

Проект по предметот Софтверски квалитет и тестирање на тема:

**Unit, Integration и End to End тестирање
на
Asp.NET Core апликација со Onion архитектура и микросервиси**

<https://github.com/AngelaMadjar/AngelaMadjar-IntegratedSystems-TicketShop>

<https://github.com/AngelaMadjar/TicketShopAdminApplication>

Изработиле:

Ангела Маџар, 181010

Петар Поповски, 181007

Содржина

1. Опис на тестираната веб апликација	3
2. Опис и тестирање на Onion архитектура	3
3. Тестирање.....	4
3.1. Unit Testing	4
xUnit	
Moq	
3.2. Integration Testing	6
In Memory Database	
WebApplicationFactory	
Postman	
3.3. End to End Testing.....	9
Cypress	

1. Опис на тестираната веб апликација

TicketShop е едноставна **E-commerce Asp.NET Core апликација** изработена во програмскиот јазик **C#** користејќи **Onion архитектура**.

Апликацијата овозможува:

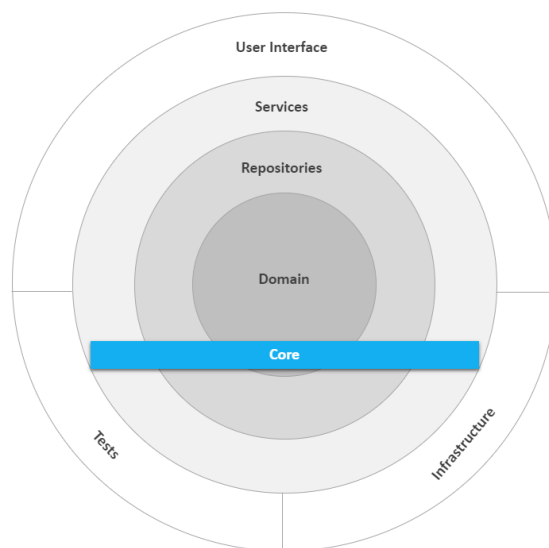
- Автентикација (регистрација и најава) на корисник;
- Преглед на достапните билети и информациите поврзани со истите;
- Креирање на нов билет, ажурирање и бришење на постоечки билет;
- Додавање на билет во кошничка со производи и бришење на билет од кошничката со производи;
- Правење нарачка со производите додадени во кошничката.

TicketShop преку **Restful API** комуницира со **TicketShopAdmin** апликацијата, каде се излистани сите нарачки направени од страна на сите корисници најавени на TicketShop, прикажувајќи го името и презимето на корисникот кој ја направил нарачката и бројот на нарачани билети. Овозможен е детален преглед на секоја нарачка, експортирање на нарачките во предефиниран Excel формат, како и експортирање на деталите за одредена нарачка (Invoice) во .pdf формат.

2. Опис и тестирање на Onion архитектура

Традиционалните архитектури често се соочуваат со проблемите на tight coupling и separation of concerns. Onion архитектурата е патерн кој овозможува подобар начин на развој на апликација од аспект на тестирање, одржување и надградување. Ваквата архитектура се соочува со предизвиците на 3-tier и n-tier архитектури со тоа што се состои од повеќе независни концентрични слоеви кои меѓусебно комуницираат со интерфејси во насока од крајниот кон централниот слој.

- **Domain Layer** – е центарот на Onion архитектурата кој ја содржи бизнис логиката и сите доменски објекти на апликацијата;
- **Repository Layer** – креира апстракција помеѓу доменските ентитети и апликациската бизнис логика.
- **Services Layer** – ги содржи интерфејсите со CRUD операции. Исто така, се користи за комуникација помеѓу Repository и UI слоевите. Притоа, интерфејсите се одделени од нивната имплементација со што се запазуваат принципите на loose coupling и separation of concerns.
- **UI Layer** – е најнадворешниот слој во хиерархијата кој содржи имплементација на dependency injection принципот, така што апликацијата со помош на loosely coupled структура комуницира со внатрешните слоеви преку интерфејси.



Слика 1. Onion Architecture

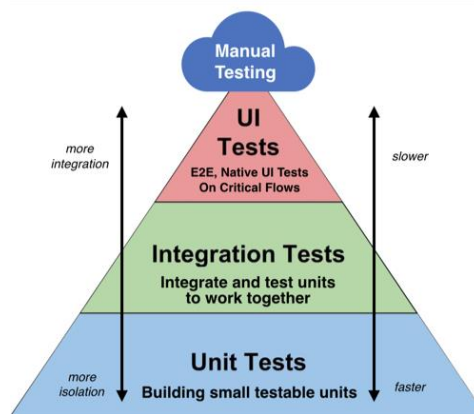
Кога станува збор за тестирање, предноста на Onion архитектурата се сведува на апстракцијата помеѓу различните слоеви на платформата, со што се намалува веројатноста за појава на грешки при тестирање.

