Глава 1

Микросервиси

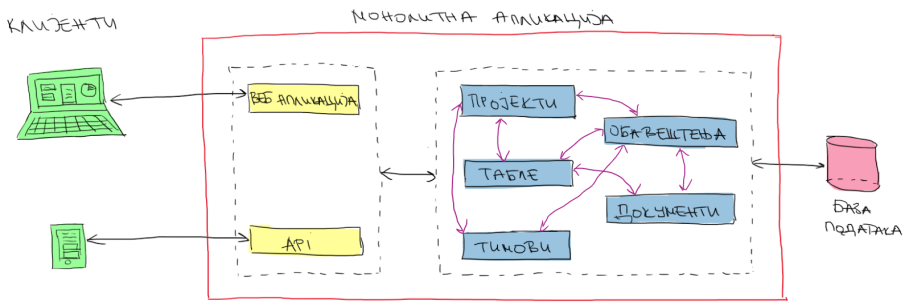
Поглавље је у изради.

Развој софтвера је сложен процес којим се од идеје добија софтверско решење. Идеја која подстиче започињање овог процеса најчешће настаје из потребе да се реши неки проблем из пословног света, да се аутоматизује постојећи процес или да се унапреде постојеће функционалности. Развој софтвера подразумева прегршт активности од управљачки-оријентисаних као што су сакупљање ресурса, управљање ресурсима, дефинисање проблема, организацију послова и одговорности и др. до имплементационо-оријентисаних као што су анализа проблема, дизајн и пројектовање решења, имплементација, тестирање, испоручивање, одржавање и др. Због тога, сматрам да нема много смисла дефинисати развој софтвера у једној или две реченице, али треба имати на уму шта све овај процес обухвата.

Ово поглавље сам започео дискусијом о процесу развоја софтвера као целини како бих направио паралелу са потпроцесом имплементације, који нас, програмере, највише интересује. Традиционално, учени смо да размишљамо о неком софтверском систему као целини, тј. као о производу који нам омогућава да извршавамо разне активности. Тако, на пример, апликација Microsoft Word нам омогућава да „пишемо електронске документе", Google Chrome да „прегледамо веб", Spotify да „слушамо музику", итд.

Сасвим природно, уколико размишљамо о томе да је потребно да имплементирамо овакве и друге апликације, ми пред собом постављамо задатке као што су: „потребно је да направим апликацију којом ћу писати електронске документе" или „потребно је да направим апликацију којом ћу водити пројекте". И ту долазимо до проблема. Овакав начин размишљања нас природно води ка томе да апликације треба развијати као целину, тј. да их развијамо као монолитне апликације.

Монолитне апликације можемо најједноставније дефинисати као апликације које раде у једном процесу. Суштина ове дефиниције значи да, уколико уклонимо било који део кода из апликације, ма колико једноставан



Слика 1.1: Графички приказ монолитне архитектуре апликације за управљање пројектима.

или узак тај део кода био, цела апликација неће бити у стању да функционише без њега. Ово је проблем. Наравно, можете рећи да сигурно неће неко доћи и тек тако обрисати код из апликације. Међутим, да ли је то баш случај у пракси? Ваши менаџери су на основу клијентских захтева одлучили да је потребно додати нову активност у апликацији и доделили Вам тај задатак. Сигурно је да ћете додати нови код који ту функционалност имплементира, на пример, нека то буде нека нова класа. Ако сте добар програмер, онда ћете можда приметити да у коду постоји класа која је врло слична оној коју сте управо написали и шта ћете у том случају урадити – рефакторисати. Дакле, *обрисаћете* код са једног места и написати га негде другде. Или, *обрисаћете* код на неком месту и уместо њега написати нови. У сваком случају, бришете код. Ко гарантује да нисте покварили процес монолитне апликације?[1]

1.1 Проблеми монолитних апликација

Први пут када сам кренуо да радим на немонолитној апликацији[2], био сам „купљен". Није било потребе да ме неко други убеђује у предности таквих организација кода у односу на монолитне апликације на којима сам до тада радио. Ипак, то не значи да ћете и Ви тако одреаговати први пут, због чега је било неопходно да препознам и наведем што више аргумената „против" развоја монолитних апликација[3]. Ево неких од њих:

[1] Одговор је тим за тестирање или систем који аутоматски покреће претходно написане тестове, под претпоставком да имате сјајан тим за тестирање којем не промиче ниједна грешка или да сте написали адеквантне тестове.

