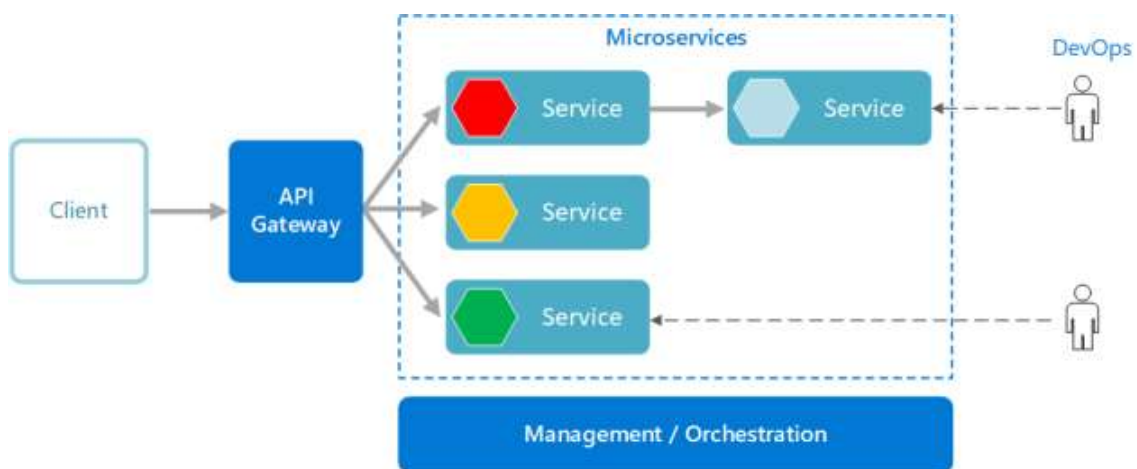


Introdução

Após o recente lançamento de .NET 8, eu assumi a responsabilidade de escrever um guia abrangente sobre a construção de microsserviços usando o ASP.NET Core 8. Esta última versão de .NET é uma versão significativa que oferece uma série de novos recursos e aprimoramentos, bem como melhorias de desempenho e suporte de longo prazo. Baseia-se nos aprimoramentos de desempenho introduzidos em .NET 7 otimizando ainda mais o compilador Just-In-Time (JIT), o coletor de lixo e o tempo de execução. O resultado são tempos de inicialização mais rápidos, melhor desempenho geral do aplicativo e menor uso de memória. Você pode descobrir mais sobre todos os novos recursos e aprimoramentos [aqui](#). Agora, vamos nos concentrar em microsserviços.



Uma arquitetura de microsserviços consiste em uma coleção de serviços pequenos, independentes e fracamente acoplados. Cada serviço é independente, implementa uma única capacidade de negócios, é responsável por persistir seus próprios dados, é uma base de código separada e pode ser implantado de forma independente.

Gateways API são pontos de entrada para clientes. Em vez de chamar serviços diretamente, os clientes chamam o gateway da API, que encaminha a chamada para os serviços apropriados.

Existem várias vantagens usando a arquitetura de microsserviços:

- Os desenvolvedores podem entender melhor a funcionalidade de um serviço.
- A falha em um serviço não afeta outros serviços.
- É mais fácil gerenciar correções de bugs e lançamentos de recursos.
- Os serviços podem ser implantados em vários servidores para melhorar o desempenho.

- Os serviços são fáceis de alterar e testar.
- Os serviços são fáceis e rápidos de implementar.
- Permite escolher a tecnologia que é adequada para uma funcionalidade específica.

Antes de escolher a arquitetura de microsserviços, aqui estão alguns desafios a serem considerados:

- Os serviços são simples, mas todo o sistema como um todo é mais complexo.
- A comunicação entre serviços pode ser complexa.
- Mais serviços é igual a mais recursos.
- Testes globais podem ser difíceis.
- Depurar pode ser mais difícil.

A arquitetura de microsserviços é ótima para grandes empresas, mas pode ser complicada para pequenas empresas que precisam criar e iterar rapidamente e não querem entrar em orquestração complexa.

Este artigo fornece um guia abrangente sobre a criação de microsserviços usando ASP.NET Core, construção de gateways de API usando Ocelot, estabelecimento de repositórios usando MongoDB, gerenciamento de JWT em microsserviços, testes de unidade de microsserviços usando xUnit e Moq, monitoramento de microsserviços usando verificações de integridade e, finalmente, implantação de microsserviços usando Docker.

Melhores Práticas

Aqui está um resumo de algumas práticas recomendadas:

- **Responsabilidade Única:** Cada microsserviço deve ter uma única responsabilidade ou propósito. Isso significa que ele deve fazer uma coisa e fazê-lo bem. Isso torna mais fácil entender, desenvolver, testar e manter cada microsserviço.
- **Armazenamento de Dados Separados:** Os microsserviços devem idealmente ter seu próprio armazenamento de dados. Este pode ser um banco de dados separado, que é isolado de outros microsserviços. Esse isolamento garante que alterações ou problemas nos dados de um microsserviço não afetem outros.
- **Comunicação Assíncrona:** Use padrões de comunicação assíncronos, como filas de mensagens ou sistemas de publicação e assinatura, para

habilitar a comunicação. Isso torna o sistema mais resiliente e separa os serviços uns dos outros.

- **Containerização:** Use tecnologias de containerização como o Docker para empacotar e implantar microsserviços. Os contêineres fornecem um ambiente consistente e isolado para seus microsserviços, facilitando o gerenciamento e a escala deles.
- **Orquestração:** Use ferramentas de orquestração de contêineres como o Kubernetes para gerenciar e dimensionar seus contêineres. O Kubernetes fornece recursos para balanceamento de carga, dimensionamento e monitoramento, tornando-o uma ótima opção para orquestrar microsserviços.
- **Construir e Implantar Separação:** Mantenha os processos de compilação e implantação separados. Isso significa que o processo de compilação deve resultar em um artefato implantável, como uma imagem de contêiner do Docker, que pode ser implantada em diferentes ambientes sem modificação.
- **Apátrida:** Os microsserviços devem ser apátridas, tanto quanto possível. Qualquer estado necessário deve ser armazenado no banco de dados ou em um armazenamento de dados externo. Serviços sem Estado são mais fáceis de escalar e manter.
- **Micro Fronteiras:** Se você estiver criando um aplicativo da Web, considere usar a abordagem de micro frontends. Isso envolve dividir a interface do usuário em componentes menores e implantáveis de forma independente que podem ser desenvolvidos e mantidos por equipes separadas.

Desenvolvimento Ambiente

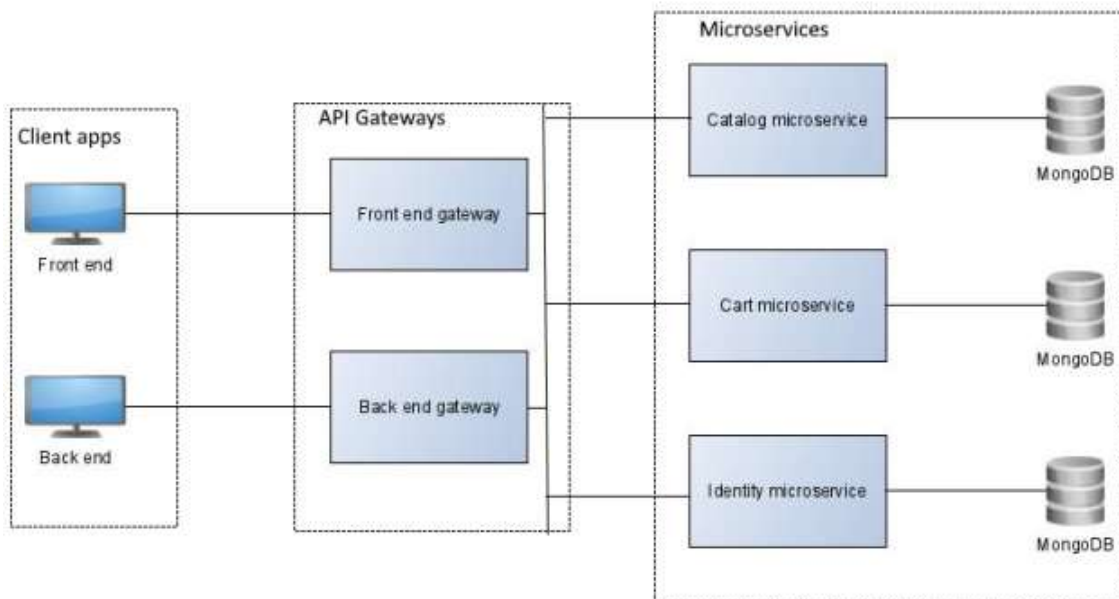
- Visual Studio 2022 >= 17.8.0
- .NET 8.0
- MongoDB
- Carteiro

Pré-requisitos

- C#
- ASP.NET Core
- Ocelot
- Espuma

- Serilog
- JWT
- MongoDB
- xUnit
- Moq

Arquitetura



Existem três microserviços:

- **Catálogo microservice:** permite gerir o catálogo.
- **Cart microservice:** permite gerir o carrinho.
- **Microserviço de identidade:** permite gerenciar autenticação e usuários.

Cada microserviço implementa uma única capacidade de negócios e tem seu próprio banco de dados dedicado. Isso é chamado de padrão de banco de dados por serviço. Esse padrão permite uma melhor separação de preocupações, isolamento de dados e escalabilidade. Em uma arquitetura de microserviços, os serviços são projetados para serem pequenos, focados e independentes, cada um responsável por uma funcionalidade específica. Para manter essa separação, é essencial garantir que cada microserviço gerencie seus dados de forma independente. Aqui estão outros prós deste padrão:

- O esquema de dados pode ser modificado sem afetar outros microserviços.

- Cada microsserviço tem seu próprio armazenamento de dados, impedindo o acesso acidental ou não autorizado aos dados de outro serviço.
- Como cada microsserviço e seu banco de dados são separados, eles podem ser dimensionados independentemente com base em suas necessidades específicas.
- Cada microsserviço pode escolher a tecnologia de banco de dados que melhor se adapte às suas necessidades, sem estar vinculado a um único banco de dados monolítico.
- Se um servidor de banco de dados estiver inativo, isso não afetará outros serviços.

Existem dois gateways API, um para o frontend e outro para o backend.

Abaixo está o gateway da API frontend:

- **GET /catálogo:** recupera itens de catálogo.
- **GET /catálogo/{id}:** recupera um item de catálogo.
- **GET /carrinho:** recupera itens do carrinho.
- **POST /carrinho:** adiciona um item de carrinho.
- **PUT /carrinho:** atualiza um item de carrinho.
- **DELETE /carrinho:** exclui um item do carrinho.
- **POST /identidade/login:** realiza um login.
- **POST /identidade/registro:** registra um usuário.
- **GET /identidade/validar:** valida um token JWT.

Abaixo está o gateway da API de back-end:

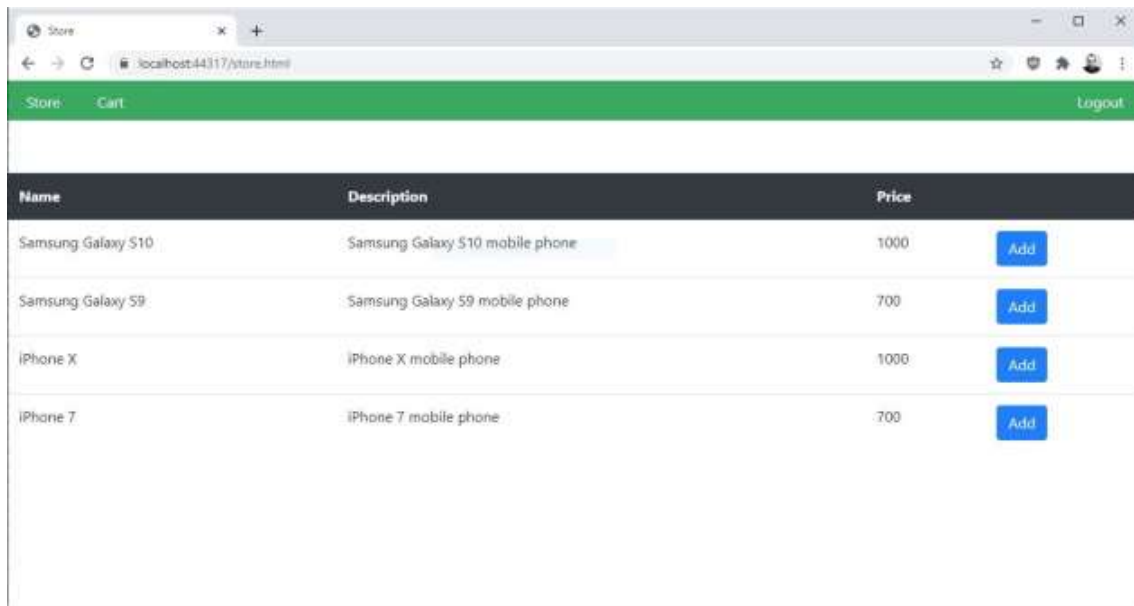
- **GET /catálogo:** recupera itens de catálogo.
- **GET /catálogo/{id}:** recupera um item de catálogo.
- **POST /catálogo:** cria um item de catálogo.
- **PUT /catálogo:** atualiza um item de catálogo.
- **DELETE /catálogo/{id}:** exclui um item de catálogo.
- **PUT /carrinho/catálogo de atualização-item:** atualiza um item de catálogo em carrinhos.
- **DELETE /cart/delete-catalog-item:** exclui referências de itens de catálogo de carrinhos.

- **POST /identidade/login:** realiza um login.
- **GET /identidade/validar:** valida um token JWT.

Finalmente, existem dois aplicativos clientes. Um front-end para acessar a loja e um back-end para gerenciar a loja.

O frontend permite que os usuários registrados vejam os itens de catálogo disponíveis, permite adicionar itens de catálogo ao carrinho e remover itens de catálogo do carrinho.

