Razvoj web-aplikacija baziran na testiranju

Rožić, Petra

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:490843

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Petra Rožić

RAZVOJ WEB-APLIKACIJA BAZIRAN NA TESTIRANJU

Diplomski rad

Voditelj rada:
Izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, veljača, 2020

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Proces TDD **2**
1.1 Testiranje 2
1.1.1 Testovi visoke razine 4
1.1.2 Testovi niže razine 6
1.2 Refaktorizacija 8
1.3 Metode razvoja softvera 9
1.4 Opis iteracije unutar TDD procesa 10
1.4.1 Crveno-Zeleno-Plavo 12

2 Izrada web-aplikacije primjenom procesa TDD **14**
2.1 Kuharica 14
2.2 Inicijalizacija projekta 15
2.2.1 Potrebni alati za izradu aplikacije 17
2.2.2 Kreiranje jednostavnih testova 18
2.3 Izrada web-aplikacije 21
2.3.1 Testiranje sadržaja web-aplikacije 21
2.3.2 Testiranje dohvaćanja podataka iz baze 34
2.3.3 Testiranje korisničkog sučelja 43
2.3.4 Testiranje potvrde dodavanja novog recepta 51
2.3.5 Refaktorizacija 60
2.4 Daljnji razvoj aplikacije 60

Bibliografija **62**

Uvod

Svaki projekt ima tri važna aspekta, to su kvaliteta, trošak i vrijeme. U današnjem konkurentnom okruženju naglasak je na razvoju projekata koji teže visokokvalitetnom rezultatu uz minimalni utrošak vremena i novčanih sredstava. Zbog pritiska s vremenom, često razmišljanje programera glasi: Ako radi, ne diraj. (eng. *If it works, don't fix it*). Posljedica takvog razmišljanja je nepregledan programski kôd kojeg je teško održavati funkcionalnim. Osim nestrukturiranog kôda, težnja za brzim razvojem softvera uvelike povećava mogućnost pogreška. Kako bismo pogreške sveli na minimum, u pomoć priskaču testovi te različite metode razvoja softvera. U ovom radu poslužit ćemo se procesom Test-Driven Development (kraće proces TDD).

Proces Test-Driven Development vezan je uz agilan razvoj softvera. Kako u svakom razvoju softvera, tako i u procesu TDD, prije svega, bitno je razumjeti zadatak. Nakon dodijeljenog zadatka, programeri prvo pišu testove kojima definiraju ponašanje aplikacije prema zahtjevima tog zadatka te pokreću razvijene testove. Budući da u tom trenutku ne postoji nikakva implementacija rješenja, očekivano je da svi testovi prijavljuju neuspjeh (tj. "padaju"). Tek tada se kreće na razvoj same aplikacije, odnosno na pisanje minimalnog kôda za koji će svi testovi prijaviti uspjeh (tj. "proći").

Cilj ovog diplomskog rada je uvesti čitatelja u metodologiju procesa Test-Driven Development te u praktičnom dijelu prikazati razvoj jednostavne web-aplikacije na temelju prethodno tumačenog procesa. Početne informacije potrebne za spomenuti sadržaj i razvoj web-aplikacije bazirane na testiranju smo preuzeli iz literature *Test-Driven Development with Python* [6].

U danjoj razradi ovog diplomskog rada proći ćemo kroz terminologiju vezanu uz proces TDD, čime se bavi prva cjelina. Osim navedenog, u prvom dijelu rada proučit ćemo sve aktivnosti koje se izvršavaju unutar procesa. U drugom poglavlju bavit ćemo se razvojem web-aplikacije u kojoj je moguć pregled kulinarskih recepata te dodavanje istih. Razvoj navedene aplikacije bazirat će se na procesu TDD.

Poglavlje 1

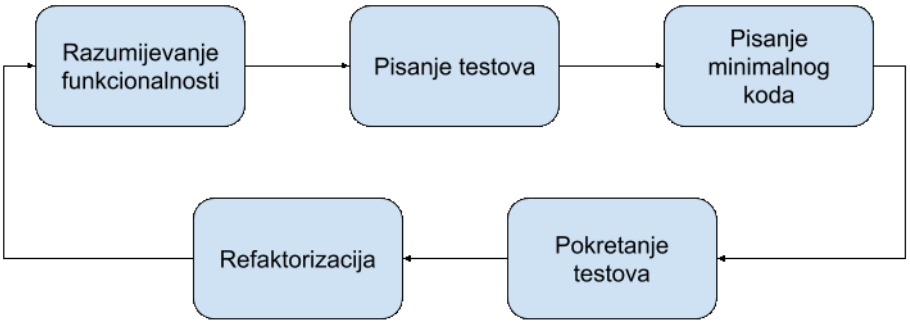
Proces TDD

Proces Test-Driven Development (kraće proces TDD) nastao je kao posljedica *test-first* pristupa razvoju unutar agilne metode Extreme Programming (kraće XP). O metodi XP te njezinim doprinosima bit će više riječi kasnije. Ideja procesa TDD je osigurati pokrivenost kôda testovima pri čemu se u svakom koraku razvoja dodaje samo onoliko programskog kôda koliko je potrebno da bi trenutni testovi prošli. Proces TDD stavlja naglasak na pisanje testova koji testiraju aplikaciju s aspekta korisnika i aspekta programera.

Prije bilo kakve akcije unutar procesa TDD bitno je razumijevanje dodijeljenog zadatka. On može zahtijevati razvoj korisničkog sučelja, uvođenje poslovnog pravila ili razvoj nove funkcionalnosti. Ukoliko su programeru jasni svi detalji vezani uz zadatak tada kreće s pisanjem testova. O samom postupku testiranja bit će riječi u poglavlju 1.1. Nakon dovršetka testova, sljedeći korak je pisanje minimalnog programskog kôda koji testove čini prolaznima. Ovaj korak smatramo riješenim tek kada se prilikom pokretanja testova ne javlja niti jedna pogreška. Prije prelaska na novu funkcionalnost potrebno je još provesti programski kôd kroz proces refaktorizacije. Ideja refaktorizacije je, ukoliko je potrebno, strukturirati programski kôd kako bi bio jasniji i čitljiviji. O refaktorizaciji bit će više riječi u poglavlju 1.2. Ovaj postupak provodi se iterativno te je jednostavnom shemom prikazan na slici 1.1.

1.1 Testiranje

Rijetka su softverska rješenja u kojima se nije potkrala niti jedna softverska pogreška. Svaka razlika između postojećih uvjeta, odnosno uvjeta koji su razvijena unutar softverskog rješenja, te uvjeta koji su definirani dokumentom o korisničkim zahtjevima smatra se *softverskom pogreškom* (eng. *software bug*), kraće pogreška. Jednostavnije rečeno, pogreška je bilo kakvo ponašanje softvera koje je suprotno od zahtjeva korisnika.



Slika 1.1: Jednostavna shema iteracija unutar procesa TDD

Softversko testiranje, kraće testiranje, postupak je kojim se utvrđuje postojanje softverskih pogrešaka i oštećenja na softveru te se uklanjaju uočene anomalije. Odnosno provjerava se zadovoljava li softversko rješenje unaprijed definirane zahtjeve korisnika. Tim postupkom postiže se kvalitetniji softver što, nadalje, utječe na zadovoljstvo korisnika. Testiranje se sastoji od izvršavanja niza pomno razrađenih testova koji su sačinjeni od blokova naredbi kojima se testira određeni dio softvera. Za test kažemo da je prošao ukoliko je očekivana povratna vrijednost jednaka onoj koju softver vraća. U suprotnom kažemo da test pada.

Ovisno o metodi koja se primjenjuje na razvoj softvera, softversko testiranje može biti postupak koji se odvija tijekom cijelog razvoja softvera ili samo jedna faza u nizu. Ako se proizvedeni softver testira samo u jednoj fazi, često je to faza prije puštanja softvera u produkciju, tada su pogreške daleko skuplje i utkane su duboko u strukturu programa te samim time kompleksnije za ispraviti. S druge strane, pojedine metode zahtijevaju da se softver testira tijekom cijelog procesa razvoja. Primjenom tog načina, pogreške se prije uočavaju te su jeftinije za ispraviti.

Testiranje se može provoditi od strane krajnjih korisnika, programera te posebnog tima testera. Kako bi osoba zadužena za testiranje znala što i kako treba testirati, od velike važnosti je razrada korisničkih zahtjeva. Jedan od načina zapisivanja korisničkih zahtjeva jest korisnička priča (eng. *User Story*, kraće US). US je kratki opis zahtjeva od strane korisnika koji definira novu funkcionalnost. Tipični predložak US glasi: *As a <type of user>, I want <some goal> so that <some reason>*. Nama bitna svojstva koje treba zadovoljavati svaka korisnička priča jesu mogućnost testiranja te da budu dovoljno mala kako bi ih sa što manje programskog koda mogli razviti.

Kriterij za ulazak u proces testiranja je detaljno razrađen dokument o korisničkim zah-

tjevima. Proučavanjem i analizom zahtjeva korisnika iz perspektive ispitivanja, započinjemo s procesom testiranja. Navedeni korak je izuzetno bitan kako bismo utvrdili jesu li korisnički zahtjevi provjerljivi ili nisu. Ukoliko pojedini zahtjev nije provjerljiv potrebno je kroz diskusiju s korisnikom doći do preciznije specifikacije zahtjeva. Idući korak je pisanje testova koji pokrivaju korisničke zahtjeve te dijelove softvera koje je potrebno testirati. S fazom pisanja testova isprepliće se faza pripremanja umjetnih podataka kojima se pokrivaju svi mogući scenariji testova. Nakon detaljno razrađenih testova slijedi pokretanje istih te analiza dobivenih podataka.

Svako pokretanje testova zahtijeva određeno vrijeme izvršavanja, a to vrijeme izvršavanja raste linearno s količinom podataka. Stoga, umjesto da iscrpno testiramo na svakom mogućem podatku, bitno je odrediti minimalni skup podataka koji pokriva sve moguće scenarije. Optimalan skup podataka je onaj koji pokriva sve rubne slučajeve te sadrži podatak koji se nalazi između ta dva rubna uvjeta.

Cilj testiranja je pomoću dobro raspisanih testova pronaći pogreške koje su se potkrale tijekom razvoja softvera. Ukoliko testeri više ne mogu pronaći pogreške, ne slijedi da je softver bez pogrešaka jer nam oni ne govore o odsutnosti pogrešaka, već o prisutnosti pogrešaka. Također, ukoliko ne možemo naći nove pogreške, ne slijedi da je softver upotrebljiv jer tek kada isti zadovoljava sve poslovne potrebe korisnika i testiranjem ne možemo naći nove pogrešaka možemo ustvrditi upotrebljivost.

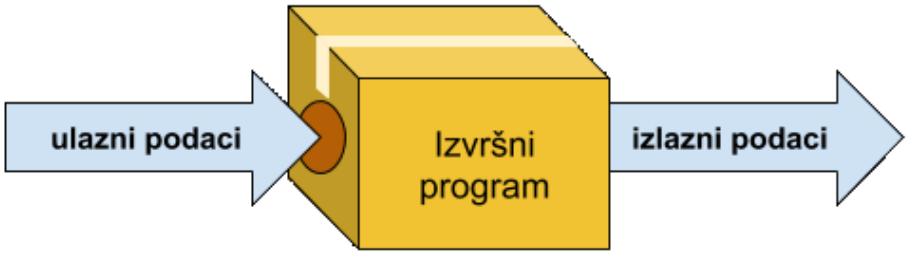
Testove u grubo možemo podijeliti na:

- testovi visoke razine (eng. *high-level tests*) - testiraju aplikaciju sa strane korisnika;
- testovi niže razine (eng. *lower-level tests*) - testiraju aplikaciju sa strane programera.

Neki od testova visoke razine su tzv. *funkcionalni testovi* i *End-to-end testovi*, dok su *jedinični testovi* i *integracijski testovi* neki od testova niže razine. Razvoj softvera putem procesa TDD zasniva se na pisanju funkcionalnih i jediničnih testova. Kako bismo što bolje testovima pokrili funkcionalnosti koristit ćemo navedene testove visoke razine i niže razine. U daljnjoj razradi posvetit ćemo se posebno testovima s aspekta korisnika te testovima s aspekta programera.

1.1.1 Testovi visoke razine

Metoda testiranja softvera tijekom koje testeri ne mare za unutarnjom strukturom, dizajnom i implementacijom, naziva se metoda testiranja crne kutije (eng. *black box testing*). Navedena metoda testira iz aspekta korisnika gdje su prilikom provedbe testova bitni ulazni podaci i povratna vrijednost softvera. Testiranje crne kutije prikazano je ilustracijom 1.2. Takvi testovi dalje se dijele na funkcionalne i nefunkcionalne testove.



Slika 1.2: Testiranje crne kutije

Testovi visoke razine, odnosno testovi s aspekta korisnika, mogu se provoditi ručno (eng. *manual*) ili automatizirano (eng. *automated*). Ručno testiranje izvršava osoba (korisnik, tester ...) koja vrši interakciju sa softverom koristeći tipkovnicu i miš, oponašajući tipičnog korisnika aplikacije. Takva osoba ne mora znati programske jezike i može biti neovisna od programera što omogućuje objektivnu perspektivu. Razmjerno složenošću softvera, takvo testiranje postaje sve skuplje te sklonije ljudskim pogreškama. S druge strane, automatizirano testiranje provode računala izvršavanjem skripti koje su razvili programeri. Cijena ovakvog testiranja ne raste razmjerno složenošću softvera, ali zahtijeva znanje programera o pisanju skripti.

Za pisanje testova potrebno je izdvojiti određeno vrijeme, stoga programeri nisu skloni pisanju testnih skripti. Testiranje složene aplikacije, bez razvijenih testnih skripti, svodi se na pokretanje korisničkog sučelja, postavljanje prijelomnih točaka (eng. *breakpoints*), unos podataka koje korisničko sučelje zahtjeva, pa takav način testiranja iziskuje daleko više vremena. Osim navedenog, takvim načinom testiranja ne stignemo provjeriti sve dijelove aplikacije uslijed promjena programskog kôda. Stoga je brže i efikasnije izdvojiti vrijeme za pisanje testova i pokretati ih prilikom svake promjene.

U praktičnom dijelu rada, tijekom pisanja testova visoke razine poslužit ćemo se funkcionalnim testovima i End-to-end testovima. Njima je zajedničko što testiraju kako aplikacija reagira na aktivnosti korisnika.

Funkcionalni testovi

Vrsta testova koja pokreću web-preglednik i simuliraju aktivnosti korisnika na aplikaciji, kao rezultat prikazuju kako aplikacija funkcionira s aspekta korisnika, nazivaju se *funkcionalni testovi* (eng. *Functional tests*, kraće FT). Funkcionalni testovi govore koje aktivnosti korisnik može izvršavati u aplikaciji te kako aplikacija reagira na pojedinu aktivnost.

Važno je naglasiti da FT-ovi ne znaju kako je aplikacija implementirana, poznate su im samo moguće akcije korisnika te nužni odgovori aplikaciji na svaku akciju. Jedan od mogućih izvora informacija za funkcionalne testove jesu korisničke priče (kraće US). S obzirom na to da FT slijede US, mogli bismo reći da su funkcionalni testovi neka vrsta specifikacije za našu aplikaciju [6].

Kako agilne metode ne teže pisanju specifikacija, funkcionalni testovi s dodatnim komentarima mogu poslužiti kao zamjena za pisanu specifikaciju. Stoga je cilj da FT-ovi budu detaljno pokriveni s komentarima kako bi bili čitljivi svima uključujući i ne-programerima (osobama koje nisu učene čitanju programskih kôdova). Tako raspisani FT-ovi mogu doprinijeti boljem razumijevanju aplikacije te kvalitetnijoj diskusiji o trenutnim i budućim zahtjevima korisnika.

End-to-end testovi

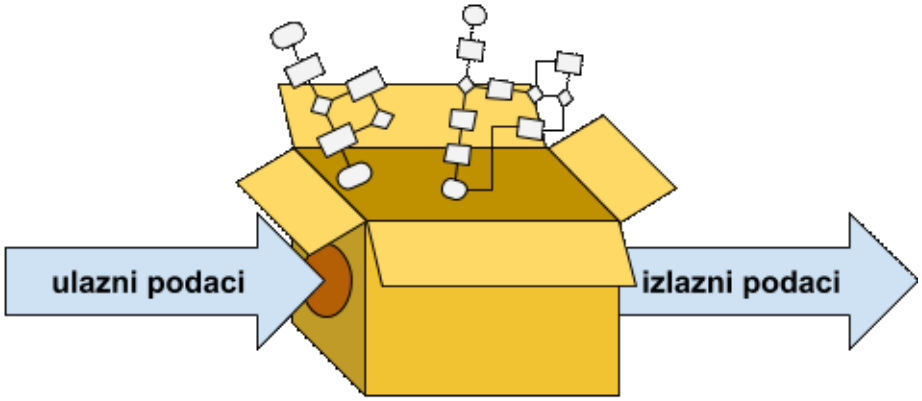
Složenošću aplikacije rastu i moguće akcije koje korisnik izvršava. Kako bismo testirali niz akcija koje krajnji korisnik izvršava, koristit ćemo tzv *End-to-end testovi* (kraće E2E). End-to-end testovi koriste se za provjeru ovisnosti sustava kao i za osiguravanje neometanog protoka podataka između brojnih podsustava i glavnog sustava. Ovi testovi skupi su za održavati te zahtijevaju više vremena kako bi se izvršili. Stoga preporučuje se nekoliko ključnih E2E testova, dok za ostale dijelove se preporučuje oslanjanje na druge tipove testova.

Kao i prethodna vrsta testova, End-to-end testovi ne znaju kako je aplikacija implementirana. Za razvoj ovih testova potrebno je znati moguće akcije korisnika te kakve podatke očekujemo nakon pojedine akcije. Svrha ovog testiranja je identificirati ovisnosti sustava i osigurati integritet podataka između različitih komponenti sustava i sveukupnog sustava.