[2] Овде мислим да микросервисну апликацију, али нисам хтео да уводим овај термин док не дођем до дефиниције касније.

[3] Далеко од тога да сам непоколебљиви противник монолитних апликација. Заправо, сасвим је јасно из дискусије у овој секцији да се у сваком аргументу трудим да наведем алтернативе које оправдавају монилитне апликације што боље умем. У каснијем тексту ћу се поново осврнути на ову тему и тада ћу дискутовати о томе када је монолитна архитектура погодна.

Један извор кода Монолитне апликације традиционално имају један пројекат који садржи сав код, поготово уколико се користе напредна развојна окружења попут Visual Studio, Qt Creator, Intellij IDEA и др. Велики број датотека којима оваква развојна окружења морају да управљају доводе до успоравања оваквих алата за развој. Неретко се присећам приче једног колеге који је радио на пројекту у којем је изграђивање апликације замрзло читав рачунар. Системи за изградњу кода попут CMake уводе модуларност у смислу да се појединачне компоненте апликације могу развијати на различитим пројектима, па чак и на различитим репозиторијумима кода. Међутим, они не решавају проблем изграђивања кода. Да би се монолитна апликација изградила у целости, потребно је довући код свих компоненти на једно место и поново их изградити. Поново, и ово је могуће раздвојити писањем специјализованих скриптова, на пример, у програмском језику Python и тиме решити и овај проблем (донекле). Међутим, то што је ово једно могуће решење, не значи да је то *добро* решење. Заправо, специјализовани скриптови уводе додатну логику и усложњавају цео пројекат. Не знам каквог сте Ви става, али ја ћу радије затворити репозиторијум и питати искуснијег колегу како систем функционише, него да дешифрујем шта скриптови раде. А шта ако искуснији колега не постоји?

Међузависност у развоју Монолитни приступ онемогућује да се компоненте испоручују независно. Не само клијентима, већ и другим тимовима у развојној организацији, као што је тим за тестирање. Продуктивност свих тимова би била утолико бржа уколико бисмо могли да им испоручујемо компоненту по компоненту.

Отежана разумљивост Врло је неугодан осећај када имате испред себе пројекат од неколико стотина хиљада линија кода, поготово ако сте се запослили као јуниор без радног искуства. Ово је нешто са чиме се, практично гледано, свако од нас сусретне у неком тренутку у каријери. Ретки су случајеви да јуниори без искуства раде на коду „од нуле". Уколико је неопходно да разумемо свих ститину хиљада линија како бисмо сагледали систем у целости, то је велики проблем. Са друге стране, уколико је могуће да нам главни програмер додели задатке везано за само једну компоненту која је потпуно одвојена од остатка апликације, вероватно ћемо бити самопоузданији приликом мењања тог дела кода. Наравно, ово није проблем који мучи само почетнике. Без обзира на сениоритет, немогуће је тако лако снаћи се у коду који видимо по први пут, а у којем су све компоненте међусобно повезане.

Јака спрегнутост компоненти и технологија Имате монолитну апликацију која управља подацима за једну банку. Ову апликацију користе и банкарски службеници и власници банкарских рачуна (звaћемо их

корисници надаље). На почетку пројекта сте се одлучили да користите SQL Server као систем за трајно чување података и написали сте доста кода који управља подацима у „релационом свету". Банкарски службеници су задовољни радом апликације, али став корисника је да би апликација могла радити брже. Како бисте повећали перформансе апликације, одлучујете се да корисничке податке изместите у нерелациони СУБП MongoDB. Сасвим сигурно, спремни сте да рефакторишете део апликације који се бави корисничким подацима из угла корисника, али шта је са делом апликације који користе банкарски службеници? Сва је прилика да овакву измену нисте могли да предвидите пре пет година када сте започели рад на апликацији. То значи да сви делови кода који раде над корисничким подацима морају бити рефакторисани у новој технологији. Додатно, сви тестови који су написани за постојећи код се морају писати изнова за нови код. Немонолитне апликације „форсирају" независност између компоненти, што олакшава промену технологије у било ком тренутку.

