaki poslani zahtevi.

```
<script src="~/Scripts/jquery.signalR-2.0.3.min.js"></script>
<script src="~/signalr/hubs"></script>
<script>
   $(function() {
      var hub = $.connection.worker2WebHub;
      hub.on('showMessage', function(result) {
         $('.message-div').css('display', 'block');
         setTimeout(function() {
         $('.message-div').fadeOut();
         }, 3000);
      });
      $.connection.hub.start(function() {
         hub.server.join($.connection.hub.id,
            '@HttpContext.Current.Session["MySessionId"].ToString()');
      });
      $.connection.hub.start().done(function () {
         $('#hdnConnectionId').val($.connection.hub.id);
```

```
        });
    });
</script>
```

Programska koda 34: SignalR povezava iz spletne vloge.

V spletni aplikaciji nato ob določenem dogodku pošljemo zahtevo AJAX želenemu krmilniku. Vključimo še potrebne podatke, ki so v našem primeru ID povezave, ID skupine in sporočilo. Primer takšnega klica AJAX je prikazuje spodnja programska koda 35, kjer naredimo klic po metodi POST in specificiramo, da želimo odgovor v obliki JSON. Ob uspehu ali napaki se izvede programska koda v za to določenem delu metode.

```
@section scripts {
    <script>
        $(document).ready(function () {
            $('#btnPoslji').click(function () {
                var hubId = $('#hdnConnectionId').val();
                var msg = $('#txtMessage').val();
                $.ajax({
                    type: 'POST',
                    url: '@Url.Action("Index","Message")',
                    contentType: "application/json; charset=utf-8",
                    dataType: 'json',
                    data: JSON.stringify({ 'connId': hubId, 'groupId':
'@HttpContext.Current.Session["MySessionId"].ToString()', 'message': msg
}),
                    success: function (data) {
                        //ob uspehu izvedi sledeče
                    },
                    error: function (xhr, errorMsg, error) {
                        //ob neuspehu izvedi sledeče
                    }
                });
                return false;
            });
        });
    </script>
}
```

Programska koda 35: Primer klica AJAX do krmilnika.

V okviru delovne vloge smo že omenili primer, ko po procesiranju sporočila posredujemo odgovor prvotnemu pošiljatelju. To naredimo s pomočjo spodnje programske kode (programska koda 36). Najprej naredimo povezavo do vozlišča s pomočjo objekta `HubConnection`. Niz URL, ki ga moramo podati pri ustvarjanju povezave, je URL, kjer se izvaja spletna aplikacija, ki gostuje dostopno vozlišče. Nato naredimo proxy, preko katerega bomo sprožili metodo za pošiljanje odgovora. Pri metodi `Invoke()` je potrebno paziti na ime klicane metode, saj se mora ujemati s tisto v vozlišču, prav tako pa ji moramo pripeti vse potrebne argumente.

```
HubConnection connection = new HubConnection(urlString);
IHubProxy proxy = connection.CreateHubProxy("worker2WebHub");
connection.Start().Wait();
```

```
//sledi vse potrebno procesiranje s strani delovne vloge
proxy.Invoke("ReplyResponse", message.Group, "Odgovor pošiljatelju.");
```

Programska koda 36: Primer povezave in klica metode dostopnega vozlišča.

Kot zadnje omenimo še izgradnjo globalnega identifikatorja seje, katerega uporabljamo za združevanje povezav. Primer izgradnje takšnega identifikatorja seje je prikazan spodaj (programska koda 37). Najbolj primerno je v datoteko `Global.asax.cs` znotraj metode `Session_Start` generirati GUID, katerega nato shranimo v sejo. V kolikor se sklicujemo na `Session.SessionID`, se le ta spreminja in ni primeren za združevanje. Za združevanje lahko uporabimo tudi kakšen drug način oz. uporabimo za generiranje identifikacijske številke seje kakšno drugo metodo.

```
void Session_Start(object sender, EventArgs e) {
   HttpContext.Current.Session.Add("MySessionId",
      Guid.NewGuid().ToString());
}
```

Programska koda 37: Izgradnja globalnega identifikatorja seje.