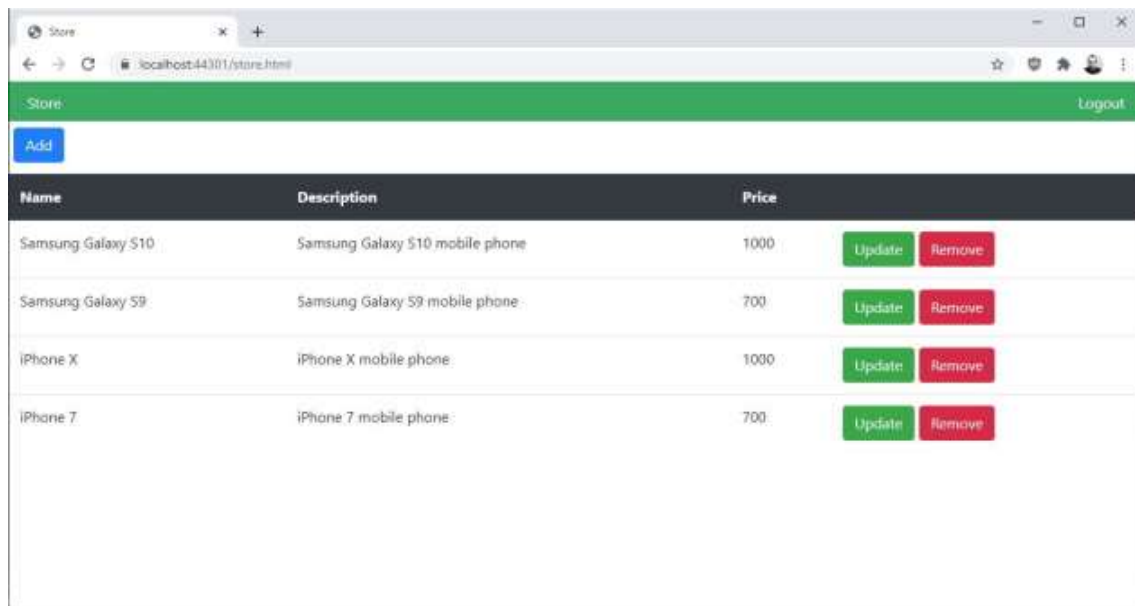
Aqui está uma captura de tela da página da loja no frontend:



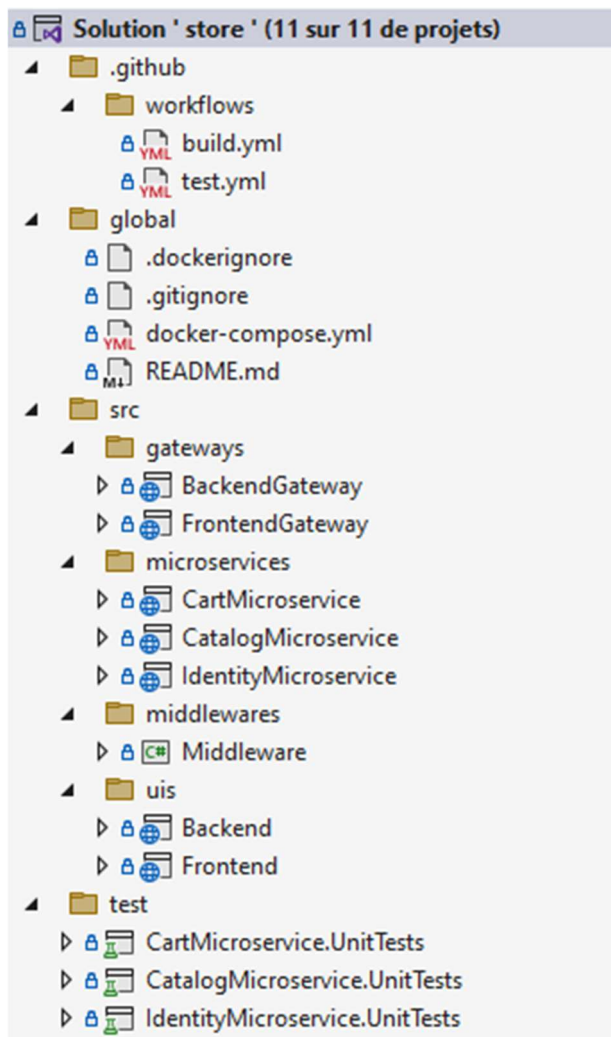
Name	Description	Price	
Samsung Galaxy S10	Samsung Galaxy S10 mobile phone	1000	<button>Add</button>
Samsung Galaxy S9	Samsung Galaxy S9 mobile phone	700	<button>Add</button>
iPhone X	iPhone X mobile phone	1000	<button>Add</button>
iPhone 7	iPhone 7 mobile phone	700	<button>Add</button>

O back-end permite que os usuários administradores vejam os itens de catálogo disponíveis, permite adicionar novos itens de catálogo, atualizar itens de catálogo e remover itens de catálogo.

Aqui está uma captura de tela da página da loja no backend:



Código Fonte



- CatalogMicroserviceo Project contém o código-fonte do microserviço que gerencia o catálogo.

- CartMicroserviceo Project contém o código-fonte do microserviço que gerencia o carrinho.
- IdentityMicroserviceo Project contém o código-fonte do microserviço que gerencia a autenticação e os usuários.
- Middlewareo Project contém o código-fonte de funcionalidades comuns usadas por microserviços.
- FrontendGatewayo Project contém o código-fonte do gateway da API frontend.
- BackendGatewayo Project contém o código-fonte do gateway da API de back-end.
- Frontendo Project contém o código-fonte do aplicativo cliente frontend.
- Backendo Project contém o código-fonte do aplicativo cliente de back-end.
- testpasta de solução contém testes unitários de todos os microserviços.

Microserviços e gateways são desenvolvidos usando ASP.NET Core e C#. Os aplicativos cliente são desenvolvidos usando HTML e Vanilla JavaScript para se concentrar em microserviços.

Microserviços

Catálogo Microservice

Vamos começar com o microserviço mais simples CatalogMicroservice.

CatalogMicroserviceé responsável pela gestão do catálogo.

Abaixo está o modelo usado por CatalogMicroservice:

```
public class CatalogItem
{
    public static readonly string DocumentName = nameof(CatalogItem);

    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; init; }

    public required string Name { get; set; }

    public string? Description { get; set; }
```



```
    public decimal Price { get; set; }  
}
```

Abaixo está a interface do repositório:

```
public interface ICatalogRepository  
{  
    IList<CatalogItem> GetCatalogItems();  
    CatalogItem? GetCatalogItem(string catalogItemId);  
    void InsertCatalogItem(CatalogItem catalogItem);  
    void UpdateCatalogItem(CatalogItem catalogItem);  
    void DeleteCatalogItem(string catalogItemId);  
}
```

Abaixo está o repositório:

```
public class CatalogRepository(IMongoDatabase db) : ICatalogRepository  
{  
    private readonly IMongoCollection<CatalogItem> _col =  
        db.GetCollection<CatalogItem>(CatalogItem.DocumentName);  
  
    public IList<CatalogItem> GetCatalogItems() =>  
        _col.Find(FilterDefinition<CatalogItem>.Empty).ToList();  
  
    public CatalogItem GetCatalogItem(string catalogItemId) =>  
        _col.Find(c => c.Id == catalogItemId).FirstOrDefault();  
  
    public void InsertCatalogItem(CatalogItem catalogItem) =>  
        _col.InsertOne(catalogItem);  
  
    public void UpdateCatalogItem(CatalogItem catalogItem) =>  
        _col.UpdateOne(c => c.Id == catalogItem.Id, Builders<CatalogItem>.Update
```

```

        .Set(c => c.Name, catalogItem.Name)
        .Set(c => c.Description, catalogItem.Description)
        .Set(c => c.Price, catalogItem.Price));

public void DeleteCatalogItem(string catalogItemId) =>
    _col.DeleteOne(c => c.Id == catalogItemId);
}

```

Abaixo está o controlador:

```

[Route("api/[controller]")]
[ApiController]
public class CatalogController(ICatalogRepository catalogRepository) :
    ControllerBase
{
    // GET: api/<CatalogController>
    [HttpGet]
    public IActionResult Get()
    {
        var catalogItems = catalogRepository.GetCatalogItems();
        return Ok(catalogItems);
    }

    // GET api/<CatalogController>/653e4410614d711b7fc953a7
    [HttpGet("{id}")]
    public IActionResult Get(string id)
    {
        var catalogItem = catalogRepository.GetCatalogItem(id);
        return Ok(catalogItem);
    }
}

```

```
// POST api/<CatalogController>
[HttpPost]
public IActionResult Post([FromBody] CatalogItem catalogItem)
{
    catalogRepository.InsertCatalogItem(catalogItem);
    return CreatedAtAction(nameof(Get), new { id = catalogItem.Id }, catalogItem);
}
```

```
// PUT api/<CatalogController>
[HttpPut]
public IActionResult Put([FromBody] CatalogItem? catalogItem)
{
    if (catalogItem != null)
    {
        catalogRepository.UpdateCatalogItem(catalogItem);
        return Ok();
    }
    return new NoContentResult();
}
```

```
// DELETE api/<CatalogController>/653e4410614d711b7fc953a7
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    catalogRepository.DeleteCatalogItem(id);
    return Ok();
}
```

```
}
```

ICatalogRepository é adicionado usando injeção de dependência em *Startup.com* para tornar o microservice testável:

```
// This method gets called by the runtime.
```

```
// Use this method to add services to the container.
```

```
public void ConfigureServices(IServiceCollection services)
```

```
{
```

```
    services.AddControllers();
```

```
    services.AddMongoDb(Configuration);
```

```
    services.AddSingleton<ICatalogRepository>(sp =>
```

```
        new CatalogRepository(sp.GetService<IMongoDatabase>() ??
```

```
            throw new Exception("IMongoDatabase not found"))
```

```
);
```

```
    services.AddSwaggerGen(c =>
```

```
{
```

```
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "Catalog", Version = "v1" });
```

```
});
```

```
    // ...
```

```
}
```

Abaixo está AddMongoDB método de extensão:

```
public static void AddMongoDb
```

```
(this IServiceCollection services, IConfiguration configuration)
```

```
{
```

```
    services.Configure<MongoOptions>(configuration.GetSection("mongo"));
```

```
    services.AddSingleton(c =>
```

```
{
```

```
        var options = c.GetService<IOptions<MongoOptions>>();
```

```

        return new MongoClient(options.Value.ConnectionString);
    });
    services.AddSingleton(c =>
    {
        var options = c.GetService<IOptions<MongoOptions>>();
        var client = c.GetService<MongoClient>();

        return client.GetDatabase(options.Value.Database);
    });
}

```

Abaixo está Configure método em *Startup.com*:

```

// This method gets called by the runtime.
// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseSwagger();

    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "Catalog V1");
    });
}

```

```
var option = new RewriteOptions();  
option.AddRedirect("^$", "swagger");  
app.UseRewriter(option);
```

```
// ...
```

```
app.UseHttpsRedirection();
```

```
app.UseRouting();
```

```
app.UseAuthorization();
```

```
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapControllers();  
});  
}
```

Abaixo está *appsettings.com*:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  "AllowedHosts": "*",  
  "mongo": {
```

```

"connectionString": "mongodb://127.0.0.1:27017",

"database": "store-catalog"

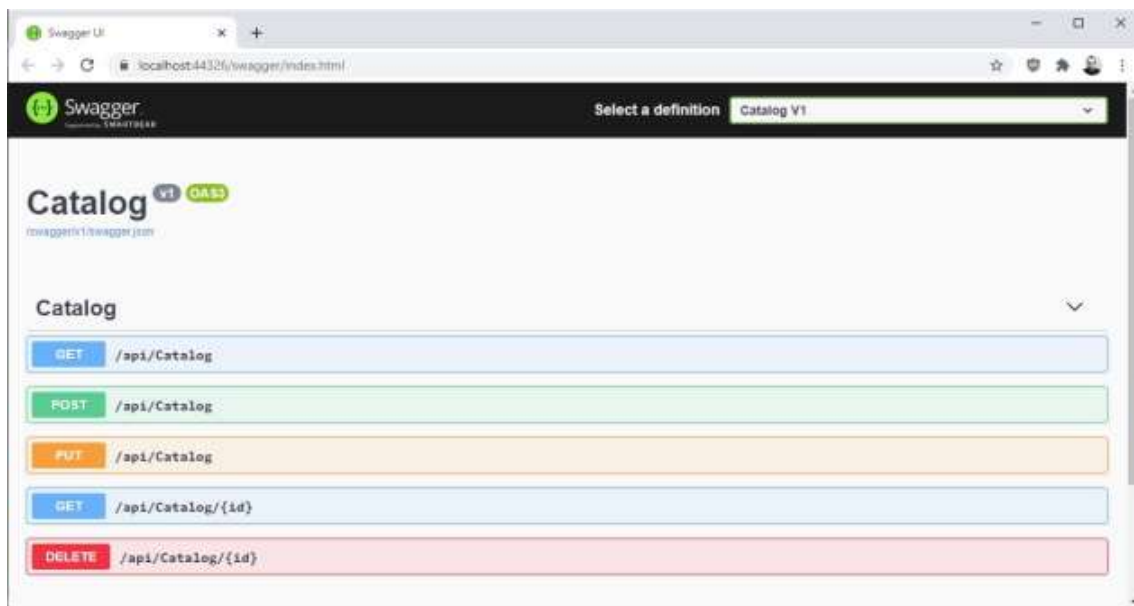
}

}

```

A documentação da API é gerada usando o Swashbuckle. O middleware do Swagger está configurado em *Startup.com*, em *ConfigureServices* e *Configure* métodos em *Startup.cs*.

Se você correr *CatalogMicroservice* projeto usando IISExpress ou Docker, você terá a UI Swagger ao acessar <http://localhost:44326/>:



Microserviço Identidade

Agora, vamos passar para *IdentityMicroservice*.

IdentityMicroservice é responsável pela autenticação e gerenciamento de usuários.

Abaixo está o modelo usado por *IdentityMicroservice*:

```

public class User
{
    public static readonly string DocumentName = nameof(User);

    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]

```

```

public string? Id { get; init; }

public required string Email { get; init; }

public required string Password { get; set; }

public string? Salt { get; set; }

public bool IsAdmin { get; init; }


public void SetPassword(string password, IEncryptor encryptor)
{
    Salt = encryptor.GetSalt();

    Password = encryptor.GetHash(password, Salt);
}


public bool ValidatePassword(string password, IEncryptor encryptor) =>
    Password == encryptor.GetHash(password, Salt);
}

```

IEncryptormiddleware é usado para criptografar senhas.

Abaixo está a interface do repositório:

```

public interface IUserRepository
{
    User? GetUser(string email);

    void InsertUser(User user);
}

```

Abaixo está a implementação do repositório:

```

public class UserRepository(IMongoDatabase db) : IUserRepository
{
    private readonly IMongoCollection<User> _col =
        db.GetCollection<User>(User.DocumentName);
}

```



```
public User? GetUser(string email) =>
    _col.Find(u => u.Email == email).FirstOrDefault();
```

```
public void InsertUser(User user) =>
    _col.InsertOne(user);
}
```

Abaixo está o controlador:

```
[Route("api/[controller]")]
```

```
[ApiController]
```

```
public class IdentityController
```

```
(IUserRepository userRepository, IJwtBuilder jwtBuilder, IEncryptor encryptor)
```

```
: ControllerBase
```

```
{
```

```
[HttpPost("login")]
```

```
public IActionResult Login([FromBody] User user,
```

```
[FromQuery(Name = "d")] string destination = "frontend")
```

```
{
```

```
var u = userRepository.GetUser(user.Email);
```

```
if (u == null)
```

```
{
```

```
return NotFound("User not found.");
```

```
}
```

```
if (destination == "backend" && !u.IsAdmin)
```

```
{
```

```
return BadRequest("Could not authenticate user.");
```

```
}
```

```
var isValid = u.ValidatePassword(user.Password, encryptor);

if (!isValid)
{
    return BadRequest("Could not authenticate user.");
}

var token = jwtBuilder.GetToken(u.Id);

return Ok(token);
}
```

```
[HttpPost("register")]
public IActionResult Register([FromBody] User user)
{
    var u = userRepository.GetUser(user.Email);

    if (u != null)
    {
        return BadRequest("User already exists.");
    }

    user.SetPassword(user.Password, encryptor);
    userRepository.InsertUser(user);

    return Ok();
}
```

```

[HttpGet("validate")]
public IActionResult Validate([FromQuery(Name = "email")] string email,
                             [FromQuery(Name = "token")] string token)
{
    var u = userRepository.GetUser(email);

    if (u == null)
    {
        return NotFound("User not found.");
    }

    var userId = jwtBuilder.ValidateToken(token);

    if (userId != u.Id)
    {
        return BadRequest("Invalid token.");
    }

    return Ok(userId);
}
}

```

IUserRepository IJwtBuilder e IEncryptor middlewares são adicionados usando injeção de dependência em *Startup.com*:

// This method gets called by the runtime.

// Use this method to add services to the container.

