Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Alen Juršič

Izdelava komponent za razvoj mikrostoritev v ogrodju .NET Core

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Matjaž Branko Jurič
Somentor: asist. Jan Meznarič

Ljubljana, 2020

COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja. Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod odprtokodno licenco MIT. Podrobnosti licence so dostopne na spletni strani https://opensource.org/licenses/MIT.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Analizirajte gradnike in delovanje mikrostoritev, identificirajte prednosti in slabosti ter naslovite področje vsebnikov. Podrobno opišite uporabo mikrostoritev v okolju .NET in pri tem naslovite izzive konfiguriranje, odkrivanja in preverjanja vitalnosti delovanja mikrostoritev. Implementirajte rešitev in jo ovrednotite.

Rad bi se zahvalil mentorju prof. dr. Matjažu Branku Juriču in somentorju asist. Janu Meznariču, ki sta me usmerjala, dajala nasvete in mi pomagala pri implementaciji diplomske naloge. Zahvaljujem se tudi vsem sorodnikom in prijateljem, ki so mi stali ob strani in me podpirali.

Kazalo

Povzetek

Abstract

1 Uvod 1

2 Gradniki in delovanje mikrostoritev 3

2.1 Kaj so mikrostoritve? 3
2.2 Kako so nastale mikrostoritve? 4
2.3 Kako deluje arhitektura mikrostoritev? 5
2.4 Prednosti in slabosti mikrostoritev 7
2.5 Prihodnost mikrostoritev 8
2.6 Docker in vsebniki 9

3 Mikrostoritve v C# in .NET Core 11

3.1 Na kratko o .NET Core 11
3.2 Izdelava mikrostoritve v .NET Core 12

4 Opis problema 17

4.1 Problem preverjanja vitalnosti mikrostoritev 17
4.2 Problem konfiguracije mikrostoritev 18
4.3 Problem odkrivanja mikrostoritev 18

5 Implementacija rešitve 21

5.1 Preverjanje vitalnosti mikrostoritev 21

5.2 Konfiguracija mikrostoritev 30
5.3 Odkrivanje mikrostoritev 46
5.4 Evalvacija 59

6 Zaključek **61**

Literatura **63**

Slike

2.1 API prehod v arhitekturi mikrostoritev. 6

3.1 Diagram arhitekture ekosistema .NET. 12
3.2 Kreiranje novega projekta za izdelavo preproste mikrostoritve. 13
3.3 Izbira tipa projekta. 13
3.4 Nastavitev lokacije, imena projekta in rešitve. 14
3.5 Izbira predloge za projekt. 15
3.6 Prikaz strukture ustvarjenega projekta. 15
3.7 Primer odgovora naše mikrostoritve na poslan zahtevek. . . . 16

5.1 Struktura implementacije mehanizma za preverjanje vitalnosti mikrostoritev. 23
5.2 Primer odgovora HTTP na zahtevo za preverbo vitalnosti mikrostoritve. 31
5.3 Struktura implementacije mehanizma za konfiguracijo mikrostoritev. 33
5.4 Prikaz vrednosti na strežniku Consul preko njegovega grafičnega vmesnika. 44
5.5 Prikaz odgovora naše mikrostoritve na zahtevo prikaza pridobljenih konfiguracijskih vrednosti. 45
5.6 Struktura implementacije mehanizma za odkrivanje mikrostoritev. 48
5.7 Primer prikaza registrirane storitve preko grafičnega vmesnika strežnika Consul. 58

Seznam uporabljenih kratic

kratica	angleško
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
REST	Representational State Transfer
SOA	Service-Oriented Architecture
MSMQ	Microsoft Message Queuing
API	Application Programming Interface
IoT	Internet of Things
IPC	Inter Process Communication
XAML	Extensible Application Markup Language
JSON	JavaScript Object Notation
URL	Uniform Resource Locator

Povzetek

Naslov: Izdelava komponent za razvoj mikrostoritev v ogrodju .NET Core

Avtor: Alen Juršič

Uporaba mikrostoritev je zadnje čase čedalje pogostejša, vendar je njihova izdelava lahko zelo zahtevna. Pri samem razvoju je potrebno paziti na veliko dejavnikov, ki nam lahko kasneje povzročajo probleme, na primer napake delovanja, sprememba konfiguracij in premestitev lokacije izvajanja mikrostoritev. V okviru te diplomske naloge so za omenjene probleme razvite tri komponente: komponenta za preverjanje vitalnosti mikrostoritev, konfiguracijo mikrostoritev in odkrivanje mikrostoritev. Komponente so razvite tako, da se lahko hitro in enostavno uporabijo pri razvoju mikrostoritev v ogrodju .NET Core, ki postaja vedno bolj priljubljeno.

Ključne besede: mikrostoritve, .NET Core, Consul, Etcd, preverjanje vitalnosti mikrostoritev, konfiguracija miksrostoritev, odkrivanje mikrostoritev.

Abstract

Title: Implementation of components for microservices development in .NET Core framework

Author: Alen Juršič

Microservices have become hugely popular in recent years, but their developement can still be very demanding. In the development itself, it is necessary to pay attention to many factors that can cause problems later. Some of these are operational errors, configuration changes, and relocation of running microservices. In this thesis, we developed three components for the mentioned problems: a component for health checking, a component for configuration and a component for service discovery. The developed components can be used quickly and easily in the development of microservices in the .NET Core framework, which is becoming more and more popular.

Keywords: microservices, .NET Core, components, microservice health check, microservice configuration, microservice discovery.

Poglavje 1

Uvod

Izdelava spletnih aplikacij z uporabo mikrostoritev je dandanes eden najbolj razširjenih načinov. Mikrostoritve pripomorejo k boljšemu delovanju, preglednosti ter nadzoru delujoče spletne aplikacije. Za izdelavo imamo na voljo veliko programskih jezikov, prav tako so za nekatere že na voljo posebna ogrodja, ki nam implementacijo spletnih aplikacij z mikrostoritvami precej olajšajo.

Za programski jezik C# je podjetje Microsoft pripravilo ogrodje .NET Core, ki poleg izdelave aplikacij za Windows, Linux in macOS platforme omogoča tudi druge stvari, med njimi tudi izdelavo mikrostoritev. Izdelava mikrostoritve sama po sebi ni preveč zapletena, je pa za dobro delovanje potrebno implementirati kar nekaj dodatnih funkcionalnosti. Nekatera ogrodja za izdelavo mikrostoritev že vsebujejo komponente za podporo standardnih funkcionalnosti, kot so preverjanje vitalnosti, konfiguracija, odkrivanje itd.

V tej diplomski nalogi bomo predstavili, kako se razvije komponente omenjenih treh funkcionalnosti za razvoj mikrostoritev v ogrodju .NET Core. Postopoma bomo razvili tri komponente, in sicer najprej komponento za preverjanje vitalnosti mikrostoritve, potem komponento za odkrivanje storitev in na koncu še komponento za konfiguracijo mikrostoritev. Uporaba razvitih komponent bo prikazana na preprosti mikrostoritvi, ki jo bomo za ta namen razvili skozi diplomsko nalogo.

Najprej bomo v 2. poglavju predstavili osnovne gradnike in arhitekturo mikrostoritev. Nato bomo v 3. poglavju na kratko opisali ogrodje .NET Core in v njem razvili preprosto mikrostoritev. V 4. poglavju bomo opisali tri probleme, za katere bomo v 5. poglavju tudi razvili rešitve in njihovo uporabo prikazali na mikrostoritvi iz poglavja 3. Za zaključek bomo v 6. poglavju še povzeli rezultate naših rešitev.

Poglavje 2

Gradniki in delovanje mikrostoritev

2.1 Kaj so mikrostoritve?

Mikrostoritve so posamezne komponente v arhitekturi mikrostoritev - to je slog programiranja, kjer je aplikacija kot celota sestavljena iz večih storitev. Glavna ideja mikrostoritev je izdelava velikih, kompleksnih aplikacij, ki se bodo verjetno s časom širile in spreminjale. Beseda mikro nakazuje na to, da naj bodo storitve v arhitekturi mikrostoritev majhne, neobsežne. Majhnost storitev pa ni glavni cilj. Pri arhitekturi mikrostoritev se poskušamo osredotočiti predvsem na to, da v posamezni storitvi rešujemo le določen tip problemov. Večje aplikacije so lahko sestavljene tudi iz nekaj sto storitev, med katerimi so lahko nekatere po velikosti dosti večje od drugih. Sam koncept mikrostoritev niti ni nova stvar, saj lahko nekaj podobnega opazimo že pri storitveno usmerjeni arhitekturi (ang. SOA - Service-oriented-architecture) [7, 5, 20, 21, 19].

2.1.1 Primerjava arhitekture mikrostoritev s storitveno usmerjeno arhitekturo

Tako arhitektura mikrostoritev kot tudi storitveno usmerjena arhitektura predstavljata storitve kot njune glavne komponente, vendar pa se glede na značilnosti mikrostoritev močno razlikujeta. Storitve pri arhitekturi mikrostoritev so enonamenske storitve, kar pomeni, da je njihova funkcionalnost namenjena opravljanju ene določene stvari, medtem ko lahko storitve pri SOA predstavljajo vse od manjših aplikacijskih storitev pa do velikih naprednih storitev. Eno glavnih načel pri SOA je deljenje storitev, medtem ko naj se pri arhitekturi mikrostoritev poskušajo narediti čim bolj neodvisne storitve, kar pomeni, da naj storitev predstavlja eno samo enoto z minimalnimi odvisnostmi. Čeprav je razlik med izbranima arhitekturama še mnogo, naj omenimo le še, da SOA temelji na protokolih za sporočanje MSMQ in SOAP, arhitektura mikrostoritev pa večinoma temelji na protokolih REST in protokolih za enostavno sporočanje. Določitev, katera arhitektura je boljša, je odvisna predvsem od namena aplikacije, ki jo izdelujemo [10, 18].

2.2 Kako so nastale mikrostoritve?

V začetkih razvoja spletnih aplikacij je bila najbolj uporabljena monolitna arhitektura, ki v eni aplikaciji vsebuje vse med sabo zelo povezane gradnike, potrebne za delovanje. Ko so se za aplikacijo pojavile zahteve po spremembah, je bilo potrebno posodobiti celotno aplikacijo. Sama aplikacija je tako s časom postajala čedalje kompleksnejša in težja za vzdrževanje. S prihodom vsebnikov (ang. container), ki so opisani v poglavju 2.6, se je način razvoja aplikacij močno spremenil. Večje, do sedaj trdo sklopljene aplikacije, so se začele izdelovati z združevanjem več manjših enot. Za lažjo ponazoritev si predstavljajte veliko kocko, ki jo nadomestijo z enako veliko kocko, sestavljeno iz več manjših kock [7, 22, 6, 18, 19].

Neka aplikacija ponavadi zajema nabor večih funkcionalnosti, kot so upravljanje uporabnikov, pregled izdelkov, izvrševanje transakcij itd. V mikrosto-

ritvah je vsak sklop podobnih funkcionalnosti implementiran v svoji storitvi. Tako so spletne aplikacije, kot so Amazon, eBay in Netflix, v ozadju razdrobljene na več mikrostoritev, uporabniku pa so na voljo kot sklopljena celota [5].

2.3 Kako deluje arhitektura mikrostoritev?

Glede na zahteve aplikacije lahko arhitektura mikrostoritev vsebuje [3]:

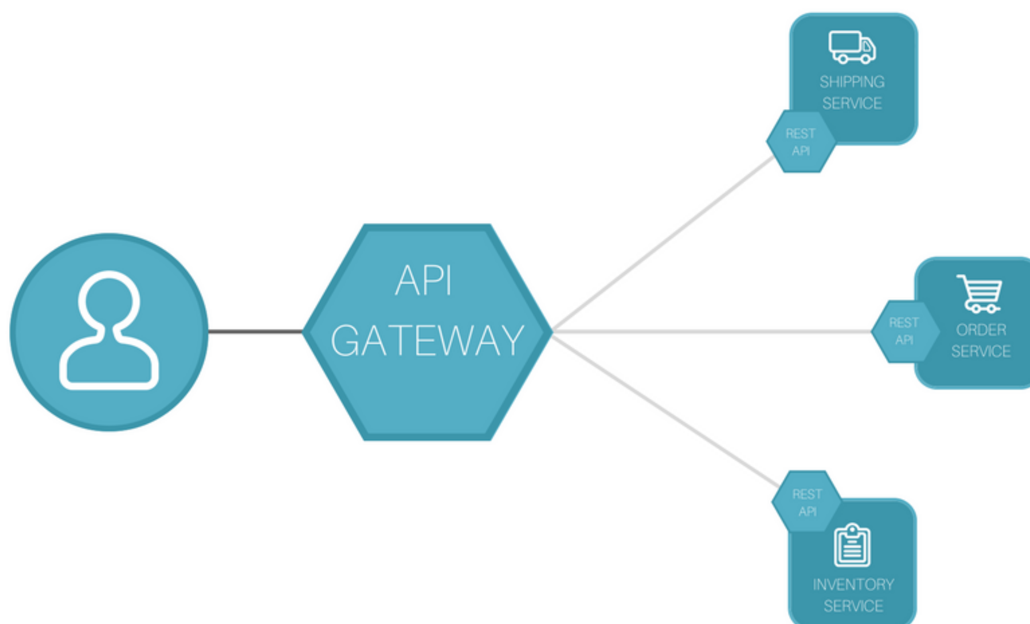
- **API prehod** (ang. API gateway), ki je vstopna točka za vse uporabnike aplikacije.
- **Mikrostoritve** so majhne storitve, ki opravljajo določeno nalogo.
- **Podatkovno bazo.**
- **Medstoritveno komunikacijo**, ki lahko uporablja razne protokole, da storitve komunicirajo med sabo.

2.3.1 API prehod

Pri monolitnih aplikacijah uporabnik naredi HTTP zahtevek na eno izmed instanc aplikacije. Kako pa uporabnik naredi zahtevek pri aplikaciji, ki je sestavljena iz mikrostoritev? Eden od možnih načinov je, da uporabnik naredi RESTful HTTP zahtevek na posamezno storitev aplikacije, ker pa lahko neka spletna stran zahteva sto ali več klicev storitev, postane ta način zelo neučinkovit in počasen [7].