Со различните нивоа на тестирање (unit, integration, end to end) кои со помош на разновидни алатки ги применивме врз нашата апликација, уште еднаш ја потврдивме независноста меѓу различните слоеви на оваа архитектура и поделбата на задачи.

3. Тестирање

Независно од методите за тестирање, патерните или архитектурата која ја избираме, со пишувањето на софистициран софтвер, доаѓа и потребата да се осигураме дека апликацијата работи исправно со додавањето на нови функционалности или со рефакторирање.

На *Слика 2*. е прикажана „пирамидата на тестирање“ која ги сумаризира основните типови на тестови кои еден софтвер треба да ги примени. Тестовите може да се категоризираат како:

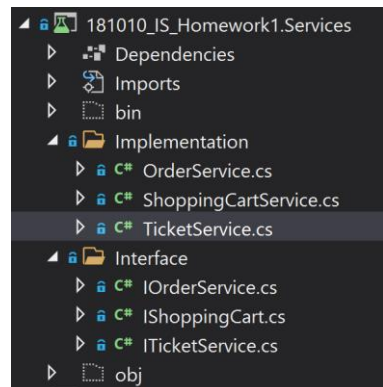


Слика 2: Testing Pyramid

- **Програмски** – кои имаат за цел да се осигураат дека бизнис логиката работи (пример: валидации, калкулации, подредување и слично). Во овие тестови спаѓаат **Unit** и **интеграциските** тестови;
- **Интеракциски** – кои служат за да се осигура корисничката интеракција со апликацијата преку симулација на евентуално сценарио со користење на некоја алатка. Дел од ваквите тестови се и **End2End** тестовите.

3.1. Unit Testing

Една од класичните и фундаментални техники е **unit тестирањето**. Концептот е прилично едноставен, а идејата е да се изолира кодот што е можно повеќе и да се овозможи unit тестирање на едноставен, брз и ефикасен начин. Доколку користењето на оваа техника е комплицирано, тоа



е индикатор дека кодот на апликацијата не е доволно фрагментиран и треба да претрпи промени.

Unit тестирање може да се примени на секаква компонента, а **во нашиот случај, го применивме на функции кои се наоѓаат во имплементацијата на Services слојот.**

Во Services слојот на TicketShop апликацијата, интерфејсите се одделени од нивната имплементација. Има три различни класи: TicketService, ShoppingCartService и OrderService кои служат за операции со билетите, кошничката и нарачките (*Слика 3*).

Слика 3. Сервиси во TicketShop апликацијата

Кога станува збор за Unit тестирање, .Net нуди неколку frameworks, меѓу кои xUnit е најнова и моментално е меѓу најпопуларните. За да бидеме во чекор со времето, за тестирање на сервисите во нашата апликација, **користевме xUnit framework** со кој се подобрува изолацијата на тестовите во споредба со NUnit и MSTests.

Дополнително, бидејќи нашите сервиси зависат од Repository слојот, односно користат инстанци од неколку репозиториуми од базата на податоци, **користевме Moq**. Moq е најпозната mocking framework за C#, .NET. Се користи во unit тестови за да се креираат „лажни“ објекти за основните зависности со што се олеснува интеракцијата меѓу методите и нивните зависности. Има за цел да го изолира парчето код кое поминува низ процес на тестирање од неговите надворешни зависности (како во нашиот случај, базата на податоци) и да се осигура дека соодветните методи се повикани за соодветните зависни објекти.

На *Слика 4.* е прикажана една функција од Ticket сервисот, а на *Слика 5.* е прикажан начинот на кој се прави unit тест за една ваква функција користејќи Moq.

```
namespace _181010_IS_Homework1.Services.Implementation
{
    5 references
    public class TicketService : ITicketService
    {
        private readonly IRepository<Ticket> _ticketRepository;
        private readonly IRepository<TicketInShoppingCart> _ticketInShoppingCartRepository;
        private readonly IUserRepository _userRepository;
        1 reference
        public TicketService(IRepository<Ticket> ticketRepository,
            IRepository<TicketInShoppingCart> ticketInShoppingCartRepository,
            IUserRepository userRepository)
        {
            _ticketRepository = ticketRepository;
            _ticketInShoppingCartRepository = ticketInShoppingCartRepository;
            _userRepository = userRepository;
        }

        0 references
        public TicketService() { }

        7 references
        public bool AddToShoppingCart(AddToShoppingCartDTO item, string userId)
        {
            var user = this._userRepository.Get(userId);
            var userShoppingCart = user.UserShoppingCart;

            if(item.TicketId != null && userShoppingCart != null)
            {
                var ticket = this.GetDetailsForTicket(item.TicketId);

                if(ticket != null)
                {
                    TicketInShoppingCart itemToAdd = new TicketInShoppingCart
                    {
                        Id = Guid.NewGuid(),
                        Ticket = ticket,
                        TicketId = ticket.Id,
                        ShoppingCart = userShoppingCart,
                        CartId = userShoppingCart.Id,
                        Quantity = item.Quantity
                    };
                    this._ticketInShoppingCartRepository.Insert(itemToAdd);
                    return true;
                }
                return false;
            }
            return false;
        }
    }
}
```