1.1.2 Testovi niže razine

Za razliku od metode testiranja crne kutije, metoda testiranje bijele kutije (eng. *white box testing*) zahtijeva poznavanje dijela ili cijele unutarnje strukture, dizajna, implementacije softvera koji se testira. Ovim testovima testira se izvorni kôd aplikacije s aspekta programera. Testeri odlučuju koje blokove naredbi žele testirati, potom određuju na kojim podacima će vršiti to testiranje te kakve izlazne podatke očekuju. Takvo testiranje prikazano je ilustracijom 1.3.

S obzirom na to da ovi testovi ne zahtijevaju korisničko sučelje, s pisanjem istih se može krenuti u ranoj fazi. Problem može nastati ukoliko se zahtjevi korisnika, samim time i implementacija, prečesto mijenjaju. Posljedica čestog mijenjanja zahtjeva korisnika jest prilagodba testnih skripti što nadalje dovodi do toga da održavanje istih postaje samo teret. Između ostalog, ovi testovi mogu postati vrlo složeni pa stoga pisanje testova zahtijeva odlično poznavanje programskog jezika i razvoja softvera.



Slika 1.3: Testiranje bijele kutije

Uobičajeno je da se prilikom razvoja softvera isti podijeli na manje module koji se dalje dodjeljuju različitim programerima ili timu programera. Kada je pojedini modul razvijen, njegovu funkcionalnost provjeravamo pomoću jediničnih testova. U konačnici, kada su svi moduli razvijeni, pomoću integracijskih testova provjeravamo interakciju između modula te grade li traženi softver.

Jedinični testovi

Jedni od testova koji prakticiraju testiranje bijele kutije jesu jedinični testovi (eng. *Unit tests*, kraće UT). Ovi testovi se često primjenjuju na pojedinoj komponenti ili pojedinoj metodi modela unutar aplikacije. Cilj je ispitati svaki dio softvera zasebno i osigurati da svaki dio radi kako se očekuje.

UT-ovi testiraju funkcionalnost manjeg dijela programskog kôda te ne bi trebali ovisiti o programskom kôdu (klasama, metodama ...) izvan područja testiranja. Ukoliko je ovisnost neizbježna, pomoću lažnih objekata (eng. *mock objects*) možemo ju zaobići. O tome će biti više riječi kasnije.

Ovi testovi usredotočuju se na testiranje funkcionalnosti pojedinih dijelova i ne otkrivaju probleme koji nastaju u interakciji između različitih dijelova softvera. Za otkrivanje takvih problema koristit ćemo iduću vrstu testova, tj. integracijske testove.

Integracijski testovi

Nakon što se provede testiranje manjih dijelova softvera, potrebno je provjeriti interakciju između tih dijelova. To ćemo učiniti pomoću integracijskih testova (eng. *Integration tests*).

Cilj ovog testiranja je provjeriti je li kombinacije dijelova softvera izvršavaju zahtjeve na očekivani način.

Budući da integracijski test provjerava komunikaciju, bilo između dijelova softvera ili drugih sustava, često ovisi o drugim vanjskim sustavima (npr. baza podataka). Stoga održavanje ovih testova je naporno i skupo. Integracijski testovi provode se na dva načina: metodom odozdo prema gore (eng. *bottom-up*) i metodom odozgo prema dolje (eng. *top-down*).

1.2 Refaktorizacija

Kao što smo vidjeli, testovi imaju veliku ulogu u razvoju kvalitetnih softvera. S vremenom količina testova se nagomila kao i količina linija produkcijskog kôda u kojima se sve teže snalaziti. Kako bismo ublažili zahtjevnost razvoja softvera, od velike je važnosti proces refaktorizacije.

Proces tijekom kojega mijenjamo dizajn ali ne i funkcionalnost aplikacije, naziva se refaktorizacija (eng. *refactoring*) [4]. Tijekom razvoja softvera programeri su usredotočeni na dodavanje novih funkcionalnosti kako bi zadovoljili definirano vrijeme isporuke te zanemaruju vrijednost strukturiranog programskog kôda. Kako bi programski kôd održali čitljivim, potrebno je nakon svake izmjene provjeriti je li potrebna refaktorizacija. Osim što pruža čitljivost kôda, adekvatnom primjenom refaktorizacije u budućnosti na efikasniji način možemo dodati nove funkcionalnosti.

Primjena refaktorizacije donosi nam niz prednosti, ali izaziva i velike rizike. Refaktorizacija se vrši na programskom kôdu koji ne sadrži pogreške te svaka nepromišljena promjena može ih uzrokovati. Kako bismo rizičnost refaktoriranja sveli na minimum, veliku ulogu ima pisanje testova. Stoga refaktoriranje se provodi na programskom kôdu koji je pokriven testovima te su svi testovi prolazni, također nakon svakog refaktoriranja potrebno je provjeriti jesu li testovi i dalje prolazni. Ukoliko se potkrala pogreška, potrebno ju je ispraviti prije nego li krenemo dalje s promjenama u programskom kôdu.

Primjena refaktorizacije na velikim dijelovima kôda često rezultira pogreškama što nadalje zahtijeva pronalazak razloga pojave pogreške, mjesto nastanka te rješavanje istih. Rizik refaktorizacije smanjuje se primjenom procesa refaktorizacije na malim dijelovima programskog kôda. Stoga je efikasnije primijeniti proces na nizu manjih dijelova nakon kojih se ispituje je li funkcionalnost aplikacije narušena. Primjenom manjih iteracija refaktorizacije lakše je uočiti gdje je izazvana pogreška.

Proučavanjem i primjenom refaktorizacije, došlo se do zaključka o principima koje se preporučuje slijediti kako bi razvoj aplikacije bio efikasan, a rezultat bio što kvalitetniji. Jedan od principa kaže: „Nemoj se ponavljati.” (eng. *Don't Repeat Yourself*), odnosno nakon tri dovoljno slična bloka naredbi slijedi refaktoriranje [6]. Nakon dva ista bloka naredbi nije preporučeno refaktoriranje, odnosno izdvajanje u posebnu metodu, jer pisa-

nje sljedećeg bloka naredbi može dovesti do dovoljno sličnog kôda koji zahtijeva manje prilagodbe. U konačnici sva tri bloka možemo izdvojiti u zasebnu metodu dodavanjem potrebnih parametra metodi.

Tijekom razvoja aplikacije često nam dolaze ideje za unaprjeđenje aplikacije te nove funkcionalnosti koje bi mogli implementirati. Potaknuti tim idejama, programeri često dodaju blokove naredbi koje će im jednoga dana možda zatrebati. Kao rezultat takvog razvoja aplikacije slijedi dugačak, nepregledan i neiskorišten kôd. Stoga u procesu razvoja aplikacije bitno je imati na umu još jedan princip razvoja: „Neće ti koristiti!" (eng. *You ain't gonna need it!*) [6].

Osim navedena dva principa kojima se treba voditi tijekom razvoja aplikacija, javlja se i treći koji kritizira čitljivost kôda. Loš odabir varijabli, preduge funkcije, dugačak popis parametra, te još niz drugih nedostataka strukture kôda se jednom frazom može nazvati trulost kôda (eng. *Bad smell*). Kako bi daljnje testiranje, razumijevanje te sami razvoj aplikacije bili što efikasniji bitno je čim ranije uočiti takve nedostatke te ući u proces refaktorizacije istih.

Kao što su testovi sastavni dio procesa razvoja kvalitetnog softvera, tako i refaktorizacija igra veliku ulogu u samom procesu. Nakon kratkog uvoda o vrsti testova i refaktorizacije, detaljno ćemo opisati jednu iteraciju unutar procesa TDD.

1.3 Metode razvoja softvera

Prilikom razvoja softvera bitno je nekoliko čimbenika kvalitete, neki od njih su razumijevanje softvera, točnost i brzina implementacije te timski rad. Kako bi se navedeni čimbenici kvalitete što optimalnije sjedinili, razvijene su razne metode razvoja softvera. Metode razvoja softvera možemo podijeliti u dvije skupine, klasične i agilne metode [5].

U klasičnim metodama naglasak je na planiranju cijeloga procesa razvoja softvera (s točno određenim početkom i krajem) te detaljno razrađenom dokumentacijom. S druge strane, agilna metoda naginje iterativnom razvoju softvera koji omogućuje brzu prilagodbu zahtjevima korisnika te se u što većoj mjeri izbjegava dokumentaciju. Jedan od boljih primjera agilne metode je *Extreme Programming* (XP).

Extreme Programming razvija softver u iteracijama te je cilj svake iteracije razviti novu funkcionalnost ili popraviti odnosno mijenjati postojeću funkcionalnost. Na početku svake iteracije korisnici svoje zahtjeve dostavljaju u obliku kratkih priča (eng. *User stories*, kraće US) koje se kasnije dijele na manje zadatke te se oni dodjeljuju programerima. Inovacije koje je XP razvio jesu test-first development, pair programming, refactoring.

Razvoj gdje test ide prvi (eng. test-first development) teži razvoju softvera u kojemu se prvo razvijaju testovi koji prekrivaju ciljanu funkcionalnost te se naposljetku razvija funkcionalnost koja se testira prethodno napisanim testovima. Bitno je nakon razvoja svake

funkcionalnosti pokrenuti sve prethodno napisane testove kako bi bili sigurni da razvojem nove funkcionalnosti nismo izazvali pogreške u starim funkcionalnostima.

Opisana inovacija, gdje test ide prvi, preteča je procesu Test-Driven Development. Osim navedenog, uz proces TDD usko je vezan i proces refaktorizacije. Stoga možemo reći da se proces TDD sastoji od tri velike akcije a to su pisanje testova, pisanje minimalnog kôda te refaktorizacije. U nastavku detaljno prolazimo kroz iteraciju unutar procesa Test-Driven Development.

1.4 Opis iteracije unutar TDD procesa

Proces TDD sastoji se od niza iteracija koje se sastoje od niza akcija. Niz akcija unutar iteracije detaljno je prikazan na ilustraciji 1.4, te daljnji opis iteracije možemo pratiti na spomenutoj ilustraciji.

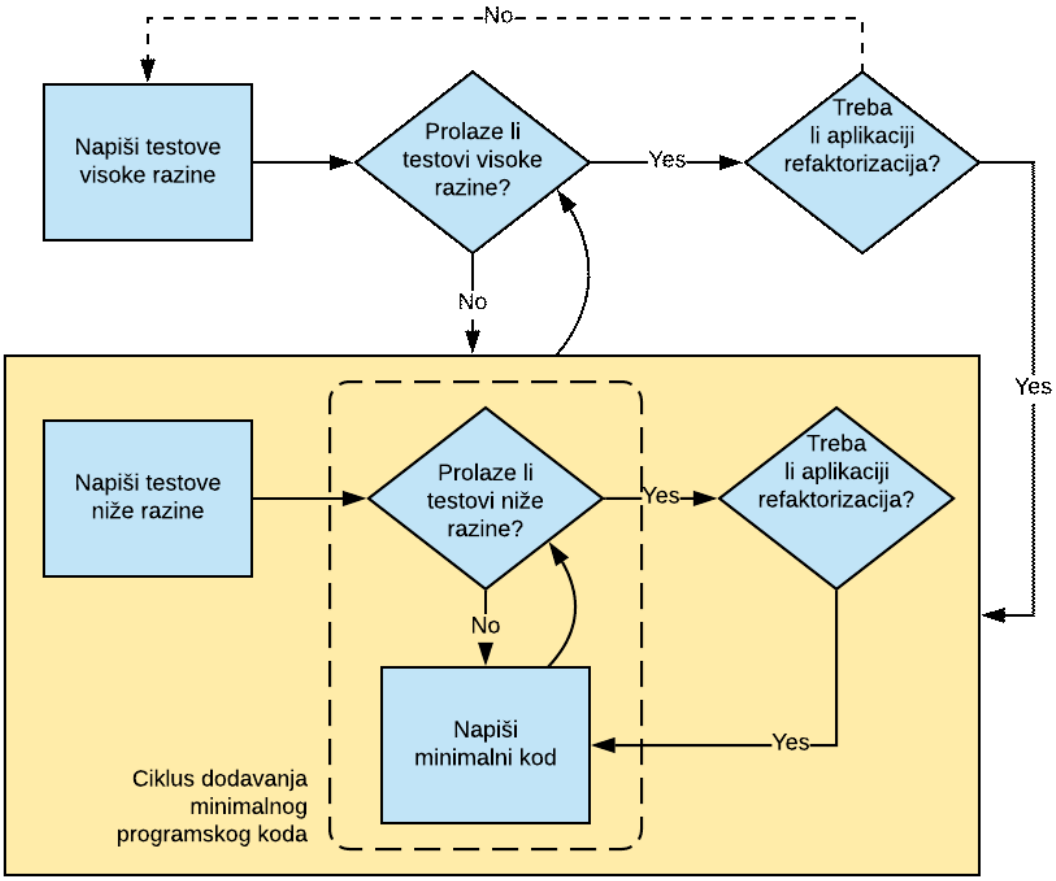
Svaka iteracija započinje pisanjem testova za funkcionalnosti koje tek trebamo razviti iz čega slijedi da će testovi sigurno padati. Ali bitno je naglasiti da oni trebaju padati na očekivani način. Odnosno pogrešku koju nam jave testovi mora biti u skladu s onom koju smo pretpostavili da ćemo dobiti.

Ukoliko testovi jave pogrešku koju nismo očekivali tada moramo djelovati te otkriti gdje se potkrala pogreška. Pogrešku otkrivamo pomoću procesa testiranja, od kojih autor Percival H. ([6]) predlaže:

- dodavanje ispisa na ekranu;
- unaprjeđenje poruke pogreške kako bi prikazala više podataka o trenutnom stanju;
- ručna provjera stranice uz koju je vezana pogreška;
- koristiti metodu koja zaustavlja izvršavanje testa na neko vrijeme kako bismo provjerili trenutno stanje stranice.

Prvi testovi koji se razvijaju jesu oni s aspekta korisnika, odnosno testovi visoke razine. Nakon razvoja testovi se pokreću s ciljem utvrđivanja padaju li na očekivani način. Ukoliko ne padaju na očekivani način tada smo došli do pogreške koju procesom testiranja uklanjamo. S druge strane, ako padaju na očekivani način tada krećemo s pisanjem testova niže razine.

Nakon što napišemo potrebne testove niže razine ulazimo u *Ciklus dodavanja minimalnog programskog kôda* koji se sastoji od dvije akcije. Prva akcija je pokretanje testova niže razine kako bismo vidjeli ruše li se na očekivani način. Ukoliko je odgovor potvrdan, krećemo na sljedeću akciju a to je pisanje minimalnog kôda. Cilj pisanja minimalnog kôda



Slika 1.4: Prikaz jedne iteracije unutar TDD procesa

je stavljanje padajućih testova u prolazno stanje. Prethodno navedene dvije akcije se ponavljaju dokle god svi testovi niže razine ne prolaze. Ukoliko testovi niže razine padaju na neočekivani način, tada krećemo s otkrivanjem pogreške.

Sljedeća akcija je postavljanje pitanja „Treba li strukturirati kôd?”. Ukoliko je odgovor potvrdan tada se vraćamo u *Ciklus dodavanja minimalnog programskog kôda* te pišemo minimalni kôd kojim postižemo pregledniju strukturu te dalje nastavljamo prema već opisanim akcijama. U suprotnom, ako nije potrebno refaktoriranje, tada ponovno pokrećemo testove visoke razine i provjeravamo jesu li oni prošli.

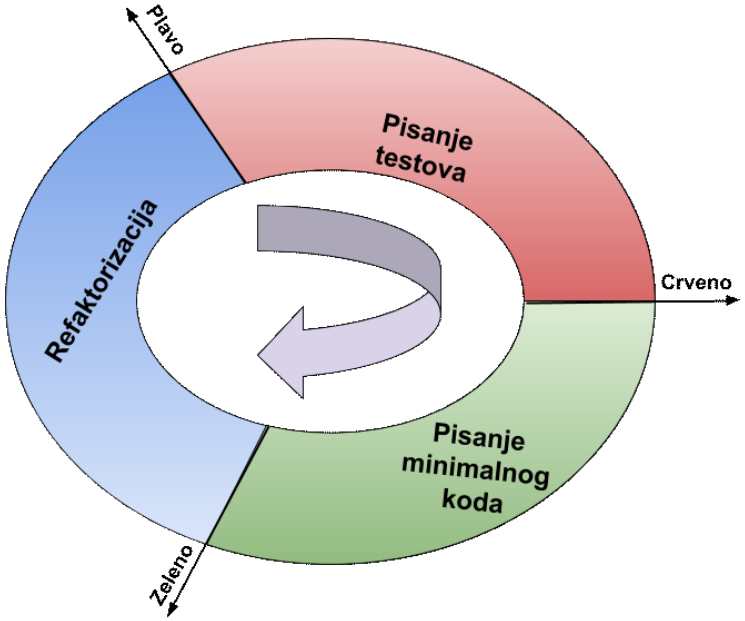
Bitno je napomenuti da u proces refaktorizacije krećemo tek kada nam prolaze svi testovi, a završavamo tek kada utvrdimo da su svi testovi i dalje prolazni. Pokretanje testova

nam osigurava da refaktorizacijom nismo narušili funkcionalnost aplikacije. Naravno, promjenom strukture aplikacije, često se naruše postojeći testovi. Stoga unutar ove akcije dozvoljeno je prilagoditi testove novoj strukturi kôda kako bi oni i dalje bili prolazni.

Dokle god testovi visoke razine nisu prolazni pokrećemo fazu pisanja testova niže razine te dalje slijede već opisane akcije. Nakon što testovi visoke razine prolaze, postavljamo si već poznato pitanje „Treba li strukturirati kôd?". Ukoliko je odgovor negativan tada nastavljamo s pisanjem novih testova visoke razine, u suprotnom nastavljamo s akcijom pisanja testova niže razine.

1.4.1 Crveno-Zeleno-Plavo

Akcije koje se izvršavaju unutar jedne iteracije procesa TDD poznate su i kao Crveno-Zeleno-Plavo (eng. *Red-Green-Blue*) [6]. Navedeno načelo prikazano je na ilustraciji 1.5.