S tem smo zaključili z implementacijo vzorcev. Vso procesiranje sporočila je odvisno od aplikacije oz. narave problema in naj ga razvijalec razvije problemu primerno.

6. ZAKLJUČEK

6.1. Rezultati in diskusija

Rezultat magistrske naloge je spletna aplikacija, ki se izvaja v javnem oblaku Windows Azure in uporablja predlagane načrtovalske vzorce. Aplikacija je bila v osnovi eno najemniška in razvita s pomočjo ogrodja .NET, po pristopu MVC. Izvajala se je lokalno in ni izkoriščala uporabljala tehnologije oblaka. Preko implementacije vzorcev, smo ji kasneje dodali še nekatere druge tehnologije, kot so SignalR, OWIN, knjižnice Azure itd., kar je omogočilo implementacijo vzorcev in prenos na oblak Windows Azure. Problem, zaradi katerega smo se lotili implementacije vzorcev, je bil v veliki meri realen. Pred implementacijo vzorcev smo imeli razvito spletno trgovino, katero smo želeli razširiti za delovanje v oblaku ter ji dodati nove funkcionalnosti, katere prinašajo predlagani načrtovalski vzorci. S problemom prenosa obstoječe aplikacije v oblak, se je na globalni ravni najverjetneje srečalo že kar nekaj podjetij, ki se ukvarja z razvojem programske opreme. Z vedno večjim promoviranjem računalništva v oblaku in nižanjem cen uporabe storitev v oblaku, obstaja možnost, da se bo vedno več podjetij odločalo za prenos aplikacij v oblak.

Prvi vzorec spreminja podatkovni kontekst, s katerim dostopamo do podatkovne baze, kar omogoča izbiro izoliranosti podatkov. S pomočjo dinamičnega objekta smo takšno spremembo naredili globalno. V nasprotnem primeru bi potrebovali dva različna podatkovna konteksta in ju ločiti s pogojnimi stavki ali v najslabšem primeru, v kolikor ne bi uporabili vzorca, zgraditi dve ločeni aplikaciji, vsako za drug način izolacije podatkov.

Drugi vzorec, ki opisuje avtentikacijo in avtorizacijo uporabnikov pri dostopu do spletnih storitev, s pomočjo Azure Active Directory, uporablja varnostni mehanizem, ki se nahaja na strani Microsofta. Prednost takšnega pristopa je v tem, da se nam ni potrebno ukvarjati z implementacijo in logiko takšnega varnostnega mehanizma. Slabost pa je, da lahko prezremo obvestilo o narejeni spremembi na storitvi in ne posodobimo lastne aplikacije. Takrat lahko pride do napak v izvajanju ali česa drugega. Vseeno pa omenjeni vzorec omogoča, da podjetja uporabijo organizacijsko strukturo in informacije o zaposlenih v takšni obliki, kot so jih uporabljali do sedaj. Kadar podjetje ne želi uporabljati aktivni imenik v oblaku, moramo v aplikaciji narediti prevezavo na lokalni aktivni imenik.

Zadnji vzorec opisuje področje dvosmerne komunikacije. Vzorci za enosmerno komunikacijo niso nič novega, tudi za dvosmerno ne. Vendar pa vzorci za povratno sporočanje uporabljajo vrste ali teme. Povratna sporočila zahtevajo lastno vrsto za vsakega pošiljatelja. S pomočjo korelacijskih identifikacijskih številk lahko olajšamo povratno sporočanje. V kolikor pa vpeljemo še tehnologijo spletne vtičnice, kot smo to naredili v predlaganem vzorcu, pa s tem spremenimo način dosedanjega povratnega sporočanja. Pošiljatelju sedaj v realnem času,

takoj, ko je odgovor pripravljen, le-tega posredujemo. Seveda pa mora imeti pošiljatelj za takšno delovanje vklopljen spletni brskalnik in biti povezan do dostopnega vozlišča, drugače ne prejme odgovora. Zaradi tega je zaželena uporaba tabel ali katere druge oblike trajne hrambe podatkov, v katere shranimo odgovor in ga lahko pošiljatelj prevzame kadarkoli.