Слика 4. Пример функција од TicketService

```
public class TicketServiceTests
{
    // sut = system under test
    private readonly TicketService _sut;

    // Mocking
    private readonly Mock<IRepository<Ticket>> _ticketRepoMock = new Mock<IRepository<Ticket>>();
    private readonly Mock<IRepository<TicketInShoppingCart>> _ticketInShoppingCartRepoMock = new Mock<IRepository<TicketInShoppingCart>>();
    private readonly Mock<IUserRepository> _userRepoMock = new Mock<IUserRepository>();

    // Initializing the instance of TicketService
    0 references
    public TicketServiceTests()
    {
        _sut = new TicketService(_ticketRepoMock.Object, _ticketInShoppingCartRepoMock.Object, _userRepoMock.Object);
    }

    // TESTING: GetDetailsForTicket(Guid? id)
    [Fact]
    0 references
    public void GetDetailsForTicket_ShouldReturnTicket_WhenTicketExists()
    {
        // Arrange
        var ticketId = Guid.NewGuid();
        var title = "Example movie";
        var image = "someurl";
        var rating = 5;
        var price = 120;
        var seat = 2;
        var dateAndTime = DateTime.Now;

        var ticket = new Ticket
        {
            Id = ticketId,
            Title = title,
            Image = image,
            Rating = rating,
            Price = price,
            Seat = seat,
            DateAndTime = dateAndTime,
            TicketsInShoppingCart = null
        };






        _ticketRepoMock.Setup(x => x.Get(ticketId))
            .Returns(ticket);

        // Act
        var resultTicket = _sut.GetDetailsForTicket(ticketId);

        // Assert
        Assert.Equal(ticketId, resultTicket.Id);
        Assert.Equal(title, resultTicket.Title);
        Assert.Equal(image, resultTicket.Image);
        Assert.Equal(rating, resultTicket.Rating);
        Assert.Equal(price, resultTicket.Price);
        Assert.Equal(seat, resultTicket.Seat);
        Assert.Equal(dateAndTime, resultTicket.DateAndTime);
    }
}
```

Слика 5. Unit тестирање на функцијата од Слика 4. со Moq

За сите сервиси **напишавме вкупно 17 unit тестови** кои успешно се извршија (Слика 6.), а кои се наоѓаат во **Services.xUnit.Moq** проектот во нашата апликација.

Test	Duration
▲  Services.xUnit.Moq (17)	605 ms
▲  Services.xUnitTests.Moq (17)	605 ms
▷  OrderServiceTests (2)	192 ms
▷  ShoppingCartServiceTests (4)	203 ms
▷  TicketServiceTests (11)	210 ms

Слика 6. Успешно извршување на Unit тестови

3.2. Integration Testing

И покрај тоа што Unit тестовите се исклучително важни, имаат одредени недостатоци. Така, со Unit тестови се постигнува апстракција на зависностите, иако многу делови од апликацијата се посветени токму на справување со тие зависности, како што се бази на податоци, библиотеки и слично. Ваквите зависности не можат да се моделираат со Unit тестови или пак да се заменат со mocks, па оттука, не би знаеле дали ги користиме соодветно доколку не примениме тестови од повисоко ниво, како што се интеграциски и End-to-End тестови.

Интеграциското тестирање е техника одговорна за спојување и комбинирање на различни делови од апликацијата кои би требало да се извршуваат последователно. Се користи за да се осигураме дали интеракцијата на различни компоненти се одвива како што сме предвиделе.

Во **TicketShop** извршивме **интеграциско тестирање на HTTP POST и HTTP GET акциите во Ticket и во Home контролерите**, во кои последователно се повикуваат повеќе функции од сервисите. Бидејќи за тестирање на ShoppingCart контролерот беше потребна автентикација на корисник, истиот го тестиравме со End to End тестирање.

Првичната идеја беше да не работиме со оригиналната, локална база на податоци, туку да креираме **In Memory Database**. Целта на ваквата база на податоци е нејзино рекреирање при извршување на секој тест, така што би се избегнал проток на податоци меѓу тестовте. За да се постигне ова, во Startup.cs ја користевме оригиналната база на податоци, но креиравме виртуелни функции кои ги преоптоваривме во класата TestStartup.cs на тој начин што при тестирање, би работеле со новокреираната база, наместо со оригиналната. In Memory Database содржи претходно дефинирани објекти (во класата PredefinedData.cs) кои одново се рекреираат при извршување на секој тест (во класата DataSeeder.cs).