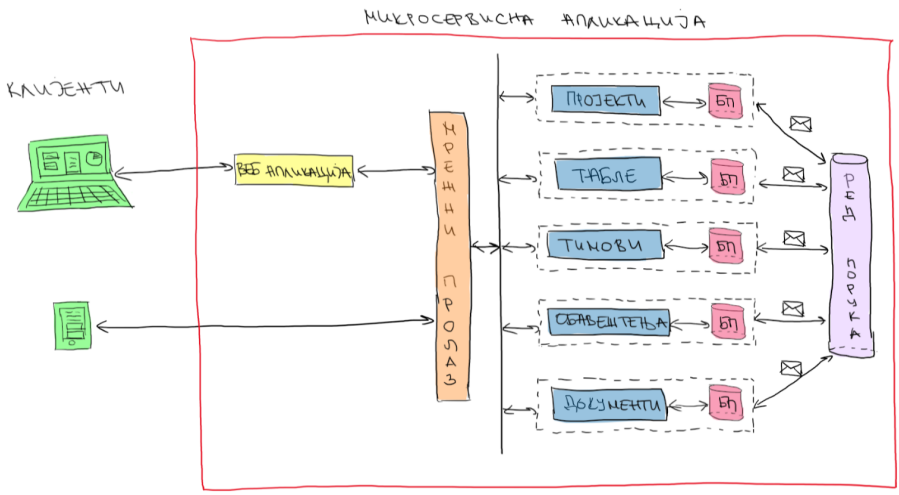
Неробусност на пропусте С обзиром да цела апликација ради у једном процесу, уколико било који део загушава рад, онда је и цела апликација загушена. Под загушењем можемо подразумевати различите пропусте: густ саобраћај, експлозија динамичких објеката, дефекти, итд. Заправо, цела апликација представља уско грло за саму себе. Видећемо како потпуно раздвајање компоненти отвара елегантне технике за решавање ових проблема.

Немогућност испоручивања делова апликације У реду, пронашли смо дефект у једном од 20 компоненти које чине нашу апликацију и успешно смо га исправили и тестирали и спремни смо да испоручимо нову верзију. Шта сада? Потребно је да изградимо поново апликацију у целости, да угасимо стару верзију и да поставимо нову верзију. У зависности од сложености овог процеса, постављање нове верзије може потрајати, због чега апликација није доступна одређени временски период. У добу када се тежи тзв. „*zero downtime*" приступу испоручивања, ово понашање је неприхватљиво. Немонолитне апликације далеко једноставније решавају овај проблем.

1.2 Микросервисне апликације као решење

Померање од монолитне архитектуре превасходно подразумева одустајање од покретања целе апликације у оквиру једног процеса. *Микросервисне апликације* су оне чије се компоненте извршавају у више процеса.

Иако се микросервисне апликације често разумеју као дистрибуиране апликације, важно је разумети да ипак постоји јасна разлика између ова два концепта. У општем случају, дистрибуиране апликације не дефинишу границе између раздвојених процеса. То значи да под дистрибуираним



Слика 1.2: Графички приказ микросервисна архитектуре апликације за управљање пројектима.

апликацијама можемо подразумевати и апликације које се извршавају у целости на више чворова у дистрибуираној мрежи. Дугим речима, можемо испоручити једну монолитну апликацију на више рачунара које ће, на неки начин, радити заједно и такву апликацију (у ширем смислу) назвати дистрибуираном апликацијом. Међутим, овакву апликацију нипошто не смемо назвати микросервисном. Суштина микросервисних апликација јесте да сваки процес (надаље користимо термин *микросервис*) извршава само онај део кода који је везан за једну компоненту унутар апликације.

Приликом дизајнирања архитектуре микросервисних апликација, важно је имати на уму неколико кључних особина сваког микросервиса. За почетак, као што смо већ напоменули, сваки микросервис имплементира једну целовиту функционалност која има јасно дефинисане границе (тј. компоненту апликације). Такође, сваки микросервис се покреће као посебан процес. Додатно, сваки сервис има своје помоћне ресурсе, као што су базе података, које не треба да дели са осталим микросервисима. Коначно, потребно је посебно обратити пажњу на сарадњу микросервиса коришћењем напредних техника као што су асинхрони редови порука. Слика 1.2 приказује једну могућу архитектуру засновану на микросервисима за апликацију која управља пројектима. Приметимо да архитектура која је описана овом сликом прати све наведене особине. Додатно, можемо приметити да клијентске апликације не приступају микросервисима директно, већ индиректно путем мрежног пролаза. О свим овим техникама, али и неким другим, биће више речено у наредним поглављима.