Tekom izdelave magistrske naloge smo prišli do ugotovitve, da na področju računalništva v oblaku načrtovalski in arhitekturni vzorci obstajajo, vendar pa jih za določena področja (dvosmerno sporočanje, neodvisnost dostopa do podatkovne baze, spreminjanje uporabniškega vmesnika v večnajemniški aplikaciji ipd.) še primanjkuje. Načrtovalski vzorci, ki so predlagani v tej magistrski nalogi, so osredotočeni na implementacijo s pomočjo javnega oblaka Windows Azure, kar bi bilo potrebno v kasnejši fazi preoblikovati v bolj generične vzorce, kateri bi lahko bili uporabljeni tudi na drugih javnih ali celo privatnih oz. hibridnih oblakih. Vseeno predlagani načrtovalski vzorci rešujejo nekatere ključne probleme (večnajemništvo, avtorizacija dostopa do virov, povratna sporočilna komunikacija, dinamično spreminjanje podatkovnega konteksta), ki se lahko pojavijo pri razvoju aplikacij, specifičnih za oblak.

6.2. Sklep

Skozi magistrsko nalogo smo si pobližje ogledali računalništvo v oblaku, ponudnike računalniških oblakov in storitve oblaka. Nadaljevali smo z opisom in analizo načrtovalskih in arhitekturnih vzorcev, ki lahko pospešijo in olajšajo rešitev določenega problema pri razvoju programskih sistemov. V nadaljevanju smo definirali tri načrtovalske vzorce, ki so namenjeni rešitvi problemov na področju računalništva v oblaku. Za implementacijo vzorcev smo uporabili primer spletne trgovine, ki smo jo postavili v javni oblak Windows Azure. Skupaj s tehnologijami .NET, HTML5, SignalR in ostalimi, smo na primeru spletne trgovine uspešno implementirali vzorce, katere pa je možno prenesti tudi na ostale spletne aplikacije.

Razvijalci so s prihodom računalništva v oblaku dobili veliko novih možnosti za razvoj programskih sistemov, saj oblak ponuja nove funkcionalnosti, katere pa do sedaj niso bile tako enostavno dostopne. Večnajemništvo, možnost enostavnega razširjanja in porazdelitev bremena, enostavno upravljanje s storitvami na enem mestu, so le nekatere funkcionalnosti računalništva v oblaku. Vendar pa je potrebno aplikacije in storitve razviti tako, da bodo v čim večji meri zadoščale lastnostim oblaka. Za učinkovito reševanje določenih problemov se razvijalci zatekajo k ostalim razvijalcem, ki so se s takšnimi problemi že srečali in jih tudi uspešno rešili. Pojavili so se načrtovalski in arhitekturni vzorci, katerih pa zaradi relativno mlade tehnologije računalniških oblakov še ni veliko.

Cilj magistrske naloge je bil predlagati nove in inovativne načrtovalske vzorce za razvoj aplikacij, specifičnih za oblak. V ta namen smo zgradili tri vzorce. Prvi omogoča izbiro

podatkovne izoliranosti v večnajemniški aplikaciji. To pomeni, da bo lahko novi najemnik v večnajemniški aplikaciji izbiral kakšno stopnjo izoliranosti podatkov bi želel. Izbira lahko med lastnimi tabelami povezanimi v podatkovno shemo in skupnimi tabelami, ločenimi s ključem najemnika. Naslednji se loteva avtentikacije in avtorizacije uporabnika pri dostopu do spletnih storitev REST s pomočjo Azure Active Directory. Zadnji vzorec gradi dvosmerno komunikacijo s pomočjo sporočilnih vrst storitvenega vodila in spletne vtičnice.