Сепак, во една понова статија која може да се погледне на следниот [линк](#), наведени се недостатоците на ваквите In Memory бази. Во статијата се потенцира фактот дека иако ваквите бази на податоци се брзи за имплементација, проблемите кои ги предизвикуваат не се лесни за

решавање. Поточно, ваквите бази се само бледа имитација на Entity Framework базите, што значи дека за целосно интеграциско тестирање, повторно мора да се дозволи пристап до реалната (raw) база на податоци. Со примена на In Memory Database, само се дуплира бројот на напишани тестови.

Оттука, одлучивме да го закоментираме целиот код напишан за замена на базата на податоци со предефинирани објекти при тестирање и продолживме да изведуваме интеграциски тестови на посовремен и поедноставен начин. Наместо мануелно подесување на HostBuilder, може едноставно да се примени **WebApplicationFactory** за да се подигне апликацијата во меморија со цел да се постигнат функционални тестови. Во нашиот случај, користиме **WebApplicationFactory** со оригиналната база на податоци подесена во класата **Startup.cs**, од претходно наведените причини.

```
// HTTP GET CREATE
[Fact]
0 references
public async Task Create_Get_ReturnsCreateHtmlPage()
{
    // Arrange
    var factory = new WebApplicationFactory<Startup>();

    // Create an HttpClient which is setup for the test host
    var client = factory.CreateClient();

    // Act
    var response = await client.GetAsync("/Tickets/Create");

    // Assert
    var responseString = await response.Content.ReadAsStringAsync();

    Assert.Contains("Create", responseString);
}
```

Слика 7. Пример интеграциски тест на GET акција

На Слика 7. е прикажан еден едноставен интеграциски тест на GET акцијата CREATE од Tickets контролерот. Се иницијализира **WebApplicationFactory** како и http клиент кој пристапува до посакуваниот end-point. Проверката за валидност на тестот се извршува така што се проверува дали во одговорот кој го добил клиентот се содржи одреден стринг.

```
// HTTP POST CREATE
[Fact]
0 references
public async Task Create_Post_CreatesNewTicket()
{
    var factory = new WebApplicationFactory<Startup>();

    // This is done to disable POST actions redirect in order to prevent losing information in the response header
    var clientOptions = new WebApplicationFactoryClientOptions();
    clientOptions.AllowAutoRedirect = true;
    clientOptions.BaseAddress = new Uri("http://localhost");
    clientOptions.HandleCookies = true;
    clientOptions.MaxAutomaticRedirections = 7;

    var client = factory.CreateClient(clientOptions);

    // await EnsureAuthenticationCookie();
    var formData = new Dictionary<string, string>
    {
        { "Id", "40022a5e-1058-4f05-8fe3-0d8175388932" },
        { "Title", "nov" },
        { "Image", "nova" },
        { "Rating", "1" },
        { "Price", "200" },
        { "Seat", "4" },
        { "DateTime", DateTime.Now.ToString() },
        { "TicketsInShoppingCart", null }
    };

    var response = await client.PostAsync("/Tickets/Create", new FormUrlEncodedContent(formData));

    var requestString = await response.Content.ReadAsStringAsync();

    // Assert
    Assert.Contains("nova", requestString);
}
```