Veliko boljši pristop je, da uporabniki naredijo čim manj zahtevkov na stran, kar lahko dosežemo z uporabo API prehoda. Ta se nahaja med uporabnikom in mikrostoritvami ter skrbi, da ima uporabnik vse API-je mikrostoritev zbrane na enem mestu (glej sliko 2.1). Uporabnik pošlje zahtevo na API prehod, ta pa zahtevo posreduje potrebnim storitvam in rezultat pošlje nazaj uporabniku. Ena najpomembnejših prednosti tega pristopa je, da čim se neka storitev razdeli na več storitev ali pa se morda dve ali več storitev združi v eno, o tem ne rabimo obveščati uporabnikov, ampak samo API

prehod. Uporabniki torej ostanejo nedotaknjeni [7, 5, 6].



Slika 2.1: API prehod v arhitekturi mikrostoritev.

2.3.2 Medstoritvena komunikacija

Pri monolitnih aplikacijah komponente med sabo komunicirajo preko navadnih metodnih klicev. Pri arhitekturi mikrostoritev je to nemogoče, saj različne storitve tečejo na različnih procesih. Storitve morajo tako uporabljati medprocesno komunikacijo (IPC) [7].

Sinhroni HTTP

Sinhroni HTTP je enostaven in pogosto uporabljen mehanizem IPC. Aplikacija mora na odgovor čakati, kar pomeni, da opravilo čaka, dokler nekaj ni opravljeno. Deluje preko interneta in je zelo enostaven za implementacijo zahtevek-odgovor načinov komunikacije. Slabost protokola HTTP pa je,

da ne podpira ostalih načinov komuniciranja, kot je na primer način objavi-naroči (ang. publish-subscribe). [2, 7].

Omejitev je tudi to, da morata biti tako odjemalec kot strežnik sočasno na voljo, kar ni vedno mogoče. Odjemalec HTTP mora tudi vedeti naslov in vrata strežnika. Aplikacija mora tako uporabljati mehanizem za odkrivanje storitev, ki ga bomo v nadaljevanju tudi implementirali [7].

Asinhrono sporočanje

Asinhrono sporočanje je neke vrste alternativa sinhronem HTTP sporočanju. Ta pristop ima veliko prednosti. Proizvajalce sporočil loči od prejemnikov. Proizvajalci sporočil komunicirajo direktno s tako imenovanim posrednikom sporočil in ne rabijo uporabljati mehanizma za odkrivanje storitev. Ena izmed slabosti tega pristopa je potreba po posredniku sporočil [2, 7].

2.4 Prednosti in slabosti mikrostoritev

Tako kot ostali arhitekturni slogi izdelave aplikacij imajo tudi mikrostoritve svoje prednosti in slabosti.

Prednosti [1, 3, 6]:

- Vsaka mikrostoritev je relativno majhna, kar razvijalcu omogoča boljši pregled nad implementacijo in lažje razumevanje njenega delovanja. Če morda v aplikaciji potrebujemo neko mikrostoritev, ki smo jo predhodno že izdelali za neko drugo aplikacijo, jo lahko enostavno vključimo tudi v to.

- Arhitekturni slog mikrostoritev nam omogoča neprekinjeno nameščanje storitev. Vsako storitev lahko skaliramo neodvisno od drugih, kar pomeni, da lahko enostavno naredimo več instanc bolj obremenjene storitve.

- Mikrostoritve odpravljajo kakršnokoli zavezanost točno določeni tehnologiji. Pri razvoju nove storitve lahko razvijalci izberejo jezik in tehnologije, ki so za to storitev najprimernejše.

Slabosti [1, 6]:

- Ena od slabosti mikrostoritev je že samo ime, saj razvijalec pogosto besedo "mikro" jemlje preveč dobesedno. Tako se bolj osredotoča na to, da bodo storitve, ki jih razvija, čim manjše, namesto, da bi izkoristil glavni cilj mikrostoritev, to je razdrobiti aplikacijo na več delov in jo tako narediti bolj prilagodljivo za razvoj in namestitev.

- Razvijalec se mora soočati z veliko kompleksnostjo izdelave porazdeljenih aplikacij. Tudi razvojna okolja so zaenkrat večinoma osredotočena na izdelavo monolitnih aplikacij in ne nudijo izrecne podpore za razvoj porazdeljenih aplikacij.

- Pogosto prve različice aplikacij niso tako obširne in kompleksne, zato razvijalec ne čuti potrebe po uporabi arhitekture mikrostoritev. Tudi samo testiranje aplikacije, ki temelji na tej arhitekturi, je kompleksnejše. Za razliko od monolitnih aplikacij mora pri testiranju mikrostoritev razvijalec pognati vse odvisne storitve, da jo lahko sploh testira.

2.5 Prihodnost mikrostoritev

Arhitektura mikrostoritev postaja med razvijalci čedalje bolj priljubljena. Bolj kot se bodo mikrostoritve uporabljale, bolj se bodo pojavljali novi problemi in tudi nove rešitve pri implementaciji mikrostoritev. Povečanje števila mikrostoritev v neki organizaciji poveča tudi delo, ki se ukvarja z reševanjem problemov, kot so spremljanje stanj storitev in njihovih atributov, avtomatsko popravljanje storitve itd. Z mislijo na te probleme morajo razvijalci mikrostoritev poleg poslovne logike narediti še mnogo drugih stvari [4].

2.6 Docker in vsebniki

Docker je orodje, namenjeno lažji izdelavi, namestitvi in izvajanju aplikacij z uporabo vsebnikov [13]. To so standardne programske enote, ki zapakirajo kodo in vse njene odvisnosti, tako da je aplikacija zanesljivo in hitro prenosljiva iz enega okolja v drugo. Slika vsebnika je samostojen izvršljiv paket, ki vsebuje programsko opremo ter vse, kar je potrebno, da se ta lahko izvaja [11]. Docker si torej lahko predstavljamo kot zelo lahko virtualno izvajalsko okolje, kar pomeni, da vsebuje samo stvari oziroma enote, potrebne za osnovno delovanje [17].

Docker temelji na arhitekturi odjemalec-strežnik. Strežnik Docker je odgovoren za vse akcije, ki so povezane z vsebniki. Strežnik dobi ukaz s strani odjemalca preko ukazne vrstice ali REST API-jev ter ga izvede. Slike so osnovna enota Dockerja. Iz njih se naredijo prej omenjeni vsebniki, v katere lahko nato namestimo aplikacije [12]. Docker je torej imenitno orodje za upravljanje, namestitev in izvajanje mikrostoritev.

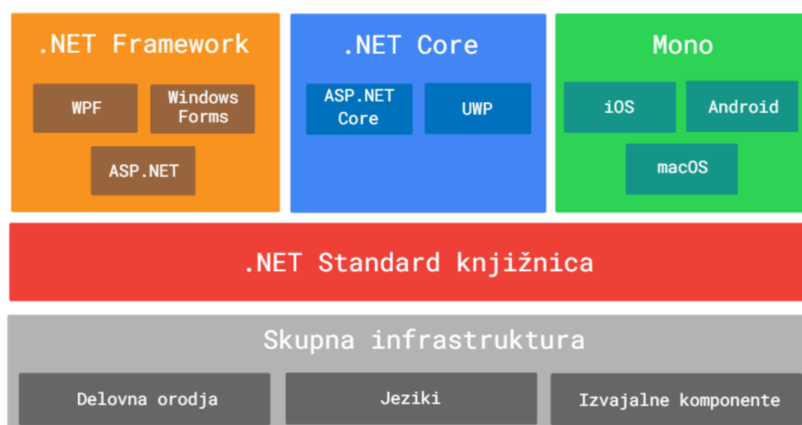
Poglavje 3

Mikrostoritve v C# in .NET Core

3.1 Na kratko o .NET Core

.NET Core je ogrodje, namenjeno izdelavi aplikacij za operacijske sisteme Windows, Linux in macOS. Njegova glavna prednost pred ostalimi ogrodji je, da se lahko uporabi za izdelavo različnih vrst programske opreme. Nekatere izmed teh so spletne aplikacije, mobilne aplikacije, namizne aplikacije, aplikacije v oblaku, mikrostoritve, aplikacijski programski vmesniki, aplikacije za IoT in še bi lahko naštevali. Je odprtokodno ogrodje, kar pomeni, da je izvorna koda dostopna vsem uporabnikom. Trenutno podpira programiranje v programskih jezikih C#, F#, XAML, VB.NET in TypeScript, ki so ravno tako odprtokodni [14].

.NET Core je ena izmed treh komponent ekosistema .NET (glej sliko 3.1). Je prenovljena in posodobljena različica ogrodja .NET Framework. Vse tri komponente temeljijo na .NET Standardu, ki je uradna specifikacija .NET API-jev [15, 16].



Slika 3.1: Diagram arhitekture ekosistema .NET.

3.2 Izdelava mikrostoritve v .NET Core

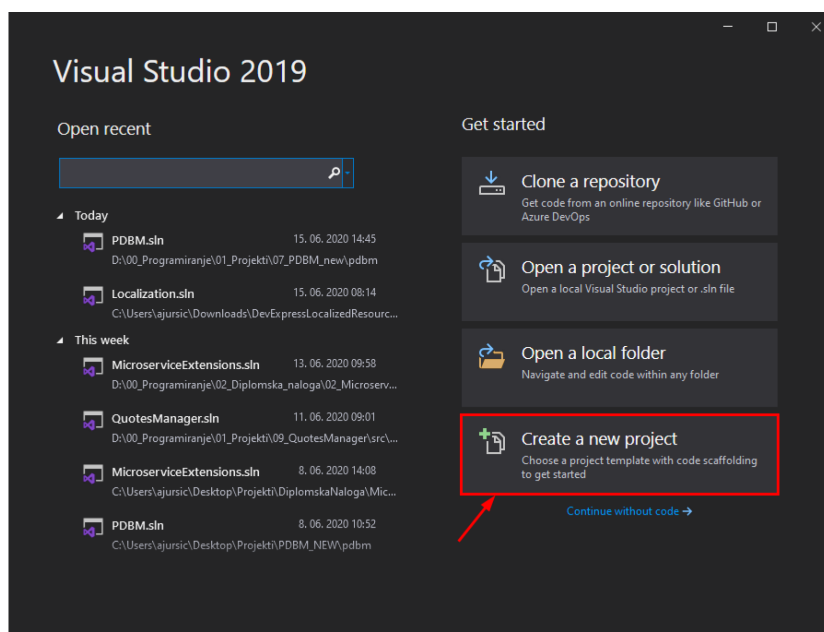
Za lažje razumevanje postopka izdelave mikrostoritev v .NET Core bomo z uporabo programskega jezika C# naredili primer preproste mikrostoritve. Za izdelavo bomo uporabili razvojno okolje Visual Studio Community 2019, ki je brezplačno za vse uporabnike.

Najprej bomo odprli Visual Studio in ustvarili nov projekt s klikom na gumb 'Create new project', kot je prikazano na sliki 3.2.

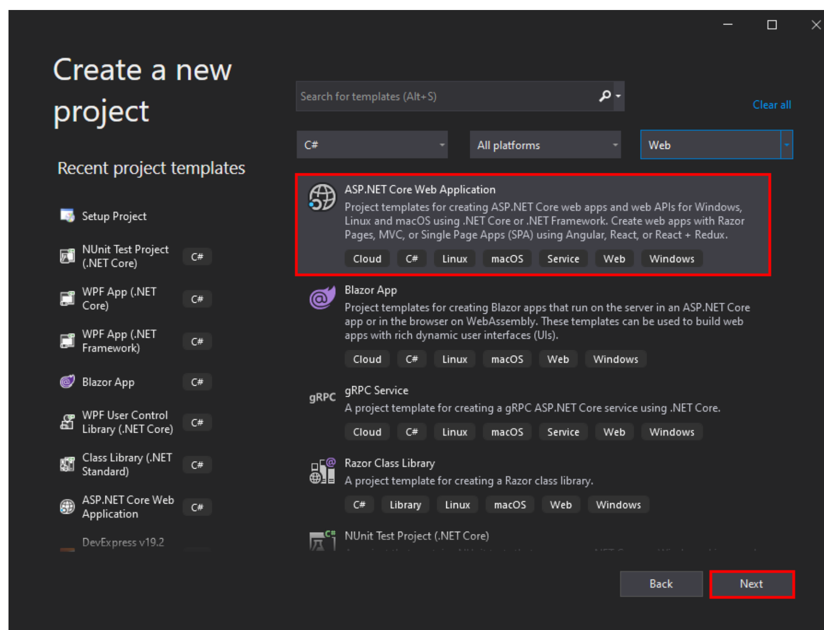
Po kliku se nam prikaže seznam vseh že obstoječih tipov projektov. Za izdelavo mikrostoritve izberimo tip ASP.NET Core Web Application, ki je namenjen izdelavi spletnih aplikacij ASP.NET Core in API-jev z uporabo .NET Core (glej sliko 3.3).

Ko izberemo tip, se nam pokažejo tri vnosna polja (glej sliko 3.4). Polje z napisom 'Project name' je namenjeno poimenovanju projekta, ki ga bomo ustvarili. V našem primeru naj bo to "SimpleMicroservice". V polju z napisom 'Location' izberemo lokacijo mape, v kateri se bo projekt nahajal. Zadnje vnosno polje pa je namenjeno imenu rešitve, ki bo vsebovala ta projekt.

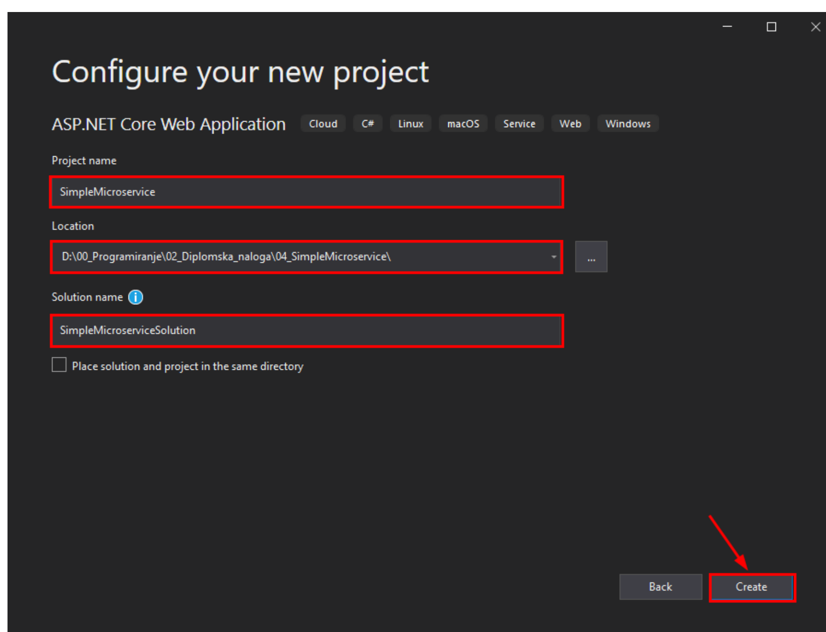
Preden se projekt ustvari, moramo samo še izbrati predlogo za izbran tip aplikacije, ki jo hočemo ustvariti. V našem primeru bomo izbrali kar



Slika 3.2: Kreiranje novega projekta za izdelavo preproste mikrostoritve.