Slika 1.5: Prikaz načela Crveno-Zeleno-Plavo unutar jedne iteracije TDD procesa

U crveno stanje ulazimo kada napišemo testove te ustanovimo da padaju na očekivani način. Za očekivati je da prethodno napisani testovi padaju jer za sada nije napisan programski kôd koji ih čini prolaznima. Slijedi niz iteracija pisanja minimalnog kôda i pokretanja testova. Nakon što utvrdimo da su testovi prošli, tada smo u zelenom stanju. Zeleno

stanje govori da napisani programski kôd funkcionira prema zahtjevima korisnika za koje smo napisali testove. Posljednja akcija unutar iteracije procesa TDD je refaktorizacija. Tijekom procesa refaktorizacije programer prolazi kroz napisani kôd i provjerava može li ga kako unaprijediti. U plavo stanje ulazimo tek kada je programer zadovoljan sa strukturom kôda.

Nakon uvida u terminologiju i načela procesa TDD primijenimo ga na izradi jednostavne aplikacije.

Poglavlje 2

Izrada web-aplikacije primjenom procesa TDD

Sada kada imamo uvid u načela i princip izvršavanja procesa TDD, primijenimo prethodno naučeno te razvimo jednostavnu aplikaciju. Kako bi čitatelj uspješno pratio daljnji sadržaj, očekuje se da je upoznat s izradom aplikacije korištenjem arhitekturalnog obrasca MVC (eng. *Model-View-Controller*), stvaranjem baze pomoću migracija te s osnovama web-aplikacije.

2.1 Kuharica

Aplikacija koju ćemo razvijati namijenjena je pregledavanju recepata te njihovih detalja od koji su istaknuti: popis sastojaka (naziv sastojka, količina i mjerna jedinica) te opis postupka pripreme recepta. Kako bi popis recepata bio ažuran, omogućit ćemo dodavanje novih recepata.

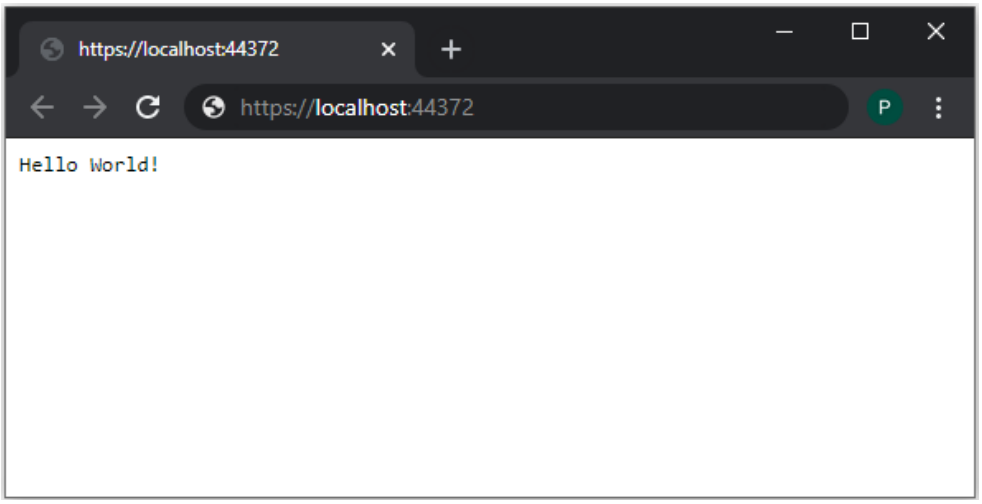
Opisanu aplikaciju razvijat ćemo u skladu s proučenim procesom Test-Driven Development. Poučeni prednostima korisničkih priča, razvijat ćemo funkcionalnosti koje one opisuju. Raspišimo neke US:

- Ja kao korisnik aplikacije, na početnom ekranu želim vidjeti naslov koji glasi *Cookbook* te popis naziva recepata kako bih imao pregled koji se nude.
- Ja kao korisnik aplikacije, klikom na pojedini recept, želim vidjeti detalje recepta kako bih imao bolji uvid u pojedini recept.
- Ja kao korisnik aplikacije, želim dodavati nove recepte u kuharicu kako bi uvijek bila ažurna.

Aplikaciju, u kojoj ćemo razviti sve prethodno navedene funkcionalnosti, nazovimo *Cookbook*. Započet ćemo s inicijalizacijom projekta Cookbook, nadalje ćemo kreirati dva projekta unutar kojih ćemo zasebno razvijati testove visoke razine i testove niže razine. Nakon kreiranja jednostavnih testova započet ćemo s razvojem testova za funkcionalnosti aplikacije te nastaviti dalje primjenjivati proces TDD.

2.2 Inicijalizacija projekta

Projekt Cookbook razvit ćemo unutar okruženja Visual Studio 2017 koristeći programski jezik C# unutar okvira ASP.NET Core. Rezultat kreiranja takvoga projekta jest web-aplikacija koju je moguće odmah izgraditi (eng. *build*) te pokrenuti bez potrebe za pisanjem kôda. Rezultat pokretanja projekta jest sadržaj stranice jednak slici 2.1.



Slika 2.1: Prvo pokretanje projekta

Nakon što smo ustanovili da unutar projekta nema pogrešaka, prilagodimo strukturu aplikacije našim potrebama. Prema zahtjevu okvira ASP.NET Core, prilikom pokretanja aplikacije, klasa *Startup* se izvodi prva te ona sadrži dvije metode:

- metodu *ConfigureServices()* unutar koje se definiraju usluge (eng. *services*) o kojima ovisi aplikacija;
- metodu *Configure()* unutar koje definiramo kako aplikacije reagira na HTTP zahtjeve.

Budući da ćemo projekt strukturirati vodeći se arhitektonskim obrascem MVC (eng. *Model-View-Controller*), prilagodimo klasu *Startup* kao što je prikazano unutar isječka kôda 2.1. Osim izmjene klase *Startup*, potrebno je kreirati direktorije: *Models*, *Views* i *Controllers*.

Listing 2.1: Cookbook/Startup.cs

```
1  public class Startup
2  {
3      public void ConfigureServices(IServiceCollection services)
4      {
5          services.AddMvc();
6      }
7
8      public void Configure(IApplicationBuilder app,
           IHostingEnvironment env)
9      {
10         if (env.IsDevelopment())
11         {
12             app.UseDeveloperExceptionPage();
13             app.UseMvcWithDefaultRoute();
14         }
15
16         app.Run(async (context) =>
17         {
18             await context.Response.WriteAsync("Hello World!");
19         });
20     }
21 }
```

Kao što se arhitektonski obrazac MVC pokazao dobrom praksom, izdvajanje testova iz produkcijskog kôda još je jedna dobra praksa koju ćemo primijeniti u izradi aplikacije. Stoga sljedeći korak je, uz projekt Cookbook, kreirati dodatna dva projekta: Cookbook.LowerLevelTests i Cookbook.HighLevelTests. Unutar projekta Cookbook.LowerLevelTests razvijat ćemo, kao što i ime kaže, testove niže razine (jedinične testove, integracijske testove), dok ćemo unutar projekta Cookbook.HighLevelTests razvijati testove više razine (funkcionalne testove, End-to-end testove). Testove ćemo razvijati pomoću paketa xUnit te ćemo unutar projekta Cookbook.HighLevelTests koristiti dodatni paket Selenium. O potrebnim paketima bit će riječi kasnije.

Nakon dodanih izmjena, struktura aplikacije izgleda ovako:

```
/
└── Solution'Cookbook'
    └── Tests
        └── Cookbook.HighLevelTests
            └── Dependencies
```

```
    Class1.cs
  Cookbook.LowerLevelTests
    Dependencies
    UnitTest1.cs
Cookbook
  Connected Services
  Dependencies
  Properties
  Controllers
  Models
  Views
  appsettings.json
  Program.cs
  Startup.cs
```

Nakon što smo projekt podijelili u tri manja projekta, pogledajmo koji paketi su nam potrebni za njihov razvoj.

2.2.1 Potrebni alati za izradu aplikacije

Kao što smo prethodno naveli, za razvojno okruženje koristit ćemo Microsoft Visual Studio Community 2017. Za izradu aplikacije koristit ćemo okvir ASP.NET Core [1].

Za razvoj projekata koji služe testiranju aplikacije potreban nam je paket xUnit. xUnit je alat otvorenog kôda koji olakšava testiranje dijelova softvera pisanih u .NET tehnologijama [3]. Alternativa ovom paketu je nUnit paket. Za xUnit smo se odlučili jer je prilagođen razvoju softvera pomoću procesa TDD. Odnosno testne metode u potpunosti su izolirane jedna od druge, tj. ne omogućava se dijeljenje podataka između različitih testnih metoda.

Kao što smo naveli u odjeljku 1.1.2 o jediničnim testovima, bitno je da su UT neovisni od dijelova softvera koji nisu u fokusu testa. Kako bismo izbjegli ovisnost o drugim dijelovima softvera, potrebno je dodati paket Moq koji nam omogućuje kreiranje lažnih objekata (eng. *mock objects*) koji simuliraju određenu klasu te definiraju njezino ponašanje. Tako kreirani lažni objekti omogućuju testiranje dijela softvera nad definiranim podacima kojima se inicira određena interakcija. Ovaj paket bit će nam potreban unutar projekta Cookbook.LowerLevelTests.

Znamo da testovi visoke razine testiraju aplikaciju s aspekta korisnika, odnosno oni otvaraju web-preglednik te simuliraju akcije korisnika. Kako bismo automatizirali ove testove, osim paketa xUnit, bit će nam potreban i paket Selenium.WebDriver. Selenium.WebDriver je paket alata za automatizaciju web-preglednika, odnosno pomoću programskog kôda možemo otvoriti web-preglednik, pretražiti elemente na web-stranici, unijeti vrijed-

nosti te u konačnici zatvoriti web-preglednik [2]. Između mnoštva web-preglednika koristit ćemo Google Chrome.

Nakon što pomoću Seleniuma otvorimo web-preglednik, potrebno je definirati na koju web-adresu želimo ići. Naravno, cilj nam je otvoriti web-adresu gdje se nalazi naša aplikacija. Već prilikom kreiranja projekta, .NET Core automatski je definirao web-adresu naše aplikacije te ju postavio na *http://localhost:58883/*.

Verzije prethodno navedenih alata i paketa, koje ćemo koristiti prilikom razvoja aplikacije možemo vidjeti u sljedećoj tablici:

Alat	**Verzija**	**Uloga**
Microsoft Visual Studio Community 2017	15.9.5	Razvojno okruženje
ASP.NET Core	2.2	Okvir za izradu aplikacije
xUnit	2.4.1	Alat za testiranje dijelova softvera
Moq	4.13.1	Alat za simulaciju klasa i njihovog ponašanja
Selenium.WebDriver	3.141.0	Alat za automatizaciju web-preglednika
Google Chrome	79.0.3945.117	Web-preglednik

Sada kada imamo pregled potrebnih alata i paketa za razvoj aplikacije baziranoj na procesu TDD, možemo krenuti s pisanjem testova. Prije pisanja testova kojima ćemo testirati funkcionalnosti koje razvijamo unutar aplikacije, kreirajmo jednostavne testove kako bismo ustanovili funkcionira li sve na očekivani način.

2.2.2 Kreiranje jednostavnih testova

Prije nego krenemo na razvoj složenijih testova kojima ćemo provjeravati funkcionalnosti naše web-aplikacije, kreirajmo jednostavne testove na kojima ćemo objasniti osnovne naredbe. Prilikom kreiranja projekta Cookbook.LowerLevelTests, automatski je generirana klasa *UnitTest1* unutar koje dodajemo jednostavne testove niže razine:

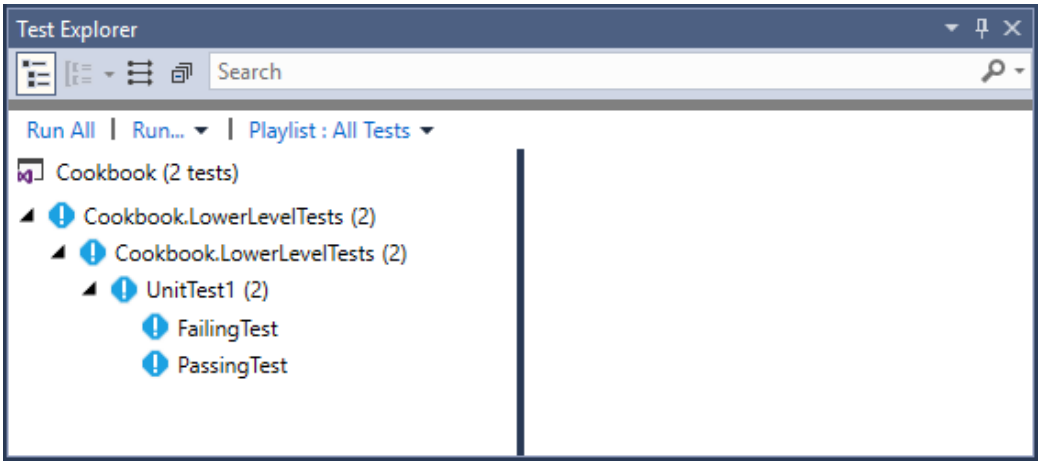
Listing 2.2: Cookbook.LowerLevelTests/UnitTest1.cs

```
1  [Fact]
2  public void PassingTest()
3  {
4      Assert.Equal(4, Add(2, 2));
5  }
6
7  [Fact]
8  public void FailingTest()
9  {
10     Assert.Equal(5, Add(2, 2));
```

```
11 }
12
13 int Add(int x, int y)
14 {
15     return x + y;
16 }
```

U navedenim testovima možemo uočiti naredbe iz paketa xUnit. Naredbom *[Fact]* definiramo sljedeću metodu kao testnu te ju koristimo u prvoj i sedmoj liniji. Osim pomoću *[Fact]*, naredbom *[Theory]* možemo naznačiti da je iduća metoda testna, o njoj će biti više riječi kasnije. Sljedeća naredba u kojoj vidimo primjenu xUnit paketa se nalazi u četvrtoj i desetoj liniji. Naredbom *Assert.Equal* provjeravamo je li dobivena vrijednost jednaka očekivanoj.

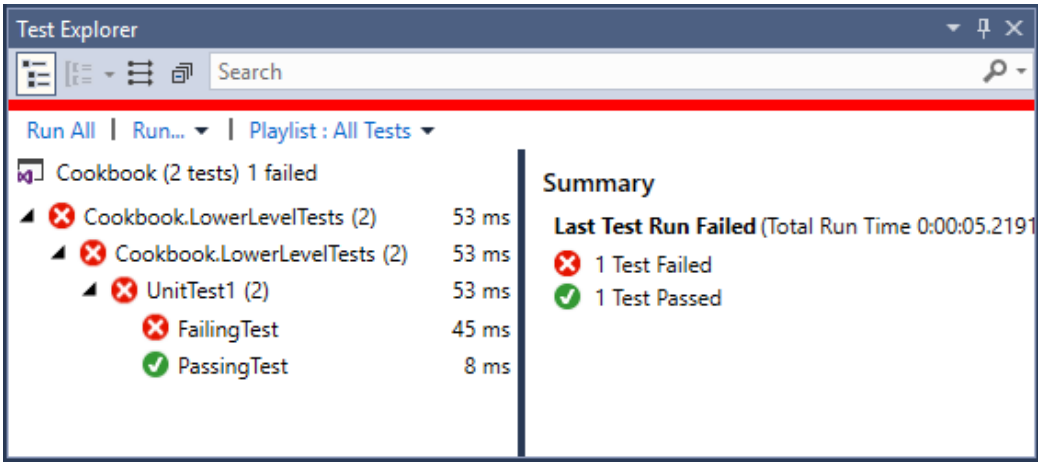
Prethodno dodana dva testa pokrenut ćemo pomoću alata koji dolazi uz Visual Studio, a to je Test Explorer čiji trenutni sadržaj je prikazan na slici 2.2. Klikom na Run All



Slika 2.2: Test Explorer nakon dodavanja prvih testova

pokrećemo sve testove te rezultat testiranja možemo vidjeti na slici 2.3. Takav rezultat je bio očekivan.

Nakon kratkog uvoda o pokretanju testova unutar projekta Cookbook.LowerLevelTests, pogledajmo kako ćemo pokretati testove unutar projekta Cookbook.HighLevelTests. Prije dodavanja testova visoke razine, potrebno je instalirati prethodno navedene pakete (xUnit i Selenium) te implementirati sučelje *IDisposable* unutar klase *Class1* koju je generirao sam projekt. Navedeno sučelje zahtijeva definiciju metode *Dispose()* koja služi zatvaranju web-preglednika kojega smo prethodno otvorili unutar konstruktora klase *Class1*. Upravo



Slika 2.3: Test Explorer nakon pokretanja prvih testova

navedeno otvaranje web-preglednika izvršava Selenium u sedmoj liniji dok ga zatvara u desetoj liniji.

Listing 2.3: Cookbook.HighLevelTests/Class1.cs

```
1  public class Class1 : IDisposable
2  {
3      private readonly IWebDriver _driver;
4
5      public Class1()
6      {
7          _driver = new ChromeDriver();
8      }
9
10     public void Dispose()
11     {
12         _driver.Quit();
13         _driver.Dispose();
14     }
15 }
```

Nakon što smo instalirali pakete te implementirali sučelje, sve je spremno za pisanje prvog testa visoke razine. Unutar klase *Class1* dodajmo test *getContent* kojim provjeravamo sadržaj stranice nakon prvog pokretanja aplikacije, kakvog smo vidjeli na slici 2.1. Dodani test možemo vidjeti u sljedećem isječku koda:

Listing 2.4: Cookbook.HighLevelTests/Class1.cs

```
1  [Fact]
```

```
2 public void getContent()
3 {
4     _driver.Navigate().GoToUrl("http://localhost:58883/");
5
6     Assert.Contains("Hello World!", _driver.PageSource.ToString());
7 }
```

U navedenom testu primjećujemo korištenje alata xUnit i Selenium. Pomoću paketa xUnit, u šestoj liniji, provjeravamo sadrži li web-stranica definirani string. U četvrtoj i šestoj liniji koristimo alat Selenium tako što prvo definiramo web-stranicu koju je potrebno otvoriti, a kasnije dohvaćamo sadržaj, odnosno izvorni kôd u HTML-u iste web-stranice.