Слика 8. Пример интеграциски тест на POST акција

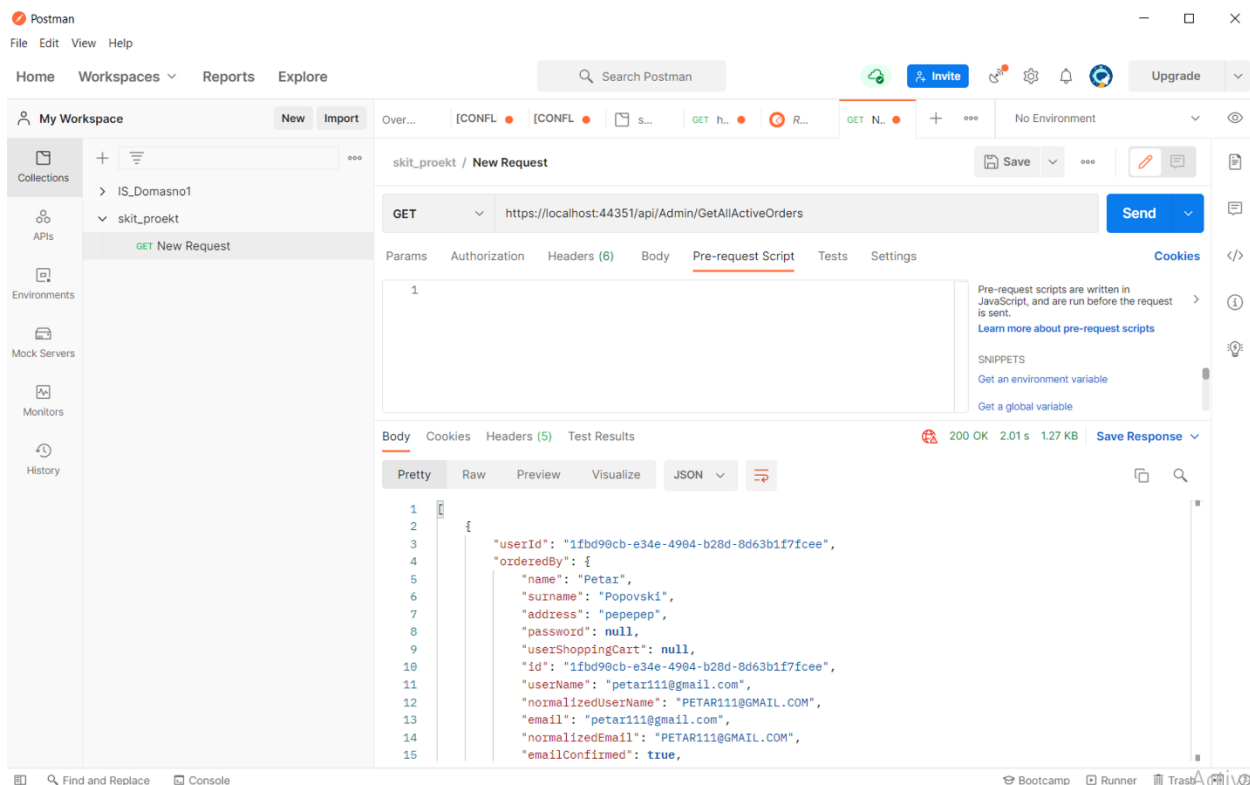
На Слика 8. е прикажан интеграциски тест на POST акцијата CREATE од Ticket контролерот. Повторно се иницијализира **WebApplicationFactory**, како и клиент на ист начин како и на Слика 7. Потоа, се креира нов објект од тип Dictionary, кој всушност ќе биде новокреираниот Ticket, за потоа да се предаде како аргумент во конструкторот на **FormUrlEncodedContent** кој ќе иницијализира објекти со зададени name, value парови. На крај, креираниот Ticket го испраќаме на /Ticket/Create, со што се пополнува формата за креирање на нов билет.

Вкупно напишавме 14 интеграциски тестови за GET и POST акции во Home и Ticket контролерите кои успешно се извршија.

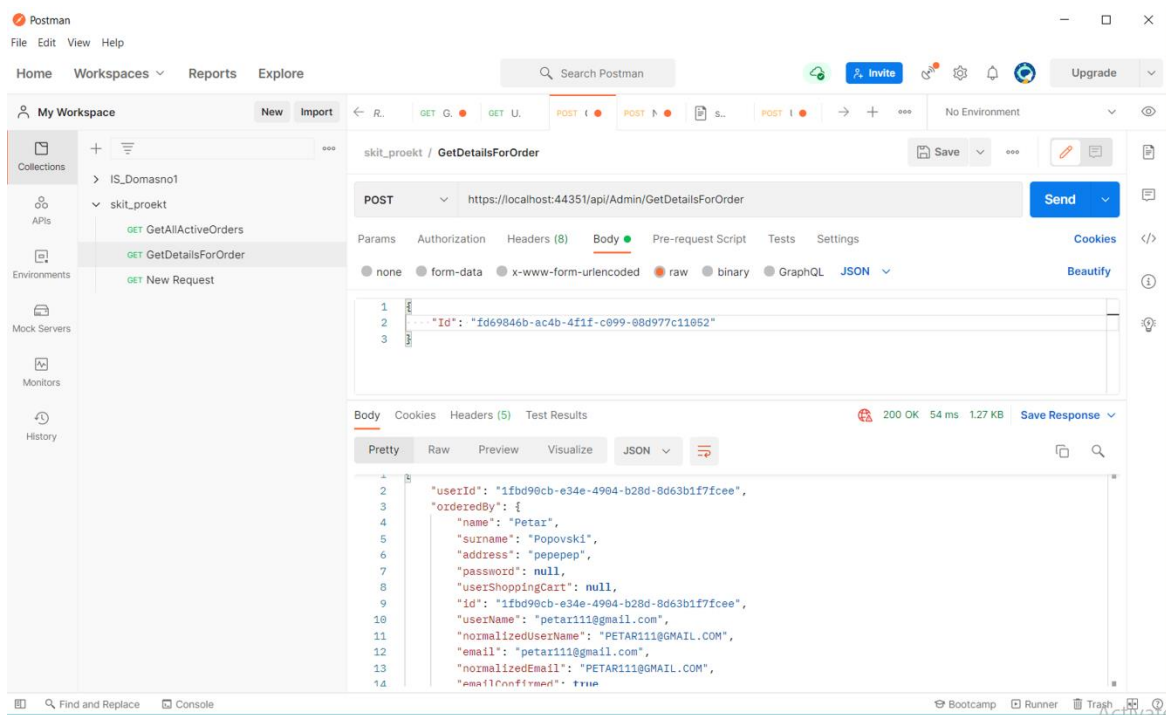
Дополнително, ја искористивме и алатката Postman која се користи за тестирање на API. Претставува API клиент кој го поедноставува креирањето, споделувањето, тестирањето и документирањето на APIs. Ова е овозможено така што корисниците може да креираат и зачуваат едноставни или комплексни HTTP/s requests, како и нивните добиени одговори. Оваа алатка ни послужи за **тестирање на комуникацијата помеѓу TicketShop и микросервисот TicketShopAdmin.**