Размотримо сада како оваква архитектура решава проблеме које сам изложио у претходној секцији:

Један извор кода по микросервису Сваки микросервис има могућност да буде потпуно одвојен од свих осталих микросервиса. Не само да на овај начин можемо омогућити јасну поделу посла у оквиру тимова, већ можемо врло једноставно омогућити неке савремене технике развоја софтвера, као што су итеративност, инкременталност, дистрибуиран рад, итд.

Нема зависности у развоју Развој једног микросервиса је потпуно независтан од развоја осталих микросервиса. Самим тим, када један тим заврши са развојем неког микросервиса, он га може испоручити осталим тимовима у развојној организацији, без потребе да чека остале тимове да заврше остале микросервисе.

Олакшана разумљивост Како је сваки микросервис независан и дефинисан јасним границама, разумевање сваког микросервиса је једноставније. Додатно, можемо говорити једноставнијим језиком када описујемо систем као целину, управо зато што назив микросервиса описује цео један подсистем.

Технолошка независност Како сваки микросервис користи своје помоћне ресурсе, то је могуће користити различите системе за управљање базама података и друге сервисе или библиотеке у сваком микросервису. Штавише, сваки микросервис може бити имплементиран у другачијем програмском језику. Овиме се омогућава да сваки микросервис користи најбоље ресурсе за извршавање својих задатака. Додатно, замена једног ресурса другим је утолико лакша пошто не утиче на остатак система.

Висока робусност система Уколико се приликом иницијалног дизајна превиди да ће неки микросервис имати густ саобраћај, могуће је једноставно покренути још један, исти такав микросервис и преполовити број захтева. Дакле, скалирање микросервиса је тривијално. Додатно, уколико неки микросервис напрасно падне, аутоматски системи могу препознати овај догађај и аутоматски покренути нову инстанцу, па инструирати мрежном пролазу да рутира захтеве на нову инстанцу. Ово је практично немогуће постићи у монолитној архитектури.

Инкрементално испоручивање Уколико се пронађе дефект у неком микросервису, развојни тим може реаговати исправљањем дефекта, а затим испоручивањем само тог микросервиса. Остали микросервиси ће и даље опслуживати захтеве без проблема током процеса замене тог једног микросервиса.

1.3 Дванаест фактора

Како би се максимизирала употребљивост микросервисне архитектуре, постављене су одређене препоруке којих се треба придржавати. Ове пре-

поруке су познате као дванаест фактора и апликације које их поштују се називају апликације са дванаест фактора (енг. *the twelve-factor app*).

1. *Репозиторијум кода* (енг. *codebase*). Сваки микросервис треба да садржи свој изворни код, у самосталном репозиторијуму. Репозиторијуми треба да буду праћени коришћењем система за верзионисање кода. Репозиторијум се може испоручити на различита окружења: тестирање, продукција, итд.

2. *Зависности* (енг. *dependencies*). Сваки микросервис је изолован и дефинише само своје зависности од пакета, библиотека, итд. Промене у зависностима неког микросервиса не треба да утичу на читав систем.

3. *Конфигурација* (енг. *config*). Информације о подешавању микросервиса треба изместити из кода самог микросервиса. Овим информацијама се управља помоћу спољашњих алата за конфигурацију. На овај начин се омогућује репликација микросервиса у различитим окружењима.

4. *Помоћни сервиси* (енг. *backing services*). Помоћни ресурси попут складишта података, система за кеширање, управљачима порука и др. морају бити доступни микросервисима путем URL адресе. На овај начин се помоћни ресурси одвајају од самих микросервиса, чиме се омогућава једноставност њихове замене другим помоћним ресурсима.

5. *Изградња, инсталација, покретање* (енг. *build, release, run*). Свако испоручивање треба да дефинише одвојене процесе за изградњу, инсталацију и покретање кода. Свака од ових фаза у процесу испоручивања треба да буде јединствено идентификована и да подржава могућност поништавања. Савремени системи за непрекидну интеграцију (CI) и непрекидно испоручивање (CD) омогућују овај фактор.