Predlagane vzorce bi bilo potrebno ponuditi širši množici razvijalcev, ki bi vzorce pregledali in preizkusili na realnih primerih. V primeru uspešne implementacije vzorcev in dobrega odziva, bi predlagane vzorce lahko prevzela celotna skupnost razvijalcev. Vseeno pa bi se lahko vzorce še izboljšalo in naredilo bolj generične, kar bi omogočilo, da bi lahko razvijalci uporabili vzorce pri različnih ponudnikih javnega oblaka, seveda ob predpogoju, da le-ti ponujajo vse potrebne storitve. Vzorci so implementirani s pomočjo ogrodja .NET in programskega jezika C#. S pomočjo ostalih tehnologij smo lahko hitro implementirali določene dele predlaganih vzorcev. Prvi vzorec tako s pomočjo ogrodja Entity Framework in z zamenjavo podatkovnega konteksta rešuje problem, katerega bi drugače morali reševati z ustvarjanjem ločenih povezav za izbrani tip podatkovne izolacije. Drugi vzorec uporablja spletni vmesnik OWIN, kateri razdvoji strežnik in aplikacijo. Tako lahko poskrbimo za povezavo z avtentikacijskim strežnikom in tudi za preoblikovanje odgovorov, ki pridejo z njegove strani. SignalR pa podira meje med strežnikom in odjemalčevim spletnim brskalnikom, tako da ga lahko uporabimo za povratno sporočanje.

Vidimo torej, kako nam lahko tehnologija, v kombinaciji z vzorci, pomaga pri hitrem in učinkovitem razvoju programskih sistemov. Spomnimo, da so razvijalci tisti, ki bodo najverjetneje morali v prihajajočem obdobju zgraditi in prenesti večje število aplikacij v oblak. Pri tem jim bodo v veliko pomoč načrtovalski vzorci in uporaba sodobne tehnologije.

VIRI IN LITERATURA

[1] R. Buyya et al, *Cloud Computing: Principles and Paradigms*, New Jersey: John Wiley & Sons, 2011, pogl. 1.

[2] P. Mell, T. Grance, *The NIST Definition of Cloud Computing*, Gaithersburg: NIST, 2011, str. 2.

[3] ZDNet (2008). HP Dismisses Cloud 'Hype'. Dostopno na: http://www.zdnet.com/news/hp-dismisses-cloud-hype/255222 (januar 2014).

[4] M. Armbrust et al, *Above the Clouds: A Berkeley View of Cloud Computing*, Berkley: University of California at Berkeley, 2009, pogl. 3.

[5] R. Giordanelli, C. Mastroianni, *The Cloud Computing Paradigm: Characteristics, Opportunities and Research Issues*, Rende: ICAR-CNR, 2010, str. 4.

[6] Salesforce (2011). A Complete History of Cloud Computing. Dostopno na: http://www.salesforce.com/uk/socialsuccess/cloud-computing/the-complete-history-of-cloud-computing.jsp (januar 2014).

[7] Business Insider (2013). The 15 Most Valuable Cloud Computing Companies In The World Are Worth Way More Than You'd Think. Dostopno na: http://www.businessinsider.com/the-15-most-valuable-cloud-computing-companies-2013-7?op=1 (januar 2014).

[8] Google (2014). What is Google App Engine? Dostopno na: https://developers.google.com/appengine/docs/whatisgoogleappengine (februar 2014).

[9] P. Louridas, "Up in the Air: Moving Your Applications to the Cloud", *Software, IEEE*, št. 27, zv. 4, str. 6-11, 2010.

[10] S. L. Garfinkel, *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*, Massachusetts: MIT Press, 1999, str. 1.

[11] Business Insider (2013). Amazon Cloud Beats IBM, Microsoft, And Google In Cloud Computing, Says Report. Dostopno na: http://www.businessinsider.com/amazon-cloud-beats-ibm-microsoft-google-2013-11 (januar 2014).