Овие две апликации меѓусебно комуницираат на начин што, регистрираниот и најавениот корисник прави нарачка на билети од TicketShop апликацијата, а вака направената нарачка се прикажува во табела во апликацијата TicketShopAdmin заедно со сите нарачки од останатите корисници. **Со Postman тестираме дали е достапна акцијата GetAllActiveOrders, со која се прикажуваат сите нарачки, и акцијата GetDetailsForOrder, со која се прегледуваат деталите за некоја направена нарачка.** На *Слика 10.* и *Слика 11.* е прикажан добиениот одговор во Postman.

TicketShop.IntegrationTests (14)	5.5 sec
TicketShop.IntegrationTests (1...	5.5 sec
HomeTest (1)	765 ms
TicketTest (13)	4.7 sec
AddToShoppingCart_Get...	157 ms
Create_Get_ReturnsCreate...	86 ms
Create_Post_CreatesNewTi...	280 ms
Delete_Get_ReturnsDelete...	137 ms
Delete_Get_WhenIdsNull	1.4 sec
Delete_Get_WhenTicketIs...	97 ms
Delete_Post_DeletesTicket	1.5 sec



Слика 10. Тестирање на акцијата GetAllActiveOrders во Postman



Слика 11. Тестирање на акцијата GetDetailsForOrder во Postman

3.3. End to End Testing

Идејата на end-to-end тестирањето е да се тестира и имитира однесувањето на еден корисник при користење на апликацијата, со интеракција со сите постоечки функционалности, од почеток до крај. Со додавање на овој слој на тестирање на нашата апликација, се осигуравме дека се покриени сите можни интеракции со апликацијата, спречувајќи било какви грешки.

.Net Core Blazor е одличен начин за да се креира front-end за C# апликација која може да се искористи за целите на end to end тестирање со помош на тестирачката алатка **Cypress**. Cypress е релативно нова, **автоматизирана front-end алатка** за тестирање која користи jQuery како DOM манипулација за интеракција со веб пребарувач. Со помош на тестови напишани во JavaScript прикажува како би се одвивала интеракцијата со корисничкиот интерфејс чекор по чекор. Оваа алатка е достоин комплемент на Selenium имајќи предност во однос на популарност и едноставна употреба. Се што треба да се направи за да се инсталира оваа алатка, е да се извршат следните команди во командна линија во root фолдерот на апликацијата (во нашиот случај, креиравме нов проект TicketShop.E2ETest):

- `npm init -y`
- `npm i cypress --save-dev`
- `npx cypress open`

По извршувањето на последната команда, во нов прозорец се отвора интерфејсот на Cypress и креира фолдери каде може да се пишуваат тест скрипти (Слика 12.).

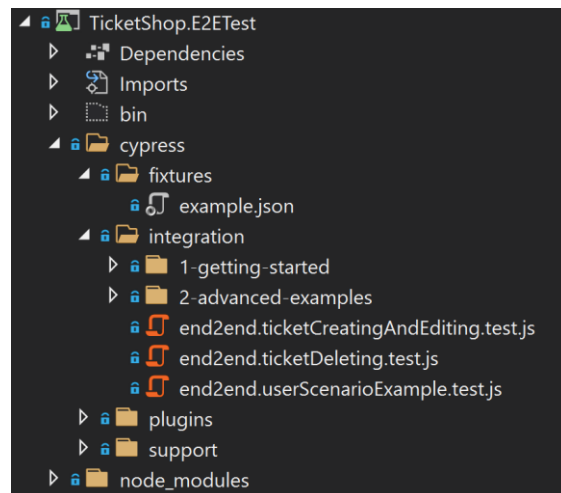
Со помош на Cypress, ја тестиравме основната TicketShop апликација, како и микросервисот со кој комуницира преку Restful API – TicketShopAdmin на начин што на html елементите додадовме id атрибути кои потоа ги користевме при пишувањето на JavaScript тестови.