Slika 3.3: Izbira tipa projekta.

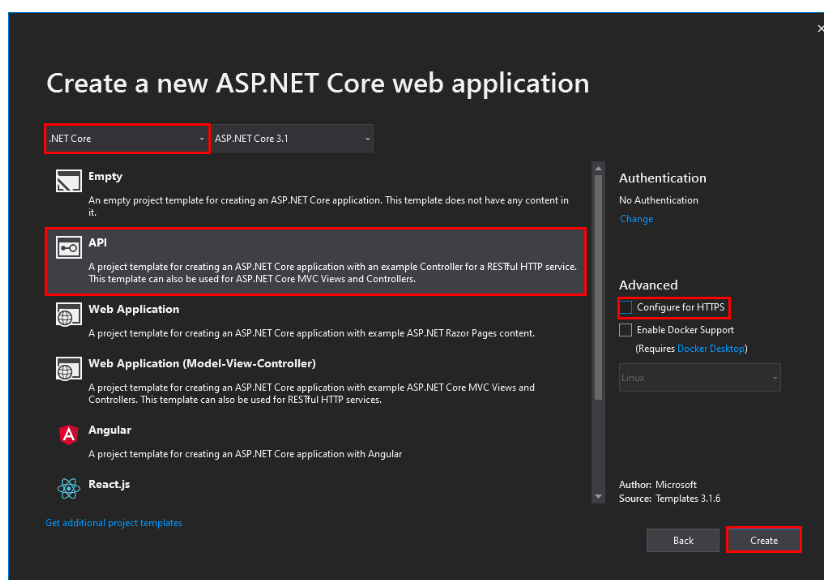


Slika 3.4: Nastavitev lokacije, imena projekta in rešitve.

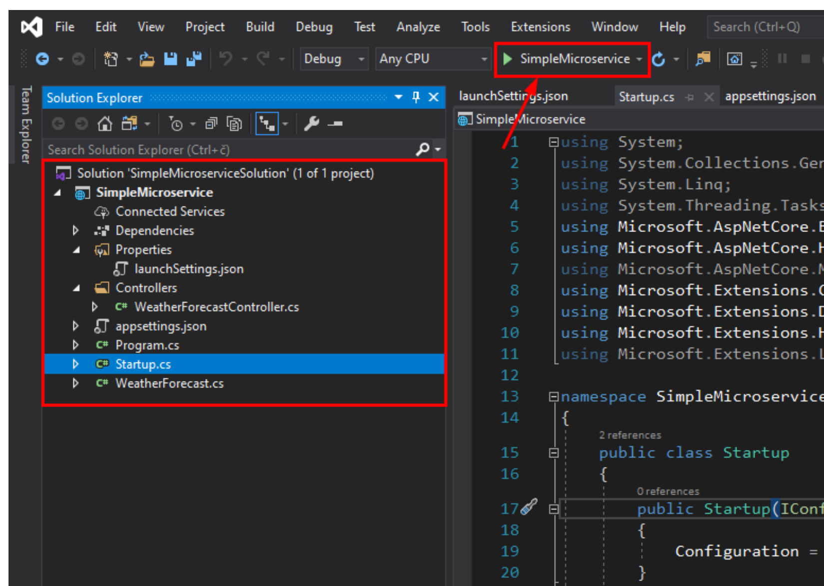
predlogo API (glej sliko 3.5). Bodite pozorni na to, da je v zgornjem levem polju izbran .NET Core, predlagamo pa tudi odstranitev kljukice pri polju z napisom "Configure for HTTPS", ki nam omogoči komunikacijo aplikacije preko protokola HTTPS.

Po pritisku na gumb 'Create' se nam v stranskem oknu z napisom "Solution Explorer" prikaže struktura ustvarjenega projekta (glej sliko 3.6). Ta vsebuje datoteki *Project.cs* in *Startup.cs*, ki skrbita za sam zagon in inicializacijo projekta. V datoteki *appsettings.json* so shranjene konfiguracijske vrednosti, v datoteki *launchSettings.json* v mapi *Properties* pa so shranjene nastavitve za sam zagon aplikacije. Ostane nam samo še datoteka *WeatherForecastController.cs*, ki je v predlogo dodana kot primer, zraven najdemo tudi datoteko *WeatherForecast.cs*. Ti dve datoteki služita za posredovanje podatkov odjemalcem aplikacije.

Sedaj lahko preizkusimo delovanje naše mikrostoritve. Pred zagonom izberimo profil, ki je enak imenu našega projekta. V našem primeru je to 'SimpleMicroservice'. Sedaj lahko mikrostoritev zaženemo.



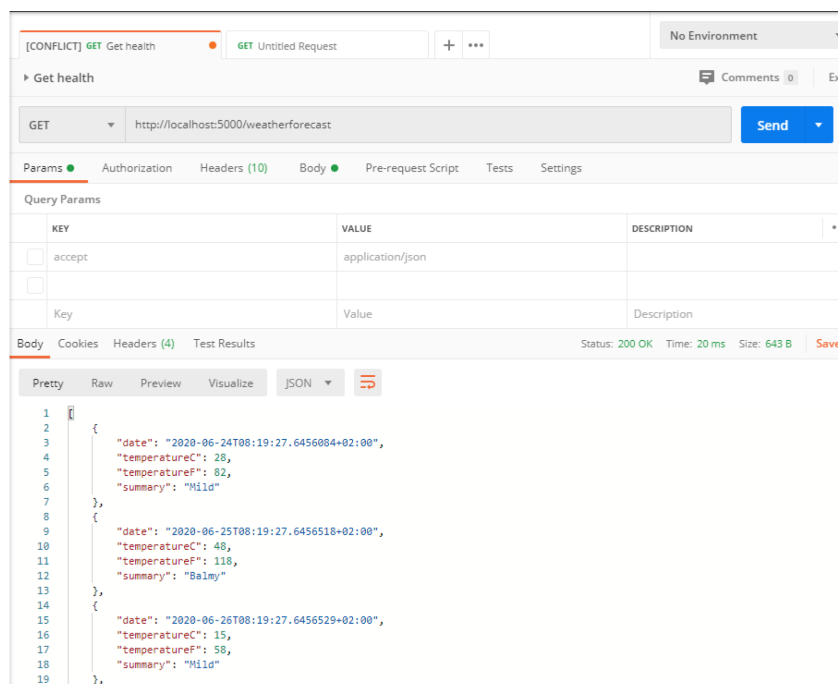
Slika 3.5: Izbira predloge za projekt.



Slika 3.6: Prikaz strukture ustvarjenega projekta.

Za preizkus delovanja bomo uporabili program Postman, ki je odličen za pošiljanje zahtevkov HTTP. Naša mikrostoritev je dostopna na naslovu http://localhost:5000/weatherforecast. Pri pošiljanju zahtevka GET na ta

naslov, nam mikrostoritev v našem primeru vrne testne podatke o vremenu, kot je prikazano na sliki 3.7.



Slika 3.7: Primer odgovora naše mikrostoritve na poslan zahtevek.

Primer te mikrostoritve bomo v nadaljevanju uporabili tudi za prikaz uporabe implementiranih rešitev pri problemih, ki so opisani v naslednjem poglavju.

Poglavje 4

Opis problema

V tem poglavju bomo predstavili najpogostejše probleme, ki jih je potrebno nasloviti pri izdelavi mikrostoritev. To so preverjanje vitalnosti mikrostoritev, konfiguracija mikrostoritev in odkrivanje mikrostoritev.

4.1 Problem preverjanja vitalnosti mikrostoritev

Eden izmed najpogostejših problemov arhitekture mikrostoritev je pravočasno zaznavanje, da se neka storitev ne izvaja pravilno oz. je neodzivna. Da preprečimo nadaljnje nepravilno izvajanje mikrostoritve, moramo to napako čim prej zaznati. Ker aplikacije, grajene v tej arhitekturi, pogosto vsebujejo veliko različnih instanc mikrostoritev, si želimo, da bi do njihovega stanja lahko dostopali preko neke skupne točke za spremljanje njihovega stanja.

Če želimo to omogočiti, mora vsaka mikrostoritev ponujati dostop do njenih informacij. Ena od teh informacij je vitalnost mikrostoritve. V ta namen se vsaki mikrostoritvi implementira vmesnik, preko katerega lahko pridobimo podatke o stanju mikrostoritve. Nekateri izmed teh podatkov so: prostor na disku, povezava z bazo, povezava z neko drugo mikrostoritvijo itd.

4.2 Problem konfiguracije mikrostoritev

Kot smo že omenili, so v arhitekturi mikrostoritev sistemi razdeljeni na več mikrostoritev, ki se pogosto izvajajo ločeno ena od druge. Vsaka od njih je lahko nameščena in skalirana neodvisno od ostalih. To pomeni, da lahko za neko mikrostoritev obstaja več instanc, ki se izvajajo sočasno. Recimo, da želimo spremeniti neko nastavitev mikrostoritve, za katero v nekem trenutku obstaja veliko instanc. Če imamo konfiguracijo vsebovano v njej, bomo morali konfiguracijo popraviti za vsako instanco in vsako od njih tudi ponovno namestiti. V primeru, da nočemo popolnoma prekiniti delovanja mikrostoritve, ki jo konfiguriramo, bo to privedlo do položaja, da bodo nekatere instance v določenem trenutku imele novo konfiguracijo, druge pa staro. Velikokrat imajo mikrostoritve povezavo do nekih zunanjih sistemov, za katere potrebujejo podatke, kot so URL, uporabniško ime, geslo ipd. V primeru, da jih hočemo spremeniti, bi bilo dobro imeti enotno konfiguracijo, ki bi jo uporabljale vse instance [8].

V ta namen se uporablja zunanja shramba, na primer datotečni sistem, sistemske spremenljivke, baza ali pa konfiguracijski strežnik, na katerem hranimo konfiguracijske vrednosti mikrostoritev. Ob zagonu mikrostoritev pridobi konfiguracijske vrednosti iz izbrane zunanje shrambe. Mikrostoritve imajo možnost ponovnega branja konfiguracijskih vrednosti tudi v času izvajanja, kar pomeni, da lahko neprestano spremljajo spremembe na konfiguracijskih virih, če slednji to omogočajo [8].

4.3 Problem odkrivanja mikrostoritev

Za arhitekturo mikrostoritev velja, da mikrostoritve dostikrat spremenijo lokacijo izvajanja. Problematično postane, ko moramo tudi ostalim mikrostoritvam, ki jo uporabljajo, povedati novi naslov, na katerem je preseljena mikrostoritev še vedno dosegljiva. To je brez uporabe mehanizma za odkrivanje storitev zelo kompleksen in zamuden postopek, saj moramo to ročno popraviti v vsaki instanci mikrostoritve, ki se izvaja [9].

Mehanizem za odkrivanje storitev omogoča, da ob preselitvi neke mikrostoritve na drug naslov ne posegamo v delovanje drugih mikrostoritev. Vse, ki se izvajajo, strežniku za odkrivanje storitev same sporočajo, na katerem naslovu se nahajajo. Ko ena izmed njih želi dostopati do izbrane mikrostoritve, pošlje zahtevek na strežnik, ki ji vrne informacije o naslovu, na katerem se iskana mikrostoritev nahaja [9].

V naslednjem poglavju bomo opisali postopek razvoja komponent za ogrodje .NET Core, s katerimi rešujemo zgoraj opisane probleme.

Poglavje 5

Implementacija rešitve

Rešitev, ki bo zagotavljala podporo za identificirane probleme v ogrodju .NET Core, je v obliki komponent. V tem poglavju bomo opisali njihovo zasnovo in implementacijo.

5.1 Preverjanje vitalnosti mikrostoritev

5.1.1 Opis zahtev

Razviti želimo komponento za preverjanje vitalnosti, ki nam bo s statusi HTTP in odgovori ("UP", "DOWN"), skladnimi s popularnimi oblačnimi platformami, kot je npr. Kubernetes, sporočala informacije o stanju naše mikrostoritve. Želimo, da nam mikrostoritev vrne HTTP status 200 in odgovor "UP", ko so rezultati preverjanja vitalnosti mikrostoritve uspešni, ter HTTP status 503 in odgovor "DOWN", ko so rezultati neuspešni.

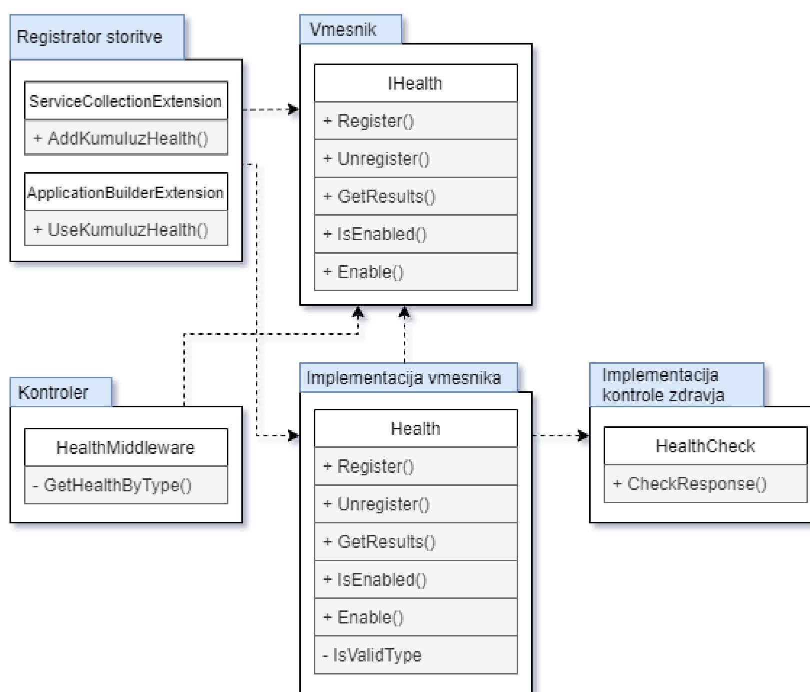
Za preverjanje vitalnosti želimo uporabiti poljubno število kontrol vitalnosti, ki so že implementirane ali pa jih implementiramo po potrebi. Nekatere od kontrol, ki jih želimo imeti, so kontrola prostora na disku, kontrola HTTP povezav, kontrola povezave z bazo itd.