```

public void ConfigureServices(IServiceCollection services)
{

```

```

services.AddControllers();

services.AddMongoDb(Configuration);

services.AddJwt(Configuration);

services.AddTransient<IEncryptor, Encryptor>();

services.AddSingleton<IUserRepository>(sp =>
    new UserRepository(sp.GetService<IMongoDatabase>() ??
        throw new Exception("IMongoDatabase not found"))
);

services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "User", Version = "v1" });
});

// ...
}

```

Abaixo está AddJwt método de extensão:

```

public static void AddJwt(this IServiceCollection services, IConfiguration
configuration)
{
    var options = new JwtOptions();

    var section = configuration.GetSection("jwt");

    section.Bind(options);

    services.Configure<JwtOptions>(section);

    services.AddSingleton<IJwtBuilder, JwtBuilder>();

    services.AddAuthentication()

        .AddJwtBearer(cfg =>
        {
            cfg.RequireHttpsMetadata = false;

```

```

    cfg.SaveToken = true;

    cfg.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = false,

        IssuerSigningKey = new SymmetricSecurityKey
            (Encoding.UTF8.GetBytes(options.Secret))

    };
});
}

```

IJwtBuilder é responsável por criar tokens JWT e validá-los:

```

public interface IJwtBuilder
{
    string GetToken(string userId);

    string ValidateToken(string token);
}

```

Abaixo está a implementação de IJwtBuilder:

```

public class JwtBuilder(IOptions<JwtOptions> options) : IJwtBuilder
{
    private readonly JwtOptions _options = options.Value;

    public string GetToken(string userId)
    {
        var signingKey = new SymmetricSecurityKey
            (Encoding.UTF8.GetBytes(_options.Secret));

        var signingCredentials = new SigningCredentials
            (signingKey, SecurityAlgorithms.HmacSha256);

        var claims = new[]
        {

```

```
        new Claim("userId", userId)
    };

    var expirationDate = DateTime.Now.AddMinutes(_options.ExpiryMinutes);
    var jwt = new JwtSecurityToken(claims: claims,
        signingCredentials: signingCredentials, expires: expirationDate);
    var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);

    return encodedJwt;
}
```

```
public string ValidateToken(string token)
{
    var principal = GetPrincipal(token);
    if (principal == null)
    {
        return string.Empty;
    }
}
```

```
ClaimsIdentity identity;
try
{
    identity = (ClaimsIdentity)principal.Identity;
}
catch (NullReferenceException)
{
    return string.Empty;
}

var userIdClaim = identity?.FindFirst("userId");
```

```
if (userIdClaim == null)
{
    return string.Empty;
}

var userId = userIdClaim.Value;

return userId;
}
```

```
private ClaimsPrincipal GetPrincipal(string token)
{
    try
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var jwtToken = (JwtSecurityToken)tokenHandler.ReadToken(token);
        if (jwtToken == null)
        {
            return null;
        }

        var key = Encoding.UTF8.GetBytes(_options.Secret);
        var parameters = new TokenValidationParameters()
        {
            RequireExpirationTime = true,
            ValidateIssuer = false,
            ValidateAudience = false,
            IssuerSigningKey = new SymmetricSecurityKey(key)
        };
        IdentityModelEventSource.ShowPII = true;

        ClaimsPrincipal principal =
```

```

        tokenHandler.ValidateToken(token, parameters, out _);
        return principal;
    }
    catch (Exception)
    {
        return null;
    }
}

```

IEncryptor é simplesmente responsável por criptografar senhas:

```

public interface IEncryptor
{
    string GetSalt();
    string GetHash(string value, string salt);
}

```

Abaixo está a implementação de IEncryptor:

```

public class Encryptor: IEncryptor
{
    private const int SALT_SIZE = 40;
    private const int ITERATIONS_COUNT = 10000;

    public string GetSalt()
    {
        var saltBytes = new byte[SALT_SIZE];
        var rng = RandomNumberGenerator.Create();
        rng.GetBytes(saltBytes);

        return Convert.ToBase64String(saltBytes);
    }
}

```



```
}
```

```
public string GetHash(string value, string salt)
{
    var pbkdf2 = new Rfc2898DeriveBytes
        (value, GetBytes(salt), ITERATIONS_COUNT, HashAlgorithmName.SHA256);

    return Convert.ToBase64String(pbkdf2.GetBytes(SALT_SIZE));
}
```

```
private static byte[] GetBytes(string value)
{
    var bytes = new byte[value.Length + sizeof(char)];
    Buffer.BlockCopy(value.ToCharArray(), 0, bytes, 0, bytes.Length);

    return bytes;
}
}
```

Abaixo está Configure método em *Startup.com*:

```
// This method gets called by the runtime.
```

```
// Use this method to configure the HTTP request pipeline.
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
}
```

```
app.UseSwagger();
```

```
app.UseSwaggerUI(c =>
```

```
{
```

```
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Catalog V1");
```

```
});
```

```
var option = new RewriteOptions();
```

```
option.AddRedirect("^$", "swagger");
```

```
app.UseRewriter(option);
```

```
// ...
```

```
app.UseHttpsRedirection();
```

```
app.UseRouting();
```

```
app.UseAuthorization();
```

```
app.UseEndpoints(endpoints =>
```

```
{
```

```
    endpoints.MapControllers();
```

```
});
```

```
}
```

Abaixo está *appsettings.com*:

```
{
```

```
  "Logging": {
```

```
    "LogLevel": {
```

```
"Default": "Information",
"Microsoft": "Warning",
"Microsoft.Hosting.Lifetime": "Information"
},
"AllowedHosts": "*",
"mongo": {
  "connectionString": "mongodb://127.0.0.1:27017",
  "database": "store-identity"
},
"jwt": {
  "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103",
  "expiryMinutes": 60
}
}
```

Agora, vamos testar IdentityMicroservice.

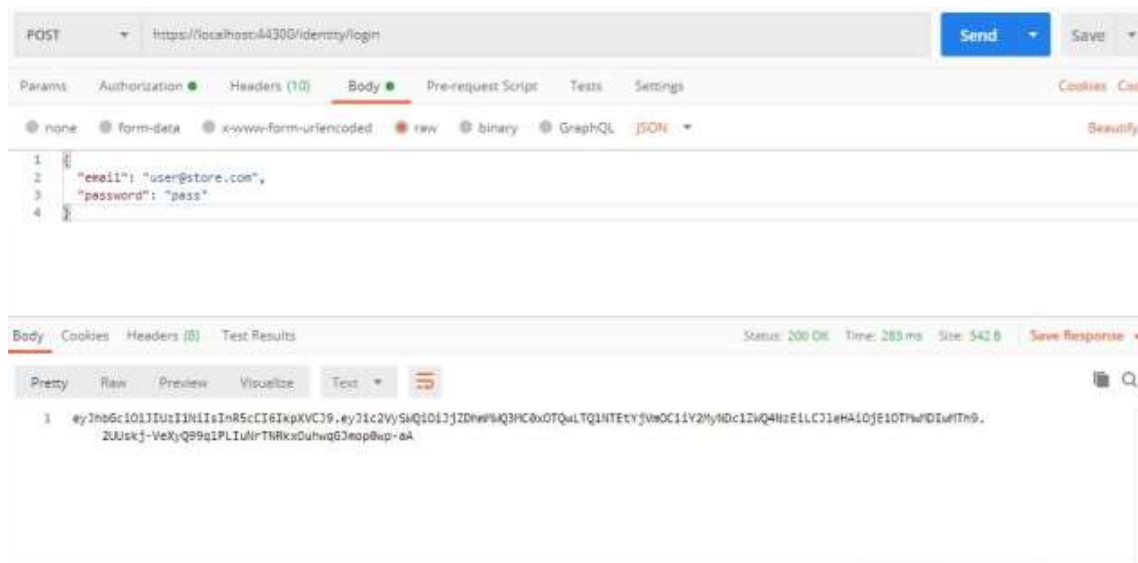
Abra o Postman e execute o seguinte POST pedido <http://localhost:44397/api/identity/register> com a seguinte carga útil para registar um utilizador:

```
{
  "email": "user@store.com",
  "password": "pass"
}
```

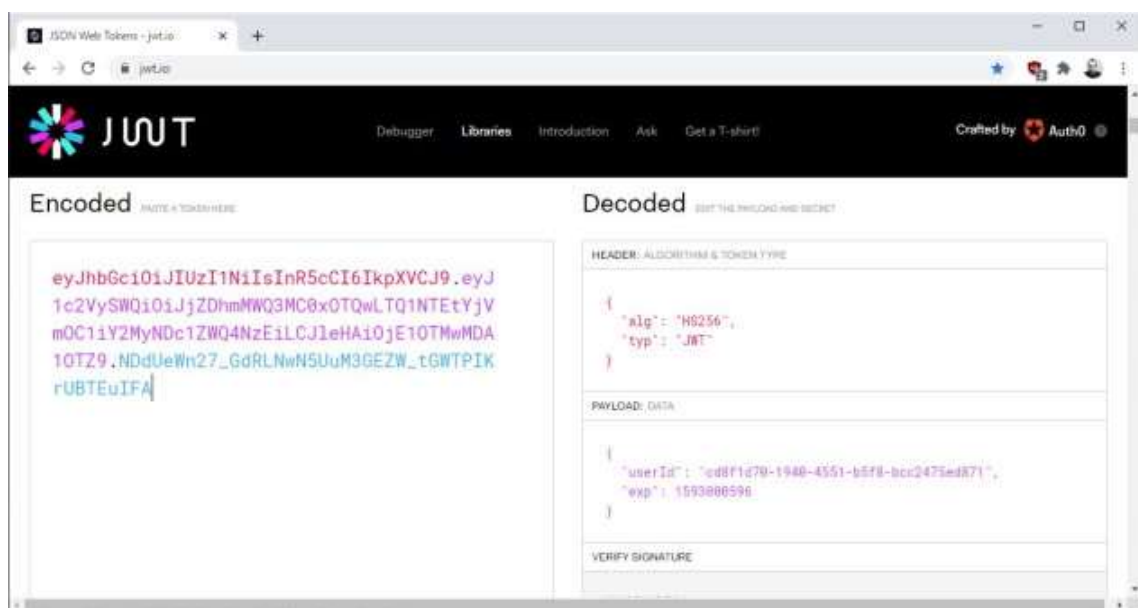
Agora, execute o seguinte POST pedido <http://localhost:44397/api/identity/login> com a seguinte carga útil para criar um token JWT:

```
{
  "email": "user@store.com",
  "password": "pass"
}
```

}

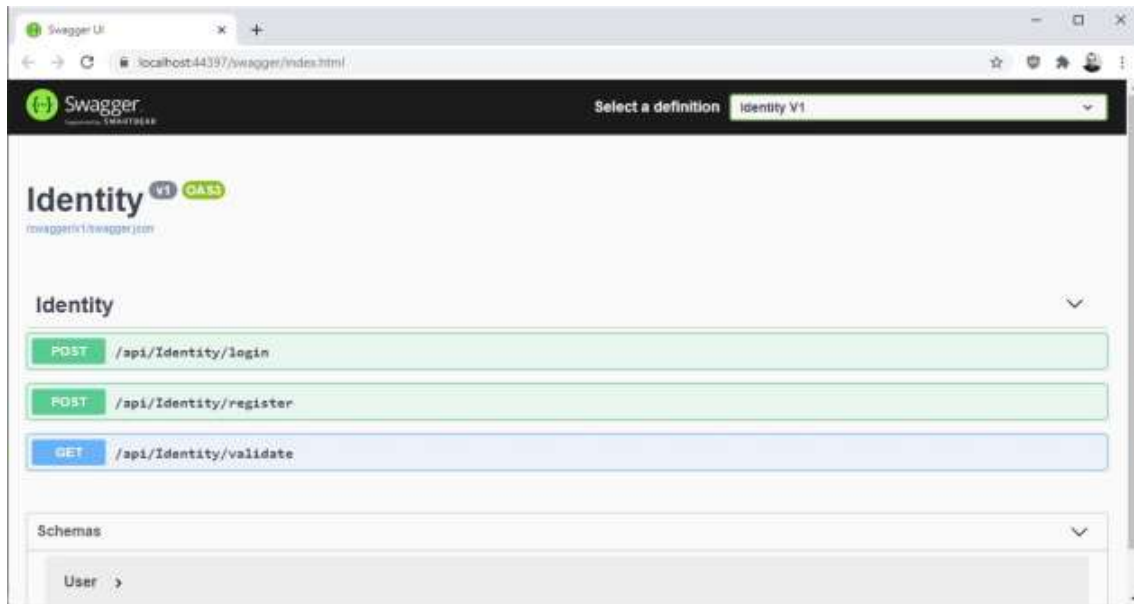


Você pode então verificar o token gerado jwt.io:



É isso. Você pode executar o seguinte GET pedido <http://localhost:44397/api/identity/validate?email={email}&token={token}> da mesma forma para validar um token JWT. Se o token for válido, a resposta será o ID do usuário, que é um ObjectId.

Se você correr IdentityMicroservice projeto usando IISExpress ou Docker, você receberá a UI Swagger ao acessar <http://localhost:44397/>:



Adicionando JWT ao Microservice do Catálogo

Agora, vamos adicionar a autenticação JWT ao microserviço de catálogo.

Primeiro, temos que acrescentar jwt seção em *appsettings.com*:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  "AllowedHosts": "*",  
  "jwt": {  
    "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103"  
  },  
  "mongo": {  
    "connectionString": "mongodb://127.0.0.1:27017",  
    "database": "store-catalog"  
  }  
}
```

```
}
```

Então, temos que adicionar a configuração JWT em ConfigureServices método em *Startup.com*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddJwtAuthentication(Configuration); // JWT Configuration

    // ...
}
```

Onde AddJwtAuthentication o método de extensão é implementado da seguinte forma:

```
public static void AddJwtAuthentication
    (this IServiceCollection services, IConfiguration configuration)
{
    var section = configuration.GetSection("jwt");
    var options = section.Get<JwtOptions>();
    var key = Encoding.UTF8.GetBytes(options.Secret);
    section.Bind(options);
    services.Configure<JwtOptions>(section);

    services.AddSingleton<IJwtBuilder, JwtBuilder>();
    services.AddTransient<JwtMiddleware>();

    services.AddAuthentication(x =>
    {
        x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    });
}
```

```

    })
    .AddJwtBearer(x =>
    {
        x.RequireHttpsMetadata = false;
        x.SaveToken = true;
        x.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });

```

```

services.AddAuthorization(x =>
{
    x.DefaultPolicy = new AuthorizationPolicyBuilder()
        .AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme)
        .RequireAuthenticatedUser()
        .Build();
});
}

```

JwtMiddleware é responsável por validar o token JWT:

```

public class JwtMiddleware(IJwtBuilder jwtBuilder) : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        // Get the token from the Authorization header
    }
}

```

```

var bearer = context.Request.Headers["Authorization"].ToString();
var token = bearer.Replace("Bearer ", string.Empty);

if (!string.IsNullOrEmpty(token))
{
    // Verify the token using the IJwtBuilder
    var userId = jwtBuilder.ValidateToken(token);

    if (ObjectId.TryParse(userId, out _))
    {
        // Store the userId in the HttpContext items for later use
        context.Items["userId"] = userId;
    }
    else
    {
        // If token or userId are invalid, send 401 Unauthorized status
        context.Response.StatusCode = 401;
    }
}

// Continue processing the request
await next(context);
}
}

```

Se o token JWT ou o ID do usuário forem inválidos, enviaremos **401 Status não autorizado**.

Então, nós registramos JwtMiddleware em Configure método em *Startup.com*:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```



```

{
    // ...

    app.UseMiddleware<JwtMiddleware>(); // JWT Middleware

    app.UseAuthentication();

    app.UseAuthorization();

    // ...
}

```

Em seguida, especificamos que exigimos autenticação JWT para nossos endpoints em *CatálogoController.cs* através [Authorize] atributo:

```

// GET: api/<CatalogController>

[HttpGet]
[Authorize]
public IActionResult Get()
{
    var catalogItems = _catalogRepository.GetCatalogItems();
    return Ok(catalogItems);
}

// ...

```

Agora, o microserviço de catálogo é protegido por meio da autenticação JWT. O microserviço de carrinho foi protegido da mesma maneira.

Finalmente, precisamos adicionar a autenticação JWT ao Swagger. Para isso, precisamos atualizar AddSwaggerGen em ConfigureServices em *Startup.com*:

```

public void ConfigureServices(IServiceCollection services)
{

```

```
//...
```

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Catalog", Version = "v1" });
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
        Description = "Please insert JWT token with the prefix Bearer into field",
        Name = "Authorization",
        Type = SecuritySchemeType.ApiKey,
        Scheme = "bearer",
        BearerFormat = "JWT"
    });
    c.AddSecurityRequirement(new OpenApiSecurityRequirement {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            new string[] { }
        }
    });
});
```

```
//...  
}
```

Agora, se você quiser executar o Postman em microserviços de catálogo ou carrinho, você precisa especificar o **Token Portador** em **Autorização** guia.

Se você quiser testar microserviços de catálogo ou carrinho com o Swagger, clique em **Autorizar** botão e digite o token JWT com o prefixo Bearer no campo de autorização.

Carrinho Microservice

CartMicroservice é responsável pela gestão do carrinho.

Abaixo estão os modelos utilizados por CartMicroservice:

```
public class Cart  
{  
    public static readonly string DocumentName = nameof(Cart);  
  
    [BsonId]  
    [BsonRepresentation(BsonType.ObjectId)]  
    public string? Id { get; init; }  
  
    [BsonRepresentation(BsonType.ObjectId)]  
    public string? UserId { get; init; }  
  
    public List<CartItem> CartItems { get; init; } = new();  
}
```

```
public class CartItem  
{  
    [BsonRepresentation(BsonType.ObjectId)]  
    public string? CatalogItemId { get; init; }  
  
    public required string Name { get; set; }  
  
    public decimal Price { get; set; }
```

```

    public int Quantity { get; set; }
}

```

Abaixo está a interface do repositório:

```

public interface ICartRepository
{
    IList<CartItem> GetCartItems(string userId);
    void InsertCartItem(string userId, CartItem cartItem);
    void UpdateCartItem(string userId, CartItem cartItem);
    void DeleteCartItem(string userId, string cartItemId);
    void UpdateCatalogItem(string catalogItemId, string name, decimal price);
    void DeleteCatalogItem(string catalogItemId);
}

```

Abaixo está o repositório:

```

public class CartRepository(IMongoDatabase db) : ICartRepository
{
    private readonly IMongoCollection<Cart> _col =
        db.GetCollection<Cart>(Cart.DocumentName);

    public IList<CartItem> GetCartItems(string userId) =>
        _col
            .Find(c => c.UserId == userId)
            .FirstOrDefault()?.CartItems ?? new List<CartItem>();

    public void InsertCartItem(string userId, CartItem cartItem)
    {
        var cart = _col.Find(c => c.UserId == userId).FirstOrDefault();
        if (cart == null)
        {

```

```

        cart = new Cart
        {
            UserId = userId,
            CartItems = new List<CartItem> { cartItem }
        };
        _col.InsertOne(cart);
    }
    else
    {
        var ci = cart
            .CartItems
            .FirstOrDefault(ci => ci.CatalogItemId == cartItem.CatalogItemId);

        if (ci == null)
        {
            cart.CartItems.Add(cartItem);
        }
        else
        {
            ci.Quantity++;
        }

        var update = Builders<Cart>.Update
            .Set(c => c.CartItems, cart.CartItems);
        _col.UpdateOne(c => c.UserId == userId, update);
    }
}

```

```

public void UpdateCartItem(string userId, CartItem cartItem)
{
    var cart = _col.Find(c => c.UserId == userId).FirstOrDefault();
    if (cart != null)
    {
        cart.CartItems.RemoveAll(ci => ci.CatalogItemId == cartItem.CatalogItemId);
        cart.CartItems.Add(cartItem);
        var update = Builders<Cart>.Update
            .Set(c => c.CartItems, cart.CartItems);
        _col.UpdateOne(c => c.UserId == userId, update);
    }
}

```

```

public void DeleteCartItem(string userId, string catalogItemId)
{
    var cart = _col.Find(c => c.UserId == userId).FirstOrDefault();
    if (cart != null)
    {
        cart.CartItems.RemoveAll(ci => ci.CatalogItemId == catalogItemId);
        var update = Builders<Cart>.Update
            .Set(c => c.CartItems, cart.CartItems);
        _col.UpdateOne(c => c.UserId == userId, update);
    }
}

```

```

public void UpdateCatalogItem(string catalogItemId, string name, decimal price)
{
    // Update catalog item in carts

```

```

var carts = GetCarts(catalogItemId);

foreach (var cart in carts)
{
    var cartItem = cart.CartItems.FirstOrDefault
        (i => i.CatalogItemId == catalogItemId);

    if (cartItem != null)
    {
        cartItem.Name = name;

        cartItem.Price = price;

        var update = Builders<Cart>.Update
            .Set(c => c.CartItems, cart.CartItems);

        _col.UpdateOne(c => c.Id == cart.Id, update);
    }
}

```

```

public void DeleteCatalogItem(string catalogItemId)
{
    // Delete catalog item from carts

    var carts = GetCarts(catalogItemId);

    foreach (var cart in carts)
    {
        cart.CartItems.RemoveAll(i => i.CatalogItemId == catalogItemId);

        var update = Builders<Cart>.Update
            .Set(c => c.CartItems, cart.CartItems);

        _col.UpdateOne(c => c.Id == cart.Id, update);
    }
}

```

```

private IList<Cart> GetCarts(string catalogItemId) =>
    _col.Find(c => c.CartItems.Any(i => i.CatalogItemId == catalogItemId)).ToList();
}

```

Abaixo está o controlador:

```
[Route("api/[controller]")]
```

```
[ApiController]
```

```
public class CartController(ICartRepository cartRepository) : ControllerBase
```

```
{
```

```
    // GET: api/<CartController>
```

```
    [HttpGet]
```

```
    [Authorize]
```

```
    public IActionResult Get([FromQuery(Name = "u")] string userId)
```

```
    {
```

```
        var cartItems = cartRepository.GetCartItems(userId);
```

```
        return Ok(cartItems);
```

```
    }
```

```
    // POST api/<CartController>
```

```
    [HttpPost]
```

```
    [Authorize]
```

```
    public IActionResult Post([FromQuery(Name = "u")] string userId,
```

```
        [FromBody] CartItem cartItem)
```

```
    {
```

```
        cartRepository.InsertCartItem(userId, cartItem);
```

```
        return Ok();
```

```
    }
```



```
// PUT api/<CartController>

[HttpPut]
[Authorize]
public IActionResult Put([FromQuery(Name = "u")] string userId,
                        [FromBody] CartItem cartItem)
{
    cartRepository.UpdateCartItem(userId, cartItem);
    return Ok();
}
```

```
// DELETE api/<CartController>

[HttpDelete]
[Authorize]
public IActionResult Delete([FromQuery(Name = "u")] string userId,
                        [FromQuery(Name = "ci")] string cartItemId)
{
    cartRepository.DeleteCartItem(userId, cartItemId);
    return Ok();
}
```

```
// PUT api/<CartController>/update-catalog-item

[HttpPut("update-catalog-item")]
[Authorize]
public IActionResult Put([FromQuery(Name = "ci")] string catalogItemId,
                        [FromQuery(Name = "n")] string name, [FromQuery(Name = "p")] decimal price)
{
    cartRepository.UpdateCatalogItem(catalogItemId, name, price);
    return Ok();
}
```

```
}
```

```
// DELETE api/<CartController>/delete-catalog-item
```

```
[HttpDelete("delete-catalog-item")]
```

```
[Authorize]
```

```
public IActionResult Delete([FromQuery(Name = "ci")] string catalogItemId)
```

```
{
```

```
    cartRepository.DeleteCatalogItem(catalogItemId);
```

```
    return Ok();
```

```
}
```

```
}
```

ICartRepository é adicionado usando injeção de dependência em *Startup.com* para tornar o microservice testável:

```
public void ConfigureServices(IServiceCollection services)
```

```
{
```

```
    services.AddControllers();
```

```
    services.AddJwtAuthentication(Configuration); // JWT Configuration
```

```
    services.AddMongoDb(Configuration);
```

```
    services.AddSingleton<ICartRepository>(sp =>
```

```
        new CartRepository(sp.GetService<IMongoDatabase>>() ??
```

```
            throw new Exception("IMongoDatabase not found"))
```

```
);
```

```
// ...
```

```
}
```

Configure método em *Startup.com* é o mesmo que o de *CatalogMicroservice*.

Abaixo está *appsettings.com*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "jwt": {
    "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103"
  },
  "mongo": {
    "connectionString": "mongodb://127.0.0.1:27017",
    "database": "store-cart"
  }
}
```

A documentação da API é gerada usando o Swashbuckle. O middleware do Swagger está configurado em *Startup.com*, em *ConfigureServices* e *Configure* métodos em *Startup.cs*.

Se você correr *CartMicroservice* projeto usando *IISExpress* ou *Docker*, você terá a UI Swagger ao acessar <http://localhost:44388/>.

Gateways API

Existem dois gateways API, um para o frontend e outro para o backend.

Vamos começar com o frontend.

ocelot.json o arquivo de configuração foi adicionado em *Programa.cs* da seguinte forma:

```

var builder = Host.CreateDefaultBuilder(args)

    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        config
            .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
            .AddJsonFile("appsettings.json", true, true)
            .AddJsonFile("ocelot.json", false, true)
            .AddJsonFile($"appsettings.
{hostingContext.HostingEnvironment.EnvironmentName}.json", true, true)
            .AddJsonFile($"ocelot.
{hostingContext.HostingEnvironment.EnvironmentName}.json",
                optional: false, reloadOnChange: true)
            .AddEnvironmentVariables();

        if (hostingContext.HostingEnvironment.EnvironmentName ==
"Development")
        {
            config.AddJsonFile("appsettings.Local.json", true, true);
        }
    })

    .UseSerilog((_, config) =>
    {
        config
            .MinimumLevel.Information()
            .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
            .Enrich.FromLogContext()
            .WriteTo.Console();
    })

```

```

        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });

builder.Build().Run();

O Serilog está configurado para gravar logs no console. Você pode, é claro,
escrever logs para arquivos de texto
usando WriteTo.File(@"Logs\store.log") e Serilog.Sinks.File pacote nuget.

Então, aqui está Startup.com:

public class Startup(IConfiguration configuration)
{
    private IConfiguration Configuration { get; } = configuration;

    // This method gets called by the runtime.
    // Use this method to add services to the container.