Сценаријата кои ги тестиравме во TicketShop се:

- Регистрирање и логирање на корисник;
- Навигација до страницата каде се прикажани достапните билети;
- Додавање на билет во кошничка;
- Бришење билет од кошничка;
- Правење нарачка;
- Додавање, ажурирање и брижење на билет;

А во TicketShopAdmin:

- Преглед на сите нарачки;
- Експортирање на нарачките во Excel;
- Преглед на нарачка;
- Експортирање .pdf Invoice на нарачка;



Слика 12. Инсталирање на Cypress

```
<div class="row">
  <div class="col-md-4">
    <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
      <h4>Create a new account.</h4>
      <hr />
      <div asp-validation-summary="All" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="Input.Email"></label>
        <input asp-for="Input.Email" class="form-control" id="register-email" />
        <span asp-validation-for="Input.Email" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.Name"></label>
        <input asp-for="Input.Name" class="form-control" id="register-name" />
        <span asp-validation-for="Input.Name" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.Surname"></label>
        <input asp-for="Input.Surname" class="form-control" id="register-surname" />
        <span asp-validation-for="Input.Surname" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.Address"></label>
        <input asp-for="Input.Address" class="form-control" id="register-address" />
        <span asp-validation-for="Input.Address" class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.Password"></label>
        <input asp-for="Input.Password" class="form-control" id="register-password" />
        <span asp-validation-for="Input.Password" class="text-danger"></span>
      </div>
    </form>
  </div>
</div>
```

Слика 13. Додавање на id атрибут во Register View-то, со цел да се пристапат HTML елементите во Cypress тестовите

```
describe('example scenario', () => {
  beforeEach(() => {
    // open the web app by accessing the localhost
    cy.visit('https://localhost:44351/')
  })

  it('displays user registration, login and purchase', () => {

    // REGISTER

    // click on the button 'Register' in the navigation bar in order to register
    cy.get('#register').click()

    // enter your email
    cy.get('#register-email').type("test@teeeeeeeeeest.com")

    // enter your name
    cy.get('#register-name').type("Angela")

    // enter your surname
    cy.get('#register-surname').type("Madjar")

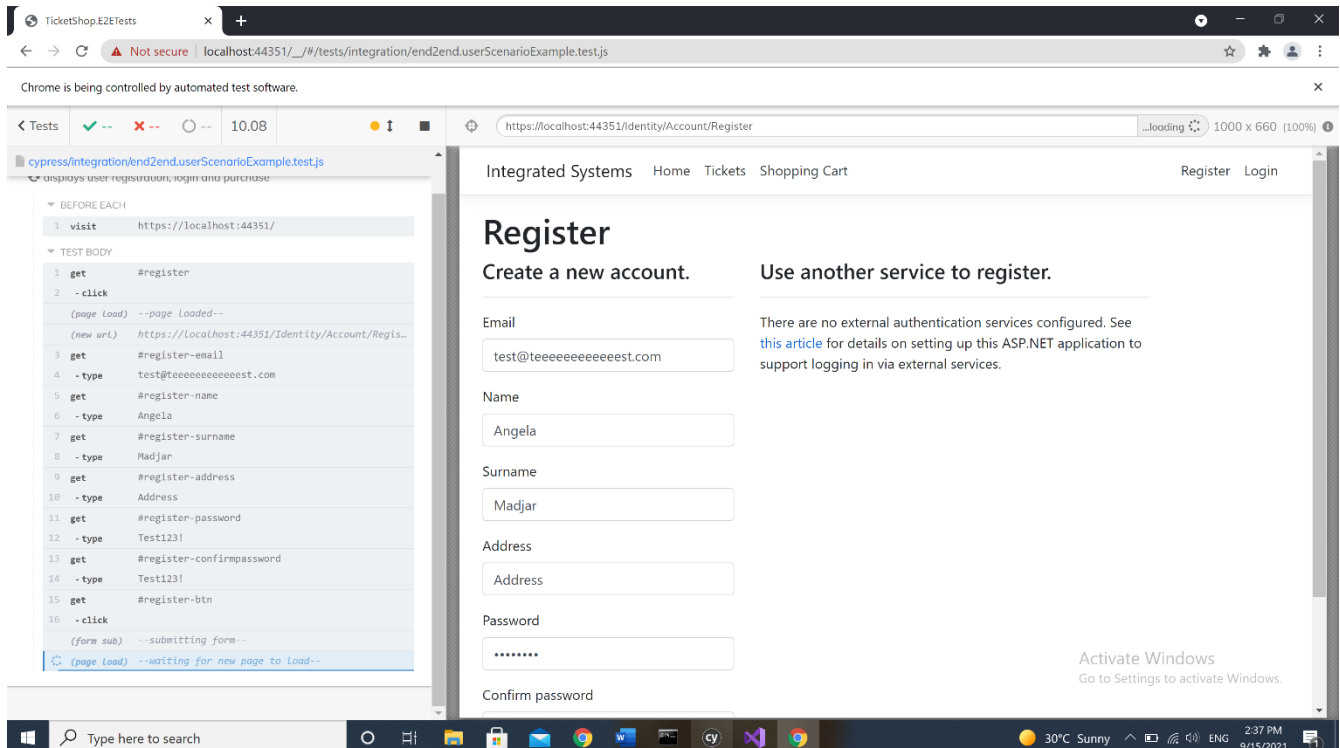
    // enter your address
    cy.get('#register-address').type("Address")

    // enter your password
    cy.get('#register-password').type("Test123!")

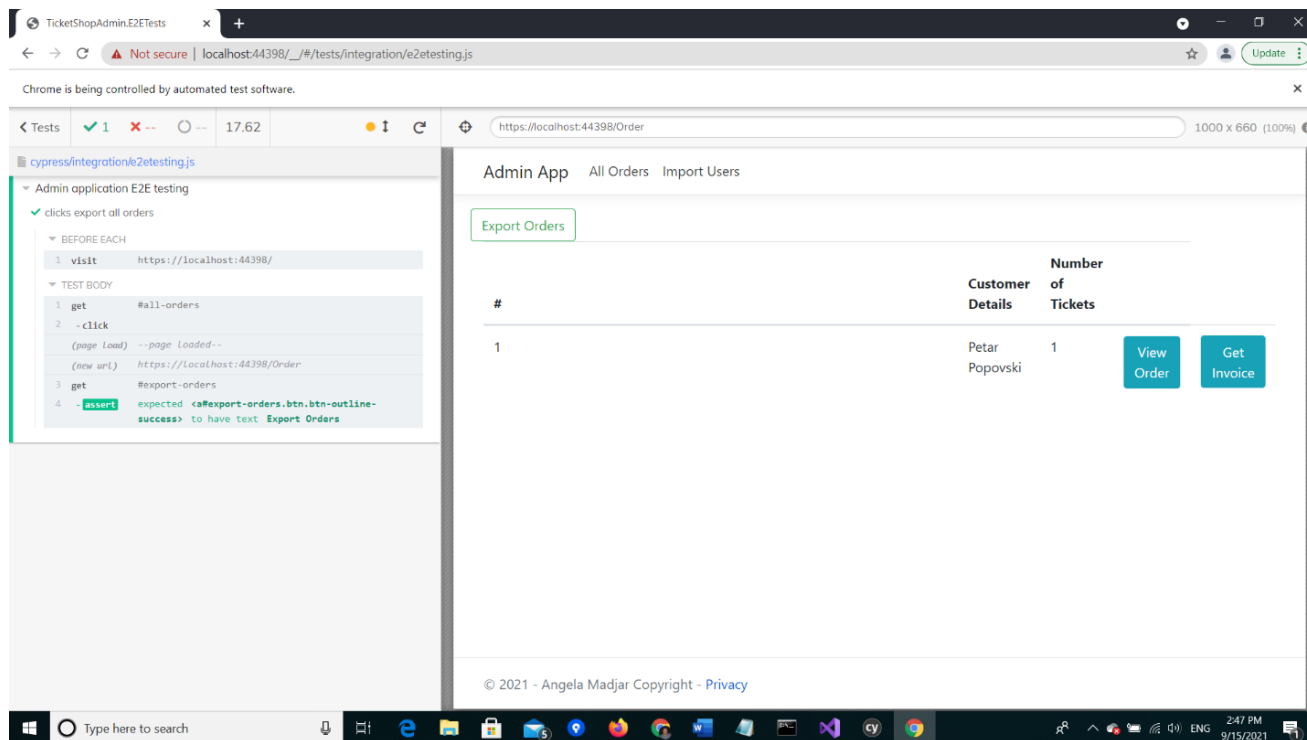
    // confirm your password
    cy.get('#register-confirmpassword').type("Test123!")

    // click on the 'Register' button
    cy.get('#register-btn').click()
  })
})
```