Pri preverjanju vitalnosti želimo imeti tudi možnost pridobivanja rezultatov vseh kontrol ali pa samo tistih kontrol, ki preverjajo zgolj pripravljenost

mikrostoritve ali delovanje mikrostoritve. V ta namen bomo uvedli dva atributa, s katerima bomo kontrolam definirali njihov namen.

5.1.2 Načrt implementacije

Na sliki 5.1 je prikazana struktura programa, ki implementira zgoraj opisane zahteve. Razred *ServiceConfig* je namenjen vključitvi mehanizma v našo mikrostoritev. Ob klicu funkcije *AddKumuluzHealth()* se z upoštevanjem podanih parametrov naredi instanca implementacije vmesnika *IHealth*, ki predstavlja glavno komponento tega mehanizma. Vmesnik za preverjanje vitalnosti zdravja mikrostoritve nam ponuja metodo *Register()*, preko katere lahko registriramo že obstoječe ali naknadno implementirane kontrole zdravja. Če želimo med delovanjem mikrostoritve odstraniti eno izmed kontrol, lahko to storimo s klicem metode *Unregister()*. Glavno delo tega mehanizma pa opravlja metoda *GetResults()*, ki pridobi rezultate vseh registriranih kontrol zdravja. Vsaka kontrola zdravja namreč vsebuje metodo *CheckResponse()*, ki vrača rezultat o stanju, za katerega je zadolžena. Ko metoda *GetResults()* pridobi vse rezultate registriranih konrol, ustvari novo vrednost, v kateri so vrednosti vseh pridobljenih rezultatov ter končno stanje mikrostoritve. Razred *HealthMiddleware* je zadolžen za vračanje rezultatov uporabniku. Preko API vmesnika izpostavlja metodo *GetHealthByType()*, ki glede na izbran tip kontrol vrne rezultate ustreznih kontrol vitalnosti.



Slika 5.1: Struktura implementacije mehanizma za preverjanje vitalnosti mikrostoritev.

5.1.3 Implementacija

Zaradi lažjega razumevanja in berljivosti tega dokumenta so podrobneje predstavljeni samo pomembnejši elementi implementacije.

Za začetek si poglejmo, kako sta implementirani metodi *AddKumuluzeHealth()* (glej izsek kode 5.1) v razredu *ServiceCollectionExtension* in *UseKumuluzeHelth()* (glej izsek kode 5.2) v razredu *ApplicationBuilderExtension*. Metoda *AddKumuluzHealth()* je lahko klicana nad instanco objekta *IServiceCollection* in sprejme delegat tipa *Action*, ki definira metodo nad tipom objekta *HealthOptions*. Najprej naredimo novo instanco tipa *HealthOptions* in nad njo izvedemo podan delegat. Nato naredimo novo instanco implementacije vmesnika *IHealth*, ki ji podamo objekt tipa *HealthOptions*. To storitev dodamo v zbirko vseh storitev kot “singleton”, kar pomeni, da bo med delovanjem mikrostoritve vedno na voljo samo ena instanca. Metoda

UseKumuluzeeHealth() pa je nekoliko kompleksnejša. Izvede se lahko nad objektom tipa *IApplicationBuilder*. Ob klicu te metode lahko poljubno podamo tudi pot, na kateri bo omogočeno pridobivanje informacij o vitalnosti mikrostoritve. Najprej preverimo, ali je podana pot veljavna in v primeru neveljavnosti vrnemo izjemo. Nato naredimo funkcijo, ki mikrostoritvi sporoči, ali je pot zahteve namenjena pregledu vitalnosti. V primeru, da to drži, za nadaljnjo izvedbo zahteve uporabimo vmesno programsko opremo, ki je implementirana v razredu *HealthMiddleware*.

```
1 public static IServiceCollection AddKumuluzHealth(this
    ↪ IServiceCollection services, Action<HealthOptions>
    ↪ options = null)
2   {
3     HealthOptions opt = new HealthOptions();
4     options?.Invoke(opt);
5     IHealth healthRegistry = new Health(opt);
6     services.AddSingleton(healthRegistry);
7     return services;
8   }
```

Izsek kode 5.1: Metoda, ki doda naš mehanizem za preverjanje vitalnosti v zbirko storitev.

```
1 public static IApplicationBuilder UseKumuluzHealth(this
    ↪ IApplicationBuilder app, string route = "/health")
2   {
3     if (app == null)
4       throw new ArgumentNullException(nameof(app));
5
6     //validation of path
7     if (route == null)
8       throw new ArgumentNullException(nameof(route), "
          ↪ Route must not be null!");
9     else if (route.Length == 0)
10      throw new ArgumentException(nameof(route), "Route
          ↪ must not be empty!");
11    else if (route == "/")
```

```
12          throw new ArgumentException(nameof(route), "Route
              ↪ must not be equals '/'!");
13
14        //prepare path
15        if (route[0] != '/')
16          route = route.Insert(0, "/");
17        if (route[route.Length - 1] == '/')
18          route = route.Remove(route.Length - 1);
19
20        //check which health type is required
21        Func<HttpContext, bool> predicate = c =>
22        {
23          List<string> validPaths = new List<string>
24          {
25            route,
26            route + "/live",
27            route + "/ready"
28          };
29
30          if (c.Request.Path.HasValue == false)
31            return false;
32
33          return c.Request.Path.HasValue &&
34              validPaths.Contains(c.Request.Path.Value);
35        };
36
37        //set which middleware to use
38        app.MapWhen(predicate, a => a.UseMiddleware<
            ↪ HealthMiddleware>());
39        return app;
40      }
```

Izsek kode 5.2: Metoda, ki omogoči delovanje mehanizma.

Če je storitev prejela zahtevo za pregled stanja vitalnosti, uporabimo razred *HealthMiddleware*. Ta ob inicializaciji pridobi instanco implementacije vmesnika *IHealth*, ki smo jo predhodno dodali v zbirko vseh storitev. Storitev sama poskrbi, da se izvede metoda *Invoke()*, ki glede na pot zahteve pridobi

rezultate kontrol vitalnosti s pomočjo metode *GetHealthByType()* in jih v formatu JSON posreduje uporabniku. Naloga metode *GetHealthType()* (glej izsek kode 5.3) je, da za podan tip vitalnosti izvede ustrezne kontrole ter na podlagi rezultatov pripravi ustrezno strukturirane podatke za odgovor.

```
1  private HealthResponse GetHealthByType(HealthType type)
2    {
3      try
4      {
5        // get results
6        var results = _healthCheckRegistry.GetResults(type);
7
8        // prepare response
9        HealthResponse healthResponse = new HealthResponse();
10       healthResponse.Checks = results;
11       healthResponse.Status = State.UP;
12
13       // check if any check is down
14       if (results.Where(e => State.DOWN.Equals(e.Status)).
             ↪ Any())
15         healthResponse.Status = State.DOWN;
16
17       return healthResponse;
18     }
19     catch
20     {
21       return null;
22     }
23   }
```

Izsek kode 5.3: Implementacije metode, ki vrne stanje vitalnosti glede na podan tip.

Sedaj si lahko podrobneje pogledamo razred *Health*, ki implementira vmesnik *IHealth*. Ob inicializaciji se pokliče konstruktor, ki prejme objekt tipa *HealthOptions*. Ta vsebuje podatke o tem, katere kontrole se bodo uporabljale. Te podatke nastavi uporabnik preko delegata v klicu metode *AddKumuluzHealth()*, ki smo jo omenili zgoraj.

Razred *Health* vsebuje tri ključne metode:

- *Register()* - Doda kontrolo vitalnosti v uporabo.
- *Unregister()* - Odstrani izbrano kontrolo vitalnosti.
- *GetResults()* - Izvede vse kontrole vitalnosti za podan tip in za vsako vrne rezultate izvedbe (glej izsek kode 5.4).

```
1  public IEnumerable<HealthCheckResponse> GetResults(
     ↪ HealthType type)
2  {
3    _logger.LogInformation("Getting health check results"
       ↪ );
4
5    foreach (var pair in _healthChecks)
6    {
7      if (!IsValidType(pair.Value, type)) continue;
8
9      var response = pair.Value?.CheckResponse();
10     if (response == null) continue;
11
12     response.Name = pair.Key;
13     yield return response;
14   }
15 }
```

Izsek kode 5.4: Implementacija metode, ki vrne rezultate vseh registriranih kontrol.

Kontrole so pri preverjanju vitalnosti storitve ključnega pomena. Celotna implementacija je narejena tako, da lahko uporabnik kadarkoli implementira nove kontrole. Za primer si poglejmo kontrolo za preverjanje dosegljivosti vsebine na določenem spletnem naslovu (glej izsek kode 5.5). Razred, ki to omogoča, se imenuje *HttpHealthCheck*. Pomembno je, da vse kontrole razširjajo osnovni razred *HealthCheck*. Nad vsakim razredom kontrole se lahko doda tudi atribut "Liveness" ali "Readiness" (možna je tudi uporaba obeh). *CheckResponse()* je metoda, ki jo mora vsaka kontrola vitalnosti

implementirati. V njej je vsebovana logika, ki preveri določeno stvar. V našem primeru bo ta metoda preverila, kakšne odgovore dobimo ob pošiljanju zahtevkov na izbrane naslove URL. V primeru, da status koda HTTP ni v rangu od vključno 200 do 300, metoda vrne neuspešen rezultat skupaj s potrebnimi informacijami.

```
1   public override HealthCheckResponse CheckResponse()
2    {
3      _data.Clear();
4
5      if (_urls == null)
6        return null;
7
8      response.Up();
9
10     foreach (string url in _urls)
11       checkHttpStatus(ref response, url);
12
13     response.Data = _data;
14     return response;
15   }
16
17   private void checkHttpStatus(ref HealthCheckResponse
        ↪ response, string url)
18   {
19     HttpWebRequest myRequest;
20     HttpWebResponse myResponse = null;
21
22     try
23     {
24       myRequest = (HttpWebRequest)WebRequest.Create(url);
25       myResponse = (HttpWebResponse)myRequest.GetResponse
            ↪ ();
26       if ((int)myResponse.StatusCode >=200 && (int)
            ↪ myResponse.StatusCode < 300)
27         _data.Add(url, State.UP);
28       else
29       {
```

```
30            response.Down();
31            _data.Add(url, State.DOWN);
32          }
33        }
34        catch
35        {
36          response.Down();
37          _data.Add(url, State.DOWN);
38        }
39        finally
40        {
41          if (myResponse != null)
42            myResponse.Close();
43        }
44      }
```

Izsek kode 5.5: Primer kontrole za preverjanje dosegljivosti spletnega naslova.

5.1.4 Primer uporabe

Uporabo implementiranega mehanizma bomo prikazali na mikrostoritvi, ki smo jo naredili v poglavju 3. Najprej projektu naše mikrostoritve dodamo referenco na projekt *HealthCore*. V razredu *Startup* se nahaja metoda *ConfigureServices()*. V tej metodi zbirki storitev dodamo mehanizem za preverjanje vitalnosti, v katerega vključimo kontrolo za preverjanje dosegljivosti spletnih naslovov in kontrolo za preverjanje razpoložljivega prostora na disku (glej izsek kode 5.6).

```
1     services.AddKumuluzHealth(options =>
2       {
3         options.RegisterHealthCheck("http_health_check",
            ↪ new HttpHealthCheck("https://github.com/"));
4         options.RegisterHealthCheck("
            ↪ disk_space_health_check", new
            ↪ DiskSpaceHealthCheck(500, SpaceUnit.Gigabyte)
            ↪ );
```

```
5       });
```

Izsek kode 5.6: Primer vključitve mehanizma za preverjanje vitalnosti v mikrostoritev.

Da omogočimo preverjanje vitalnosti, moramo v metodi *Configure()*, ki se ravno tako nahaja v razredu *Startup*, mikrostoritvi povedati, naj uporablja dodan mehanizem (glej izsek kode 5.7). Metoda, ki jo pokličemo, sprejme tudi opcijski parameter, ki kaže na del naslova, na katerem bodo dosegljive informacije o vitalnosti naše mikrostoritve.

```
1     app.UseKumuluzHealth();
```

Izsek kode 5.7: Klic metode *UseKumuluzHealth()*, ki omogoči delovanje mehanizma.

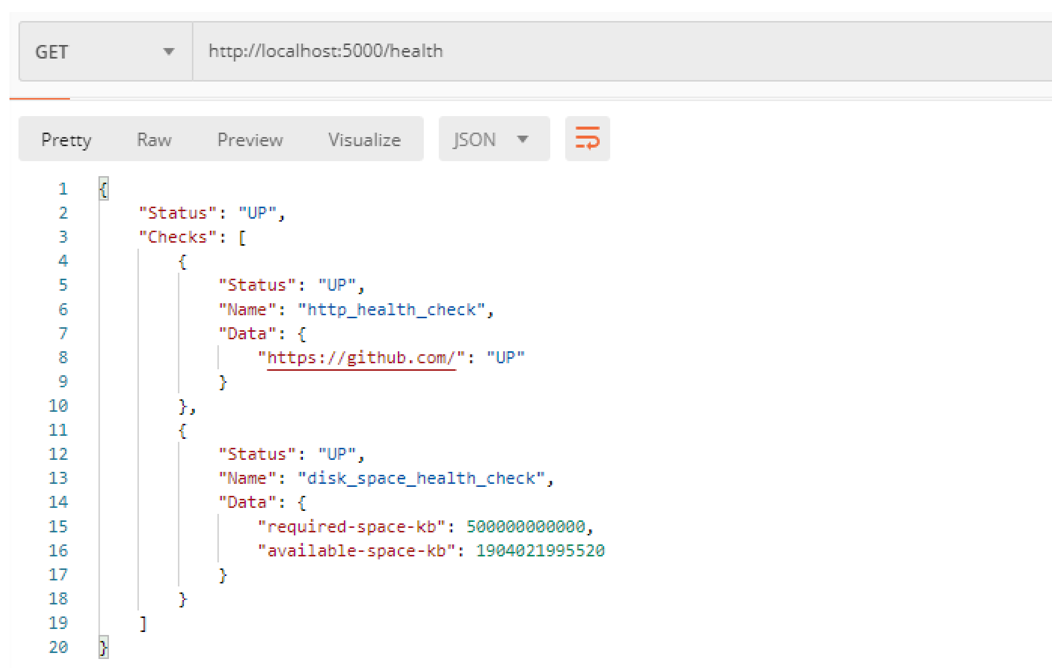
Sedaj lahko preko zahtevka HTTP preverimo vitalnost naše mikrostoritve. V našem primeru se te informacije nahajajo na spletnem naslovu "http://localhost:5000/health". V programu Postman lahko vidimo odgovor na zahtevek GET, ki je bil poslan na ta naslov (glej sliko 5.2). Iz odgovora lahko razberemo, da naša mikrostoritev deluje uspešno.