Kako bismo pokrenuli testove visoke razine potrebno je otvoriti dodatnu instancu Visual Studia te pokrenuti server projekta Cookbook. Nakon što pokrenemo server potrebno je u prvom Visual Studiu unutar Test Explorera pokrenuti test *Class1.getContent*. Pokretanjem testa otvara se web-preglednik koji je kontroliran od strane Seleniuma, provjerava se sadržaj stranice te se zatvara web-preglednik. Rezultat ovoga testa prikazan je unutar Test Explorera te uočavamo da on prolazi što je očekivano.

Nakon uvoda o pokretanju testova te uvida u potrebne alate krenimo s razvojem naše aplikacije.

2.3 Izrada web-aplikacije

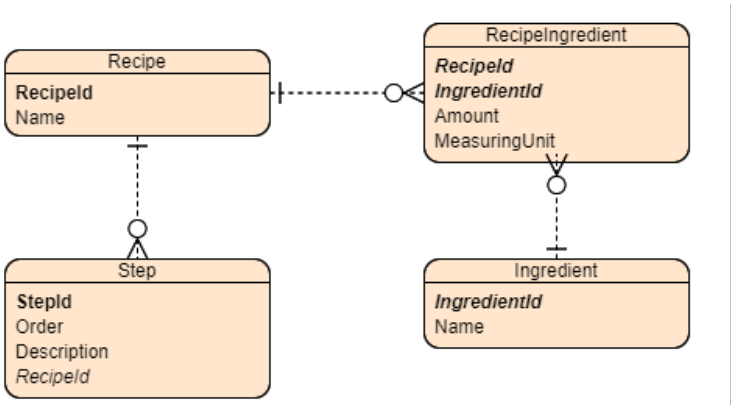
Prije no što krenemo na razvoj aplikacije, potrebno je definirati bazu podataka. Svakako nam je potreban model *Recipe* koji će opisivati recept. Budući da se recept sastoji od liste koraka i liste sastojaka, izdvojimo ih u posebne modele *Step* i *Ingredient*. Svaki sastojak se može naći u više recepata te svaki recept može sadržavati više sastojaka. Stoga dodajmo model *RecipeIngredient* koji će povezivati ova dva modela.

Opisanu bazu podataka s popisom atributa možemo vidjeti na ilustraciji 2.4. Od čitatelja očekujemo da sam razvije navedene modele, pomoću migracija kreira bazu te razvije funkciju *Seed* kojom popunjava praznu bazu. Rezultat kreiranja baze podataka možemo vidjeti na slici 2.5.

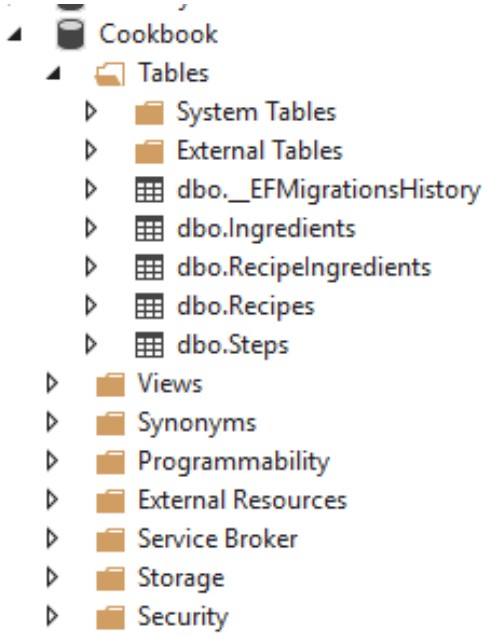
Nakon opisa baze krenimo s razvojem prve korisničke priče koja zahtjeva pregled naslova te prikaz recepta koji se nalaze u bazi.

2.3.1 Testiranje sadržaja web-aplikacije

Započinjemo s implementacijom prve korisničke priče, kojom ćemo pokazati kako provesti testiranje sadrži li web-aplikacija specificirani sadržaj.



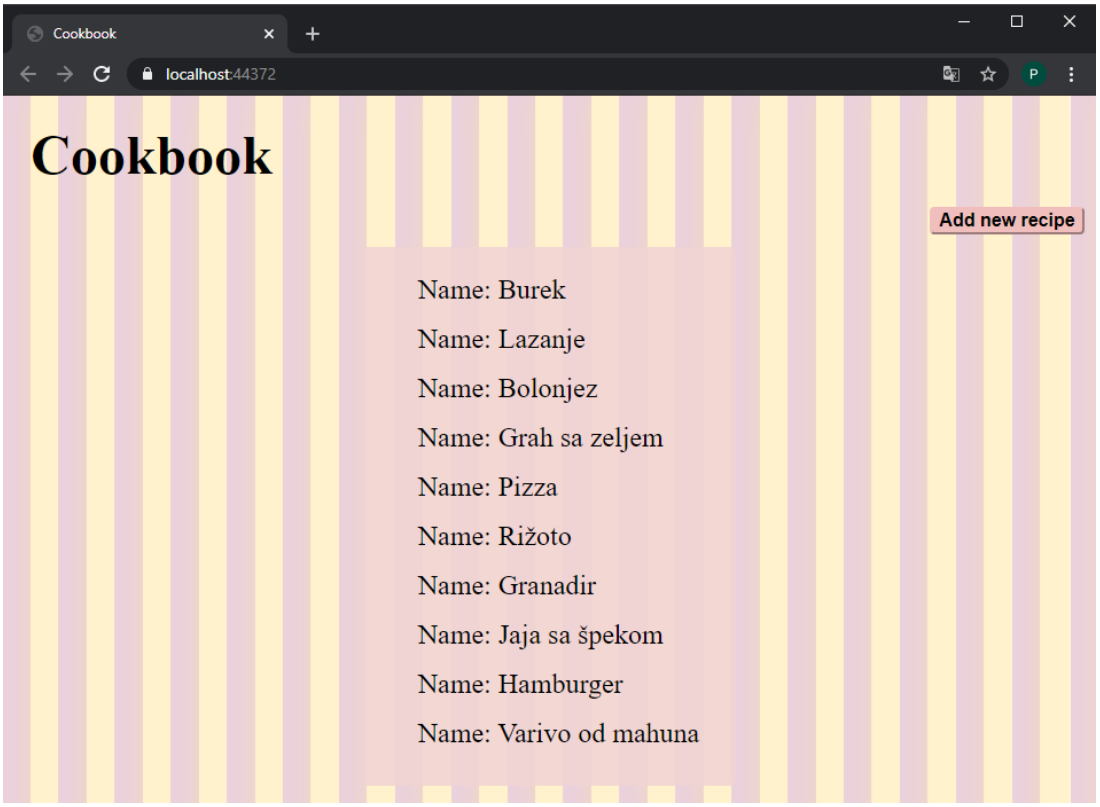
Slika 2.4: Shema baze podataka



Slika 2.5: Popis tablica unutar baze

US1: Ja kao korisnik aplikacije, na početnom ekranu želim vidjeti naslov koji glasi *Cookbook* te popis naziva recepata kako bih imao pregled koji se nude.

Opis: Na početnom ekranu aplikacije treba prikazati naslov *Cookbook* te popis naziva recepata koje povlačimo iz baze podataka. Implementacijom opisane funkcionalnosti, razvit ćemo korisničko sučelje kao što je prikazano na slici 2.6.



Slika 2.6: Početna stranica aplikacije

Prije nego li krenemo na razvoj korisničke priče, obrišimo dokument *UnitTest1.cs* te preimenujmo automatski generiranu klasu *Class1*, unutar Cookbook.HighLevelTests, u *HighLevelTest*. Sukladno opisanom procesu Test-Driven Development, započnimo razvoj funkcionalnosti pisanjem testa visoke razine.

Listing 2.5: Cookbook.HighLevelTests/HighLevelTest.cs

```
1 [Fact]
2 public void getContent()
3 {
4     _driver.Navigate().GoToUrl("http://localhost:58883/");
5
6     Assert.Contains("Cookbook", _driver.PageSource.ToString());
7     Assert.Equal("Cookbook", _driver.Title);
8 }
```

Prvi test visoke razine za ovu korisničku priču, sličan je testu 2.4. Jedina razlika je što sada na ekranu očekujemo naslov *Cookbook* koji provjeravamo u šestoj liniji testa. Nadalje, osim u naslovu stranice, isti možemo prikazati i u nazivu web-aplikacije koju obično definiramo u zaglavlju HTML dokumenta elementom <*title*>. Naziv web-aplikacije provjeravamo u sedmoj liniji kôda.

Pokrenemo li ovaj test, očekujemo da će pasti jer još nismo kreirali izmjene unutar same aplikacije. Nakon pokretanja testa uočimo poruku pogreške koju nam javlja Test Explorer.

```
Cookbook.HighLevelTest.getContent
   Line: 6
   Message: Assert.Contains() Failure
   Not found: Cookbook
   In value: <html><head></head><body><pre style="word-wrap: break-word;
white-space: pre-wrap;">Hello World!</pre></body></html>
```

Kako bismo ovaj test učinili prolaznim najjednostavnije je unutar klase *Startup* prilagoditi metodu *Configure* tako da, umjesto *Hello World!*, vraća sadržaj stranice jednak *Cookbook*. Naravno, tek smo započeli s razvojem ove funkcionalnosti. Već sljedeći test bi zahtijevao da nadopunimo HTML dokument u obliku stringa unutar iste metode. Stoga izdvojimo naslov stanice unutar dokumenta *Index.cshtml* čiji ćemo sadržaj postepeno nadopunjavati. Budući da koristimo strukturu MVC, početnu stranicu aplikacije (index) ćemo dodati unutar mape Views/Home te će nam biti potreban kontroler *HomeController* koji prosljeđuje kreirani pogled (eng. *view*).

Prije kreiranja kontrolera i pogleda, kreirajmo jedinični test koji će provjeravati vraća li kontroler odgovarajući tip podatka, a koji bi se trebao podudarati s tipom *ViewResult*:

Listing 2.6: Cookbook.LowerLevelTests/HomePageTest.cs

```
1 [Fact]
2 public void TestIndex()
3 {
4     var homeController = new HomeController();
5     var result = homeController.Index();
6
7     Assert.IsType<ViewResult>(result);
8 }
```

Kako bismo mogli izgraditi projekt potrebno je dodati referencu na projekt Cookbook te kreirati traženu klasu *HomeContoller*. Ponovno pozivajući se na arhitekturalni obrazac MVC, neka direktorij *Controllers* sadrži dokument *HomeContoller.cs* unutar kojeg je razvijen kontroler *HomeContoller* koji nasljeđuje klasu *Controller* te sadrži metodu *Index*. Rezultat ovih izmjena je kreirana klasa sadržaja:

Listing 2.7: Cookbook/Controllers/HomeController.cs

```
1 public class HomeController : Controller
2 {
3     public IActionResult Index()
4     {
5         return View();
6     }
7 }
```

Pokretanjem testa niže razine možemo uočiti kako je test prošao. U ovom trenutku struktura aplikacije izgleda ovako:

```
/
└── Solution'Cookbook'
    ├── Tests
    │   └── (...)
    └── Cookbook
        ├── (...)
        ├── Controllers
        │   └── HomeController.cs
        ├── Models
        │   ├── Ingredient.cs
        │   ├── Recipe.cs
        │   ├── RecipeIngredient.cs
        │   └── Step.cs
        ├── Views
        │   └── Home
        │       └── Index.cshtml
        └── (...)
```

Potrebno je i test više razine učiniti prolaznim. Unutar dokumenta *Views/Home/Index.cshtml* dodajmo sljedeći sadržaj:

Listing 2.8: Cookbook/Views/Home/Index.cshtml

```
1 <html>
2     <head>
3     </head>
4     <body>
5         Cookbook
6     </body>
7 </html>
```

Nakon dodanog sadržaja i pokretanja testova visoke razine, možemo uočiti da je prvo ispitivanje unutar testa prošlo. Sljedeća pogreška koja se javlja glasi:

```
Cookbook.HighLevelTest.getContent
    Message: Assert.Equal() Failure
    Expected: Cookbook
    Actual: localhost
```

Odnosno naslov web-aplikacije trenutno je jednak *localhost*, dok mi očekujemo naslov *Cookbook*. Dodajmo sljedeći programski isječak:

Listing 2.9: Cookbook/Views/Home/Index.cshtml

```
1 <html>
2     <head>
3         <title>Cookbook</title>
4     </head>
5 (...)
```

Nakon dodanog programskog isječka, test visoke razine (2.5) prolazi, tj. obje vrste testova prolaze. Stoga prijeđimo na sljedeći korak a to je ispis recepata na ekran. Ovaj korak možemo implementirati na elegantan način tako da nazive nekih konkretnih recepata popišemo (eng. *hardcode*) unutar dokumenta *Views/Home/Index.cshtml*. Navedeni način preporučuje se početnicima ili ako nemamo uvid u naredne funkcionalnosti koje treba razviti. Iz kratkog opisa aplikacije uviđamo da će jedna od funkcionalnosti biti čitanje recepata iz baze. Stoga, umjesto popisivanja naziva recepta, dohvatimo nazive recepata iz prethodno kreirane baze podataka.

Prema algoritmu za proces TDD, prvo krećemo s pisanjem testova visoke razine. Nadopunimo test tako da u HTML dokumentu očekujemo element kojemu je dodijeljen id *Recipes* te unutar njega očekujemo barem jedan recept kojemu je dodijeljena klasa *recipe*.

Listing 2.10: Cookbook.HighLevelTests/HighLevelTest.cs

```
1 (...)
2 var recipes = _driver.FindElement(By.Id("Recipes")).FindElements(By.
      ClassName("recipe"));
3 Assert.True(recipes.Count > 0, "No recipe is displayed on page");
```

Pokretanjem testa javlja se pogreška:

```
Cookbook.HighLevelTest.getContent
    Message: OpenQA.Selenium.NoSuchElementException : no such element: Unable to locate element: ”method”:”css selector”,”selector”:”Recipes”.
```

Odnosno na HTML stranici nije moguće pronaći identifikator *Recipes*. Kako bismo popravili ovu grešku, potrebno je unutar HTML dokumenta dodati:

Listing 2.11: Cookbook/Views/Home/Index.cshtml

```
(...)
    <body>
        Cookbook
        <div id="Recipes">

        </div>
    </body>
</html>
```

Uočimo, unutar elementa kojemu je dodijeljen id *Recipes* nismo dodali ni jedan element klase *recipe*, stoga nam se javlja sljedeća pogreška:

> Cookbook.HighLevelTest.getContent
> Message: No recipe is displayed on page
> Expected: True
> Actual: False

Prije no što razriješimo problem s ovom pogreškom, dodajmo test niže razine kojim želimo provjeriti prosljeđujemo li listu recepta u pogled *Views/Home/Index.cshtml*. Prosljeđivanje podataka može se vršiti pomoću *ViewBag*, vlastitog modela i drugih. *ViewBag* je dinamički tip čija se uporaba predlaže pri slanju malog skupa privremenih podataka. Svrha opisanog tipa odgovara našim potrebama, stoga proslijedimo listu recepata pomoću *ViewBag*-a.

Listing 2.12: Cookbook.LowerLevelTests/HomePageTest.cs

```
[Fact]
public void IndexReturnListOfRecipe()
{
    Mock<IRecipeRepository> recipeRepository = new Mock<
        IRecipeRepository>();
    recipeRepository.Setup(x => x.GetAllRecipe())
        .Returns(GetAllRecipe());
    HomeController homeController = new HomeController(
        recipeRepository.Object);
    var result = homeController.Index();
    var viewResult = Assert.IsType<ViewResult>(result);
    var recipes = Assert.IsAssignableFrom<List<Recipe>>(
        homeController.ViewBag.Recipes);
    Assert.True(recipes.Count > 0, "Recipes from database not
        retrieved");
```

```
12      Assert.Null(homeController.ViewBag.NoRecipe);
13  }
14
15  private IEnumerable GetAllRecipe()
16  {
17      return new List
18      {
19          new Recipe (...)
20      };
21  }
```

U četvrtoj i petoj liniji testa, možemo uočiti korištenje paketa Moq. Prvo kreiramo lažni objekt, odnosno lažni repozitorij recepata. Zatim kreiranom objektu definiramo njegovo ponašanje, to jest prilikom poziva funkcije *RecipeRepository.GetAllRecipe* objekt će vraćati vrijednost privatne funkcije *GetAllRecipe*. Za pokretanje prethodno dodanog testa potrebno je:

- Kreirati direktorij *Interface* te unutar njega sučelje *IRecipeRepository* koje sadrži deklaraciju metode *GetAllRecipe*.
- Kreirati direktorij *Repositories* te unutar njega klasu *RecipeRepository* koja implementira sučelje *IRecipeRepository*.
- Dodati konstruktor kontroleru *HomeController* koji će primati repozitorij recepata.

Listing 2.13: Cookbook/Interfaces/IRecipeRepository.cs

```
1      public interface IRecipeRepository
2      {
3          IEnumerable GetAllRecipe();
4      }
```

Listing 2.14: Cookbook/Repositories/RecipeRepository.cs

```
1  public class RecipeRepository : IRecipeRepository
2  {
3      public IEnumerable GetAllRecipe()
4      {
5          throw new NotImplementedException();
6      }
7  }
```

Listing 2.15: Cookbook/Controllers/HomeController.cs

```
1 private readonly IRecipeRepository _recipeRepository;
2
3 public HomeController(IRecipeRepository recipeRepository)
4 {
5     _recipeRepository = recipeRepository;
6 }
7
8 public IActionResult Index(){(...)}
```

Nakon kreiranih izmjena, pogledajmo strukturu aplikacije:

```
/
└── Solution'Cookbook'
    ├── Tests
    │   └── (...)
    └── Cookbook
        ├── Connected Services
        ├── Dependencies
        ├── Properties
        ├── Controllers
        │   └── HomeController.cs
        ├── Interfaces
        │   └── IRecipeRepository.cs
        ├── Models
        │   └── (...)
        ├── Repositories
        │   └── RecipeRepository.cs
        ├── Views
        │   └── (...)
        ├── appsettings.json
        ├── Program.cs
        └── Startup.cs
```

Prilagodbom aplikacije za pokretanje testa *HomePageTest.IndexReturnListOfRecipe* narušili smo test *HomePageTest.TestIndex* (2.6). Ako usporedimo testove možemo uočiti da novi test nadopunjuje stari test, stoga obrišimo test *HomePageTest.TestIndex*.