[12] B. Sosinsky, *Cloud Computing Bible*, Indianapolis: Wiley Publishing, 2011, pogl. 1, 4, str. 216-217.

[13] M.P. McGrath, *Understanding PaaS*, California: O'Reilly, 2012, pogl. 1.

[14] T. J. Bittman, *Private Cloud Computing: An Essential Overview*, Gartner Research, 2010, str. 3. Dostopno na: https://www.gartner.com/doc/1476032/private-cloud-computing-essential-overview (februar 2014).

[15] ZDNet (2013). Microsoft finds a new way to deliver a private cloud in a box. Dostopno na: http://www.zdnet.com/microsoft-finds-a-new-way-to-deliver-a-private-cloud-in-a-box-7000016279/ (februar 2014).

[16] Gartner (2012). Mind the Gap: Here Comes Hybrid Cloud. Dostopno na: http://blogs.gartner.com/thomas_bittman/2012/09/24/mind-the-gap-here-comes-hybrid-cloud/ (februar 2014).

[17] W. Forrest, *Clearing the air on cloud computing*, McKinsey & Company, 2009, str. 12. Dostopno na: http://www.isaca.org/Groups/Professional-English/cloud-computing/GroupDocuments/McKinsey_Cloud%20matters.pdf (februar 2014).

[18] IDC (2008). Defining "Cloud Services" and "Cloud Computing". Dostopno na: http://blogs.idc.com/ie/?p=190 (februar 2014).

[19] E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley, 1995, pogl. 1, 3, 4, 5.

[20] Amazon (2014). Amazon Web Services. Dostopno na: http://aws.amazon.com/documentation/ (februar 2014).

[21] Microsoft (2014). Windows Azure Services. Dostopno na: http://www.windowsazure.com/en-us/services/ (februar 2014).

[22] Z. Hill et al, "Early Observations on the Performance of Windows Azure", v zborniku *19th ACM International Symposium on High Performance Distributed Computing*, New York, New York, jun. 2010, str. 367-376.

[23] B. Wilder, *Cloud Architecture Patterns*, California: O'Reilly, 2012, str. 77.

[24] F. Buschman et al, *Pattern-Oriented Software Architecture: A System of Patterns. Vol 1*, Chichester: John Wiley & Sons, 1996, str. xii, 3, 25-30, 97-98, 123-124, 169-170.

[25] G. Rasool, P. Maeder, I. Philippow, "Evaluation of design pattern recovery tools", *Procedia Computer Science*, št. 3, str. 813-819, 2011.

[26] J.C. Franchitti. Software Engineering: Design Patterns, Architectural Patterns. New York: New York University, str. 7-9. Dostopno na: http://www.nyu.edu/classes/jcf/g22.2440-001_sp06/slides/session8/g22_2440_001_c82.pdf (marec 2014).

[27] C. Pehling et al, "An Architectural Pattern Language of Cloud-Based Applications", v zborniku *18th Conference on Pattern Languages of Programs*, New York, New York, 2011, članek št. 2.

[28] D. Betts et al, *Developing Multi-tenant Applications for the Cloud on Microsoft Windows Azure, 3rd Edition*, Redmond: Microsoft, 2012, str. 32-34, 74-77.

[29] W3C (2004) Web Services Architecture. Dostopno na: http://www.w3.org/TR/ws-arch/ (april 2014).

[30] T. Reimer, P. Abraham, Q. Tan, "Federated Identity Access Broker Pattern for Cloud Computing", v zborniku *16th International Conference on Network-Based Information Systems*, Gwangju, South Korea, sept. 2013, str. 134-140.

[31] IETF (2001). SOAP Extensions: Basic and Digest Authentication. Dostopno na: http://tools.ietf.org/html/draft-cunnings-salz-soap-auth-00 (april 2014).

[32] P.J. Danielsen, A. Jeffrey, "Validation and Interactivity of Web API Documentation", v zborniku *20th IEEE International Conference on Web Services*, Santa Clara, California, jun. 2013, str. 523-530.