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();

        services.AddOcelot(Configuration);

        services.AddJwtAuthentication(Configuration); // JWT Configuration

        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy",
                builder => builder.AllowAnyOrigin()
                    .AllowAnyMethod()

```

```

        .AllowAnyHeader());
    });

    services.AddHealthChecks()
        .AddMongoDb(
            mongodbConnectionString: (
                Configuration.GetSection("mongo").Get<MongoOptions>()
                ?? throw new Exception("mongo configuration section not found")
            ).ConnectionString,
            name: "mongo",
            failureStatus: HealthStatus.Unhealthy
        );

    services.AddHealthChecksUI().AddInMemoryStorage();
}

// This method gets called by the runtime. Use this method
// to configure the HTTP request pipeline.
public async void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseMiddleware<RequestResponseLogging>();

    app.UseCors("CorsPolicy");

```

```
app.UseAuthentication();
```

```
app.UseHealthChecks("/healthz", new HealthCheckOptions  
{  
    Predicate = _ => true,  
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse  
});
```

```
app.UseHealthChecksUI();
```

```
var option = new RewriteOptions();  
option.AddRedirect("^$", "healthchecks-ui");  
app.UseRewriter(option);
```

```
await app.UseOcelot();  
}  
}
```

RequestResponseLogging middleware é responsável por registrar solicitações e respostas:

```
public class RequestResponseLogging(RequestDelegate next,  
ILogger<RequestResponseLogging> logger)  
{  
    public async Task InvokeAsync(HttpContext context)  
    {  
        context.Request.EnableBuffering();  
        var builder = new StringBuilder();  
        var request = await FormatRequest(context.Request);  
        builder.Append("Request: ").AppendLine(request);
```

```

builder.AppendLine("Request headers:");

foreach (var header in context.Request.Headers)
{
    builder.Append(header.Key).Append(": ").AppendLine(header.Value);
}

var originalBodyStream = context.Response.Body;
using var responseBody = new MemoryStream();
context.Response.Body = responseBody;
await next(context);

var response = await FormatResponse(context.Response);
builder.Append("Response: ").AppendLine(response);
builder.AppendLine("Response headers: ");

foreach (var header in context.Response.Headers)
{
    builder.Append(header.Key).Append(": ").AppendLine(header.Value);
}

logger.LogInformation(builder.ToString());

await responseBody.CopyToAsync(originalBodyStream);
}

private static async Task<string> FormatRequest(HttpRequest request)
{

```



```

using var reader = new StreamReader(
    request.Body,
    encoding: Encoding.UTF8,
    detectEncodingFromByteOrderMarks: false,
    leaveOpen: true);
var body = await reader.ReadToEndAsync();
var formattedRequest = $"{request.Method}
{request.Scheme}://{request.Host}{request.Path}{request.QueryString}
{body}";
request.Body.Position = 0;
return formattedRequest;
}

private static async Task<string> FormatResponse(HttpResponse response)
{
    response.Body.Seek(0, SeekOrigin.Begin);
    string text = await new StreamReader(response.Body).ReadToEndAsync();
    response.Body.Seek(0, SeekOrigin.Begin);
    return $"{response.StatusCode}: {text}";
}
}

```

Usamos o log no gateway para não precisarmos verificar os logs de cada microserviço.

Aqui está *ocelot.Desenvolvimento.json*:

```

{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/catalog",
      "DownstreamScheme": "http",

```

```
"DownstreamHostAndPorts": [  
  {  
    "Host": "localhost",  
    "Port": 44326  
  }  
,  
  "UpstreamPathTemplate": "/catalog",  
  "UpstreamHttpMethod": [ "GET" ],  
  "AuthenticationOptions": {  
    "AuthenticationProviderKey": "Bearer",  
    "AllowedScopes": []  
  }  
},  
{  
  "DownstreamPathTemplate": "/api/catalog/{id}",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 44326  
    }  
  ],  
  "UpstreamPathTemplate": "/catalog/{id}",  
  "UpstreamHttpMethod": [ "GET" ],  
  "AuthenticationOptions": {  
    "AuthenticationProviderKey": "Bearer",  
    "AllowedScopes": []  
  }  
}
```

```
},  
  
{  
  "DownstreamPathTemplate": "/api/cart",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 44388  
    }  
  ],  
  "UpstreamPathTemplate": "/cart",  
  "UpstreamHttpMethod": [ "GET" ],  
  "AuthenticationOptions": {  
    "AuthenticationProviderKey": "Bearer",  
    "AllowedScopes": []  
  }  
},  
  
{  
  "DownstreamPathTemplate": "/api/cart",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 44388  
    }  
  ],  
  "UpstreamPathTemplate": "/cart",  
  "UpstreamHttpMethod": [ "POST" ],
```

```
"AuthenticationOptions": {
  "AuthenticationProviderKey": "Bearer",
  "AllowedScopes": []
},
{
  "DownstreamPathTemplate": "/api/cart",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "localhost",
      "Port": 44388
    }
  ],
  "UpstreamPathTemplate": "/cart",
  "UpstreamHttpMethod": [ "PUT" ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer",
    "AllowedScopes": []
  },
  {
    "DownstreamPathTemplate": "/api/cart",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      {
        "Host": "localhost",
        "Port": 44388
      }
    ]
  }
}
```

```
}  
],  
"UpstreamPathTemplate": "/cart",  
"UpstreamHttpMethod": [ "DELETE" ],  
"AuthenticationOptions": {  
  "AuthenticationProviderKey": "Bearer",  
  "AllowedScopes": []  
}  
},  
{  
  "DownstreamPathTemplate": "/api/identity/login",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 44397  
    }  
  ],  
  "UpstreamPathTemplate": "/identity/login",  
  "UpstreamHttpMethod": [ "POST" ]  
},  
{  
  "DownstreamPathTemplate": "/api/identity/register",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    {  
      "Host": "localhost",  
      "Port": 44397
```

```

    }
  ],
  "UpstreamPathTemplate": "/identity/register",
  "UpstreamHttpMethod": [ "POST" ]
},
{
  "DownstreamPathTemplate": "/api/identity/validate",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "localhost",
      "Port": 44397
    }
  ],
  "UpstreamPathTemplate": "/identity/validate",
  "UpstreamHttpMethod": [ "GET" ]
}
],
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:44300/"
}
}

```

E finalmente, abaixo está *appsettings.com*:

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",

```

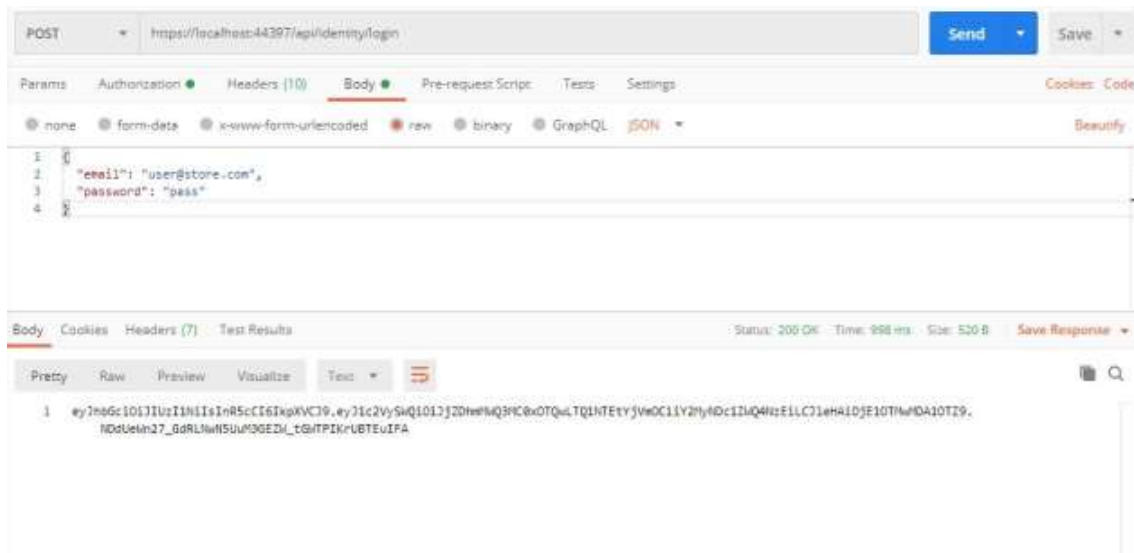
```
"Microsoft.Hosting.Lifetime": "Information"
}
},
"AllowedHosts": "*",
"jwt": {
  "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103"
},
"mongo": {
  "connectionString": "mongodb://127.0.0.1:27017"
}
}
```

Agora, vamos testar o gateway frontend.

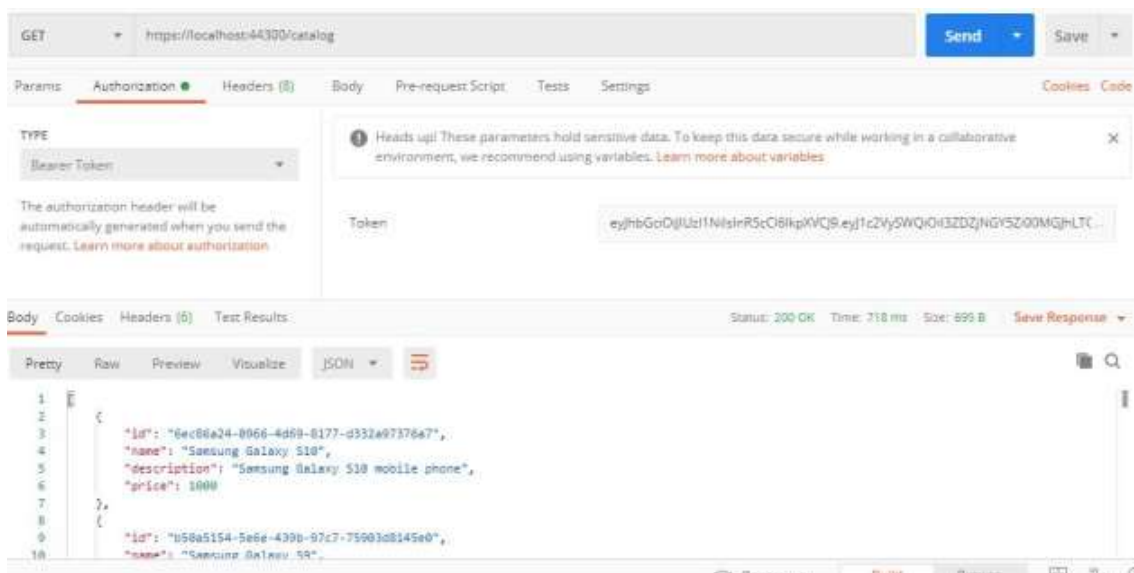
Primeiro, execute o seguinte POST pedido <http://localhost:44300/identidade/login> com a seguinte carga útil para criar um token JWT:

```
{
  "email": "user@store.com",
  "password": "pass"
}
```

Nós já criamos esse usuário durante o teste IdentityMicroservice. Se você não criou esse usuário, você pode criar um executando o seguinte POST pedido <http://localhost:44300/identity/register> com a mesma carga acima.

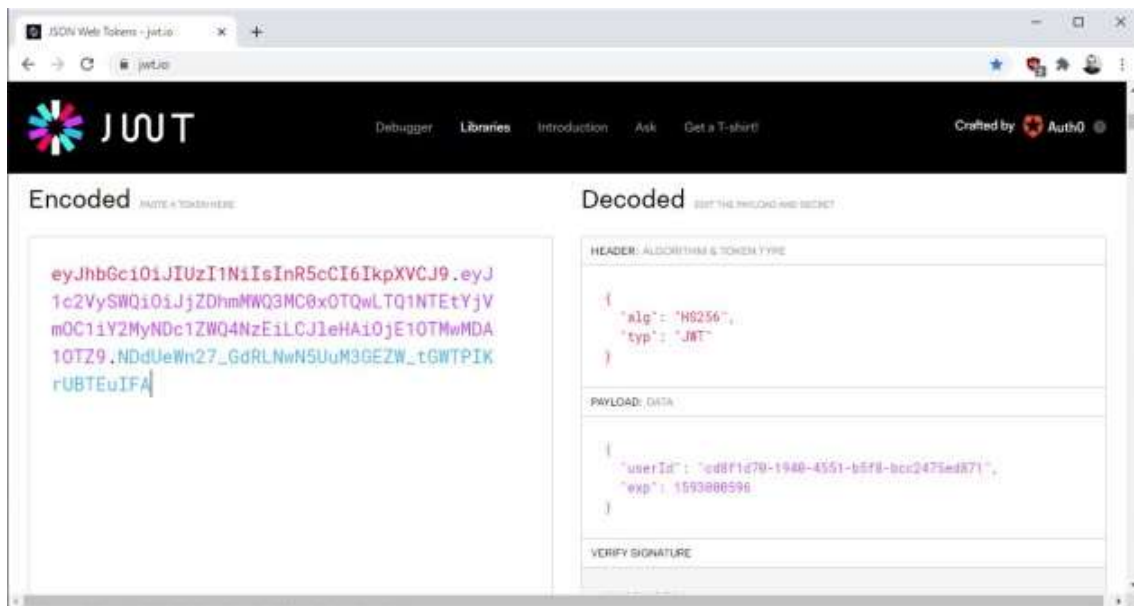


Em seguida, vá para a guia Autorização no Postman, selecione **Token Portador** digite e copie o token JWT **Token** campo. Em seguida, execute o seguinte GET pedido para recuperar o catálogo <http://localhost:44300/catalog>:

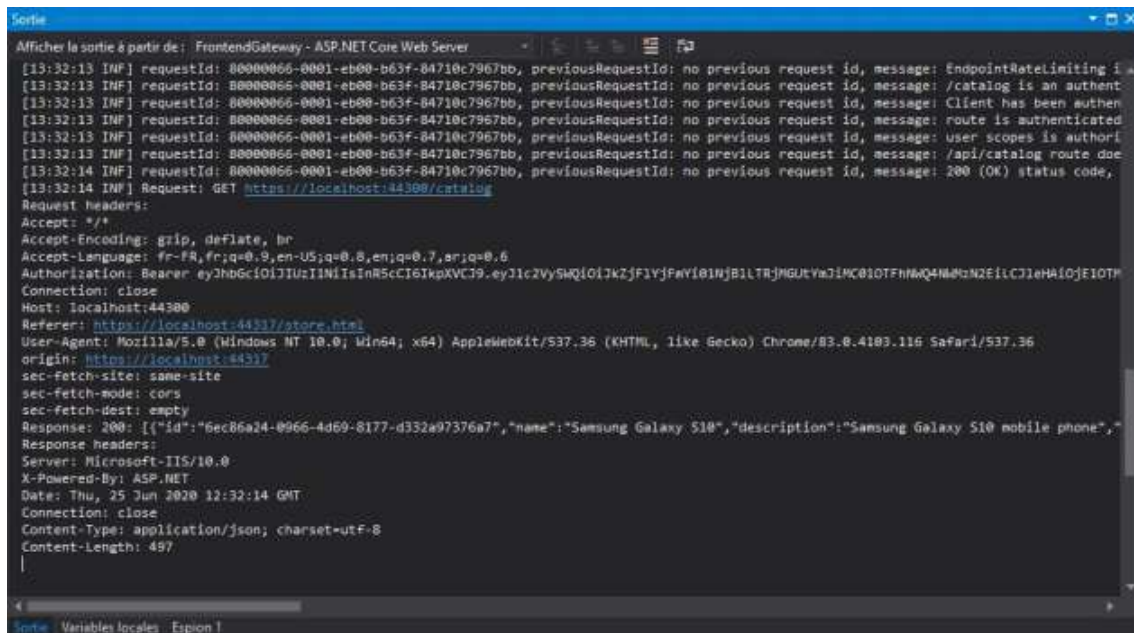


Se o token JWT não for válido, a resposta será **401 Não autorizado**.

Você pode verificar os tokens em jwt.io:



Se abirmos o console no Visual Studio, poderemos ver todos os logs:



É isso! Você pode testar os outros métodos de API da mesma maneira.

O gateway de back-end é feito praticamente da mesma maneira. A única diferença está em `ocelot.json` arquivo.

Aplicativos Cliente

Existem dois aplicativos clientes. Um para o frontend e outro para o backend.

Os aplicativos cliente são feitos usando HTML e Vanilla JavaScript por uma questão de simplicidade.