Sada možemo izgraditi projekt Cookbook.LowerLevelTests te ga pokrenuti. Prva pogreška koja nam se javlja glasi:

Cookbook.LowerLevelTests.HomePageTest.IndexReturnListOfRecipe
Message: Assert.IsAssignableFrom() Failure
Expected: typeof(System.Collections.Generic.List)
Actual: (null)

Ova pogreška je rezultat pristupanja elementu *Recipes* unutar *ViewBag*-a kojeg nismo inicijalizirali. Stoga prilagodimo metodu *Index* unutar kontrolera *HomeController*:

Listing 2.16: Cookbook/Controllers/HomeController.cs

```
1 public IActionResult Index()
2 {
3     ViewBag.Recipes = new List();
4     return View();
5 }
```

Nakon što smo kreirali test za ispitivanje šaljemo li listu recepata u pogled, kreirajmo test koji provjerava prosljeđujemo li poruku *No recipes to display* u pogled ako je lista recepata prazna.

Listing 2.17: Cookbook.LowerLevelTests/HomePageTest.cs

```
1 [Fact]
2 public void IndexReturnMessageToDisplay()
3 {
4     Mock recipeRepository = new Mock();
5     recipeRepository.Setup(x => x.GetAllRecipe())
6                     .Returns(new List());
7     HomeController homeController = new HomeController(
          recipeRepository.Object);
8     var result = homeController.Index();
9     var viewResult = Assert.IsType(result);
10    Assert.Null(homeController.ViewBag.Recipes);
11    Assert.Equal("No recipes to display", homeController.ViewBag.
          NoRecipe);
12 }
```

Pokretanjem ovoga testa dolazimo do pogreške:

Cookbook.LowerLevelTests.HomePageTest.IndexReturnMessageToDisplay
Line: 10
Message: Assert.Null() Failure
Expected: (null)
Actual: List []

Uvođenjem dodatnog uvjeta unutar metode *HomeController.Index* (2.18) lako možemo riješiti prethodnu pogrešku.

Listing 2.18: Cookbook/Controllers/HomeController.cs

```
1 public IActionResult Index()
2 {
3     var recipes = _recipeRepository.GetAllRecipe().ToList();
4     if (recipes.Any()) ViewBag.Recipes = recipes;
5
6     return View();
7 }
```

Sljedeća pogreška glasi:

> Cookbook.LowerLevelTests.HomePageTest.IndexReturnMessageToDisplay
> Line: 11
> Message: Assert.Equal() Failure
> Expected: No recipes to display
> Actual: (null)

Kako test ne bi javljao pogrešku, prilagodimo metodu pomoću sljedećeg isječka kôda.

Listing 2.19: Cookbook/Controllers/HomeController.cs

```
1 public IActionResult Index()
2 {
3     var recipes = _recipeRepository.GetAllRecipe().ToList();
4     if (recipes.Any()) ViewBag.Recipes = recipes;
5     else ViewBag.NoRecipe = "No recipes for display";
6
7     return View();
8 }
```

Nakon kreiranih izmjena testovi niže razine 2.17 i 2.12 prolaze.

Vratimo se na testove visoke razine. Kako bismo imali bolji uvid u pogrešku koju javlja test *HighLevelTest.getContent* pogledajmo što nam se ispisuje na ekranu:

> InvalidOperationException: Unable to resolve service for type 'Cookbook.Interfaces.IRecipeRepository' while attempting to activate 'Cookbook.Controllers.HomeController'.

Ova pogreška se javlja jer nismo konfigurirali uslugu unutar metode *Startup.ConfigureServices*. Odnosno potrebno je definirati da se zahtjevi na sučelja rješavaju pomoću instanci repozitorija. Pogrešku rješavamo dodavanjem treće linije kôda iz sljedećeg isječka:

Listing 2.20: Cookbook/Startup.cs

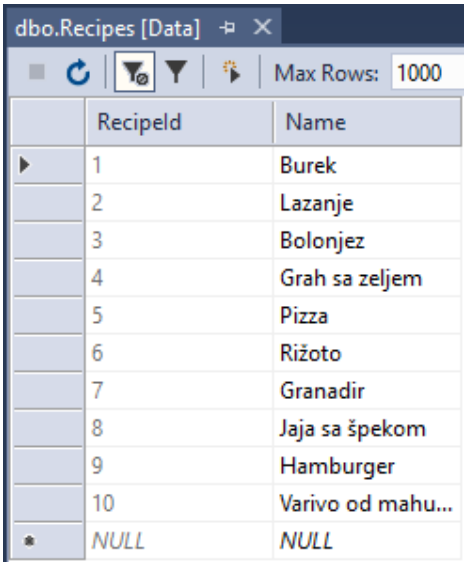
```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddTransient<IRecipeRepository, RecipeRepository>();
4
5     services.AddMvc();
6 }
```

Ponovnim pokretanjem aplikacije izbacuje se iznimka:

> System.NotImplementedException: 'The method or operation is not implemented.'

Ta iznimka je očekivana jer nismo implementirali metodu *RecipeRepository.GetAllRecipe* već smo Visual Studiu dopustili da to učini umjesto nas (prethodno kada smo pokretali test niže razine *HomePageTest.IndexReturnListOfRecipe*). Vrijeme je da ju razvijemo kako bi naši zahtjevi bili ispunjeni.

Kako bismo razvili ovu metodu, pretpostavimo da smo kreirali model i bazu te u bazu dodali nekoliko zapisa kao što su navedeni na slici 2.7.



Slika 2.7: Popunjena tablica Recipe

Kada imamo kreiranu bazu te u njoj nekoliko zapisa možemo razviti repozitorij *RecipeRepository*. Unutar njega potrebno je ostvariti komunikaciju s bazom što ćemo postići upotrebom tehnologije Entity Framework Core. Entity Framework Core (kraće EFCore)

verzija je popularne tehnologije Entity Framework koja objektno-relacijski (ORM) preslikava zapise iz baze u kolekciju objekata.

Listing 2.21: Cookbook/Repositories/RecipeRepository.cs

```
1  public class RecipeRepository : IRecipeRepository
2  {
3      private readonly AppDbContext _appDbContext;
4
5      public RecipeRepository(AppDbContext appDbContext)
6      {
7          _appDbContext = appDbContext;
8      }
9
10     public IEnumerable GetAllRecipe()
11     {
12         return _appDbContext.Recipes;
13     }
14 }
```

Upotrebu EFCore uočavamo u dvanaestoj liniji gdje iz baze podataka čita sve zapise iz tablice *Recipe*, te vraća kolekciju objekata tipa *Recipe*.

Kako bismo provjerili jesmo li uklonili pogrešku, pokrenimo test visoke razine *getContent*. Nakon pokretanja javlja nam se i dalje ista pogreška:

```
HighLevelTest.getContent
    Message: No recipe is displayed on page
    Expected: True
    Actual: False
```

Pogreška se i dalje javlja iz razloga što pogled ne ispisuje podatke, iako ih kontroler ispravno prosljeđuje putem *ViewBag*-a. Implementirajmo ispis podataka na sljedeći način:

Listing 2.22: Cookbook/Views/Home/Index.cshtml

```
1  (...)
2      <div id="Recipes">
3          @{
4              foreach (var recipe in ViewBag.Recipes)
5              {
6                  <div class="recipe">
7                      Name: @recipe.Name
8                  </div>
9              }
10         }
11     </div>
```

```
12 (...)
```

Ponovnim pokretanjem testova uočavamo da oni prolaze iz čega zaključujemo da je funkcionalnost razvijena. Međutim, propustili smo provjeriti je li repozitorij recepata na ispravan način povukao zapise iz baze. Kako bi provjerili prethodno rečeno potrebno je kreirati integracijski test čime se bavimo u idućoj cjelini.

2.3.2 Testiranje dohvaćanja podataka iz baze

Rijetke su aplikacije koje ne zahtijevaju komunikaciju s vanjskim sustavom. Kako bismo provjerili izvršava li se komunikacija na očekivani način, služimo se integracijskim testovima. Pretpostavimo da smo definirali novu bazu podataka unutar klase *IntegrationTestBase* čija je struktura jednaka strukturi produkcijske baze. Novo kreirana baza podataka stvara se prilikom poziva integracijskih testova te uništava po završetku integracijskih testova.

Unutar klase *IntegrationTests* razvijmo test kojim provjeravamo dohvaćamo li metodom *RecipeRepository.GetAllRecipe* sve recepte iz baze podataka, to jest komunicira li naša aplikacija na očekivan način s bazom podataka.

Listing 2.23: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1  public class IntegrationTests: IntegrationTestBase
2  {
3      [Fact]
4      public void RetrivesAllDataFromDB()
5      {
6          var context = GivenAppContext();
7          var ingredientRepo = new IngredientRepository(context);
8          var recipeRepo = new RecipeRepository(context, ingredientRepo
               );
9
10         Recipe recipe1 = new Recipe
11         {
12             Name = "RetrivesAllDataFromDB1",
13             (...)
14         };
15         Recipe recipe2 = new Recipe
16         {
17             Name = "RetrivesAllDataFromDB2",
18             (...)
19         };
20
21         List<Recipe> recordsBeforeAdding = Assert.IsType<List<Recipe
               >>(recipeRepo.GetAllRecipe().ToList());
22
```

```
23          context.Recipes.Add(recipe1);
24          context.Recipes.Add(recipe2);
25          context.SaveChanges();
26
27          List<Recipe> recordsAfterAdding = Assert.IsType<List<Recipe
                >>(recipeRepo.GetAllRecipe().ToList());
28          Assert.Equal(2 + recordsBeforeAdding.Count(),
                recordsAfterAdding.Count());
29
30          Recipe record = null;
31          foreach(var x in recordsAfterAdding)
32          {
33              if (x.Name == recipe1.Name)
34              {
35                  record = x;
36                  break;
37              }
38          }
39          Assert.NotNull(record);
40          Assert.Equal(recipe1.Name, record.Name);
41      }
42  }
```

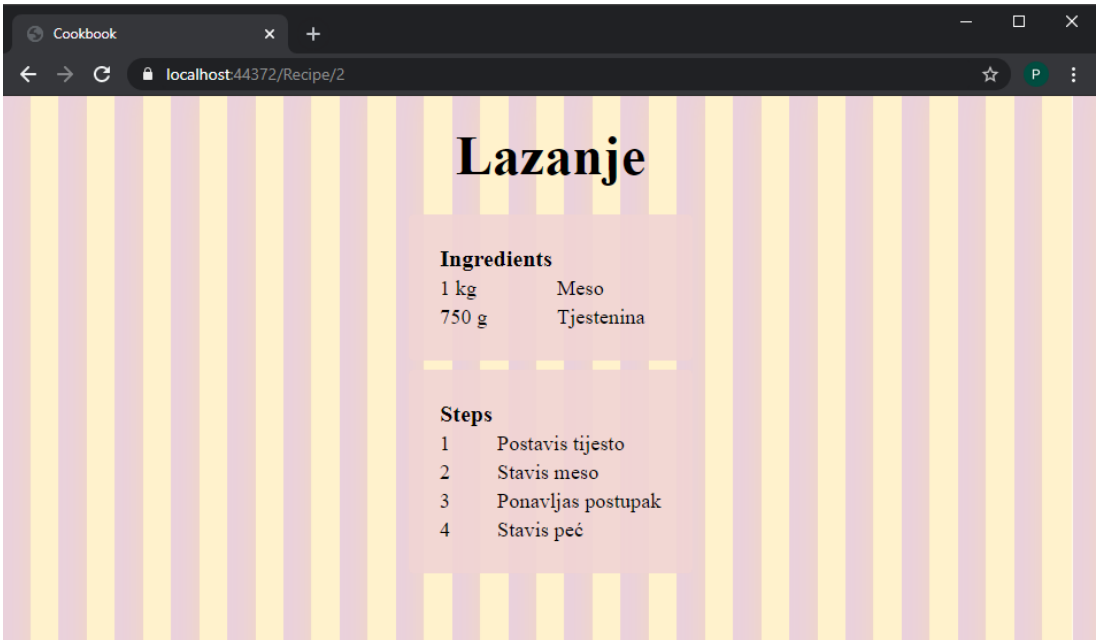
Budući da nam prethodni test ne javlja pogrešku i svi testovi prolaze, možemo zaključiti da programski kôd funkcionira prema zahtjevima koje smo definirali na početku. Iako bi refaktorizacija bila sljedeća akcija u procesu TDD, naš programski kôd je još uvijek dovoljno jednostavan te zasad ne vidimo mogućnost unaprjeđenja njegove strukture. Stoga prelazimo na implementaciju nove funkcionalnosti koju možemo birati između:

- klikom na recept otvara se stranica s detaljima o istom receptu;
- omogućavanje dodavanja novog recepta.

Jedan od zahtjeva druge funkcionalnost glasi da nakon potvrde za dodavanje novoga recepta se prikazuje pogled s detaljima o istom. Stoga, kako bismo izbjegli velike izmjene unutar testova te produkcijskog kôda, započnimo s prikazom detalja recepta.

US2: Ja kao korisnik aplikacije, klikom na pojedini recept, želim vidjeti detalje recepta kako bih imao bolji uvid u pojedini recept.

Opis: Kada na početnoj stranici aplikacije kliknemo na naslov recepta, potrebno je prikazati novu stranicu na kojoj se nalaze detalji o receptu. Odnosno iz baze podataka je potrebno povući te na novoj stranici prikazati: naslov recepta, naslov sekcije *Ingredients* unutar koje se nalazi popis sastojaka (naziv sastojka, količina i mjerna jedinica) i naslov sekcije *Steps* unutar koje se nalazi popis koraka (redni broj i opis koraka). Nakon razvoja funkcionalnosti, nova stranica će izgledati kao što je prikazano na slici 2.8.



Slika 2.8: Stranica na kojoj su prikazani detalji recepta

Prema opisu procesa TDD, prvo je potrebno razviti testove visoke razine. Unutar testa potrebno je definirati sljedeći niz akcija:

- otvoriti web-preglednik na adresi http://localhost:58883/;
- provjeriti da je svaki recept link na novu stranicu;
- kliknuti na recept;
- provjeriti da je naslov web-aplikacije jednak *Cookbook*;
- provjeriti prikazuje li se naslova recepta na koji smo kliknuli;
- provjeriti prikazuju li se naslovi sekcija *Ingredients* i *Steps*.

Prethodno opisani niz akcija i očekivane vrijednosti predstavljaju jedan od slučajeva. Sljedeće što možemo pomoću testova visoke razine provjeriti jest slučaj gdje korisnik želi pristupiti receptu s id-em koji ne postoji u bazi. Na takav zahtjev ne očekujemo prethodne ispise na ekranu, već očekujemo samo ispis poruke koja govori da nije moguće prikazati recept.

Analogno testovima visoke razine 2.5 i 2.10, možemo razviti prethodna dva opisana testa. Također, ispravljajući redom pogreške koje nam se pojavljuju, jednostavno možemo

razviti funkcionalnost koju opisuju testovi. Stoga se nećemo fokusirati na razvoj ovih testa visoke razine već se posvetimo razvoju testova niže razine potrebnih za implementaciju ove funkcionalnosti.

Izvršavamo li redom opisane akcije unutar prethodno opisanog testa visoke razine, nakon klika na recept, otvara nam se stranica s greškom (HTTP error 404) s porukom *'This localhost page can't be found'*. Očito ne postoji kontroler koji odgovara na HTTP zahtjev nastao nakon klika na recept. Stoga započnimo s fazom pisanja testova niže razine.

Testirajmo kontroler koji na *GET* zahtjev vraća pogled s detaljima recepta čiji je id proslijeđen kao parametar. Kako ovim testom pokrivamo novi kontroler *RecipeController*, preporučuje se kreiranje dodatne klase *RecipePageTest* unutar koje ćemo razvijati testove kojima provjeravamo navedeni kontroler.

Listing 2.24: Cookbook.LowerLevelTests/RecipePageTest.cs

```
1  [Fact]
2  public void ReturnRecipeDetails()
3  {
4      Mock<IRecipeRepository> recipeRepository = new Mock<
           IRecipeRepository>();
5
6      int recipeId = 1;
7      string recipeName = "Burek";
8      List<IngredientDTO> ingredients = new List<IngredientDTO> { (...)
           };
9      List<StepDTO> steps = new List<StepDTO> { (...) };
10
11     recipeRepository.Setup(x => x.GetRecipeById(recipeId))
12                     .Returns(new RecipeDetailDTO
13                     {
14                         RecipeId = recipeId,
15                         Name = recipeName,
16                         (...)
17                     });
18
19     RecipeController recipeController = new RecipeController(
           recipeRepository.Object);
20     var result = recipeController.Index(recipeId);
21     var viewResult = Assert.IsType<ViewResult>(result);
22
23     Assert.IsType<RecipeDetailDTO>(recipeController.ViewBag.Recipe);
24     Assert.IsType<string>(recipeController.ViewBag.Recipe.Name);
25     Assert.IsType<int>(recipeController.ViewBag.Recipe.RecipeId);
26     (...)
27
28     Assert.Equal(recipeName, recipeController.ViewBag.Recipe.Name);
29     Assert.Equal(recipeId, recipeController.ViewBag.Recipe.RecipeId);
```

```
30      (...)
31      Assert.Null(recipeController.ViewBag.NoRecipe);
32  }
```

Primijetimo, kao povratnu vrijednost metode *RecipeRepository.GetRecipeById(recipeId)* očekujemo *Data Transfer Objects* (kraće DTOs). DTO objekti služe kako bismo prosljeđivali samo podatke relevantne za određeni pogled ili kontroler. Kreirajmo klasu *RecipeDetailDTO* koja će sadržavati sve potrebne podatke recepta, a to su: naziv, id, lista sastojaka, lista koraka. Neka se lista sastojaka sastoji od objekata klase *IngredientDTO* koja sadrži naziv, iznos i mjernu jedinicu; dok se lista koraka sastoji od objekata klase *StepDTO* koja opisuje svaki korak te sadrži redni broj i opis. Razvoj ovih klasa prepuštamo čitatelju.

Prije pokretanja prethodnog testa potrebno je implementirati metode koje nedostaju. Unutar *IRecipeRepositorya* nedostaje deklaracija metode *GetRecipeById(recipeId)* što dalje zahtijeva implementaciju iste metode unutar *RecipeRepository*. Dopustit ćemo Visual Studiu da umjesto nas deklarira metodu unutar sučelja te napravi inicijalnu implementaciju metode, što rezultira sljedećim isječcima kodova:

Listing 2.25: Cookbook/Interfaces/IRecipeRepository.cs