Слика 14. Сypress тест во кој се пристапуваат додадените id атрибути во Register View



Слика 15. Визуелен приказ на извршување на Сypress тестот од Слика 14.



Слика 16. Визуелен приказ на извршување на Cypress тестот од Слика 14.

Референци

<https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-mvc/>

<https://lawrey.medium.com/unit-tests-ui-tests-integration-tests-end-to-end-tests-c0d98e0218a6>

<https://dev.to/betoyanes/unit-vs-integration-vs-e2e-tests-4b2o>

https://jimmybogard.com/avoid-in-memory-databases-for-tests/?fbclid=IwAR2dbHH0plT5LmYFbUnJBst_TuUknclSAVziMk_EAtLMQ48p8-2QRmFamw4

https://www.dotnetcurry.com/aspnet-core/1420/integration-testing-aspnet-core?fbclid=IwAR3WHYyzOBS5Qm_zME06r9UdNBqZOGInb2vbUy77uNL1vW40xmpDJqSEavQ

<https://adamstorr.azurewebsites.net/blog/integration-testing-with-aspnetcore-3-1>

<https://kagawacode.medium.com/blazor-end-to-end-test-with-cypress-55bcf1a7a079>

<https://adamstorr.azurewebsites.net/blog/integration-testing-with-aspnetcore-3-1-remove-the-boiler-plate>

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.testing.webapplicationfactory-1?view=aspnetcore-3.0>