6. *Процеси* (енг. *processes*). Сваки микросервис се мора извршавати у оквиру засебног процеса, изоловано од осталих сервиса. Сви подаци који треба да буду запамћени се не чувају у оквиру ових процеса, већ у екстерним помоћним ресурсима (као што су базе података).

7. *Везивање портова* (енг. *port binding*). Све функционалности које неки микросервис нуди чине део његовог јавног интерфејса. Овај интерфејс је доступан спољашњем окружењу путем јединственог порта који је придружен том микросервису.

8. *Конкурентност* (енг. *concurrency*). Микропроцеси скалирају коришћењем великог броја мањих идентичних копија насупрот повећању ресурса једне инстанце на најмоћнијој машини.

9. *Једнократност* (енг. *disposability*). Микросервиси морају бити такви да је њихово покретање или заустављање брзо и грациозно. Процеси треба да минимизују време покретања (идеално, покретање једног микросервиса не би требало да буде дуже од неколико секунди). Грациозно заустављање подразумева престанак ослушкивања нових захтева, могућност да се добије одговор на тренутно активне захтеве и, коначно, гашење процеса. Ово омогућава брзу измену кода или конфигурације и робусност продукционих испоручивања.

10. *Једнакост развоја/продукције* (енг. *dev/prod parity*). Окружења у којима раде програмери, окружења за тестирање и продукциона окружења треба да буду што је могуће сличнија. У микросервисној архитектури, програмер који у развојном окружењу пише нови део кода може након неколико сати да испоручи тај код у продукцији. У традиционалним приступима развоја, ово је често немогуће.

11. *Вођење дневника* (енг. *logging*). Дневници дају увид у понашање апликације која се извршава. Дневнике можемо разумети као ток агрегираних, временских догађаја који се сакупљају над свим извршним процесима и помоћним сервисима. Микросервисне апликације не би требало да воде рачуна о писању или управљању дневницима. Довољно је исписивати поруке на стандардни излаз. У тестним и продукцијским окружењима, токови дневника из свих процеса се сакупљају и прослеђују једном сервису или више њих, као што су: терминал за праћење догађаја у реалном времену, помоћни сервиси за складиштење сервиса, сервиси који алгоритмима машинског учења анализирају дневнике и дају извештаје о раду апликације, и др.

12. *Администраторски процеси* (енг. *admin processes*). Административне или управљачке процеса треба покретати као једнократне процесе. Ови задаци могу укључивати: миграције података, чишћење података, довлачење аналитичких података за креирање извештаја итд. Код администраторских процеса се испоручује заједно са остатком микросервиса и извршавају се на идентичном окружењу у продукцији.

1.4 Елементарни обрасци развоја

Како микросервисна архитектура представља супротан приступ развоја монолитним архитектурама, потребно је дефинисати нове технике и обрасце пројектовања који ће се користити у развоју микросервисних апликацијама. У поглављима који следе се бавимо напредним техникама развоја, међутим, постоје и неке елементарније технике на које треба скренути пажњу. Ове технике се, наравно, могу користити у развоју апликација на било којој архитектури, али интересантно је то што се ове технике најчешће примењују у развоју микросервисних архитектура.

1.4.1 Инверзија зависности

Претпоставимо да имамо две функционалности које зависе једна од друге, на пример, једна компонента апликације омогућава кориснику да уноси податке за електронску пошту, а друга компонента имплементира механизам слања електронске поште (на пример, помоћу Gmail провајдера). Нека су ове функционалности учаурене у две класе: `EmailController` и `GmailEmailService`[4]. Једна могућа имплементација зависности ових компоненти би изгледала овако:

```
class EmailController
{
    private GmailEmailService _emailService;

    EmailController(GmailEmailService emailService)
    {
        _emailService = emailService;
    }

    boolean CollectDataAndSendEmail()
    {
        var from = ...;
        var to = ...;
        var subject = ...;
        var body = ...;

        // Валидација прочитаних поља пре слања поруке...

        return _emailService.SendMailViaGmail(
            from, to, subject, body);
    }
}

class GmailEmailService
{
    private GmailProvider _gmailProvider;

    boolean SendMailViaGmail(string from,
                             string to,
                             string subject,
                             string body)
    {
```

[4]Приметимо да и класа `GmailEmailService` зависи од треће класе `GmailProvider`. Међутим, за сада се фокусирамо само на однос између класа `EmailController` и `GmailEmailService`, али техника коју будемо приказали биће, наравно, применљива и за све остале зависности у нашем пројекту.

```
        return _gmailProvider.Send(from, to, subject, body);
    }
}
```