```
1 public interface IRecipeRepository
2 {
3     IEnumerable GetAllRecipe();
4     RecipeDetailDTO  GetRecipeById(int recipeId);
5 }
```

Listing 2.26: Cookbook/Repositories/RecipeRepository.cs

```
1 public RecipeDetailDTO GetRecipeById(int recipeId)
2 {
3     throw new NotImplementedException();
4 }
```

Nadalje, analogno kontroleru *HomeController* (2.15) možemo razviti kontroler *RecipeController*. Jedina razlika je u parametru kojeg očekuje metoda *RecipeController.Index(recipeId)*. Dodatno, potrebno je definirati rutu (eng. *route*) koja vodi do metode *RecipeController.Index(recipeId)*. Definiciju rute možemo provjeriti sljedećim testom:

Listing 2.27: Cookbook.LowerLevelTests/RecipePageTest.cs

```
1 [Fact]
2 public void TestRouteOnIndexMethod()
3 {
4     var method = typeof(RecipeController).GetMethod("Index");
5     var attribute = method.GetCustomAttributes(typeof(RouteAttribute)
          , false)
```

```
6                       .Cast<RouteAttribute>()
7                       .SingleOrDefault();
8       Assert.NotNull(attribute);
9       Assert.Contains("Recipe/{recipeId}", attribute.Template);
10  }
```

U četvrtoj liniji prethodnog testa, dohvaćamo određenu metodu klase specifičnog tipa. Odnosno dohvaćamo metodu *Index* unutar repozitorija recepata. Zatim dohvaćamo atribute koje smo dodijelili toj metodi. S obzirom na to da testiramo rutu, dohvatimo samo atribute koji ju opisuju te u konačnici provjerimo odgovaraju li očekivanoj ruti.

Slično razvoju jediničnog testa 2.12 te pregledom pogrešaka koje se javljaju možemo efikasno razviti minimalni kôd koji razvijene jedinične testove čini prolaznima.

Osim opisanog scenarija, potrebno je kontroleru definirati ponašanje kada kao parametar zaprimi id jednak nuli ili negativnom broju ili id koji trenutno ne postoji u bazi. Ukoliko dođe do takvog zahtjeva neka kontroler vrati poruku o nemogućnosti prikaza recepta s navedenim id-om. Dodajmo test niže razine *ReturnErrorMessage* koji testira takve scenarije.

Kako ne bismo kreirali test za svaki scenarij zasebno, kreirajmo *[Theory]*. Kao što smo već napomenuli, pomoću naredbe *[Theory]* naznačujemo da je sljedeća metoda testna. Tako naznačenoj testnoj metodi potrebno je proslijediti parametre unutar atributa *[InlineData(...)]* te za svaki atribut kreira se instanca testa. U ovom testu kao parametar prosljeđujemo id-jeve recepata.

Listing 2.28: Cookbook.LowerLevelTests/RecipePageTest.cs

```
1  [Theory]
2  [InlineData(0)]
3  [InlineData(-1)]
4  [InlineData(100)]
5  public void ReturnErrorMessage(int requestedId)
6  {
7      Mock<IRecipeRepository> recipeRepository = new Mock<
           IRecipeRepository>();
8
9      RecipeController recipeController = new RecipeController(
           recipeRepository.Object);
10
11     var result = recipeController.Index(requestedId);
12     var viewResult = Assert.IsType<ViewResult>(result);
13
14     Assert.Null(recipeController.ViewBag.Recipe);
15     Assert.Equal("The requested recipe cannot be displayed",
           recipeController.ViewBag.NoRecipe);
16 }
```

Pokrenemo li test *ReturnErrorMessage* (2.28), on će generirati tri zasebna testa. Sva tri testa javljaju iste pogreške. Razumijevanjem poruka o pogreškama lako možemo razviti traženu funkcionalnost kako bi testovi prošli. Iteracijom između razumijevanja poruka o pogreškama i dodavanja minimalnog kôda, dolazimo do razvoja kontrolera *RecipeController* koji glasi:

Listing 2.29: Cookbook/Controllers/RecipeController.cs

```
1  public class RecipeController : Controller
2  {
3      private readonly IRecipeRepository _recipeRepository;
4
5      public RecipeController(IRecipeRepository recipeRepository)
6      {
7          _recipeRepository = recipeRepository;
8      }
9
10     [Route("Recipe/{recipeId}")]
11     public IActionResult Index(int recipeId)
12     {
13         var recipe = _recipeRepository.GetRecipeById(recipeId);
14         if (recipe != null) ViewBag.Recipe = recipe;
15         else ViewBag.NoRecipe = "The requested recipe cannot be
               displayed";
16
17         return View();
18     }
19 }
```

Prisjetimo se, prethodno smo dopustili Visual Studiu da umjesto nas implementira metodu *RecipeRepository.GetRecipeById(recipeId)* na način da izbacuje iznimku *NotImplementedException*. Prilagodimo ju potrebama naše funkcionalnosti, ali u skladu s metodologijom TDD prvo je potrebno napisati integracijske testove. Unutar prvog testa kreiramo recept te ga dodajemo u bazu, a u konačnici ga dohvaćamo metodom *GetRecipeById(recipeId)* te provjeravamo jesu li svi atributi dohvaćeni na ispravan način. U idućem testu provjeravamo povratnu vrijednost iste metode dok ju pozivamo s parametrom koji nije validan.

Listing 2.30: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1 [Fact]
2 public void RetrivesRecipeById()
3 {
4     var context = GivenAppContext();
5     var ingredientRepo = new IngredientRepository(context);
6     var recipeRepo = new RecipeRepository(context, ingredientRepo);
```

```
7
8       Recipe recipe = new Recipe
9       {
10          Name = "RetrivesRecipeById",
11          RecipeIngredients = new List<RecipeIngredient> { (...) },
12          Steps = new List<Step> { (...) }
13      };
14
15      context.Recipes.Add(recipe);
16      context.SaveChanges();
17
18      string recipeName = "RetrivesRecipeById";
19      List<IngredientDTO> ingredients= new List<IngredientDTO> {(...)};
20      List<StepDTO> steps = new List<StepDTO>  { (...) };
21
22      RecipeDetailDTO recipeDetail = Assert.IsType<RecipeDetailDTO>(
            recipeRepo.GetRecipeById(recipe.RecipeId));
23      Assert.Equal(recipeName, recipe.Name);
24  }
```

Listing 2.31: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1  [Theory]
2  [InlineData(-1)]
3  [InlineData(0)]
4  [InlineData(9999)]
5  public void GetRecipeByIdReturnNull(int requestedId)
6  {
7      var context = GivenAppContext();
8      var ingredientRepo = new IngredientRepository(context);
9      var recipeRepo = new RecipeRepository(context, ingredientRepo);
10
11     Assert.Null(recipeRepo.GetRecipeById(requestedId));
12 }
```

Kako bismo prethodne dvije metode učinili prolaznima razvijmo metodu *GetRecipeById(recipeId)* na sljedeći način. Prvo dohvatimo zapis iz tablice Recipes čiji je id jednak traženom a zatim preuzmemo zapise koji se vežu na dohvaćeni zapis. Tako dohvaćeni recept je tipa *Recipe* dok je povratna vrijednost naše metode tipa *RecipeDetailDTO*. Time je potrebno još dobiveni zapis pretvoriti u tip *RecipeDetailDTO* što prepuštamo čitatelju. Dio implementacije opisane metode možemo vidjeti u sljedećem isječku.

Listing 2.32: Cookbook/Repositories/RecipeRepository.cs

```
1  public RecipeDetailDTO GetRecipeById(int recipeId)
2  {
```

```
3       Recipe recipe= _appDbContext.Recipes
4                           .Where(r => r.RecipeId == recipeId)
5                           .Include(r => r.RecipeIngredients)
6                               .ThenInclude(r => r.Ingredient)
7                           .Include(r => r.Steps)
8                           .FirstOrDefault();
9
10      if (recipe == null) return null;
11
12      RecipeDetailDTO recipeDetailDTO = new recipeDetailDTO();
13      //convert type Recipe to type RecipeDetailDTO
14
15      return recipeDetailDTO;
16  }
```

U konačnici, HTML dokument kojeg zahtijeva metoda *RecipeController.Index(recipeId)* glasi:

Listing 2.33: Cookbook/Views/Recipe/Index.cshtml

```
1  <html>
2      <head>
3          <title>Cookbook</title>
4      </head>
5      <body>
6          @if (ViewBag.Recipe != null)
7          {
8              <h1>@ViewBag.Recipe.Name</h1>
9              <h3>Ingredients</h3>
10             <div>
11                 @foreach(var ingredient in
                       ViewBag.Recipe.Ingredients)
12                 {
13                     <div>
14                         <span>@ingredient.Amount
                               @ingredient.MeasuringUnit</span><span>
                               @ingredient.Name</span>
15                     </div>
16                 }
17             </div>
18             <h3>Steps</h3>
19             <div>
20                 @foreach(var step in ViewBag.Recipe.Steps)
21                 { (...) }
22             </div>
23         }
24         <h3>@ViewBag.NoRecipe</h3>
25     </body>
```

```
26 </html>
```

Nakon niza izmjena, testovi visoke razine i testovi niže razine prolaze. Sljedeći korak je razviti funkcionalnost koja definira mogućnost dodavanja novog recepta.

2.3.3 Testiranje korisničkog sučelja

Današnje želje korisnika nad korisničkim sučeljem aplikacija uvelike zahtijevaju razvoj rješenja pomoću JavaScript-a (kraće JS-a) koji daje bezbrojne mogućnosti. U dosadašnjem razvoju naše aplikacije, mogućnosti JavaScript-a smo koristili samo prilikom definiranja akcije klika na naziv recepta. Stoga posvetimo se unaprjeđenju korisničkog sučelja. S obzirom na to da ćemo ga koristiti za pisanje skripti koje definiraju ponašanje sučelja, testirat ćemo ga pomoću testova visoke razine.

US3: Ja kao korisnik aplikacije, želim dodavati nove recepte u kuharicu kako bi uvijek bila ažurna.

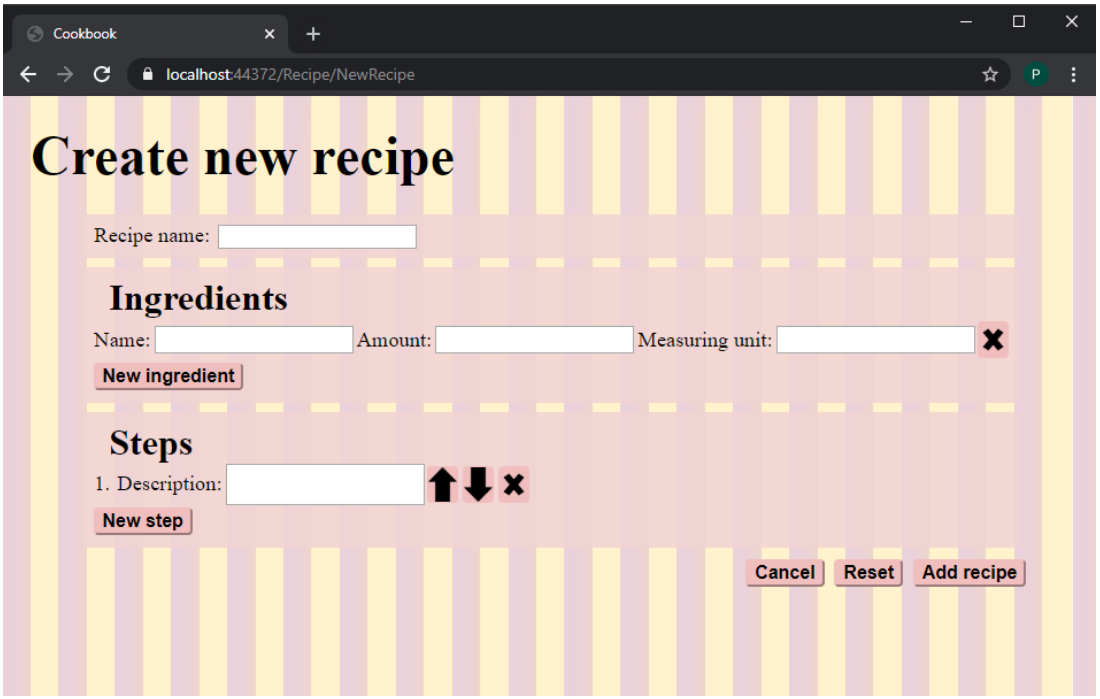
Opis: Klikom na gumb *Add recipe* otvara se stranica koja omogućuje dodavanje novoga recepta. Podaci koje je moguće unijeti su naziv recepta, naziv sastojka, mjerna jedinica i količina sastojka te niz koraka. Osim dodavanja navedenih vrijednosti, klikom na određeni gumb može se dodati novi sastojak te novi korak unutar stranice. Također, moguće je svaki korak i sastojak obrisati. Dodatno, kod liste koraka moguće je i mijenjati redoslijed koraka. Rezultat razvoja ove funkcionalnosti prikazan je na slici 2.9.

Prije kreiranja pogleda *NewRecipe*, potrebno je na početnoj stranici aplikacije dodati poveznicu (link) koji vodi na novi pogled. Stoga kreirajmo test unutar kojeg ćemo simulacijom interakcije korisnika provjeriti opisanu funkcionalnost. Test implementira sljedeće akcije:

- otvorimo početnu stranicu aplikacije, odnosno otvorimo web-preglednik na adresi *http://localhost:58883/*;
- pronađimo link (HTML element <*a*>) koji vodi na adresu */Recipe/NewRecipe* te kliknimo na njega;
- provjerimo sadrži li novootvorena stranica naslov aplikacije *Cookbook* te *Create new recipe* u HTML kôdu.

Listing 2.34: Cookbook.HighLevelTests/HighLevelTest.cs

```
1 [Fact]
2 public void DisplayNewRecipeForm()
3 {
4     _driver.Navigate().GoToUrl("http://localhost:58883/");
```



Slika 2.9: Stranica na kojoj je moguće dodati novi recept

```
5       _driver.FindElement(By.XPath("//a[@href='/Recipe/NewRecipe']")).
            Click();
6
7       Assert.Equal("Cookbook", _driver.Title);
8       Assert.Contains("Create new recipe", _driver.PageSource);
9   }
```

Pokrenemo li ovaj test, javlja se pogreška:

> HighLevelTest.DisplayNewRecipeForm
> Message: OpenQA.Selenium.NoSuchElementException : no such element: Unable to locate element: ”method”:”xpath”,”selector”:”//a[@href=’/Recipe/NewRecipe’]”

Ova pogreška je očekivana jer takav link trenutno ne postoji na stranici. Pomoću tagova iz paketa Microsoft.AspNetCore.Mvc.TagHelpers definirajmo link kojeg test zahtijeva. Tagom *asp-controller* određujemo u koji kontroler šaljemo HTTP zahtjev, dok tagom *asp-action* određujemo koja metoda će obraditi taj zahtjev. Sljedećim isječkom kôda ispravljamo prethodnu pogrešku.

Listing 2.35: Cookbook/Views/Home/Index.cshtml

```
1 @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
2 (...)
3  <a asp-controller="Recipe" asp-action="NewRecipe" class="
       navbar-brand">Add new recipe</a>
```

Nakon dodavanja odgovarajućeg linka i klika na njega, očekujemo pogrešku koja javlja kako nije pronađena stranica na web-adresi *https://localhost:44372/Recipe/NewRecipe*. No izbacuje se sljedeća pogreška:

> HighLevelTest.DisplayNewRecipeForm
> Message: Assert.Contains() Failure
> Not found: Create new recipe
> In value:<html><head> <title>Cookbook</title></head><body> <h3>The requested recipe cannot be displayed</h3></body></html>

HTTP zahtjev, koji smo kreirali klikom na link, proslijeđen je odgovarajućem kontroleru *RecipeControler* ali metoda koja ga obrađuje jest *Index(recipeId)* te je parametar *recipeId* jednak *NewRecipe*. Naravno, takav id recepta ne postoji te shodno tome nam vraća pogled s odgovarajućom porukom. Kako dalje ne bi dolazilo do zabune, ažurirajmo rutu metode *Index(recipeId)* tako da glasi *[Route("Recipe/recipeId:int")]* te pripadajući test *TestRouteOnIndexMethod*.

Nakon kreiranih izmjena te ponavljanja akcija koje su uzrokovale prethodnu pogrešku, javlja nam se pogreška koju smo očekivali:

> This localhost page can't be found. No webpage was found for the web-address: https://localhost:44372/Recipe/NewRecipe

Ono što nam nedostaje je metoda *RecipeController.NewRecipe* koja će obrađivati HTTP zahtjev kreiran klikom na link *Add new recipe*. Analogno već razvijenim metodama kontrolera možemo razviti navedenu metodu. Osim razvoja kontrolera, prema prethodnim primjerima, možemo zaključiti da će biti potrebno kreirati novi pogled */Views/Recipe/NewRecipe.cshtml*. Za početak unutar novog pogleda dodajmo naslov web-aplikacije te naslov stranice *Create new recipe*. Razvoj prethodno opisane metode i novog pogleda prepuštamo čitatelju.

Sada kada imamo kreirani pogled i odgovarajuću akciju unutar kontrolera, krenimo s daljnjim razvojem istog pogleda. S obzirom na to koje je funkcionalnosti potrebno razviti unutar ove korisničke priče, kreirajmo popis mogućih testova:

- utvrditi osnovne elemente (polja za unos naziva recepta, sastojaka i koraka recepta) unutar pogleda - *FindBasicElementsInNewRecipeForm*

- što se treba dogoditi nakon klika na gumb za
  - dodavanje novoga sastojka - *AddNewIngredient*
  - dodavanje novoga koraka - *AddNewStep*
  - brisanje određenog sastojka - *DeleteIngredient*
  - brisanje određenog koraka - *DeleteStep*
  - pomicanje određenog koraka gore - *MoveUpStep*
  - pomicanje određenog koraka dolje - *MoveDownStep*
  - odustajanje od dodavanja novoga recepta - *CancelForm*
  - ponovno popunjavanje vrijednosti novoga recepta - *ResetForm*
  - spremanje novoga recepta - *SaveNewRecipe*

No nemojmo se zadržavati na osmišljavanju testova jer se tijekom razvoja aplikacije možemo sjetiti te ažurirati listu testova kojima ćemo unaprijediti funkcionalnost. Stoga krenimo s razvojem testova.

Utvrđivanje osnovnih elemenata unutar pogleda

Kao što je navedeno u kratkom opisu korisničke priče, testom *FindBasicElementsInNewRecipeForm* provjeravamo nalazi li se na ekranu:

- oznaka i mjesto za unos naslova recepta,
- oznake i mjesta za unos imena sastojka, količine te mjerne jedinice,
- oznaka i mjesto za unos opisa prvog koraka.

Kasnije, kada dodatno razvijemo aplikaciju, bit će potrebno prilagoditi ovaj test. Odnosno dodatno ćemo na ekranu očekivati:

- gumb za dinamičko kreiranje oznaka i mjesta za unos novog sastojka,
- gumb za brisanje pojedinog sastojka,
- analogno, gumbe za dodavanje i brisanje koraka,
- gumbe za promjenu redoslijeda koraka.

Prethodno smo koristili *ViewBag* za prosljeđivanje vrijednosti iz kontrolera u pogled. Međutim, obrnuto nije moguće. Prosljeđivanje vrijednosti s pogleda u kontroler vršit ćemo pomoću modela koji ćemo sami kreirati. Dovoljan nam je već prethodno kreirani model *RecipeDetailDTO*, ali kako ne bismo narušili strukturu aplikacije kreirajmo dokument *ViewModels/RecipeViewModel.cs* koji sadrži model *RecipeDetailDTO*. Navedeni model *RecipeViewModel* potrebno je uključiti u pogled *Views/Recipe/NewRecipe.cshtml*.

Prethodno opisani test možemo lako razviti korak po korak koristeći ranije opisane tehnike. Radi kratkoće, prikazat ćemo samo konačni rezultat pogleda *Recipe/NewRecipe.cshtml*.

Listing 2.36: Cookbook/Views/Recipe/NewRecipe.cshtml