5.2 Konfiguracija mikrostoritev

5.2.1 Opis zahtev

Razviti želimo komponento za konfiguracijo mikrostoritev, ki bo omogočala hitro in enostavno spreminjanje konfiguracije mikrostoritev. Vrednosti konfiguracije želimo brati iz konfiguracijskih datotek, sistemskih spremenljivk ali pa konfiguracijskih strežnikov, kot sta Consul in Etcd.

Ker konfiguracijska strežnika Consul in Etcd omogočata zaznavanje sprememb konfiguracijskih vrednosti med delovanjem, želimo, da lahko komponenta za konfiguracijo ob zaznani spremembi vrednosti samodejno spremeni konfiguracijo mikrostoritve. Ker lahko mikrostoritev konfiguriramo preko več konfiguracijskih virov, potrebujemo tudi način, s katerim bomo določili



Slika 5.2: Primer odgovora HTTP na zahtevo za preverbo vitalnosti mikrostoritve.

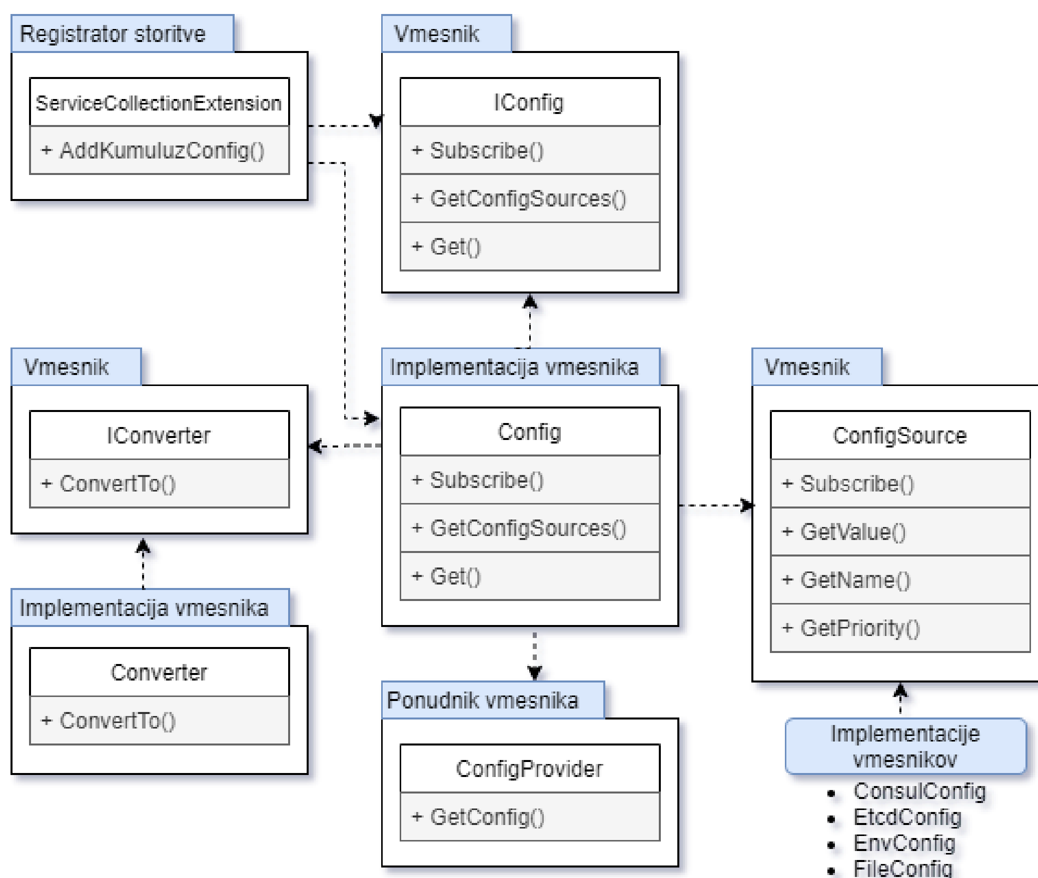
vrednost prioritet, ki bodo odločile, katera konfiguracijska vrednost se bo uporabila.

Iz konfiguracijskih virov želimo razbrati tudi vrednosti v drugih podatkovnih tipih. Za to potrebujemo možnost implementacije pretvornikov, ki bodo vrednost tipa *string* znali pretvoriti v poljuben tip.

5.2.2 Načrt implementacije

Na sliki 5.3 je prikazana struktura programa, ki implementira zgoraj opisane zahteve. Kot pri implementaciji kontrole vitalnosti mikrostoritve je tudi tukaj razred *ServiceConfig* namenjen zgolj vključitvi mehanizma v našo mikrostoritev. Ob klicu metode *AddKumuluzConfig()* se z upoštevanjem podanih parametrov naredi instanca razreda *Config*, ki je implementacija vmesnika *IConfig* in predstavlja glavno komponento v tem mehanizmu. Omenjeni vmesnik nam ponuja metodo *Subscribe()*, preko katere lahko neko vrednost

naročimo na spremembe. To pomeni, da čim se vrednost za določen ključ na konfiguracijskem strežniku spremeni, se s tem spremeni tudi vrednosti spremenljivke, ki je na ta ključ naročena. Obstaja tudi metoda *GetConfigSources()*, ki nam vrne vse konfiguracijske vire, vsebovane v razredu *Config*, in metoda *Get()*, katere naloga je, da za podani ključ preveri, kateri izmed konfiguracijskih virov vsebuje ustrezno vrednost ter jo tudi vrne. Zaenkrat so v mehanizmu vključeni razredi, ki implementirajo razred *ConfigSource* za štiri različne tipe konfiguracije. To so konfiguracija preko datotek, sistemskih spremenljivk, Etcd strežnika in Consul strežnika. Razred *ConfigSource* vsebuje metodo *Subscribe()*, ki spreminja vrednost naročene spremenljivke, metodo *GetValue()*, ki vrne vrednost tipa string za podan ključ, metodo *GetName()*, ki vrača ime konfiguracijskega vira, in metodo *GetPriority()*, ki vrača prioritetno vrednost posamezne konfiguracije. Razred *Converter*, ki je implementacija vmesnika *IConverter*, je namenjen pretvorbi rezultata tipa *string*, ki ga pridobi razred *Config*, v poljuben tip. Uporabniku je omogočeno tudi dodajanje novih pretvornikov. To naredi tako, da ustvari nov razred, ki mora implementirati vmesnik *ITypeConverter*. Kot najmanj pomemben pa je tu še razred *ConfigProvider*, ki je namenjen zgolj lažjemu pridobivanju implementacije vmesnika *IConfig*.



Slika 5.3: Struktura implementacije mehanizma za konfiguracijo mikrostoritev.

5.2.3 Implementacija

Implementacija te komponente je dokaj obsežna, zato si bomo podrobneje pogledali samo ključne elemente implementacije.

Najprej si poglejmo metodo *AddKumuluzConfig()* (glej izsek kode 5.8), preko katere delovanje komponente vključimo v našo mikrostoritev. Ta metoda se izvede nad objektom tipa *IServiceCollection* in kot dodaten parameter sprejme delegat nad objektom tipa *ConfigOptions*. Najprej naredimo novo instanco tipa *ConfigOptions*, nad katerim nato izvedemo delegat. Zatem naredimo instanco implementacije vmesnika *IConfig* in jo dodamo v zbirko

storitev.

```
1 public static IServiceCollection AddKumuluzConfig(this
    ↪ IServiceCollection services, Action<ConfigOptions>
    ↪ options)
2     {
3         ConfigOptions opt = new ConfigOptions();
4         options?.Invoke(opt);
5         IConfig config = new Config(opt);
6         ConfigProvider._config = config;
7         services.AddSingleton(config);
8 
9         return services;
10    }
```

Izsek kode 5.8: Metoda, ki doda naš mehanizem za konfiguracijo v zbirko storitev.

Razred *Config* implementira vmesnik *IConfig*. Za pravilno implementacijo mora vsebovati metode *Subscribe()*, *GetCongiSources()* in *Get()*, ki jih bomo podrobneje opisali v nadaljevanju. Razred *Config* vsebuje konstruktor, ki sprejme objekt tipa *ConfigOptions*. V njem so shranjeni podatki o poti konfiguracijske datoteke, tipi konfiguracijskih virov, ki naj se uporabijo itd. V konstruktorju (glej izsek kode 5.9) se naredi nova instanca objekta tipa *Converter*, ki ga bomo obravnavali kasneje, nato pa se ustvarijo potrebne instance konfiguracijskih virov. V vsakem primeru sta narejena konfiguracijska vira za branje konfiguracijskih datotek in sistemskih spremenljivk, dodatno pa se lahko dodata še vira za branje s strežnika Consul ali Etcd. Ko imamo narejene instance vseh potrebnih virov, jih samo še uredimo po prioritetnem vrstnem redu. Prioriteta vsakega vira je definirana v njegovi implementaciji.

```
1     public Config(ConfigOptions options)
2   {
3     //if logger is null, create new logger
4     _logger = options.Logger ?? new NullLogger<Config>();
5 
6     //create new converter
7     _converter = new Converter(_logger);
```

```
8
9       _configSources = new List<ConfigSource>();
10
11      //create and add new evironment configuration
12      _configSources.Add(new EnvConfig(_logger));
13
14      //create and add new file configuration
15      FileConfig fileConfig = new FileConfig(_logger, options
           ↪ .ConfigFilePath);
16      if (fileConfig != null)
17        _configSources.Add(fileConfig);
18      else
19        _logger.LogInformation("Default path 'config.yaml' is
             ↪  used for configuration file.");
20
21      //sort current list of config sources by their priority
           ↪ ;
22      SortConfigSources();
23
24      foreach (var extension in options.Extensions)
25      {
26        switch (extension)
27        {
28          case Extension.Consul:
29            //create and add new consul configuration
30            ConsulConfig consulConfig = new ConsulConfig(this
                 ↪ , _logger, _converter);
31            _configSources.Add(consulConfig);
32            break;
33          case Extension.Etcd:
34            //create and add new etcd configuration
35            EtcdConfig etcdConfig = new EtcdConfig(this,
                 ↪ _logger, _converter);
36            _configSources.Add(etcdConfig);
37            break;
38          default:
39            _logger.LogWarning($"There is no implementation
                 ↪ for '{extension}' extension.");
```

```
40              break;
41          }
42      }
43
44      //sort current list of config sources by their priority
         ↪ ;
45      SortConfigSources();
46  }
```

Izsek kode 5.9: Implementacija konstruktorja razreda *Config*.

Celotna implementacija komponente za konfiguracijo je pripravljena za enostavno dodajanje novih konfiguracijskih virov. Vsak konfiguracijski vir mora implementirati vmesnik *ConfigSource*, kar pomeni, da mora vsebovati metode *GetName()*, *GetPriority()*, *GetValue()* in *Subscribe()*.

Ker sta metodi *GetName()* in *GetPriority()* zelo enostavni, si raje poglejmo primer implementacije metod *GetValue()* in *Subscribe()* na primeru konfiguracijskega vira za strežnik Consul. Pri kreiranju instance tega vira se v konstruktorju naredi nov odjemalec za komunikacijo s strežnikom Consul (glej izsek kode 5.10). Tu je potrebno poudariti, da imamo predhodno že narejena vira za branje iz datotek in sistemskih spremenljivk. Preko njiju lahko tako pridobimo ostale podatke za kreiranje odjemalca.

```
1     public ConsulConfig(IConfig config, ILogger logger,
         ↪ IConverter converter)
2   {
3     _logger = logger ?? throw new ArgumentNullException(
         ↪ nameof(logger));
4     _config = config ?? throw new ArgumentNullException(
         ↪ nameof(config));
5     _converter = converter ?? throw new
         ↪ ArgumentNullException(nameof(converter));
6
7     //getting source address
8     string sourceAddress = _config.Get<string>("kumuluzee.
         ↪ config.consul.hosts");
9     if (!string.IsNullOrWhiteSpace(sourceAddress))
```

```
10      {
11        sourceAddress = ADDRESS;
12        client = new ConsulClient(c =>
13        {
14          c.Address = new Uri(sourceAddress);
15        });
16      }
17      else
18        client = null;
19
20      //if client is null, return
21      if (client == null)
22      {
23        _logger.LogWarning("Thare was problem creating consul
            ↪  client.");
24        return;
25      }
26
27      //get service configuration values from configuration
          ↪ sources
28      ServiceConfigurationValues scv = Common.
          ↪ LoadServiceConfiguration(_config);
29      startRetryDelay = scv.startRetryDelay;
30      maxRetryDelay = scv.maxRetryDelay;
31      nametag = $"environments/{ scv.envName}/services/{scv.
          ↪ name}/{scv.version}/config";
32
33
34      var resultNametag = _config.Get<string>("kumuluzee.
          ↪ config.namespace");
35      if (resultNametag != null)
36        nametag = resultNametag;
37
38      if (!string.IsNullOrWhiteSpace(nametag))
39        nametag = nametag + "/";
40    }
```

Izsek kode 5.10: Implementacija konstruktorja razreda *ConsulConfig*.

Metoda *GetValue()* (glej izsek kode 5.11) s pomočjo narejenega odjemalca pridobi vrednosti s strežnika Consul, če te seveda obstajajo. Če vrednost za podan ključ na konfiguracijskem viru ne obstaja, potem metoda vrne null. Ključ vrednosti je sestavljen iz besed, ki so združene z znakom '.' (npr.: "uporabnik.ime"), medtem ko so besede ključa na konfiguracijskem viru združene z znakom '/'.

```
1     public string GetValue(string key)
2   {
3     IKVEndpoint kv = this.client?.KV;
4
5     try
6     {
7       key = key.Replace('.', '/');
8
9       var value = kv.Get(nametag + key)?.Result?.Response?.
          ↪ Value;
10      if (value == null)
11        return null;
12
13      return Encoding.UTF8.GetString(value, 0, value.Length
          ↪ );
14    }
15    catch
16    {
17      _logger.LogInformation($"Unable to get value of key
          ↪ '{nametag + key}'");
18      return null;
19    }
20  }
```

Izsek kode 5.11: Primer implementacije metode *GetValue()* za konfiguracijo preko strežnika Consul.