Као што видимо, класа `EmailController` зависи од имплементације класе `GmailEmailService` да изврши посао слања електронске поруке. Зависност између класа је феномен који се природно јавља у објектно-оријентисаној парадигми и неопходан је у ситуацијама када желимо да раздвојимо одговорности између класа. У примеру изнад, класа `EmailController` је задужена само за сакупљање и валидацију информација, док је одговорност за слање електронске поруке учаурена у класу `GmailEmailService`. Спајање ових класа у једну би нарушило један од основних постулата објектно-оријентисане парадигме.

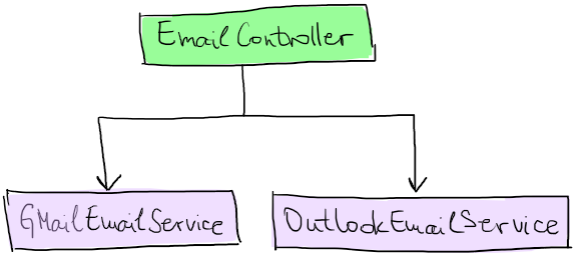
Дакле, овај приступ функционише без проблема. Међутим, шта уколико сутрадан одлучимо да променимо провајдера тако да се електронске поруке шаљу путем, на пример, Outlook провајдера? Или, шта уколико је потребно да омогућимо да се у различитим окружењима користе различити провајдери? Један могући приступ би био чување *заставице* (енг. *flag*) на нивоу класе `EmailController` којом се одређује који провајдер ће бити контактиран, на пример:

```
class EmailController
{
    private GmailEmailService _gmailService;
    private OutlookEmailService _outlookService;
    private boolean _sendViaGmail;

    EmailController(GmailEmailService gmailService,
                    OutlookEmailService _outlookService,
                    boolean sendViaGmail)
    {
        _gmailService = gmailService;
        _outlookService = outlookService;
        _sendViaGmail = sendViaGmail;
    }

    boolean CollectDataAndSendEmail()
    {
        var from = ...;
        var to = ...;
        var subject = ...;
        var body = ...;

        // Валидација прочитаних поља пре слања поруке...

        return _sendViaGmail ?
```

```
            _gmailService.SendMailViaGmail(from, to, subject, body) :
            _outlookService.SendMailViaOutlook(from, to, subject, body);
    }
}

class GmailEmailService
{
    private GmailProvider _gmailProvider;

    boolean SendMailViaGmail(string from,
                             string to,
                             string subject,
                             string body)
    {
        return _gmailProvider.Send(from, to, subject, body);
    }
}

class OutlookEmailService
{
    private OutlookProvider _outlookProvider;

    boolean SendMailViaOutlook(string from,
                               string to,
                               string subject,
                               string body)
    {
        _outlookProvider.ConnectToServer();
        _outlookProvider.SetFromField(from);
        _outlookProvider.SetToField(to);
        _outlookProvider.SetSubjectField(subject);
        _outlookProvider.SetBody(body);
        return _outlookProvider.SendMail();
    }
}
```

Приступи који користе заставице и сличне механизме не скалирају добро. Уколико је потребно користити већи број провајдера, онда нам заставице нису довољне, па бисмо морали да користимо енумераторе. Генерално, могли бисмо да смислимо додатне захтеве чија решења би захтевала нове променљиве о којима треба водити рачуна, што смањује степен разумевања кода.

Кључан је моменат приметити да класа `EmailController` не мора да зна за детаље имплементације класа `GmailEmailService` и `OutlookEmailService`. Заправо, она се на ове класе ослања на два начина, у зависности од фазе



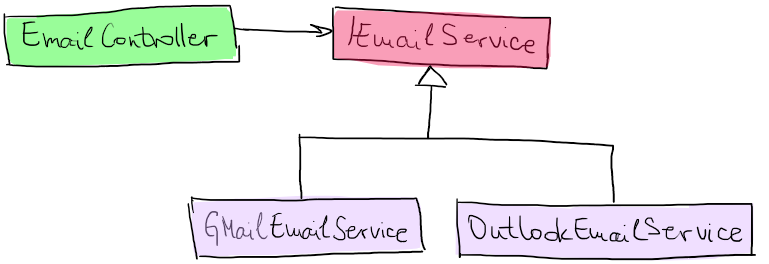
Слика 1.3: Класа `EmailController` директно зависи од класа `GmailEmailService` и `OutlookEmailService`.