Vamos escolher a página de login do frontend, por exemplo. Aqui está o HTML:

```
<!DOCTYPE html>

<html>

<head>

  <meta charset="utf-8" />

  <title>Login</title>

  <link rel="icon" href="data:,">

  <link href="css/bootstrap.min.css" rel="stylesheet" />

  <link href="css/login.css" rel="stylesheet" />

</head>

<body>

  <div class="header"></div>

  <div class="login">

    <table>

      <tr>

        <td>Email</td>

        <td><input id="email" type="text" autocomplete="off"

          class="form-control" /></td>

      </tr>

      <tr>

        <td>Password</td>

        <td><input id="password" type="password" class="form-control" /></td>

      </tr>

      <tr>

        <td></td>

        <td>

          <input id="login" type="button" value="Login"

            class="btn btn-primary" />

          <input id="register" type="button" value="Register">

        </td>

      </tr>

    </table>

  </div>

</body>

</html>
```

```

        class="btn btn-secondary" />

    </td>

</tr>

</table>

</div>

<script src="js/login.js" type="module"></script>

</body>

</html>

```

Aqui está *configurações.js*:

```

export default {
  uri: "http://localhost:44300/"
};

```

E aqui está *login.js*:

```

import settings from "./settings.js";
import common from "./common.js";

```

```

window.onload = () => {

```

```

  "use strict";

```

```

  localStorage.removeItem("auth");

```

```

  function login() {

```

```

    const user = {

```

```

      "email": document.getElementById("email").value,

```

```

      "password": document.getElementById("password").value

```

```

    };

```

```

    common.post(settings.uri + "identity/login?d=frontend", (token) => {

```

```

const auth = {
  "email": user.email,
  "token": token
};

localStorage.setItem("auth", JSON.stringify(auth));

location.href = "/store.html";

}, () => {
  alert("Wrong credentials.");
}, user);
};

```

```

document.getElementById("login").onclick = () => {
  login();
};

```

```

document.getElementById("password").onkeyup = (e) => {
  if (e.key === 'Enter') {
    login();
  }
};

```

```

document.getElementById("register").onclick = () => {
  location.href = "/register.html";
};
};

```

common.js contém funções para execução GET POST e DELETE pedidos:

```

export default {
  post: async (url, callback, errorCallback, content, token) => {

```

```
try {  
  const headers = {  
    "Content-Type": "application/json;charset=UTF-8"  
  };  
  if (token) {  
    headers["Authorization"] = `Bearer ${token}`;  
  }  
  const response = await fetch(url, {  
    method: "POST",  
    headers,  
    body: JSON.stringify(content)  
  });  
  if (response.ok) {  
    const data = await response.text();  
    if (callback) {  
      callback(data);  
    }  
  } else {  
    if (errorCallback) {  
      errorCallback(response.status);  
    }  
  }  
} catch (err) {  
  if (errorCallback) {  
    errorCallback(err);  
  }  
}  
},
```

```
get: async (url, callback, errorCallback, token) => {  
  try {  
    const headers = {  
      "Content-Type": "application/json;charset=UTF-8"  
    };  
    if (token) {  
      headers["Authorization"] = `Bearer ${token}`;  
    }  
    const response = await fetch(url, {  
      method: "GET",  
      headers  
    });  
    if (response.ok) {  
      const data = await response.text();  
  
      if (callback) {  
        callback(data);  
      }  
    } else {  
      if (errorCallback) {  
        errorCallback(response.status);  
      }  
    }  
  } catch (err) {  
    if (errorCallback) {  
      errorCallback(err);  
    }  
  }  
}
```

```

},
delete: async (url, callback, errorCallback, token) => {
  try {
    const headers = {
      "Content-Type": "application/json;charset=UTF-8"
    };
    if (token) {
      headers["Authorization"] = `Bearer ${token}`;
    }
    const response = await fetch(url, {
      method: "DELETE",
      headers
    });

    if (response.ok) {
      if (callback) {
        callback();
      }
    } else {
      if (errorCallback) {
        errorCallback(response.status);
      }
    }
  } catch (err) {
    if (errorCallback) {
      errorCallback(err);
    }
  }
}

```

```
}
```

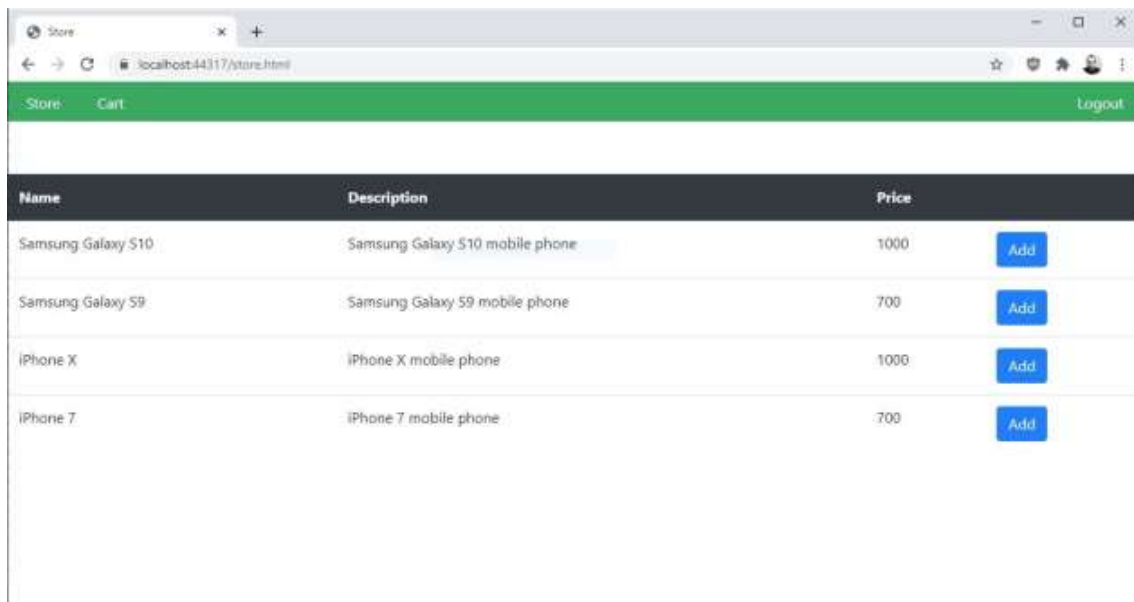
```
};
```

As outras páginas no frontend e no backend são feitas praticamente da mesma maneira.

No frontend, há quatro páginas. Uma página de login, uma página para registrar usuários, uma página para acessar a loja e uma página para acessar o carrinho.

O frontend permite que os usuários registrados vejam os itens de catálogo disponíveis, adicionem itens de catálogo ao carrinho e removam itens de catálogo do carrinho.

Aqui está uma captura de tela da página da loja no frontend:



Name	Description	Price
Samsung Galaxy S10	Samsung Galaxy S10 mobile phone	1000
Samsung Galaxy S9	Samsung Galaxy S9 mobile phone	700
iPhone X	iPhone X mobile phone	1000
iPhone 7	iPhone 7 mobile phone	700

No backend, há duas páginas. Uma página de login e uma página para gerenciar a loja.

O back-end permite que os usuários administradores vejam os itens de catálogo disponíveis, criem novos itens de catálogo, atualizem itens de catálogo e removam itens de catálogo.

Aqui está uma captura de tela da página da loja no backend:

Name	Description	Price	
Samsung Galaxy S10	Samsung Galaxy S10 mobile phone	1000	<button>Update</button> <button>Remove</button>
Samsung Galaxy S9	Samsung Galaxy S9 mobile phone	700	<button>Update</button> <button>Remove</button>
iPhone X	iPhone X mobile phone	1000	<button>Update</button> <button>Remove</button>
iPhone 7	iPhone 7 mobile phone	700	<button>Update</button> <button>Remove</button>

Testes de Unidade

Nesta seção, testaremos todos os microserviços usando xUnit e Moq.

Quando a lógica do controlador de teste de unidade, apenas o conteúdo de uma única ação é testado, não o comportamento de suas dependências ou da própria estrutura.

o XUnit simplifica o processo de teste e nos permite gastar mais tempo nos concentrando em escrever nossos testes.

Moq é uma estrutura de simulação popular e amigável para .NET. Vamos usá-lo para simular repositórios e serviços de middleware.

Para o microserviço de catálogo de teste unitário, primeiro um projeto de teste xUnit CatalogMicroservice.UnitTests foi criado. Em seguida, uma classe de teste de unidade CatalogControllerTest foi criado. Esta classe contém métodos de teste unitários do controlador de catálogo.

Uma referência do projeto CatalogMicroservice foi adicionado a CatalogMicroservice.UnitTests projeto.

Em seguida, o Moq foi adicionado usando o gerenciador de pacotes Nuget. Neste ponto, podemos começar a nos concentrar em escrever nossos testes.

Uma referência de CatalogController foi adicionado a CatalogControllerTest:

```
private readonly CatalogController _controller;
```

Então, no construtor de nossa classe de teste unitário, um repositório mock foi adicionado da seguinte forma:

```
public CatalogControllerTest()
```

```

{
    var mockRepo = new Mock<ICatalogRepository>();
    mockRepo.Setup(repo => repo.GetCatalogItems()).Returns(_items);
    mockRepo.Setup(repo => repo.GetCatalogItem(It.IsAny<string>()))
        .Returns<string>(id => _items.FirstOrDefault(i => i.Id == id));
    mockRepo.Setup(repo => repo.InsertCatalogItem(It.IsAny<CatalogItem>()))
        .Callback<CatalogItem>(_items.Add);
    mockRepo.Setup(repo => repo.UpdateCatalogItem(It.IsAny<CatalogItem>()))
        .Callback<CatalogItem>(i =>
        {
            var item = _items.FirstOrDefault(catalogItem => catalogItem.Id == i.Id);
            if (item != null)
            {
                item.Name = i.Name;
                item.Description = i.Description;
                item.Price = i.Price;
            }
        });
    mockRepo.Setup(repo => repo.DeleteCatalogItem(It.IsAny<string>()))
        .Callback<string>(id => _items.RemoveAll(i => i.Id == id));
    _controller = new CatalogController(mockRepo.Object);
}

```

onde _items é uma lista de CatalogItem:

```
private static readonly string A54Id = "653e4410614d711b7fc953a7";
```

```
private static readonly string A14Id = "253e4410614d711b7fc953a7";
```

```
private readonly List<CatalogItem> _items = new()
```

```

{
    new()

```

```

{
    Id = A54Id,
    Name = "Samsung Galaxy A54 5G",
    Description = "Samsung Galaxy A54 5G mobile phone",
    Price = 500
},
new()
{
    Id = A14Id,
    Name = "Samsung Galaxy A14 5G",
    Description = "Samsung Galaxy A14 5G mobile phone",
    Price = 200
}
};

```

Aqui está o teste de **GET api/catálogo**:

[Fact]

```

public void GetCatalogItemsTest()
{
    var okObjectResult = _controller.Get();
    var okResult = Assert.IsType<OkObjectResult>(okObjectResult);
    var items = Assert.IsType<List<CatalogItem>>(okResult.Value);
    Assert.Equal(2, items.Count);
}

```

Aqui está o teste de **GET api/catálogo/{id}**:

[Fact]

```

public void GetCatalogItemTest()
{
    var id = A54Id;

```

```

var okObjectResult = _controller.Get(id);

var okResult = Assert.IsType<OkObjectResult>(okObjectResult);

var item = Assert.IsType<CatalogItem>(okResult.Value);

Assert.Equal(id, item.Id);
}

```

Aqui está o teste de **POST api/calatlog**:

[Fact]

```

public void InsertCatalogItemTest()
{
    var createdResponse = _controller.Post(
        new CatalogItem
        {
            Id = "353e4410614d711b7fc953a7",
            Name = "iPhone 15",
            Description = "iPhone 15 mobile phone",
            Price = 1500
        }
    );

    var response = Assert.IsType<CreatedAtActionResult>(createdResponse);

    var item = Assert.IsType<CatalogItem>(response.Value);

    Assert.Equal("iPhone 15", item.Name);
}

```

Aqui está o teste de **PUT api/catálogo**:

[Fact]

```

public void UpdateCatalogItemTest()
{
    var id = A54Id;

    var okObjectResult = _controller.Put(

```

```

        new CatalogItem
        {
            Id = id,
            Name = "Samsung Galaxy S23 Ultra",
            Description = "Samsung Galaxy S23 Ultra mobile phone",
            Price = 1500
        });
Assert.IsType<OkResult>(okObjectResult);
var item = _items.FirstOrDefault(i => i.Id == id);
Assert.NotNull(item);
Assert.Equal("Samsung Galaxy S23 Ultra", item.Name);
okObjectResult = _controller.Put(null);
Assert.IsType<NoContentResult>(okObjectResult);
}

```

Aqui está o teste de **DELETE api/catálogo/{id}**:

[Fact]

```

public void DeleteCatalogItemTest()
{
    var id = A54Id;
    var item = _items.FirstOrDefault(i => i.Id == id);
    Assert.NotNull(item);
    var okObjectResult = _controller.Delete(id);
    Assert.IsType<OkResult>(okObjectResult);
    item = _items.FirstOrDefault(i => i.Id == id);
    Assert.Null(item);
}

```

Testes unitários de microserviço de carrinho e microserviço de identidade foram escritos da mesma maneira.