```
1  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
2  @using Cookbook.ViewModels
3  @model RecipeViewModel
4  <html>
5  <head>
6      <title>Cookbook</title>
7  </head>
8  <body>
9      <h1>Create new recipe</h1>
10     <label for="Recipe_Name">Recipe name:</label>
11     <input asp-for="Recipe.Name" />
12     <label for="Ingredients">Ingredients</label>
13     <div id="Ingredients">
14         <label asp-for="Recipe.Ingredients[0].Name">Name:</label>
15         <input asp-for="Recipe.Ingredients[0].Name" />
16         (...)
17     </div>
18     <label for="Steps">Steps</label>
19     <div id="Steps">
20         <label>1.</label>
21         <label asp-for="Recipe.Steps[0].Description">Description:</
               label>
22         <input asp-for="Recipe.Steps[0].Description" />
23     </div>
24 </body>
25 </html>
```

Dinamičko kreiranje inputa za novi sastojak i korak

Sljedeću funkcionalnost koju razvijamo unutar pogleda *Recipe/NewRecipe.cshtml* jest dinamičko dodavanje potrebnih oznaka i mjesta za unos novog sastojka.

Lista sastojaka okružena je tagom * * čiji je identifikator jednak *Ingredients*. Kako bi testiranje bilo jednostavnije, svaki sastojak uokvirimo tagom * * te mu dodijelimo klasu *ingredient*. To će nam omogućiti bolji pregled sastojaka, vizualno ljepši prikaz, jednostavnije testove te u konačnici lakši razvoj idućih funkcionalnosti.

Testom *AddNewIngredient* provjeravamo prethodno opisanu funkcionalnost te unutar njega razvijmo sljedeće naredbe:

- otvorimo web-preglednik na adresi *http://localhost:58883/Recipe/NewRecipe*,
- pronađimo gumb opisan id-em *addInputsForIngredient* te ga kliknimo,
- provjerimo nalaze li se na ekranu osnovni elementi te dodane oznake i mjesta za unos novog sastojka.

Prepuštamo čitatelju da sam razvije opisani test. Krenemo li s ispravljanjem pogrešaka unutar istog, prva koja se javlja govori o tome kako na ekranu ne postoji gumb čiji je id *addInputsForIngredient*. Nakon dodavanja gumba u pogled, potrebno je pomoću jQuery-a povezati gumb s funkcijom koju treba pozivati. Prema predlošku prvog sastojka, unutar funkcije kreiramo tag * *. Unutar kreiranog taga * * dodamo potrebne elemente te u konačnici kreirani tag pridodamo postojećem tagu * * čiji je id *Ingredients*. Opisanim postupkom test treba biti prolazan.

Pogledamo li prethodno opisani test *AddNewIngredient* uočavamo veliku sličnost s testom *FindBasicElementsInNewRecipeForm*. Razlika je u tome što ovoga puta, nakon klika na odgovarajući gumb očekujemo dva sastojka umjesto jednog. Analogno, dodavanju inputa za novi sastojak, možemo razviti test *AddNewStep* te funkcionalnost dodavanja inputa za novi korak.

Trenutno imamo tri testa (*FindBasicElementsInNewRecipeForm*, *AddNewIngredient* i *AddNewStep*) koja su izuzetno slična što znači da se desetak linija kôda ponavlja. Nakon što nam svi testovi prolaze te vodeći se načelom TDD procesa koji kaže „Nemoj se ponavljati.”, refaktorirajmo naš kôd. Posljedice refaktoriranja jesu:

- podjela klase *Cookbook.HighLevelTests.HighLevelTest* na nekoliko manjih (*Cookbook.HighLevelTests/HomePageTest.cs*, *Cookbook.HighLevelTests/NewRecipeTest.cs* i *Cookbook.HighLevelTests/RecipePageTest.cs*),
- unutar nove klase *Cookbook.HighLevelTests.NewRecipeTest* izdvojili smo blokove naredbi koji su slični između prethodna tri testa te ih implementirali kao zasebne pomoćne funkcije.

Brisanje unosa pojedinog sastojka i koraka

Nakon dodavanja mogućnosti unosa dodatnog sastojka i koraka, logični nastavak je razvoj funkcionalnosti za brisanje sastojka te koraka iz recepta. Krenimo s razvojem testa *DeleteStep* kojim utvrđujemo ispravno brisanje unosa za korak.

Radi lakšeg testiranja, razvijmo test pod pretpostavkom: svakom tagu * * čija je klasa *step* dodijeljen je id koraka jednak rednom broju koraka te je svakom koraku dodijeljen gumb za brisanje. Funkcija *deleteStep* prema id-u koraka, koja se poziva klikom na gumb za brisanje, će znati koji korak je potrebno obrisati. Bitno je testirati svaki slučaj, stoga ćemo testirati sljedeća tri scenarija:

- korisnik dodaje dva koraka te briše prvi korak;
- korisnik dodaje dva koraka te briše srednji korak;
- korisnik dodaje dva koraka te briše zadnji korak.

S obzirom na pretpostavku potrebno je prilagoditi testove *FindBasicElementsInNewRecipeForm*, *AddNewIngredient* i *AddNewStep*.

Ponovno, nećemo razvijati test *DeleteStep* već ćemo prepustiti čitatelju da to učini samostalno. Akcije koje je potrebno izvesti unutar testa su:

- dva puta kliknemo na gumb za dodavanje novog koraka,
- kreiramo listu stringova unutar kojih se nalaze opisi koraka te ih proslijedimo u odgovarajuće HTML elemente tipa input,
- obrišemo jedan od koraka simulacijom klika na odgovarajući gumb,
- provjerimo da je sada jedan manje korak prikazan na ekranu,
- provjerimo odgovara li redoslijed opisa koraka onome što očekujemo.

Promotrimo detaljnije neke od pogrešaka koja će nam se javiti prilikom implementacije funkcionalnosti brisanja koraka. Jedna od njih glasi:

```
HighLevelTest.DeleteStep
    Line: Assert.Equal(2, steps.Count);
    Message: Assert.Equal() Failure
    Expected: 2
    Actual: 3
```

Drugim riječima, nakon klika na gumb koji je zadužen za brisanje koraka, i dalje su prikazani svi koraci. Kako bismo ispravili ovu pogrešku, potrebno je definirati funkciju na sljedeći način:

Listing 2.37: Cookbook/Views/Recipe/NewRecipe.cshtml

```
(...)
<script>
    (...)
    function deleteStep() {
        $("#Steps").children().last().remove();
    }
</script>
```

Pokretanjem testa nailazimo na novu pogrešku, što bi značilo da smo prethodnu pogrešku uspješno riješili. Nova pogreška nam govori kako opisi koraka nisu odgovarajući. Takvu pogrešku mogli smo očekivati jer brišemo zadnji korak neovisno o tome na koji je kliknuto. Za unaprjeđenje trenutne funkcionalnosti potrebno je odrediti na koji se korak kliknulo te, ukoliko je potrebno, sve opise koraka pomaknuti za jedan prema gore. Opisani postupak možemo izvesti na sljedeći način:

Listing 2.38: Cookbook/Views/Recipe/NewRecipe.cshtml

```
(...)
<script>
    (...)
    function deleteStep() {
        var stepId = event.srcElement.parentElement.id;

        for (var k = parseInt(stepId); k < $("#Steps").children().
            length - 1; k++) {
            kNext = k + 1;
            $("#Recipe_Steps_" + k + "__Description").val($("#
                Recipe_Steps_" + kNext + "__Description").val());
        }

        $("#Steps").children().last().remove();
    }
</script>
```

Nakon prethodno opisanog niza iteracija test *DeleteStep* u sva tri testna slučaja prolazi. No moguć je i četvrti slučaj kada korisnik odluči obrisati sve unose za opis koraka. Znamo da recept bez opisa postupka izrade nije potpun. Stoga dodajmo novi test *DeleteLastStep* koji neće dopustiti brisanje unosa za korak ukoliko je on jedini, nego će isprazniti pripadni HTML element tipa input. Razvoj testa i funkcionalnosti prepuštamo čitatelju. Osim testa *DeleteLastStep*, čitatelju prepuštamo i razvoj funkcionalnosti brisanja sastojka.

Sljedeći korak je razviti testove *MoveUpStep* i *MoveDownStep* koji opisuju promjenu redoslijeda unesenih vrijednosti u HTML elemente tipa input koji služe za opis koraka. Tijekom razvoja ova dva testa možemo si postaviti pitanje što se dogodi ukoliko zadnji

opis koraka pokušamo pomaknuti za jedno mjesto dolje, ili ako prvi opis koraka pokušamo pomaknuti za jedno mjesto gore. Kao odgovor na to pitanje potrebno je razviti dva dodatna testa. S obzirom na to da smo ranije prošli kroz nekoliko razvoja testova te pripadajućih funkcionalnosti, lako možemo reproducirati niz iteracija pogrešaka i minimalnog kôda te razviti funkcionalnost aplikacije.

Dodavanje osnovnih funkcionalnosti unutar forme

Osnovne funkcionalnosti svake forme jesu mogućnost odustajanja, ponovnog postavljanja te potvrde forme, odnosno Cancel, Reset i Submit. Funkcionalnost Submit zahtijeva i promijene unutar kontrolera stoga ju ostavimo za posebnu cjelinu.

Klikom na odustajanje od popunjavanja forme potrebno je korisnika vratiti na početnu stranicu aplikacije. Test za ovu funkcionalnost je vrlo jednostavan kao i minimalni kôd koji je potrebno nadodati. Stoga prepuštamo čitatelju razvoj te krenimo odmah na sljedeću funkcionalnost.

Sljedeće što je potrebno razviti je ponovno postavljanje forme. Klikom na gumb *Reset* očekujemo da će se prikazati forma za dodavanje novog sastojka jednaka početnom stanju forme. Ponovno, ovaj test je jednostavno razviti te ga jednom linijom kôda možemo učiniti prolaznim, pa prepuštamo čitatelju razvoj.

2.3.4 Testiranje potvrde dodavanja novog recepta

Sljedeća veća funkcionalnost koju je potrebno implementirati jest potvrda unosa forme. Ono što se treba dogoditi nakon klika na dodavanje recepta jest prosljeđivanje podataka u kontroler, validacija istih, spremanje recepta te prosljeđivanje na pogled na kojem se nalaze detalji dodanog recepta.

Razvijmo test visoke razine, *SaveNewRecipe*, tako da definira sljedeće akcije:

- otvoriti pogled *NewRecipe*,
- dodati naziv recepta,
- dodati nekoliko sastojka i koraka, što ćemo učiniti pomoćnu dodatnih funkcija SetIngredients i SetSteps,
- provjeriti odgovara li definicija forme očekivanim vrijednostima,
- kliknuti na gumb za potvrdu forme,
- provjeriti prikazuje li se sada ekran s detaljima o prethodno dodanom receptu,
- provjeriti odgovaraju li svi podaci onima koje smo dodali, slično testu visoke razine kojim smo provjeravali ispis na stranici s detaljima o receptu.

Dio prethodno opisanog testa glasi:

Listing 2.39: Cookbook.HighLevelTests/NewRecipeTest.cs

```
1  [Fact]
2  public void SaveNewRecipe()
3  {
4      _driver.Navigate().GoToUrl("http://localhost:58883/Recipe/
           NewRecipe");
5
6      _driver.FindElement(By.XPath("//input[@id=(//label[contains(text
           (), 'Recipe name')]/@for)]"))
7          .SendKeys("Cufte u paradajiz sosu");
8      var ingredients = SetIngredients();
9      var steps = SetSteps();
10
11     var form = _driver.FindElement(By.TagName("form"));
12     Assert.Equal("post", form.GetAttribute("method"));
13     Assert.Matches(@".*\/Recipe\/AddRecipe", form.GetAttribute("
           action"));
14
15     _driver.FindElement(By.XPath("//input[@type='submit' and @value =
            'Add recipe']")).Click();
16
17     Assert.Matches(@"http:\/\/localhost:58883\/Recipe\/[1-9][0-9]*$",
            _driver.Url);
18
19     (...)
20 }
```

Od šeste do devete linije pripremamo podatke koji su potrebni za izvršavanja testa, odnosno pomoću funkcije *IWebElement.SendKeys()* u odgovarajuće HTML elemente dodajemo vrijednosti. Stoga između tih naredbi ne očekujemo pogreške. Kako bismo pronašli HTML tag <*form*> te njemu dodijeljene atribute, odnosno sljedeće tri naredbe učinili prolaznima, potrebno je dodati navedeni HTML tag unutar pogleda *Recipe/NewRecipe.cshtml* na sljedeći način:

Listing 2.40: Cookbook/Views/Recipe/NewRecipe.cshtml

```
1  <html>
2  <head>
3      <title>Cookbook</title>
4  </head>
5  <body>
6      <h1>Create new recipe</h1>
7      <form asp-action="AddRecipe" method="post">
8      (...)
```

```
9       </form>
10  </body>
11  </html>
```

Sljedeće što test zahtijeva jest HTML element *<input>* koji je tipa *submit*. Ovu pogrešku jednostavno riješimo dodavanjem *<input type="submit" value="Add recipe" />* unutar HTML taga *<form>* u pogledu. Nakon navedenih izmjena stigli smo do sljedeće pogreške koju javlja sedamnaesta linija testa:

NewRecipeTest.SaveNewRecipe
Message: Assert.Matches() Failure
Regex: http:localhost:58883Recipe[1-9][0-9]*$
Value: http://localhost:58883/Recipe/AddRecipe

Navedena pogreška uzrokovana je nepostojanjem metode unutar kontrolera koja obrađuje kreirani HTML zahtjev. Kreirani zahtjev traži metodu *AddRecipe*, unutar kontrolera *RecipeController*, koja prima parametar *RecipeViewModel*. Unutar nove metode potrebno je validirati podatke (time se nećemo baviti u ovoj cjelini), dodati recept u bazu te preusmjeriti korisnika na prikaz novostvorenog recepta, odnosno na akciju *RecipeRepository.Index(recipeId)*. Prije razvoja kontrolera dodajmo test niže razine, *AddRecipeShould*, koji će provjeriti opisanu metodu.

Listing 2.41: Cookbook.LowerLevelTests/RecipePageTest.cs

```
1  [Fact]
2  public void AddRecipeShould()
3  {
4      Mock<IRecipeRepository> recipeRepository = new Mock<
           IRecipeRepository>();
5      RecipeController recipeController = new RecipeController(
           recipeRepository.Object);
6
7      RecipeDetailDTO recipe = new RecipeDetailDTO
8      {
9          Name = "Test recipe",
10         Ingredients = new List<IngredientDTO> { (...) },
11         Steps = new List<StepDTO> { (...) }
12     };
13     RecipeViewModel recipeViewModel = new RecipeViewModel { Recipe =
           recipe };
14
15     var result = recipeController.AddRecipe(recipeViewModel);
16     var redirectToActionResult = Assert.IsType<RedirectToActionResult
           >(result);
17
```

```
18      Assert.Equal("Index", redirectToActionResult.ActionName);
19      Assert.Equal("Recipe", redirectToActionResult.ControllerName);
20      Assert.Contains("recipeId", redirectToActionResult.RouteValues.
            Keys);
21  }
```

Kako bismo mogli pokrenuti ove testove, potrebno je dodati kontroler *AddRecipe* što ćemo prepustiti Visual Studiu da učini umjesto nas.

Listing 2.42: Cookbook/Controllers/RecipeController.cs

```
1 public IActionResult AddRecipe(RecipeViewModel recipeViewModel)
2 {
3     throw new NotImplementedException();
4 }
```

Već znamo, pokrenemo li ovaj test, izbacit će nam iznimku: *The method or operation is not implemented*. Stoga razvijmo metodu tako da u bazu doda recept, zaprimljen unutar modela *RecipeViewModel*, naredbom *_recipeRepository.AddRecipe(recipe)*. Do sada smo strogo odvajali logiku unutar aplikacije te ćemo se nastaviti voditi time. Neka funkcija *RecipeRepository.AddRecipe(recipe)* zaprima parametar tipa *Recipe*. S obzirom na to da, unutar *RecipeViewModel*-a, dobivamo tip *RecipeDetailDTO* potrebno je tip *RecipeDetailDTO* pretvoriti u tip *Recipe*. To ćemo učiniti pomoću metode *ConvertRecipeDetailDTOToRecipe* čiji razvoj prepuštamo čitatelju. Kada kreiramo pomoćnu metodu možemo nadopuniti *RecipeController.AddRecipe(recipeViewModel)*:

Listing 2.43: Cookbook/Controllers/RecipeController.cs