Metoda *Subscribe()* je namenjena zaznavanju sprememb vrednosti na konfiguracijskem viru. Ob klicu je potrebno podati ključ, tip vrednosti, ki jo za podani ključ pričakujemo, in povratno metodo, v kateri je definirana nadaljnja uporaba pridobljene vrednosti. Metoda deluje tako, da vsake toliko časa

preveri vrednost za podani ključ in v primeru spremembe izvede povratno metodo (glej izsek kode 5.12). Časovni intervali med preverjanjem vrednosti so odvisni od pogostosti sprememb vrednosti.

```
1    public void Subscribe<T>(string key, Action<T> callback
       ↪ )
2  {
3    Watch(key, callback);
4  }
5
6  public async void Watch<T>(string key, Action<T> callback
     ↪ , string oldValue = "", int retryDelay = 0)
7  {
8    //stop watching changes if callback is null
9    if (callback == null)
10     return;
11
12   var cb = new Action<object>(o => callback((T)o)); //
       ↪ When using converter...is this even needed?
13
14   key = key.Replace('.', '/');
15
16   //trying to get response from consul configuration
17   QueryResult<KVPair> response;
18   try
19   {
20     await Task.Delay(TimeSpan.FromMilliseconds(retryDelay
         ↪ ));
21     response = await client.KV.Get(nametag + key);
22   }
23   catch
24   {
25     response = null;
26   }
27
28   if (response == null || response.StatusCode !=
       ↪ HttpStatusCode.OK) {
29     if (retryDelay == 0)
```

```
30          retryDelay = startRetryDelay;
31        Watch(key, callback, oldValue, Math.Min(2 *
             ↪ retryDelay, maxRetryDelay));
32        return;
33      }
34
35      var kvPair = response.Response;
36      if (kvPair?.Value != null)
37      {
38        var value = Encoding.UTF8.GetString(kvPair.Value, 0,
             ↪ kvPair.Value.Length);
39        if (value != oldValue)
40          cb(_converter.ConvertTo<T>(value));
41        Watch(key, callback, value, startRetryDelay);
42        return;
43      }
44
45      Watch(key, callback, oldValue, startRetryDelay);
46      return;
47    }
```

Izsek kode 5.12: Primer implementacije metode *Subscribe()* za konfiguracijo preko strežnika Consul.

Rezultat metode *GetValue()*, ki smo jo že spoznali, je vedno tipa *string*. Tip rezultata bi kot uporabniki funkcionalnosti konfiguracije želeli definirati že ob zahtevi. V ta namen je na konfiguracijskem vmesniku definirana metoda *Get()*, ki ji tip lahko podamo (glej izsek kode 5.13). V implementaciji te metode najprej pridobimo vrednost za podani ključ, ki je tipa *string*, nato pa jo z uporabo razreda *Converter* poskusimo pretvoriti v želeni tip.

```
1       //return value for given key as type of T
2       public T Get<T>(string key)
3       {
4         //get value from configuration sources
5         var value = GetValue(key);
6         //convert string value to type of T
7         return _converter.ConvertTo<T>(value);
```

```
8     }
```

Izsek kode 5.13: Primer metode *Get()* za določen tip.

Sedaj je primeren čas, da si pogledamo, kako deluje metoda *ConvertTo()* v razredu *Converter* (glej izsek kode 5.14). *Converter* vsebuje nabor pretvornikov, ki neko vrednost tipa *string* pretvorijo v drug tip. V osnovi so narejeni pretvorniki za osnovne tipe, kot so *bool*, *int*, *double* itd. Potrebno je izpostaviti predvsem značilnost pretvornika za tip *bool*. Ta ima definiranih nekaj različnih vrednosti, ki se lahko uporabijo za vrednosti "true" ali "false" kot so naprimer "1", "0", "yes", "no", "y" in "n". Uporabnik lahko naredi svoj pretvornik za poljuben tip objekta. To naredi tako, da ustvari nov razred, ki razširja abstraktni razred *BaseTypeConverter*. Ta vsebuje le metodo, ki pretvori vrednosti tipa *string* v poljuben tip, in lastnost *Priority*, po kateri se v primeru dveh pretvornikov za isti tip določi, kateri bo uporabljen. Če si pogledamo celoten postopek pretvarjanja tipa v metodi *ConvertTo()*, vidimo, da ta sprejme parameter tipa *string*. V primeru, da je ta null, vrnemo privzeto vrednost tega tipa. Če parameter ima vrednost, gremo skozi vse narejene pretvornike in poskusimo z njimi pretvoriti parameter. Če za to ne obstaja primeren pretvornik, poskusimo uporabiti kar metodo *ChangeType()* iz sistemske knjižnice. V primeru, da nam tudi ta ne uspe pretvoriti parametra, vrnemo privzeto vrednost želenega tipa.

```
1     //convert value from string to type T
2   public T ConvertTo<T>(string value)
3   {
4     //if value is null, return default value for required
        ↪ type
5     if (value == null)
6       return default(T);
7
8     //firstly check if there is any custom converter that
        ↪ can convert string to type T
9     foreach(var converter in _converters)
10    {
11      try
```

```
12        {
13          if (converter.GetType().GetMethod("Convert").
              ↪ ReturnType == typeof(T))
14          {
15            var method = converter.GetType().GetMethod("
                ↪ Convert");
16            return (T)method.Invoke(converter, new object[] {
                ↪  value });
17          }
18        }
19        catch { }
20      }
21
22      //if there is no useful converter, then use Convert.
          ↪ ChangeType method
23      try
24      {
25        FormatValueDependOnType(ref value, typeof(T));
26        return (T)Convert.ChangeType(value, typeof(T));
27      }
28      catch
29      {
30        //id value convertion has failed, return default
            ↪ value of type T
31        return default(T);
32      }
33    }
```

Izsek kode 5.14: Implementacija metode *ConvertTo()*, ki pretvori vrednosti tipa *string* v izbrani tip.

5.2.4 Primer uporabe

Naši mikrostoritvi dodajmo še mehanizem za konfiguracijo mikrostoritev. Tako kot pri uporabi mehanizma za preverjanje vitalnosti mikrostoritve, moramo tudi tukaj najprej dodati referenco na naš projekt za konfiguracijo. Tokrat moramo pred samo uporabo najprej ustvariti konfiguracijsko dato-

teko, ki bo vsebovala potrebne podatke za pravilno delovanje mehanizma (glej izsek kode 5.15). V našem primeru se bo ta datoteka imenovala *config.yaml*.

```
1 kumuluzee:
2   # name of our service
3   name: test-service
4   # version of our service
5   version: 1.0.0
6   env:
7     name: dev
8   config:
9     namespace: ''
10    start-retry-delay-ms: 100
11    max-retry-delay-ms: 500
12    consul:
13      # address of consul server
14      hosts: http://localhost:8500
```

Izsek kode 5.15: Primer vsebine konfiguracijske datoteke za potrebe mehanizma za konfiguracijo mikrostoritev.

Nato v metodi *ConfigureServices()* zbirki storitev dodamo naš mehanizem in podatek o tem, kje se nahaja konfiguracijska datoteka in kateri konfiguracijski strežnik naj uporablja (glej izsek kode 5.16). V našem primeru bomo uporabili Consul.

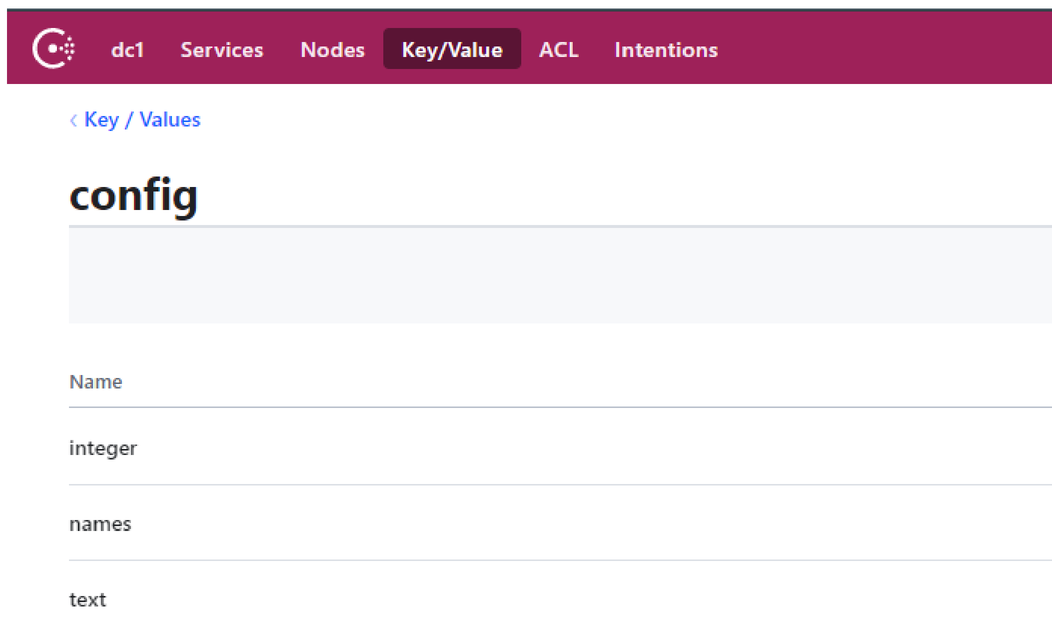
```
1     public services.AddKumuluzConfig(options =>
2   {
3     options.SetConfigFilePath(Path.GetFullPath("config.yaml
        ↪ "));
4     options.SetExtensions(Extension.Consul);
5   });
```

Izsek kode 5.16: Primer dodajanja mehanizma za konfiguracijo v mikrostoritev.

Ko to naredimo, je mehanizem pripravljen za uporabo. Uporabo bomo prikazali kar na obstoječem kontrolerju, ki se nahaja v razredu *WeatherFo-*

recastController. Najprej moramo v konstruktorju sprejeti parameter tipa *IConfig*, ki predstavlja naš mehanizem za konfiguracijo. Dobra praksa je, da si ta parameter shranimo kot privatno spremenljivko.

Sedaj na izbranem konfiguracijskem viru definirajmo nekaj ključev, katerih vrednosti so lahko poljubne (glej sliko 5.4).



Slika 5.4: Prikaz vrednosti na strežniku Consul preko njegovega grafičnega vmesnika.

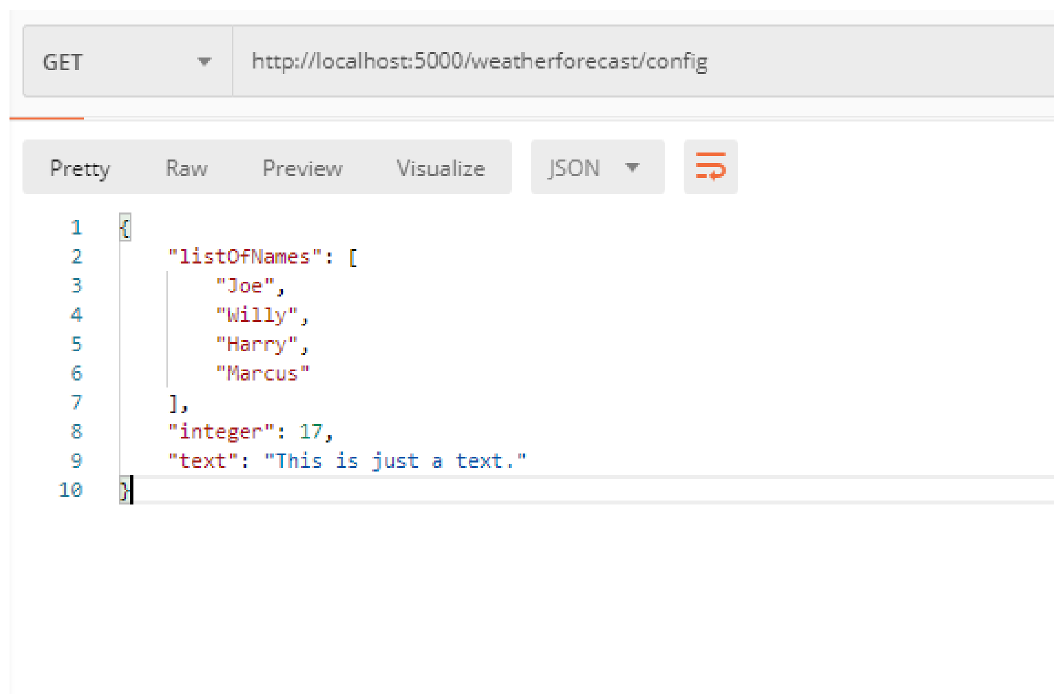
Za enostavnejši prikaz uporabe bomo naredili novo metodo *GetConfigurationValues()*, v kateri bomo iz konfiguracijskih virov prebrali nekaj vrednosti (glej izsek kode 5.17).

```
1     [HttpGet("config")]
2     public IActionResult GetConfigurationValues()
3     {
4       var result = new
5       {
6         names = _config.Get<string[]>("config.names"),
7         integer = _config.Get<int>("config.integer"),
8         text = _config.Get<string>("config.text")
```

```
9         };
10
11        return Ok(result);
12    }
```

Izsek kode 5.17: Primer pridobivanje vrednosti iz konfiguracijskega vira.

Preko programa Postman pošljemo zahtevek HTTP na spletni naslov "http://localhost:5000/weatherforecast/config". Vidimo, da nam naša mikrostoritev vrne odgovor, v katerem se nahajajo podatki, ki smo jih shranili na strežnik Consul (glej sliko 5.5).



Slika 5.5: Prikaz odgovora naše mikrostoritve na zahtevo prikaza pridobljenih konfiguracijskih vrednosti.

Sedaj si samo še poglejmo, kako spremenljivki dodeliti konfiguracijsko vrednost, ki se bo ob spremembi na strežniku Consul samodejno spremenila tudi v naši mikrostoritvi. To naredimo tako, da spremenljivko najprej definiramo, potem pa izvedemo konfiguracijsko metodo *Subscribe()*, ki sprejme vrednost ključa in delegat, ki se izvede ob spremembi (glej izsek kode 5.18).