Aqui estão os testes unitários do microserviço de carrinho:

```
public class CartControllerTest
{
    private readonly CartController _controller;

    private static readonly string UserId = "653e43b8c76b6b56a720803e";
    private static readonly string A54Id = "653e4410614d711b7fc953a7";
    private static readonly string A14Id = "253e4410614d711b7fc953a7";
    private readonly Dictionary<string, List<CartItem>> _carts = new()
    {
        {
            UserId,
            new()
            {
                new()
                {
                    CatalogItemId = A54Id,
                    Name = "Samsung Galaxy A54 5G",
                    Price = 500,
                    Quantity = 1
                },
                new()
                {
                    CatalogItemId = A14Id,
                    Name = "Samsung Galaxy A14 5G",
                    Price = 200,
                    Quantity = 2
                }
            }
        }
    }
}
```

```

    }
};

public CartControllerTest()
{
    var mockRepo = new Mock<ICartRepository>();
    mockRepo.Setup(repo => repo.GetCartItems(It.IsAny<string>()))
        .Returns<string>(id => _carts[id]);
    mockRepo.Setup(repo => repo.InsertCartItem(It.IsAny<string>(),
        It.IsAny<CartItem>()))
        .Callback<string, CartItem>((userId, item) =>
        {
            if (_carts.TryGetValue(userId, out var items))
            {
                items.Add(item);
            }
            else
            {
                _carts.Add(userId, new List<CartItem> { item });
            }
        });
    mockRepo.Setup(repo => repo.UpdateCartItem(It.IsAny<string>(),
        It.IsAny<CartItem>()))
        .Callback<string, CartItem>((userId, item) =>
        {
            if (_carts.TryGetValue(userId, out var items))
            {
                var currentItem = items.FirstOrDefault

```

```

        (i => i.CatalogItemId == item.CatalogItemId);
    if (currentItem != null)
    {
        currentItem.Name = item.Name;
        currentItem.Price = item.Price;
        currentItem.Quantity = item.Quantity;
    }
}

});

mockRepo.Setup(repo => repo.UpdateCatalogItem
    (It.IsAny<string>(), It.IsAny<string>(), It.IsAny<decimal>()))
    .Callback<string, string, decimal>((catalogItemId, name, price) =>
    {
        var cartItems = _carts
            .Values
            .Where(items => items.Any(i => i.CatalogItemId == catalogItemId))
            .SelectMany(items => items)
            .ToList();

        foreach (var cartItem in cartItems)
        {
            cartItem.Name = name;
            cartItem.Price = price;
        }
    });

mockRepo.Setup(repo => repo.DeleteCartItem
    (It.IsAny<string>(), It.IsAny<string>()))
    .Callback<string, string>((userId, catalogItemId) =>

```



```

    {
        if (_carts.TryGetValue(userId, out var items))
        {
            items.RemoveAll(i => i.CatalogItemId == catalogItemId);
        }
    });

mockRepo.Setup(repo => repo.DeleteCatalogItem(It.IsAny<string>()))
    .Callback<string>((catalogItemId) =>
    {
        foreach (var cart in _carts)
        {
            cart.Value.RemoveAll(i => i.CatalogItemId == catalogItemId);
        }
    });

_controller = new CartController(mockRepo.Object);
}

```

[Fact]

```

public void GetCartItemsTest()
{
    var okObjectResult = _controller.Get(userId);
    var okResult = Assert.IsType<OkObjectResult>(okObjectResult);
    var items = Assert.IsType<List<CartItem>>(okResult.Value);
    Assert.Equal(2, items.Count);
}

```

[Fact]

```

public void InsertCartItemTest()

```

```

{
    var okObjectResult = _controller.Post(
        UserId,
        new CartItem
        {
            CatalogItemId = A54Id,
            Name = "Samsung Galaxy A54 5G",
            Price = 500,
            Quantity = 1
        }
    );
    Assert.IsType<OkResult>(okObjectResult);
    Assert.NotNull(_carts[UserId].FirstOrDefault(i => i.CatalogItemId == A54Id));
}

```

```

[Fact]
public void UpdateCartItemTest()
{
    var catalogItemId = A54Id;
    var okObjectResult = _controller.Put(
        UserId,
        new CartItem
        {
            CatalogItemId = A54Id,
            Name = "Samsung Galaxy A54",
            Price = 550,
            Quantity = 2
        }
    );
}

```

```
);  
Assert.IsType<OkResult>(okObjectResult);  
var catalogItem = _carts[UserId].FirstOrDefault  
    (i => i.CatalogItemId == catalogItemId);  
Assert.NotNull(catalogItem);  
Assert.Equal("Samsung Galaxy A54", catalogItem.Name);  
Assert.Equal(550, catalogItem.Price);  
Assert.Equal(2, catalogItem.Quantity);  
}
```

```
[Fact]  
public void DeleteCartItemTest()  
{  
    var id = A14Id;  
    var items = _carts[UserId];  
    var item = items.FirstOrDefault(i => i.CatalogItemId == id);  
    Assert.NotNull(item);  
    var okObjectResult = _controller.Delete(UserId, id);  
    Assert.IsType<OkResult>(okObjectResult);  
    item = items.FirstOrDefault(i => i.CatalogItemId == id);  
    Assert.Null(item);  
}
```

```
[Fact]  
public void UpdateCatalogItemTest()  
{  
    var catalogItemId = A54Id;  
    var okObjectResult = _controller.Put(  

```

```

        A54Id,

        "Samsung Galaxy A54",

        550

    );

    Assert.IsType<OkResult>(okObjectResult);

    var catalogItem = _carts[UserId].FirstOrDefault(
        i => i.CatalogItemId == catalogItemId);

    Assert.NotNull(catalogItem);

    Assert.Equal("Samsung Galaxy A54", catalogItem.Name);

    Assert.Equal(550, catalogItem.Price);

    Assert.Equal(1, catalogItem.Quantity);
}

```

[Fact]

```

public void DeleteCatalogItemTest()
{
    var id = A14Id;

    var items = _carts[UserId];

    var item = items.FirstOrDefault(i => i.CatalogItemId == id);

    Assert.NotNull(item);

    var okObjectResult = _controller.Delete(id);

    Assert.IsType<OkResult>(okObjectResult);

    item = items.FirstOrDefault(i => i.CatalogItemId == id);

    Assert.Null(item);
}
}

```

Aqui estão os testes unitários do microservice de identidade:

```

public class IdentityControllerTest

```

```

{
    private readonly IdentityController _controller;

    private static readonly string AdminUserId = "653e4410614d711b7fc951a7";
    private static readonly string FrontendUserId = "653e4410614d711b7fc952a7";
    private static readonly User UnknownUser = new()
    {
        Id = "653e4410614d711b7fc957a7",
        Email = "unknown@store.com",
        Password =
"4kg245EBBE+1IF20pKSBafiNhE/+WydWZo41cfThUqh7tz7+n7Yn9w==",
        Salt =
"2lApH7EgXLHjYAvImPIDAaQ5ypyXlH8PBVmOI+0zhMBu5HxZqIH7+w==",
        IsAdmin = false
    };

    private readonly List<User> _users = new()
    {
        new()
        {
            Id = AdminUserId,
            Email = "admin@store.com",
            Password =
"Ukg255EBBE+1IF20pKSBafiNhE/+WydWZo41cfThUqh7tz7+n7Yn9w==",
            Salt =
"4lApH7EgXLHjYAvImPIDAaQ5ypyXlH8PBVmOI+0zhMBu5HxZqIH7+w==",
            IsAdmin = true
        },
        new()
        {
            Id = FrontendUserId,

```

```

        Email = "jdoe@store.com",

        Password =
"Vhq8Klm83fCVILYhCzp2vKUJ/qSB+tmP/a9bD3leUnp1acBjS2l5jg==",

        Salt =
"7+UwBowz/iv/sW7q+eYhJSfa6HiMQtJXyHuAShU+c1bUo6QUL4LIPA==",

        IsAdmin = false

    }

};

```

```

private static IConfiguration InitConfiguration()
{
    var config = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .AddEnvironmentVariables()
        .Build();

    return config;
}

```

```

public IdentityControllerTest()
{
    var mockRepo = new Mock<IUserRepository>();
    mockRepo.Setup(repo => repo.GetUser(It.IsAny<string>()))
        .Returns<string>(email => _users.FirstOrDefault(u => u.Email == email));
    mockRepo.Setup(repo => repo.InsertUser(It.IsAny<User>()))
        .Callback<User>(_users.Add);

    var configuration = InitConfiguration();
    var jwtSection = configuration.GetSection("jwt");
    var jwtOptions = Options.Create(new JwtOptions
    {

```

```

        Secret = jwtSection["secret"],
        ExpiryMinutes = int.Parse(jwtSection["expiryMinutes"] ?? "60")
    });
    _controller = new IdentityController
        (mockRepo.Object, new JwtBuilder(jwtOptions), new Encryptor());
}

```

[Fact]

```

public void LoginTest()
{
    // User not found
    var notFoundObjectResult = _controller.Login(UnknownUser);
    Assert.IsType<NotFoundObjectResult>(notFoundObjectResult);

    // Backend failure
    var user = new User
    {
        Id = FrontendUserId,
        Email = "jdoe@store.com",
        Password = "aaaaaa",
        IsAdmin = false
    };
    var badRequestObjectResult = _controller.Login(user, "backend");
    Assert.IsType<BadRequestObjectResult>(badRequestObjectResult);

    // Wrong password
    user.Password = "bbbbbb";
    badRequestObjectResult = _controller.Login(user);
}

```

```

Assert.IsType<BadRequestObjectResult>(badRequestObjectResult);

// Frontend success
user.Password = "aaaaaa";
var okObjectResult = _controller.Login(user);
var okResult = Assert.IsType<OkObjectResult>(okObjectResult);
var token = Assert.IsType<string>(okResult.Value);
Assert.NotEmpty(token);

// Backend success
var adminUser = new User
{
    Id = AdminUserId,
    Email = "admin@store.com",
    Password = "aaaaaa",
    IsAdmin = true
};
okObjectResult = _controller.Login(adminUser, "backend");
okResult = Assert.IsType<OkObjectResult>(okObjectResult);
token = Assert.IsType<string>(okResult.Value);
Assert.NotEmpty(token);
}

[Fact]
public void RegisterTest()
{
    // Failure (user already exists)
    var user = new User

```



```

{
    Id = FrontendUserId,
    Email = "jdoe@store.com",
    Password = "aaaaaa",
    IsAdmin = false
};

var badRequestObjectResult = _controller.Register(user);
Assert.IsType<BadRequestObjectResult>(badRequestObjectResult);

// Success (new user)
user = new User
{
    Id = "145e4410614d711b7fc952a7",
    Email = "ctaylor@store.com",
    Password = "cccccc",
    IsAdmin = false
};

var okResult = _controller.Register(user);
Assert.IsType<OkResult>(okResult);
Assert.NotNull(_users.FirstOrDefault(u => u.Id == user.Id));
}

```

```

[Fact]
public void ValidateTest()
{
    // User not found

    var notFoundObjectResult = _controller.Validate(UnknownUser.Email,
string.Empty);

```

```

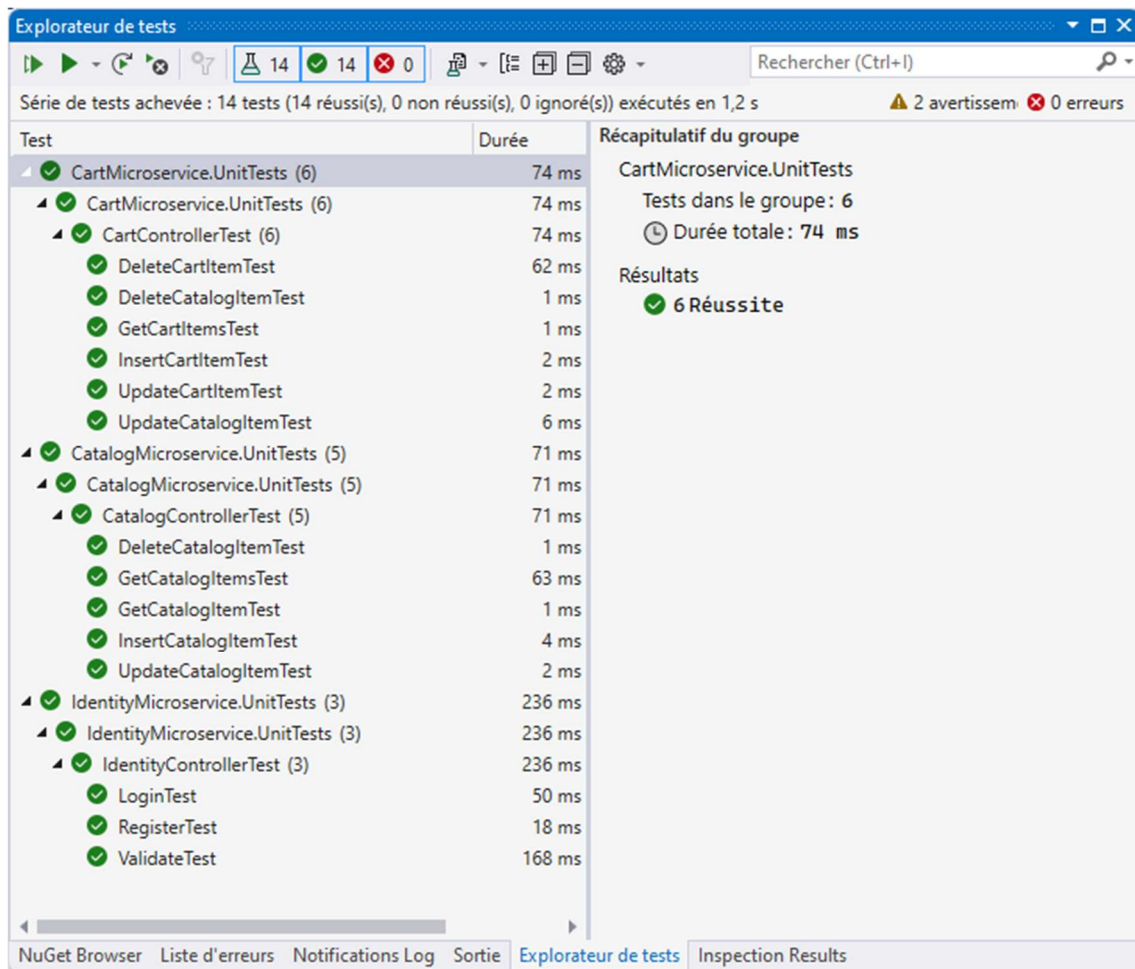
Assert.IsType<NotFoundObjectResult>(notFoundObjectResult);

// Invalid token
var badRequestObjectResult = _controller.Validate("jdoe@store.com", "zzz");
Assert.IsType<BadRequestObjectResult>(badRequestObjectResult);

// Success
var user = new User
{
    Id = FrontendUserId,
    Email = "jdoe@store.com",
    Password = "aaaaaa",
    IsAdmin = false
};
var okObjectResult = _controller.Login(user);
var okResult = Assert.IsType<OkObjectResult>(okObjectResult);
var token = Assert.IsType<string>(okResult.Value);
Assert.NotEmpty(token);
okObjectResult = _controller.Validate(user.Email, token);
okResult = Assert.IsType<OkObjectResult>(okObjectResult);
var userId = Assert.IsType<string>(okResult.Value);
Assert.Equal(user.Id, userId);
}
}

```

Se executarmos os testes unitários, notaremos que todos eles passam:



Você pode encontrar os resultados dos testes unitários em [Ações do GitHub](#).

Monitoramento usando Verificações de Saúde

Nesta seção, veremos como adicionar verificações de integridade ao microserviço de catálogo para fins de monitoramento.

Verificações de integridade são endpoints fornecidos por um serviço para verificar se o serviço está sendo executado corretamente.

As verificações de saúde são usadas para monitorar serviços como:

- Banco de dados (SQL Server, Oracle, MySQL, MongoDB, etc.)
- Conectividade externa da API
- Conectividade de disco (leitura/gravação)
- Serviço de cache (Redis, Memcached, etc.)

Se você não encontrar uma implementação adequada, poderá criar sua própria implementação personalizada.

Para adicionar verificações de integridade ao microserviço de catálogo, os seguintes pacotes nuget foram adicionados:

- `AspNetCore.HealthChecks.MongoDb`
- `AspNetCore.HealthChecks.UI`
- `AspNetCore.HealthChecks.UI.Client`
- `AspNetCore.HealthChecks.UI.InMemory.Storage`

`AspNetCore.HealthChecks.MongoDb` pacote é usado para verificar a integridade do MongoDB.

`AspNetCore.HealthChecks.UI` os pacotes são usados para usar a IU de verificação de integridade que armazena e mostra os resultados das verificações de integridade configuradas `HealthChecks` uris.