```
1 public IActionResult AddRecipe(RecipeViewModel recipeViewModel)
2 {
3     Recipe recipe = ConvertRecipeDetailDTOToRecipe(recipeViewModel.
          Recipe);
4     _recipeRepository.AddRecipe(recipe);
5
6     return null;
7 }
```

Nadalje, nailazimo na pogrešku koja govori da unutar sučelja nemamo deklariranu funkciju *AddRecipe*. Ponovno, dopustimo Visual Studiu da je generira unutar sučelja te doda inicijalnu implementaciju u repozitorij recepata:

Listing 2.44: Cookbook/Interfaces/IRecipeRepository.cs

```
1 public interface IRecipeRepository
2 {
3     IEnumerable<Recipe> GetAllRecipe();
```

```
4     RecipeDetailDTO GetRecipeById(int recipeId);
5     void AddRecipe(Recipe recipe);
6 }
```

Listing 2.45: Cookbook/Repositories/RecipeRepository.cs

```
1 public void AddRecipe(Recipe recipe)
2 {
3     throw new NotImplementedException();
4 }
```

Prije nego li prethodnu metodu prilagodimo našim potrebama bit će potrebno razviti integracijske testove. No pogledajmo dokle smo stigli s testom *AddRecipeShould*:

```
LowerLevelTests.AddRecipeShould
   Message: Assert.IsType() Failure
   Expected: Microsoft.AspNetCore.Mvc.RedirectToActionResult
   Actual: (null)
```

Odnosno osamnaesta linija testa *AddRecipeShould* javlja nam kako je, umjesto prosljeđivanja akcije, naša povratna vrijednost jednaka *null*. Da bismo ispunili naredne tri linije toga testa, potrebno je nadograditi metodu *RecipeController.AddRecipe(recipeViewModel)*:

Listing 2.46: Cookbook/Controllers/RecipeController.cs

```
1 public IActionResult AddRecipe(RecipeViewModel recipeViewModel)
2 {
3     Recipe recipe = ConvertRecipeDetailDTOToRecipe(recipeViewModel.
          Recipe);
4     _recipeRepository.AddRecipe(recipe);
5
6     return RedirectToAction("Index", "Recipe", new { recipeId =
          recipe.RecipeId });
7 }
```

Budući da ova metoda samo zaprima i sprema podatke u bazu, dodajmo joj *POST* atribut koji testiramo pomoću testa niže razine *AddRecipeOnlyPOSTareAllowed*.

Listing 2.47: Cookbook.LowerLevelTests/RecipePageTest.cs

```
1 [Fact]
2 public void AddRecipeOnlyPOSTareAllowed()
3 {
4     var method = typeof(RecipeController).GetMethod("AddRecipe");
5     var attribute = method.GetCustomAttributes(typeof(
          HttpPostAttribute), false)
```

```
6                       .Cast<HttpPostAttribute>()
7                       .SingleOrDefault();
8       Assert.NotNull(attribute);
9       Assert.Contains("POST", attribute.HttpMethods);
10  }
```

Kako bismo prethodni test učinili prolaznim, potrebno je dodati traženi atribut:

Listing 2.48: Cookbook/Controllers/RecipeController.cs

```
1  [HttpPost]
2  public IActionResult AddRecipe(RecipeViewModel recipeViewModel)
3  {
4      (...)
5  }
```

Sada kad smo razvili potrebne dijelove, pokrenimo test visoke razine *SaveNewRecipe*. Izbacuje nam se iznimka koja govori da metoda *RecipeRepository.AddRecipe(recipe)* nije implementirana. Vrijeme je da ju implementiramo, ali prije razmislimo kako ćemo razviti dodavanje recepta.

Radi izbjegavanja dupliciranja naziva sastojaka, kreirali smo odvojenu tablicu *Ingredients* unutar baze podataka. Stoga prije dodavanja recepta u bazu provjerimo podudaraju li se koji nazivi sastojaka s već postojećim. Ako u bazi postoji sastojak s određenim imenom, tada umjesto kreiranja novog sastojka dodajmo postojeći na recept. Kako bismo izveli ovu funkcionalnost, bit će nam potrebna metoda koja dohvaća sva imena sastojka te metoda koja prema imenu sastojka vraća model *Ingredient*. Ove metode implementirat ćemo unutar *IngredientRepository*-a. Dok imamo na umu sve prethodno izrečeno, kreirajmo integracijski test *AddRecipe* te *ItIsNotPossibleToCreateIngredientsWithTheSameName*.

Listing 2.49: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1  [Fact]
2  public void AddRecipe()
3  {
4      var context = GivenAppContext();
5      var ingredientRepo = new IngredientRepository(context);
6      var recipeRepo = new RecipeRepository(context, ingredientRepo);
7
8      var recordsBeforeAdding = recipeRepo.GetAllRecipe().Count();
9
10     Recipe recipe = new Recipe
11     {
12         Name = "IntegrationTest recipe",
13         RecipeIngredients = new List<RecipeIngredient> { (...) },
14         Steps = new List<Step> { (...) }
```

```
15      };
16
17      recipeRepo.AddRecipe(recipe);
18
19      var x = recipeRepo.GetAllRecipe();
20      Assert.Equal(1 + recordsBeforeAdding, x.Count());
21
22      Assert.NotEqual(0, recipe.RecipeId);
23
24      var recipeDetailDTO = recipeRepo.GetRecipeById(recipe.RecipeId);
25      Assert.Equal(recipe.Name, recipeDetailDTO.Name);
26  }
```

Listing 2.50: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1  [Fact]
2  public void ItIsNotPossibleToCreateIngredientsWithTheSameName()
3  {
4      var context = GivenAppContext();
5      var ingredientRepo = new IngredientRepository(context);
6      var recipeRepo = new RecipeRepository(context, ingredientRepo);
7
8      var recordsBeforeAdding = ingredientRepo.GetAllIngredientName().
           Count();
9
10     Recipe recipe1 = new Recipe
11     {
12         Name = "IntegrationTest recipe1",
13         RecipeIngredients = new List<RecipeIngredient>
14         {
15             new RecipeIngredient{ Ingredient = new Ingredient{ Name =
                   "TheSameName1"}, Amount = 1, MeasuringUnit = "kg"},
16             new RecipeIngredient{ Ingredient = new Ingredient{ Name =
                   "TheSameName2"}, Amount = 2, MeasuringUnit = "dag"}
17         },
18
19         Steps = new List<Step> {(...)}
20     };
21     Recipe recipe2 = new Recipe
22     {
23         Name = "IntegrationTest recipe2",
24         RecipeIngredients = new List<RecipeIngredient>
25         {
26             new RecipeIngredient{ Ingredient = new Ingredient{ Name =
                   "TheSameName2"}, Amount = 3, MeasuringUnit = "g"},
27         },
28
```

```
29          Steps = new List<Step> {(...)}
30      };
31
32      recipeRepo.AddRecipe(recipe1);
33      recipeRepo.AddRecipe(recipe2);
34
35      var ingredients = ingredientRepo.GetAllIngredientName();
36      Assert.NotEmpty(ingredients);
37      Assert.Equal(2 + recordsBeforeAdding, ingredients.Count());
38  }
```

Kako bismo mogli pokrenuti prethodne testove niže razine potrebno je kreirati sučelje *IIngredientRepository* i odgovarajući repozitorij te ga kao parametar dodati konstruktoru repozitorija *RecipeRepository*. Nakon dodavanja parametra, javljaju se pogreške unutar ostalih integracijskih testova koje činimo prolaznim dodavanjem prvih tri naredbi prethodnog testa niže razine.

Pokrenemo li prethodne integracijske testove javlja se iznimka *System.NotImplementedException*, odnosno metoda *RecipeRepository.AddRecipe(recipe)* nije implementirana. Implementirajmo metodu na sljedeći način:

Listing 2.51: Cookbook/Repositories/RecipeRepository.cs

```
1 public void AddRecipe(Recipe recipe)
2 {
3     recipe.RecipeIngredients = SetRecipeIngredients(recipe.
          RecipeIngredients.ToList());
4     _appDbContext.Recipes.Add(recipe);
5     _appDbContext.SaveChanges();
6 }
```

Kako smo prethodno napomenuli, metoda *SetRecipeIngredients(List<RecipeIngredient>)* provjerava postoje li imena sastojaka unutar baze. Kako bismo mogli izgraditi aplikaciju, potrebno je unutar sučelja *IIngredientRepository* deklarirati metode: *GetAllIngredientName*, *GetIngredientByName*. Za početak, prepustimo generiranje tih metoda Visual Studiu.

Već prva pogreška nakon pokretanja integracijskog testa *ItIsNotPossibleToCreateIngredientsWithTheSameName*, javlja kako je potrebno razviti metodu *GetAllIngredientName()*. U nastavku provedbe TDD procesa javit će se ista pogreška sa zahtjevom za razvoj metode *GetIngredientByName(name)*. Prije implementacije metoda kreirajmo integracijski test koji provjerava funkcionalnost obiju metoda, odnosno funkcionalnost repozitorija sastojaka:

Listing 2.52: Cookbook.LowerLevelTests/IntegrationTests.cs

```
1 [Fact]
```

```
2  public void IngredientRepositoryShould()
3  {
4      var context = GivenAppContext();
5      var ingredientRepo = new IngredientRepository(context);
6
7      var recordsBeforeAdding = Assert.IsType<List<String>>(
           ingredientRepo.GetAllIngredientName().ToList()).Count();
8
9      List<string> listOfNames = new List<string>
10     {
11         "IngredientRepositoryShould1",
12         "IngredientRepositoryShould2",
13         "IngredientRepositoryShould3",
14         "IngredientRepositoryShould4",
15         "IngredientRepositoryShould5"
16     };
17
18     foreach (var x in listOfNames)
19     {
20         context.Ingredients.Add(new Ingredient { Name = x });
21     }
22     context.SaveChanges();
23
24     var recordsAfterAdding = Assert.IsType<List<String>>(
           ingredientRepo.GetAllIngredientName().ToList());
25     foreach(var x in listOfNames)
26     {
27         Assert.Contains(x, recordsAfterAdding);
28     }
29     Assert.Equal(listOfNames.Count() + recordsBeforeAdding,
           recordsAfterAdding.Count());
30
31     var ingredient = Assert.IsType<Ingredient>(ingredientRepo.
           GetIngredientByName(listOfNames[2]));
32     Assert.Equal(listOfNames[2], ingredient.Name);
33     Assert.NotEqual(0, ingredient.IngredientId);
34 }
```

Na osnovi prethodno kreiranog testa niže razine, razvijmo metode *GetAllIngredient-Name* i *GetIngredientByName(name)*.

Listing 2.53: Cookbook/Repositories/IngredientRepository.cs

```
1  public IEnumerable<string> GetAllIngredientName()
2  {
3      return _appDbContext.Ingredients
4                          .Select(i => i.Name);
5  }
```

Listing 2.54: Cookbook/Repositories/IngredientRepository.cs

```
1 public Ingredient GetIngredientByName(string name)
2 {
3     return _appDbContext.Ingredients
4                         .Where(i => i.Name == name)
5                         .First();
6 }
```

Nakon implementacije metoda, integracijski testovi *AddRecipe*, *IngredientRepositoryShould* i *ItIsNotPossibleToCreateIngredientsWithTheSameName* prolaze.

Sada kada prolaze novo dodani integracijski testovi i svi testovi niže razine, vratimo se na testove visoke razine. Pokretanjem testova visoke razine nalazimo na pogrešku:

> InvalidOperationException: Unable to resolve service for type 'Cookbook.Interfaces.IIngredientRepository' while attempting to activate 'Cookbook.Repositories.RecipeRepository'.

Prethodna pogreška nam je dobro poznata. Potrebno je ažurirati klasu *Startup* dodavanjem tranzicije između sučelja *IIngredientRepository* i repozitorija *IngredientRepository*.

Na kraju kada smo funkcionalnosti pokrili testovima, te oni prolaze, vrijeme je za refaktorizaciju.

2.3.5 Refaktorizacija

Nakon razvoja svih testova, unutar klase *Cookbook.HighLevelTests.NewRecipeTest* imamo 21 testova kojima se prvih nekoliko naredbi ponavlja. Da ne bismo skretali pozornost s razloga testiranja, predlažemo izdvajanje naredbi u konstruktor klase.

Dodatno, dobra praksa je izdvojiti JavaScript iz pogleda. Kreirajmo direktorij *wwwroot* unutar kojega će se nalaziti svi statički dokumenti kao što su js, css, img. Rezultat izdvajanja JS-a iz pogleda, *Home/Index.cshtml* i *Recipe/NewRecipe.cshtml*, su dokumenti: *HomeIndex.js* i *NewRecipe.js*. Osim navedenog, unutar svih pogleda određeni dio sadržaja se ponavlja. Kako bi svaki pogled ostao fokusiran na svoju svrhu kreirajmo dokument koji će biti baza HTML stranice te unutar kojega ćemo generirati poglede prema zahtjevu.

2.4 Daljnji razvoj aplikacije

U ovo poglavlju prikazali smo kako pomoću testova razviti web-aplikaciju. Pomoću testova visoke razine provjerili smo sadržaj korisničkog sučelja te rezultat interakcije korisnika sa sučeljem. S druge strane, testovima niže razine provjeravali smo svaku akciju

kontrolera, javne metode repozitorija te komunikaciju naše web-aplikacije s bazom podataka. Odnosno u ovom poglavlju prikazali smo kako pomoću metodologije TDD razviti sve važnije aspekte koji se pojavljuju prilikom izrade web-aplikacije.

Razvili smo nekoliko funkcionalnosti web-aplikacije, no ova aplikacija se može dalje razvijati. Nekoliko ideja za razvoj dodatnih funkcionalnosti:

- vezati dodani recept na korisnika, odnosno dodati *Identity*;
- omogućiti ažuriranje te brisanje recepta;
- omogućiti pretraživanje recepta prema naslovu i/ili sastojcima;

Naravno, ova lista nije konačna. Sada kada smo, bazirajući se na procesu TDD, iskusili razvoj nekih funkcionalnosti web-aplikacije možemo prepustiti čitatelju daljnji razvoj.

Bibliografija

[1] *ASP.NET Core documentation*, `https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2`, preuzeto: siječanj, 2020.

[2] *Selenium documentation*, `https://selenium.dev/documentation/en/`, preuzeto: siječanj, 2020.

[3] *xUnit documentation*, `https://xunit.net/`, preuzeto: siječanj, 2020.

[4] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-WesleyProfessional, 2018.

[5] Robert Manger, *Softversko inženjerstvo, 1. izdanje*, Element d.o.o., 2016.

[6] Harry Percival, *Test-Driven Development with Python, 2nd Edition*, O'Reilly Media, 2017.

Sažetak

U ovom diplomskom radu obrađen je razvoj web-aplikacije baziran na pisanju testova više i niže razine. Testovima više razine provjeravamo web-aplikaciju s aspekta korisnika te smo u tu svrhu koristili funkcionalne i End-to-end testove. S druge strane, za testiranje aplikacije s aspekta programera koristili smo jedinične i integracijske testove, odnosno testove niže razine. Kako bi web-aplikacija bila u cijelosti pokrivena testovima preporučuje se korištenje oba tipa testova. Iteracije između pisanja testova više razine, testova niže razine, pisanja minimalnog programskog kôda i refaktorizacije dobro su definirane unutar procesa Test-Driven Development (kraće proces TDD). Više o testovima i procesu TDD možemo pročitati u prvom dijelu rada.

U drugom dijelu rada, razvijali smo jednostavnu web-aplikaciju primjenjujući načela i metodologiju izvršavanja procesa TDD. Web-aplikaciju razvijali smo unutar Visual Studia te slijedili arhitekturalni obrazac MVC (eng. *Model-View-Controller*). Tijekom razvoja web-aplikacije slijedili smo niz korisničkih priča koje smo prije opisali. Svaku korisničku priču podijelili smo na manje dijelove te za njih razvijali testove više razine, potom testove niže razine. Napravljen je pregled svih testova kojima smo provjeravali funkcionalnosti, dok je samo dio njih u potpunosti razvijen.

Summary

In this master thesis, the development of web-applications is based on high-level and lower-level tests. With high-level tests, we test the web-applications from the user's aspect using Functional and End-to-end tests. On the other hand, for testing from the developers aspect Unit and Integration tests were used, which are lower-level tests. For web-applications to be thoroughly tested, it is recommended to use both test types. Test-Driven Development (TDD) has a well defined iteration process for writing tests between high-level tests, lower-level tests, write minimal code and code refactoring. More about the tests and the process of TDD can be read in the first part of this thesis.

In the second part of the thesis, a simple web-application is developed applying the principles and methodology of the TDD process. The simple application was developed following the architectural pattern of Model-View-Controller and pre-written user stories, built up in the development environment Visual Studio. Each user story was divided into smaller parts, where we developed a high level test and afterwards a lower-level test for each individual part. An overview of all tests was made, where some were fully developed, while others were just described.

Životopis

Petra Rožić rođena je 12. studenog 1995. godine u Zagrebu. Tijekom osnovnoškolskog obrazovanja, u Velikoj Gorici, razvija ljubav prema matematici. Nakon završetka osnovne škole u Velikoj Gorici 2010. godine, upisuje Prirodoslovno-matematičku gimnaziju Velika Gorica gdje stječe osnovno znanje o programiranju. U želji da postane izvrstan nastavnik upisuje Preddiplomski sveučilišni studij Matematika, smjer nastavnički, na Prirodoslovno-matematičkom fakultetu u Zagrebu. Tijekom studija pobliže se upoznaje s čarima programerskog svijeta te se okreće razvoju sebe kao programera. Titulu sveučilišne prvostupnice, stekla je 2017. godine te iste godine upisala Diplomski sveučilišni studij Računarstva i matematike na istome fakultetu.