V tem delegatu lahko tako spremenimo vrednost naši spremenljivki. V našem primeru naredimo statično spremenljivko, metodo pa pokličimo kar v konstruktorju.

```
1     private static string change = null;
2     public WeatherForecastController(ILogger<
         ↪ WeatherForecastController> logger, IConfig config
         ↪ )
3   {
4     _logger = logger;
5         _config = config;
6
7     if (change == null)
8       _config.Subscribe<string>("config.change", (value) =>
           ↪  change = value);
9     }
```

Izsek kode 5.18: Primer spremljanja sprememb vrednosti na konfiguracijskem viru.

5.3 Odkrivanje mikrostoritev

5.3.1 Opis zahtev

Razviti želimo komponento za odkrivanje mikrostoritev, ki bo s pomočjo strežnika Consul znala dostopati do ostalih mikrostoritev. Mikrostoritev se mora samodejno registrirati na strežniku Consul, preko katerega lahko potem ostale storitve pridobijo spletni naslov, kjer je dostopna.

Mikrostoritev mora periodično osveževati svoj obstoj na strežniku Consul, saj jo v nasprotnem primeru deregistrira. Želimo si, da se mikrostoritve identificirajo z uporabniku prijaznim imenom, zato mora vsaka mikrostoritev ob registraciji podati tudi ime, preko katerega jo ostale mikrostoritve lahko najdejo.

V primeru, da izvajanje mikrostoritve prestavimo na drug naslov, jo mora strežnik Consul samodejno deregistrirati, mikrostoritev pa se samodejno re-

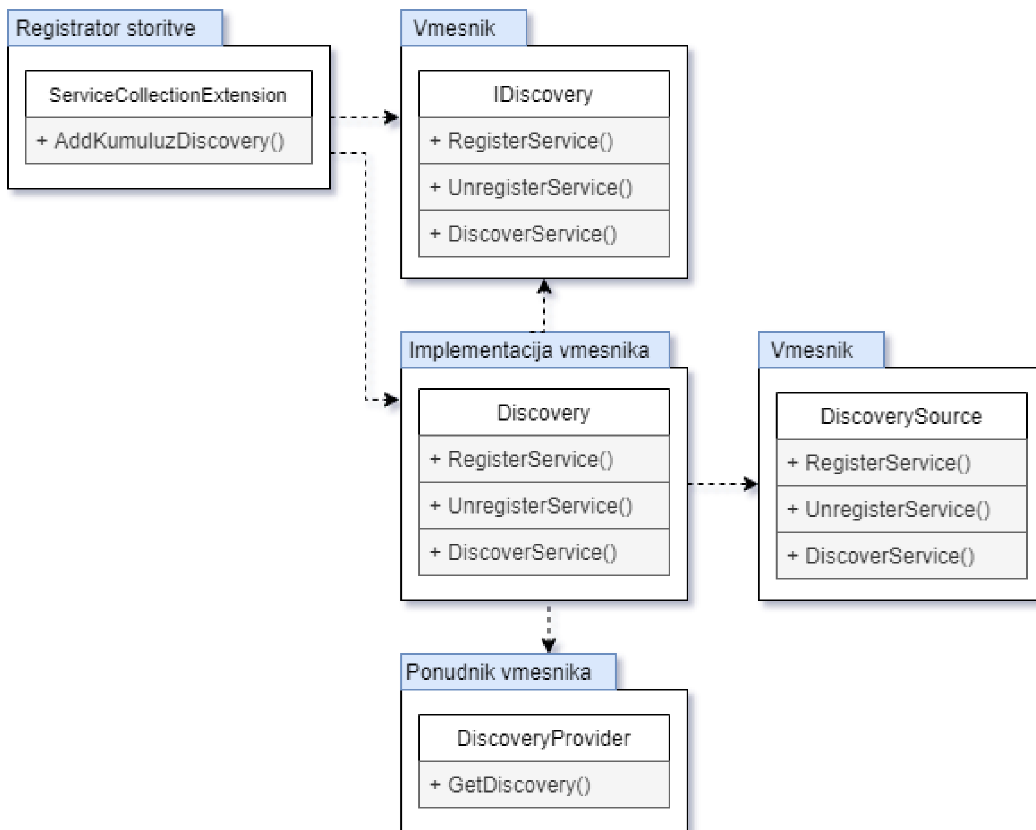
gistrira takoj, ko se prične ponovno izvajati.

5.3.2 Načrt implementacije

Na sliki 5.6 je prikazana struktura programa, ki implementira zgoraj opisane zahteve. Razred *ServiceConfig* je namenjen vključitvi mehanizma odkrivanja v našo mikrostoritev. Ob klicu metode *AddKumuluzDiscovery()* se glede na parametre naredi instanca razreda *Discovery*, ki je implementacija vmesnika *IDiscovery*. Omogoča nam registracijo, odjavo in odkrivanje mikrostoritev. Našo mikrostoritev lahko registriramo s klicem metode *RegisterService()*, ki ji po želji podamo tudi parametre v objektu tipa *RegisterOptions*. Razred *Discovery* vsebuje implementacije mehanizma odkrivanja za različne konfiguracijske strežnike. Zaenkrat naš mehanizem podpira samo konfiguracijski strežnik Consul. Ob klicu metode *Register()* se preko privzete implementacije na strežnik Consul shranijo podatki o naši mikrostoritvi, ob klicu metode *Unregister()* pa se ti podatki z njega odstranijo. Najpogosteje uporabljena metoda pa je *DiscoverService()*, ki nam na podlagi podanih parametrov vrne naslov iskane mikrostoritve, če je ta registrirana na konfiguracijskem strežniku.

5.3.3 Implementacija

Mehanizem odkrivanja se lahko v našo mikrostoritev doda s klicem metode *AddKumuluzDiscovery()* (glej izsek kode 5.19). Ta metoda se izvede nad objektom tipa *IServiceCollection* in kot parameter sprejme akcijo nad tipom objekta *InitializationOptions*. To je objekt, ki vsebuje pot do konfiguracijske datoteke, instanco objekta *ILogger*, ki se uporablja za zapisovanje dogodkov, in ime konfiguracijskega vira, ki se bo uporabljal za odkrivanje mikrostoritev. V trenutno obravnavani metodi se najprej nad objektom tipa *InitializationOptions* izvede podan delegat, ki temu objektu nastavi vrednosti. Nato se naredi nova instanca razreda *Discovery*, ki se ji ob inicializaciji poda prej kreiran objekt. Ko je objekt inicializiran, se ta doda tudi v zbirko storitev.



Slika 5.6: Struktura implementacije mehanizma za odkrivanje mikrostoritev.

```
1 public static IServiceCollection AddKumuluzDiscovery(this
    ↪ IServiceCollection services, Action<
    ↪ InitializationOptions> options)
2 {
3   InitializationOptions opt = new InitializationOptions();
4   options(opt);
5   var discovery = new Discovery(opt);
6   DiscoveryProvider._discovery = discovery;
7   services.AddSingleton(discovery);
8
9   return services;
```

```
10 }
```

Izsek kode 5.19: Metoda, ki doda naš mehanizem za konfiguracijo v zbirko storitev.

Sedaj si lahko pogledamo, kako izgleda razred *Discovery*, ki je implementacija vmesnika *IDiscovery*. Ob njegovi inicializaciji se najprej izvede koda, ki se nahaja v konstruktorju (glej izsek kode 5.20). Ta sprejme parameter tipa *InitializationOptions*, ki smo ga že omenili. S pomočjo podatkov, shranjenih v tem parametru, si ustvari objekt tipa *ConfigOptions*, ki kasneje služi kot parameter za kreiranje konfiguracije mikrostoritev, ki smo jo obravnavali v prejšnjem poglavju. Na koncu se v konstruktorju naredi še implementacija odkrivanja storitev za izbran vir, ki je v našem primeru lahko le Consul.

```
1 public Discovery(InitializationOptions options)
2 {
3   _logger = options.Logger ?? new NullLogger<Discovery>();
4
5   _configOptions = new ConfigOptions();
6   _configOptions.SetConfigFilePath(options.ConfigFilePath);
7   _configOptions.SetExtension(options.Extension);
8   _configOptions.SetLogger(_logger);
9
10  switch(options.Extension)
11  {
12    case Extension.Consul: _discoverySource = new
        ↪ ConsulDiscovery(_configOptions, _logger);
13      break;
14    default:
15      break;
16  }
17 }
```

Izsek kode 5.20: Implementacija konstruktorja razreda *Discovery*.

Razred *Discovery* poleg konstruktorja vsebuje tudi metode, ki kličejo ostale metode v implementaciji odkrivanja za izbrani vir, zato bo najbolje, da si jih pogledamo kar na primeru odkrivanja s strežnikom Consul.

Implementacija za vir Consul se nahaja v razredu *ConsulDiscovery*. Vse implementacije različnih virov morajo implementirati vmesnik *DiscoverySource*. Ob inicializaciji objekta tipa *ConsulDiscovery* se najprej pokliče konstruktor (glej izsek kode 5.21). V njem se najprej ustvari instanca objekta za konfiguracijo (glej prejšnje poglavje), preko katere se nato pridobijo potrebni podatki za sam proces odkrivanja storitev. Nato pa se ustvari odjemalec, ki lahko v našem primeru komunicira s strežnikom Consul.

```
1 public ConsulDiscovery(ConfigOptions configOptions, ILogger
    ↪ logger)
2 {
3   _config = new Config(configOptions);
4   _logger = logger;
5
6   (_startRetryDelay, _maxRetryDelay) = Common.
      ↪ GetRetryDelays(_config);
7
8   //getting source address
9   string sourceAddress = _config.Get<string>("kumuluzee.
      ↪ discovery.consul.hosts");
10  if (string.IsNullOrWhiteSpace(sourceAddress))
11    sourceAddress = ADDRESS;
12
13  //creating consul client
14  try
15  {
16    _client = new ConsulClient(c =>
17    {
18      c.Address = new Uri(sourceAddress);
19    });
20  }
21  catch
22  {
23    _client = null;
24  }
25
26  if (_client == null)
```

```
27      _logger.LogWarning("Thare was problem creating consul
           ↪ client.");
28 }
```

Izsek kode 5.21: Implementacija konstruktorja razreda *ConsulDiscovery*.

Po končani izvedbi konstruktorja je odkrivanje storitev preko strežnika Consul pripravljeno za uporabo. Sedaj si lahko pogledamo, kako izgledajo metode *RegisterService()*, *UnregisterService()* in *DiscoverService()*.

Začnimo kar z metodo *RegisterService()* (glej izsek kode 5.22), ki je odgovorna, da našo storitev registrira na strežnik Consul. Metoda ob klicu sprejme parameter tipa *RegisterOptions*, v katerem so shranjeni podatki o naši storitvi. Najprej preverimo, ali smo našo mikrostoritev predhodno mogoče že kdaj registrirali. V primeru, da ne, se ustvari nova instanca objekta tipa *ConsulServiceInstance*, ki predstavlja našo storitev. Nato se kliče metoda *Run()*, ki se izvaja neprestano. V tej metodi se najprej preveri, ali je naša storitev že registrirana na strežniku Consul. Če ni, se izvede registracija na strežnik, sicer pa se pokliče metoda *SendHearthbeat()*, ki strežniku Consul sporoči, da je naša storitev še aktivna. Po končani izvedbi katerekoli od teh dveh metod se izvede čakanje predhodno določenega časovnega intervala, potem pa se metoda *Run()* rekurzivno ponovno izvede.

```
1 public string RegisterService(RegisterOptions options)
2 {
3   if (_consulServiceInstance != null)
4   {
5     _logger.LogInformation("Service is already registered."
         ↪ );
6     return _consulServiceInstance.Id;
7   }
8
9   var regConfig = Common.GetServiceRegisterConfiguration(
       ↪ _config, options);
10  _consulServiceInstance = new ConsulServiceInstance(
       ↪ regConfig);
11
12  _canRun = true;
```

```
13   Run(regConfig.Discovery.PingInterval * 1000);
14
15   return _consulServiceInstance.Id;
16 }
17 private async void Run(int pingIntervalMs)
18 {
19   if (!_canRun)
20     return;
21
22   //if service is already registered
23   if (_consulServiceInstance.IsRegistered)
24     await SendHeartbeat();
25   else
26     await Register(_startRetryDelay);
27
28   await Task.Delay(pingIntervalMs);
29   Run(pingIntervalMs);
30 }
```

Izsek kode 5.22: Primer implementacije metode *RegisterService()*, ki registrira našo storitev na strežnik Consul.

Predpostavimo, da je bila registracija naše mikrostoritve uspešna. Sedaj jo želimo odregistrirati, za kar se uporablja metoda *UnregisterService()* (glej izsek kode 5.23). Ta je zelo preprosta. Najprej se preveri, ali je mikrostoritev sploh registrirana. Če to drži, se jo s pomočjo odjemalca strežnika Consul odregistrira preko njenega identifikatorja. Metoda pred koncem izvajanja le še preveri njeno stanje izvedbe, ki ga posreduje njenemu klicatelju.

```
1 public async Task<ExecutionStatus> UnregisterService()
2 {
3   ExecutionStatus status;
4
5   if(_consulServiceInstance == null)
6   {
7     status = ExecutionStatus.Bad();
8     status.Message = $"No service is registered";
9     return status;
```

```
10      }
11      _canRun = false;
12
13      try
14      {
15        var result = await _client.Agent.ServiceDeregister(
            ↪ _consulServiceInstance?.Id);
16        if (result.StatusCode == HttpStatusCode.OK)
17        {
18          status = ExecutionStatus.Good();
19          status.Message = $"Service with id {
              ↪ _consulServiceInstance.Id} was successfully
              ↪ unregistred.";
20        }
21        else
22        {
23          status = ExecutionStatus.Bad();
24          status.Message = $"There were some problems
              ↪ unregistering service with id {
              ↪ _consulServiceInstance.Id}.";
25        }
26
27      }
28      catch
29      {
30        status = ExecutionStatus.Bad();
31        status.Message = $"There were some problems
            ↪ unregistering service.";
32      }
33
34      return status;
35    }
```

Izsek kode 5.23: Primer implementacije metode *UnregisterService()*, ki odregistrira našo storitev iz strežnik Consul.