[33] Microsoft (2014). About Traffic Manager Load Balancing Methods. Dostopno na: http://msdn.microsoft.com/en-US/library/azure/dn339010.aspx (april 2014).

[34] Microsoft (2012). How to Authenticate Web Users with Azure Active Directory Access Control. Dostopno na: http://azure.microsoft.com/en-us/documentation/articles/active-directory-dotnet-how-to-use-access-control/ (april 2014).

[35] IETF (2012). The OAuth 2.0 Authorization Framework. Dostopno na: http://tools.ietf.org/html/rfc6749 (april 2014).

[36] Microsoft (2014). Windows Azure Queues and Windows Azure Service Bus Queues – Compared and Contrasted. Dostopno na: http://msdn.microsoft.com/en-us/library/hh767287.aspx (april 2014).

[37] Microsoft (2006). Multi-Tenant Data Architecture. Dostopno na: http://msdn.microsoft.com/en-us/library/aa479086.aspx (april 2014).

[38] Microsoft (2014). Adding, Updating, and Removing an Application. Dostopno na: http://msdn.microsoft.com/en-us/library/azure/dn132599.aspx (april 2014).

[39] Microsoft (2014). SignalR Documentation. Dostopno na: https://github.com/SignalR/SignalR/wiki (april 2014).

[40] Gartner (2008). Cloud Computing. Dostopno na: http://www.gartner.com/it-glossary/cloud-computing/ (maj 2014).

[41] IBM (2009). Cloud computing for the enterprise: Capturing the cloud. Dostopno na http://www.ibm.com/developerworks/websphere/techjournal/0904_amrhein/0904_amrhein.html (maj 2014).

[42] S. Krishnan, *Programming Windows Azure*, California: O'Reilly, 2010, pogl. 1.

[43] R. Hill, L. Hirsch, P. Lake, S. Moshiri, *Guide to Cloud Computing Principles and Practice*, London: Springer, 2013, pogl. 1.

[44] S. S. Yau, H. G. An, "Software Engineering Meets Services and Cloud Computing", *Computer*, št. 44, zv. 10, str. 47-53, 2011.

[45] EzeCastle (2010). The History Of Cloud Computing. Dostopno na: http://www.eci.com/cloudforum/cloud-computing-history.html (februar 2014).

[46] Microsoft (2014). Microsoft Azure pricing overview. Dostopno na: http://azure.microsoft.com/en-us/pricing/overview/ (maj 2014).

[47] A. Ampatzoglou, S. Charalampidou, I. Stamelos, "Research State of the Art on GoF Design Patterns: A Mapping Study", *Journal of Systems and Software*, št. 86, zv. 10, jul. 2013, str. 1945-1964.

[48] Microsoft (2014). Azure Service Bus - Service Bus Fundamentals. Dostopno na: http://azure.microsoft.com/en-us/documentation/articles/fundamentals-service-bus-hybrid-solutions/ (marec 2014).

[49] A. V. Katherine, K. Alagarsamy, "Conventional Software Testing Vs. Cloud Testing", *International Journal Of Scientific & Engineering Research*, št. 3, zv. 9, str. 145-149, 2012.

[50] TechTarget (2013). REST vs. SOAP: How to choose the best Web Service. Dostopno na: http://searchsoa.techtarget.com/tip/REST-vs-SOAP-How-to-choose-the-best-Web-service (marec 2014).

[51] L. Li, W. Chou, "Design and Describe REST API without Violating REST: A Petri Net Based Approach", v zborniku *2011 IEEE International Conference on Web Services*, Washington, USA, jul. 2011, str. 508-515.

[52] Springer (2014). Tenant-Isolated Component. Dostopno na: http://www.cloudcomputingpatterns.org/Tenant-isolated_Component (maj 2014).

[53] T. Erl, *SOA Design Patterns*, Boston: Prentice Hall, 2009, str. 90, 489.