животног циклуса апликације. Све што је класи `EmailController` неопходно *у фази превођења* кода јесте да над објектима које користи постоји одговарајући метод који она може да позове, а шта ће тај метод тачно радити ће бити израчунато тек *у фази извршавања* кода. Другим речима, у фази превођења кода, класи `EmailController` је довољно да чува референцу на интерфејс који спецификује „уговор" између те класе и других класа које користи, а прослеђивање објекта конкретне фазе се може одложити за фазу извршавања кода.

Како бисмо ово понашање имплементирали, додајемо интерфејс `IEmailService` која има метод који је неопходан класи `EmailController` за слање електронске поруке. Све остале класе које имплементирају конкретан механизам слања електронских порука (то су класе `GmailEmailService` и `OutlookEmailService`) морају да имплементирају овај интерфејс на свој, спефицичан начин. Наравно, класа `EmailController` сада може да чува само једну инстанцу интерфејса `IEmailService` – која ће јој конкретна инстанца бити прослеђена биће одлучено у фази извршавања.

```
interface IEmailService
{
    boolean SendMail(string from,
                     string to,
                     string subject,
                     string body);
}

class EmailController
{
    private IEmailService _emailService;

    EmailController(IEmailService emailService)
    {
        _emailService = emailService;
```

```
    }

    boolean CollectDataAndSendEmail()
    {
        var from = ...;
        var to = ...;
        var subject = ...;
        var body = ...;

        // Валидација прочитаних поља пре слања поруке...

        return _emailService.SendMail(from, to, subject, body);
    }
}

class GmailEmailService : IEmailService
{
    private GmailProvider _gmailProvider;

    boolean SendMail(string from,
                     string to,
                     string subject,
                     string body)
    {
        return _gmailProvider.Send(from, to, subject, body);
    }
}

class OutlookEmailService : IEmailService
{
    private OutlookProvider _outlookProvider;

    boolean SendMail(string from,
                     string to,
                     string subject,
                     string body)
    {
        _outlookProvider.ConnectToServer();
        _outlookProvider.SetFromField(from);
        _outlookProvider.SetToField(to);
        _outlookProvider.SetSubjectField(subject);
        _outlookProvider.SetBody(body);
        return _outlookProvider.SendMail();
    }
}
```



Слика 1.4: Инверзија зависности класе `EmailController` и класа `GmailEmailService` и `OutlookEmailService` помоћу интерфејса `IEmailService`.

Ако мало поразмислимо шта смо овиме урадили, видимо да смо променили смер зависности између класа. Сада, уместо да класа `EmailController` зависи од класа `GmailEmailService` и `OutlookEmailService`, постигли смо да друге две класе зависе од „уговора" који намеће класа `EmailController` интерфејсом `IEmailService`. Дакле, извршили смо *инверзију зависности* између ових класа.

1.4.2 Убризгавање зависности

Како би се инверзија зависности остварила што безболније, потребно је да постоји механизам који препознаје зависности неке класе дефинисане интерфејсима, а затим тој класи прослеђује конкретне објекте – инстанце класа које имплементирају те интерфејсе. Идеално, овај механизам треба да функционише аутоматски, уз само једну наредбу којом се спецификује која инстанца конкретне класе ће бити прослеђена класи која користи неки интерфејс. Ова наредба се, очигледно, позива тек у фази извршавања и то најчешће као део фазе припреме рада апликације, односно, пре коришћења било које друге класе која дефинише зависности од неких других класа путем интерфејса. Овакав механизам је познат под називом *убризгавање зависности*.

Велики број библиотека и радних оквира долази са предефинисаним механизмима убризгавања зависности и .NET 5.0 радни оквир је један од њих, због чега је и популаран избор за креирање микросервисних апликација, које се снажно ослањају на механизам инверзије зависности. С обзиром да нема много смисла говорити о конкретној имплементацији механизма убризгавања зависности када још нисмо ни почели да програмирамо први микросервис, демонстрацију убризгавања зависности ћемо оставити за касније. Ипак, важно је било продискутовати да је овакав механизам од изузетног значаја како би се омогућило једноставно имплементирање инверзије зависности.