Poglejmo si še zadnjo ključno metodo pri odkrivanju storitev. To je metoda *DiscoverService()* (glej izsek kode 5.24). Ta ob klicu sprejme parameter

tipa *DiscoverOptions*, ki hrani podatke o tem, katero storitev poskušamo najti. Najprej preko odjemalca strežnika Consul pridobimo vse storitve, ki ustrezajo našim zahtevam. V primeru, da nam pridobivanje storitev s strežnika spodleti, za nadaljevanje uporabimo kar predhodno najdene storitve, ki ustrezajo trenutnim zahtevam. Nato gremo s pomočjo zanke skozi vse storitve. V zanki za iskano storitev iz konfiguracije preberemo prehod URL in si ga shranimo v seznam. Naknadno nastavimo še spremljanje sprememb v konfiguraciji za njegovo vrednost. Na koncu le še pokličemo metodo *GetRandomServiceInstance()*, ki glede na podane zahteve naključno izbere ustrezno najdeno storitev, in vrnemo naslov, na katerem je dosegljiva.

```
1 public async Task<string> DiscoverService(DiscoverOptions
     ↪ options)
2 {
3   options.CompleteDiscoverOptions();
4
5   try
6   {
7     var res = await _client.Health.Service(options.
         ↪ SearchServiceKey, "", true);
8     if (res == null || res.StatusCode != HttpStatusCode.OK)
9     {
10      throw new Exception();
11    }
12
13    _discoveredServices[options.SearchServiceKey] =
14      res.Response.Select(e => new DiscoveredService(e)).
           ↪ Where(e => e.Version != null).ToList();
15  }
16    catch
17    {
18    _logger.LogError("Error retrieving healthy service
         ↪ instances from Consul.");
19  }
20
21  var allDiscoveries = _discoveredServices.ContainsKey(
       ↪ options.SearchServiceKey) ?
```

```
22      _discoveredServices[options.SearchServiceKey] : new
            ↪ List<DiscoveredService>();
23
24    foreach (var discovery in allDiscoveries)
25    {
26      var watchNamespace = $"/environments/{options.
            ↪ Environment}/services/{options.ServiceName}/{
            ↪ discovery.Version}";
27
28      if(_gatewayURLs.Where(e => e.Id == watchNamespace).Any
            ↪ () == false)
29      {
30        _logger.LogInformation($"Creating a watch for {
              ↪ watchNamespace}");
31        var key = "gatewayUrl";
32
33        _gatewayURLs.Add(new GatewayURLWatch()
34        {
35          Id = watchNamespace,
36          URL = _config.Get<string>(key)
37        });
38
39        _config.Subscribe(key, (string val) =>
40        {
41          foreach(var gatewayURL in _gatewayURLs)
42          {
43            if(gatewayURL.Id == watchNamespace)
44            {
45              gatewayURL.URL = val;
46            }
47          }
48        });
49      }
50    }
51
52    var service = Common.GetRandomServiceInstance(
          ↪ allDiscoveries, _gatewayURLs, options);
53    return service ?? "";
```

```
54 }
```

Izsek kode 5.24: Implementacija metode *DiscoverService()*, ki na strežniku Consul poišče naslov za dostop do želene storitve.

5.3.4 Primer uporabe

Za uporabo mehanizma za odkrivanje storitev moramo najprej dodati referenco na naš projekt za odkrivanje mikrostoritev. Tokrat konfiguracijsko datoteko *config.yaml* dopolnimo tako, da bo vsebovala vse potrebne podatke za delovanje mehanizma (glej izsek kode 5.25).

```
1 kumuluzee:
2   # name of our service
3   name: test-service
4   # version of our service
5   version: 1.0.2
6   server:
7     # url where our service will live
8     base-url: http://localhost:5000
9     http:
10      port: 5000
11      address: http://localhost
12  env:
13    name: dev
14  config:
15    namespace: ''
16    start-retry-delay-ms: 100
17    max-retry-delay-ms: 500
18    consul:
19      hosts: http://localhost:8500
20  discovery:
21    ttl: 30
22    ping-interval: 20
23    consul:
24      # address of consul server
25      hosts: http://localhost:8500
```

```
26        deregister-critical-service-after-s: 120
```

Izsek kode 5.25: Primer vsebine konfiguracijske datoteke za potrebe mehanizma za odkrivanje mikrostoritev.

Nato v metodi *ConfigureServices()* zbirki storitev dodajmo še mehanizem, ki mu podamo podatek o lokaciji konfiguracijske datoteke in kateri konfiguracijski strežnik naj uporablja (glej izsek kode 5.26). V našem primeru bomo uporabili Consul.

```
1     services.AddKumuluzDiscovery(options =>
2   {
3     options.SetConfigFilePath(Path.GetFullPath("config.yaml
        ↪ "));
4     options.SetExtension(DiscoveryExtension.Consul);
5   });
```

Izsek kode 5.26: Primer dodajanja mehanizma za odkrivanje mikrostoritev v našo mikrostoritev.

Mehanizem za odkrivanje mikrostoritev je sedaj dodan. Če hočemo, da se naša mikrostoritev registrira na strežnik Consul, moramo izvesti registracijo. To naredimo s klicem metode *RegisterService()* (glej izsek kode 5.27), ki jo implementira naš mehanizem.

```
1     DiscoveryProvider.GetDiscovery().RegisterService();
```

Izsek kode 5.27: Registracija naše mikrostoritve.

Preko grafičnega vmesnika strežnika Consul lahko preverimo, ali je bila registracija naše mikrostoritve uspešna in ali pravilno deluje (glej sliko 5.7).

Za pridobivanje spletnega naslova mikrostoritve z želenim imenom enostavno pokličemo metodo *DiscoverService()*, ki sprejme parameter tipa *DiscoverOptions()* (glej izsek kode 5.28).

```
1     var discoverOptions = new DiscoverOptions
2             {
3                 ServiceName = "test-service",
4                 Environment = "dev",
5                 AccessType = AccessType.Direct,
```

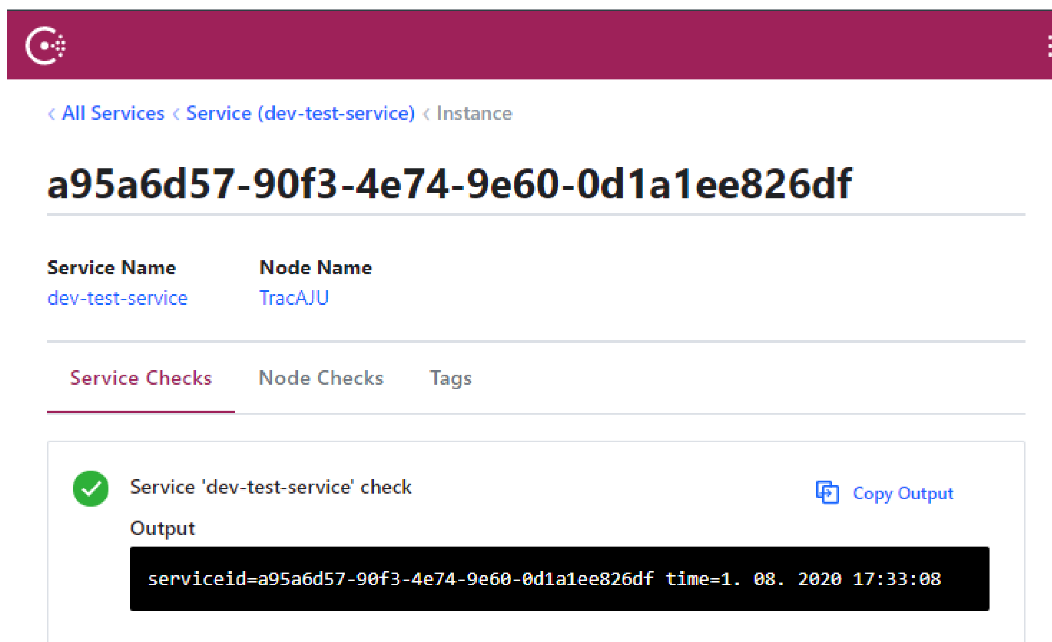
```
6                   Version = ">=1.0.0"
7               };
8       var address = DiscoveryProvider.GetDiscovery().
        ↪ DiscoverService(discoverOptions);
```

Izsek kode 5.28: Primer pridobivanja naslova za dostop do želene mikrostoritve.

Če pa želimo storitev odregistrirati s strežnika Consul, pokličemo metodo *UnregisterService()* (glej izsek kode 5.29).

```
1       DiscoveryProvider.GetDiscovery().UnregisterService();
```

Izsek kode 5.29: Primer deregistracije naše mikrostoritve.



Slika 5.7: Primer prikaza registrirane storitve preko grafičnega vmesnika strežnika Consul.

5.4 Evalvacija

Z zgoraj implementiranimi rešitvami smo izpolnili vse zastavljene cilje oz. rešili vse probleme, opisane v poglavju 4. Komponente, ki smo jih razvili v tem poglavju, lahko enostavno in učinkovito uporabimo pri razvoju mikrostoritev v ogrodju .NET Core. Njihov razvoj je bil kar zahteven, predvsem zaradi uporabe strežnikov Consul in Etcd. Potrebno je bilo najti knjižnico, ki z njima omogoča enostavno komuniciranje. Struktura implementacije rešitev je čista in modularna, kar pomeni, da nam dodajanje novih funkcionalnosti oziroma spreminjanje teh ne predstavlja večjih težav. S prihodom novejših verzij jezika C#, bo mogoče implementirane rešitve še dodatno poenostaviti. Na primerih uporabe smo pokazali, da razvite komponente uspešno rešujejo omenjene probleme. Z uporabo razvitih komponent za preverjanje vitalnosti, konfiguracijo in odkrivanje mikrostoritev lahko torej dosežemo, da bo vzdrževanje naših mikrostoritev enostavnejše in učinkovitejše. Njihova uporaba nima negativnega vpliva na učinkovitost izvajanja mikrostoritev.

Poglavje 6

Zaključek

V diplomski nalogi smo spoznali arhitekturo mikrostoritev in postopek izdelave mikrostoritev v ogrodju .NET Core z uporabo jezika C#. Na podlagi pridobljenega znanja smo opisali tri probleme, ki se pogosto pojavljajo pri tej arhitekturi, in za vsakega implementirali rešitev, za katero smo prikazali tudi enostaven primer uporabe.

Pri rešitvi za preverjanje vitalnosti storitev smo implementirali enostaven modularen mehanizem, ki nam vrne podatke o stanju storitve. Implementirana rešitev vsebuje osnoven nabor kontrol zdravja, ki pa se lahko po potrebi še nadgradijo ali enostavno dodajo. V prihodnosti bi bilo uporabno narediti tudi grafični vmesnik, preko katerega bi uporabnik lažje preveril vitalnost mikrostoritve.

Izdelava mehanizma za konfiguracijo storitev nam služi kot rešitev za problem kompleksne konfiguracije v arhitekturi mikrostoritev. Pri implementaciji smo ga poskušali narediti čim bolj modularnega ter fleksibilnega. V primeru, da se kdaj pokaže potreba po uporabi nekega drugega konfiguracijskega vira, se lahko ta doda brez pretiranega spreminjanja trenutnega mehanizma. V prihodnosti bi bilo dobro imeti podporo pridobivanja vrednosti iz konfiguracijskih virov preko atributov, ampak trenutno je to v jeziku C# preveč kompleksno.

Kot zadnjo smo implementirali rešitev za odkrivanje mikrostoritev. Izmed

vseh rešitev je bila implementacija te najzahtevnejša. V njej smo uporabili tudi našo rešitev za konfiguracijo, preko katere se pridobijo potrebni parametri za inicializacijo tega mehanizma. V prihodnosti bi bilo tako kot pri mehanizmu za konfiguracijo mikrostoritev dobro dodati še kakšen drug vir za odkrivanje.

Z izdelavo teh treh mehanizmov smo rešili zgolj nekaj glavnih problemov pri arhitekturi mikrostoritev. Cilj v prihodnosti je ustvariti rešitve tudi za ostale probleme, ki jih v tej diplomski nalogi nismo imeli priložnost omeniti.

Literatura

[1] Advantages and disadvantages of microservices architecture. Dosegljivo: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/. [Dostopano: 10. 8. 2020].

[2] Difference between synchronous and asynchronous transmission. Dosegljivo: https://www.geeksforgeeks.org/difference-between-synchronous-and-asynchronous-transmission/. [Dostopano: 3. 1. 2020].

[3] Everything you should know about microservices. Dosegljivo: https://www.leewayhertz.com/microservices-architecture-examples/. [Dostopano: 3. 1. 2020].

[4] Future of microservices. Dosegljivo: https://subscription.packtpub.com/book/application_development/9781785885082/1/ch01lvl1sec11/future-of-microservices. [Dostopano: 7. 7. 2020].

[5] Introduction to microservices. Dosegljivo: https://www.nginx.com/blog/introduction-to-microservices/. [Dostopano: 2. 1. 2020].

[6] Introduction to microservices: Part i. Dosegljivo: https://dzone.com/articles/introduction-to-microservices-part-1. [Dostopano: 2. 1. 2020].

[7] Microservices: Decomposing applications for deployability and scalability. Dosegljivo: https://www.infoq.com/articles/microservices-intro/. [Dostopano: 2. 1. 2020].

[8] Microservices: Externalized configuration. Dosegljivo: https://dzone.com/articles/microservices-externalized-configuration. [Dostopano: 9. 7. 2020].

[9] Microservices: Service registration and discovery. Dosegljivo: https://dzone.com/articles/getting-started-with-microservices-2. [Dostopano: 9. 7. 2020].

[10] Microservices vs soa: What's the difference? Dosegljivo: https://dzone.com/articles/microservices-vs-soa-whats-the-difference. [Dostopano: 2. 1. 2020].

[11] What is a container? Dosegljivo: https://www.docker.com/resources/what-container. [Dostopano: 3. 1. 2020].

[12] What is a container? Dosegljivo: https://devopscube.com/what-is-docker/. [Dostopano: 3. 1. 2020].

[13] What is docker? Dosegljivo: https://opensource.com/resources/what-docker. [Dostopano: 3. 1. 2020].

[14] What is .net core. Dosegljivo: https://www.c-sharpcorner.com/article/what-is-dot-net-core/. [Dostopano: 7. 7. 2020].

[15] What is .net core. Dosegljivo: https://stackify.com/net-ecosystem-demystified/. [Dostopano: 7. 7. 2020].

[16] What is .net core. Dosegljivo: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. [Dostopano: 7. 7. 2020].

[17] C. Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, May 2015.

[18] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, 2017.

[19] Sam Newman. *Building Microservices : Designing Fine-Grained Systems*. O'Reilly Media, Inc, USA, 2015.

[20] A. Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[21] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.

[22] Miran Varga. Arhitektura, rojena za oblak. Dosegljivo: `https://www.monitor.si/clanek/arhitektura-rojena-za-oblak/188415/`. [Dostopano: 2. 1. 2020].