Então, `ConfigureServices` método em *Startup.com* foi atualizado da seguinte forma:

```
services.AddHealthChecks()
    .AddMongoDb(
        mongodbConnectionString: (
            Configuration.GetSection("mongo").Get<MongoOptions>()
            ?? throw new Exception("mongo configuration section not found")
        ).ConnectionString,
        name: "mongo",
        failureStatus: HealthStatus.Unhealthy
    );
```

```
services.AddHealthChecksUI().AddInMemoryStorage();
```

E `Configure` método em *Startup.com* foi atualizado da seguinte forma:

```
app.UseHealthChecks("/healthz", new HealthCheckOptions
{
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
```

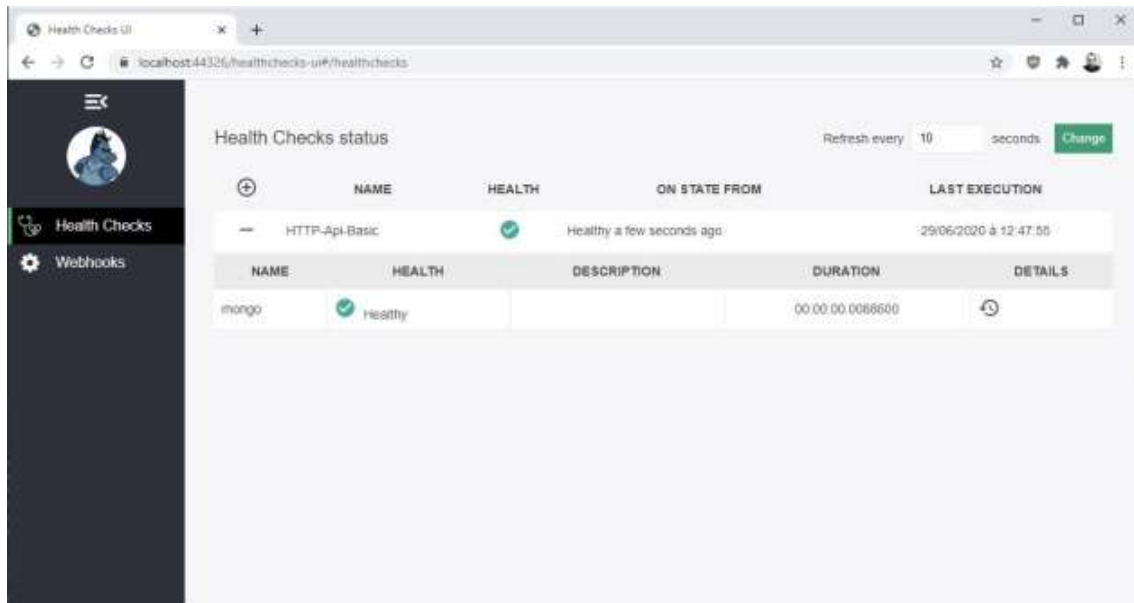
```
app.UseHealthChecksUI();
```

Finalmente *appsettings.com* foi atualizado da seguinte forma:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "mongo": {
    "connectionString": "mongodb://127.0.0.1:27017",
    "database": "store-catalog"
  },
  "jwt": {
    "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103",
    "expiryMinutes": 60
  },
  "HealthChecksUI": {
    "HealthChecks": [
      {
        "Name": "HTTP-API-Basic",
        "Uri": "http://localhost:44397/healthz"
      }
    ],
    "EvaluationTimeOnSeconds": 10,
    "MinimumSecondsBetweenFailureNotifications": 60
  }
}
```

```
}  
  
}
```

Se executarmos o microserviço de catálogo, obteremos a seguinte UI ao acessar <http://localhost:44326/healthchecks-ui>:



É isso. As verificações de saúde de outros microserviços e gateways foram implementadas da mesma maneira.

Como Executar o Aplicativo

Para executar o aplicativo, abra a solução *store.sln* no Visual Studio 2022 como administrador.

Você precisará instalar o MongoDB se ele não estiver instalado.

Primeiro, clique com o botão direito do mouse na solução, clique em propriedades e selecione vários projetos de inicialização. Selecione todos os projetos como projetos de inicialização, exceto projetos de Middleware e testes de unidade.

Então, pressione **F5** para executar o aplicativo.

Você pode acessar o frontend de <http://localhost:44317/>.

Você pode acessar o backend de <http://localhost:44301/>.

Para fazer login no frontend pela primeira vez, basta clicar em **Registrar** para criar um novo usuário e login.

Para fazer login no back-end pela primeira vez, você precisará criar um usuário administrador. Para fazer isso, abra o Swagger <http://localhost:44397/> e registre ou abra o Postman e execute o

seguinte POST pedido <http://localhost:44397/api/identity/register> com a seguinte carga útil:

```
{  
  "email": "admin@store.com",  
  "password": "pass",  
  "isAdmin": true  
}
```

Finalmente, você pode fazer login no back-end com o usuário administrador que você criou.

Se você quiser modificar a cadeia de conexão do MongoDB, você precisa atualizar *appsettings.com* de microsserviços e gateways.

Abaixo estão todos os pontos finais:

- Frontend: <http://localhost:44317/>
- Backend: <http://localhost:44301/>
- Gateway frontend: <http://localhost:44300/>
- Gateway de back-end: <http://localhost:44359/>
- Microsserviço de identidade: <http://localhost:44397/>
- Micro-serviço de catálogo: <http://localhost:44326/>
- Micro-serviço de carrinho: <http://localhost:44388/>

Como Implantar o Aplicativo

Você pode implantar o aplicativo usando o Docker.

Você precisará instalar o Docker que não está instalado.

Primeiro, copie o código-fonte para uma pasta em sua máquina.

Em seguida, abra um terminal, vá para essa pasta (onde *store.sln* o arquivo está localizado) e execute o seguinte comando:

```
docker-compose up
```

É isso, o aplicativo será implantado e será executado.

Então, você pode acessar o frontend de <http://:44317/> e o backend de <http://:44301/>.

Aqui está uma captura de tela do aplicativo em execução no Ubuntu:

```
aelassas@ubuntu: ~/Docker/store/store
File Edit View Search Terminal Help
backendgw_1 | [20:14:42 INF] Sending HTTP request GET http://backendgw/healthz
backendgw_1 | [20:14:42 INF] Received HTTP response after 36.6379ms - OK
backendgw_1 | [20:14:43 INF] End processing HTTP request after 128.3205ms - OK
cart_1      | info: System.Net.Http.HttpClient.health-checks.LogicalHandler[100]
cart_1      | Start processing HTTP request GET http://cart/healthz
cart_1      | info: System.Net.Http.HttpClient.health-checks.ClientHandler[100]
cart_1      | Sending HTTP request GET http://cart/healthz
cart_1      | info: System.Net.Http.HttpClient.health-checks.ClientHandler[101]
cart_1      | Received HTTP response after 12.1201ms - OK
cart_1      | info: System.Net.Http.HttpClient.health-checks.LogicalHandler[101]
cart_1      | End processing HTTP request after 17.8295ms - OK
frontendgw_1 | [20:14:46 INF] Start processing HTTP request GET http://frontendgw/healthz
frontendgw_1 | [20:14:46 INF] Sending HTTP request GET http://frontendgw/healthz
frontendgw_1 | [20:14:46 INF] Received HTTP response after 9.6962ms - OK
frontendgw_1 | [20:14:46 INF] End processing HTTP request after 10.6891ms - OK
catalog_1    | info: System.Net.Http.HttpClient.health-checks.LogicalHandler[100]
catalog_1    | Start processing HTTP request GET http://catalog/healthz
catalog_1    | info: System.Net.Http.HttpClient.health-checks.ClientHandler[100]
catalog_1    | Sending HTTP request GET http://catalog/healthz
catalog_1    | info: System.Net.Http.HttpClient.health-checks.ClientHandler[101]
catalog_1    | Received HTTP response after 9.3928ms - OK
catalog_1    | info: System.Net.Http.HttpClient.health-checks.LogicalHandler[101]
catalog_1    | End processing HTTP request after 9.6554ms - OK
```

Para aqueles que querem entender como a implantação é feita, aqui está *docker-compose.yml*:

version: "3.8"

services:

mongo:

image: mongo

ports:

- 27017:27017

catalog:

build:

context: .

dockerfile: src/microservices/CatalogMicroservice/Dockerfile

depends_on:

- mongo

ports:

- 44326:80

cart:

build:

context: .

dockerfile: src/microservices/CartMicroservice/Dockerfile

depends_on:

- mongo

ports:

- 44388:80

identity:

build:

context: .

dockerfile: src/microservices/IdentityMicroservice/Dockerfile

depends_on:

- mongo

ports:

- 44397:80

frontendgw:

build:

context: .

dockerfile: src/gateways/FrontendGateway/Dockerfile

depends_on:

- mongo

- catalog

- cart

- identity

ports:

- 44300:80

backendgw:

build:

context: .

dockerfile: src/gateways/BackendGateway/Dockerfile

depends_on:

- mongo

- catalog

- identity

ports:

- 44359:80

frontend:

build:

context: .

dockerfile: src/uis/Frontend/Dockerfile

ports:

- 44317:80

backend:

build:

context: .

dockerfile: src/uis/Backend/Dockerfile

ports:

- 44301:80

Então, *appsettings.Produção.json* foi usado em microserviços e gateways, e *ocelot.Produção.json* foi usado em gateways.

Por exemplo, aqui está *appsettings.Produção.json* de microserviço de catálogo:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "mongo": {
    "connectionString": "mongodb://mongo",
    "database": "store-catalog"
  },
  "HealthChecksUI": {
    "HealthChecks": [
      {
        "Name": "HTTP-API-Basic",
        "Uri": "http://catalog/healthz"
      }
    ],
    "EvaluationTimeOnSeconds": 10,
    "MinimumSecondsBetweenFailureNotifications": 60
  }
}
```

Aqui está *Dockerfile* de microserviço de catálogo:

```
# syntax=docker/dockerfile:1
```

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY ["src/microservices/CatalogMicroservice/CatalogMicroservice.csproj",

"microservices/CatalogMicroservice/"]

COPY src/middlewares middlewares/

RUN dotnet restore

"microservices/CatalogMicroservice/CatalogMicroservice.csproj"

WORKDIR "/src/microservices/CatalogMicroservice"

COPY src/microservices/CatalogMicroservice .

RUN dotnet build "CatalogMicroservice.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "CatalogMicroservice.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

ENV ASPNETCORE_URLS=http://+:80

EXPOSE 80

EXPOSE 443

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "CatalogMicroservice.dll"]

A compilação de vários estágios é explicada [aqui](#). Isso ajuda a tornar o processo de construção de contêineres mais eficiente e torna os contêineres menores, permitindo que eles contenham apenas os bits que seu aplicativo precisa em tempo de execução.

Aqui está *ocelot.Produção.json* do gateway frontend:

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/catalog",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "catalog",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/catalog",
      "UpstreamHttpMethod": [ "GET" ],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "Bearer",
        "AllowedScopes": []
      }
    },
    {
      "DownstreamPathTemplate": "/api/catalog/{id}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "catalog",
          "Port": 80
        }
      ],
    },
  ],
}
```

```
"UpstreamPathTemplate": "/catalog/{id}",
"UpstreamHttpMethod": [ "GET" ],
"AuthenticationOptions": {
  "AuthenticationProviderKey": "Bearer",
  "AllowedScopes": []
}
},
{
  "DownstreamPathTemplate": "/api/cart",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "cart",
      "Port": 80
    }
  ],
  "UpstreamPathTemplate": "/cart",
  "UpstreamHttpMethod": [ "GET" ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer",
    "AllowedScopes": []
  }
},
{
  "DownstreamPathTemplate": "/api/cart",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
```

```
    "Host": "cart",
    "Port": 80
  },
  "UpstreamPathTemplate": "/cart",
  "UpstreamHttpMethod": [ "POST" ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "Bearer",
    "AllowedScopes": []
  },
  {
    "DownstreamPathTemplate": "/api/cart",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      {
        "Host": "cart",
        "Port": 80
      }
    ],
    "UpstreamPathTemplate": "/cart",
    "UpstreamHttpMethod": [ "PUT" ],
    "AuthenticationOptions": {
      "AuthenticationProviderKey": "Bearer",
      "AllowedScopes": []
    }
  },
  {
```

```
"DownstreamPathTemplate": "/api/cart",
"DownstreamScheme": "http",
"DownstreamHostAndPorts": [
  {
    "Host": "cart",
    "Port": 80
  }
],
"UpstreamPathTemplate": "/cart",
"UpstreamHttpMethod": [ "DELETE" ],
"AuthenticationOptions": {
  "AuthenticationProviderKey": "Bearer",
  "AllowedScopes": []
}
},
{
  "DownstreamPathTemplate": "/api/identity/login",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "identity",
      "Port": 80
    }
  ],
  "UpstreamPathTemplate": "/identity/login",
  "UpstreamHttpMethod": [ "POST" ]
},
{
```



```
"DownstreamPathTemplate": "/api/identity/register",
"DownstreamScheme": "http",
"DownstreamHostAndPorts": [
  {
    "Host": "identity",
    "Port": 80
  }
],
"UpstreamPathTemplate": "/identity/register",
"UpstreamHttpMethod": [ "POST" ]
},
{
  "DownstreamPathTemplate": "/api/identity/validate",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "identity",
      "Port": 80
    }
  ],
  "UpstreamPathTemplate": "/identity/validate",
  "UpstreamHttpMethod": [ "GET" ]
}
],
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:44300/"
}
}
```

Aqui está *appsettings.Produção.json* do gateway frontend:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "jwt": {
    "secret": "9095a623-a23a-481a-aa0c-e0ad96edc103"
  },
  "mongo": {
    "connectionString": "mongodb://mongo"
  },
  "HealthChecksUI": {
    "HealthChecks": [
      {
        "Name": "HTTP-API-Basic",
        "Uri": "http://frontendgw/healthz"
      }
    ],
    "EvaluationTimeOnSeconds": 10,
    "MinimumSecondsBetweenFailureNotifications": 60
  }
}
```

E finalmente, aqui está *Dockerfile* do gateway frontend:

```
# syntax=docker/dockerfile:1
```

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
```

```
WORKDIR /app
```

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
```

```
WORKDIR /src
```

```
COPY ["src/gateways/FrontendGateway/FrontendGateway.csproj",  
"gateways/FrontendGateway/"]
```

```
COPY src/middlewares middlewares/
```

```
RUN dotnet restore "gateways/FrontendGateway/FrontendGateway.csproj"
```

```
WORKDIR "/src/gateways/FrontendGateway"
```

```
COPY src/gateways/FrontendGateway .
```

```
RUN dotnet build "FrontendGateway.csproj" -c Release -o /app/build
```

```
FROM build AS publish
```

```
RUN dotnet publish "FrontendGateway.csproj" -c Release -o /app/publish
```

```
FROM base AS final
```

```
WORKDIR /app
```

```
ENV ASPNETCORE_URLS=http://+:80
```

```
EXPOSE 80
```

```
EXPOSE 443
```

```
COPY --from=publish /app/publish .
```

```
ENTRYPOINT ["dotnet", "FrontendGateway.dll"]
```

As configurações de outros microsserviços e o gateway de back-end são feitas praticamente da mesma maneira.