

Disciplina	Arquitetura de Integração e MicroServiços
Professor	Daniel Souza Makiyama

1. Apresentação do Case

A Geek Burger Company é uma cadeia de fast-foods, reconhecida por um sistema inovador de venda e produção. Na Geek Burger é possível fazer pedidos em totens de auto-atendimento, e a produção dos lanches utiliza as mais avançadas técnicas de automação industrial. Desta forma, a Geek Burger oferece e entrega um complexo menu de hambúrgueres em tempo recorde e com muita qualidade, mesmo com um time operacional reduzido e com pouca experiência profissional.

Durante o workshop anual da empresa, que tem objetivo compartilhar as novidades com os franquizados e identificar novas demandas de negócio, foi identificado, em um debate com os gerentes de franquias, que nos últimos anos aumentou o número de pessoas com alergias ou restrições alimentares médicas que buscaram auxílio do gerente antes de fazerem seus pedidos, mas acabaram desistindo da compra.

Além da falta de informação sobre os ingredientes de cada produto no restaurante, os clientes reclamaram da falta de um protocolo de produção para lidar com as restrições, evitando a contaminação cruzada entre alimentos nas chapas, fritadeiras e esteiras de montagem.

Após a aplicação de uma pesquisa mais detalhada descobriu-se que 30% dos clientes no último mês declararam possuir algum tipo de restrição alimentar.

2. Demanda de Projeto

Para atender à esta demanda foi solicitado à TI o ajuste do sistema GeekBurger (sistema de vendas da Geek Burger) para ser allergy friendly: permitir que o usuário informe suas restrições alimentares na máquina de auto-atendimento, obtenha um menu filtrado com os produtos seguros para o seu consumo e que podem ser produzidos naquele restaurante, naquele momento, sem o risco da contaminação cruzada.

Após uma longa reunião com o departamento de TI descobriu-se que não seria possível ajustar o sistema na arquitetura atual (monolítica), pois as máquinas de auto-atendimento já estavam rodando no máximo de sua capacidade e qualquer novo componente comprometeria totalmente a experiência do usuário. Na máquina de auto-atendimento rodavam os módulos de Produtos, Usuários, Ordens e Interface Visual. Até o momento as máquinas atendiam à demanda pois a lista de produtos era pequena e padrão entre os restaurantes e o volume de vendas era bem menor. Ao longo do tempo a lista de produtos foi diversificando e o volume de atendimentos triplicou, tornando inviável qualquer melhoria no sistema sem upgrade das máquinas, o que geraria um alto custo aos franquizados.

A empresa então contratou um time para reimplementar o sistema utilizando microserviços. Foi fornecida a seguinte lista de requisitos funcionais e não funcionais:

As máquinas de auto-atendimento devem rodar somente a interface visual do sistema;



O login nas máquinas de auto-atendimento deve ocorrer por reconhecimento facial. O reconhecimento facial ocorre em 2 etapas: primeiro a imagem é submetida para verificar se existe um rosto na imagem. Caso sim, o rosto ganha um ID. Com este ID é possível submetê-lo para encontrar um rosto similar. Caso um rosto similar seja encontrado, o id do rosto similar é utilizado para obter-se o registro do usuário em questão. Caso um rosto similar não seja encontrado, o mesmo é submetido para uma lista de rostos na nuvem;

Caso seja um novo usuário (rostos similares não foram encontrados), o usuário deve informar no auto-atendimento a lista dos ingredientes que possui restrição;

Ao ser reconhecido no auto-atendimento o mesmo deve apresentar ao usuário uma lista de produtos filtrados pelas restrições do usuário (caso ele possua restrição) e áreas de produção disponível para elaboração do produto com a restrição;

Caso o usuário não possua restrição, mas todas as áreas de fritura estejam desligadas, por exemplo, o auto-atendimento não deve mostrar produtos que demandem fritura;

Um servidor será disponibilizado por loja para armazenar um catálogo local de usuários, produtos e ingredientes compartilhado;

A base completa de produtos e ingredientes de todas as lojas deve ser centralizada na nuvem;

Para reduzir o trabalho manual, a matriz irá escanear o rótulos de cada componente dos produtos, atualizando o catálogo de ingredientes na nuvem. Será construído um console application (Label Loader) responsável por varrer uma pasta de imagens de rótulos e enviar para o serviço de ingredients.

As áreas de produção devem enviar uma mensagem quando forem iniciadas. Nesta mensagem devem ser incluído o tipo de produção (fritadeira, chapa ou esteira) e os alimentos que são proibidos naquela área. Desta forma é possível identificar as áreas de produção que estão disponíveis ou indisponíveis para atender a cada restrição;

A cada mudança de produto o catálogo local deve ser atualizado, caso a atualização seja em um produto que a loja oferece;

A UI depende dos catálogos disponíveis no servidor, portanto o catálogo deve avisar a UI quando estiver disponível para ser utilizado;

O cliente então deve selecionar N produtos e clicar em comprar. Uma mensagem com a ordem é disparada e recebida pelos microserviços de Ordens e Produção. O serviço de Produção inicia a montagem do pedido enquanto o serviço de Ordens verifica o pagamento. A Produção aguarda confirmação do pagamento e finalização da produção para enviar a mensagem de ordem finalizada.

Caso o pagamento seja negado, o serviço de Ordens envia uma mensagem que permite o cancelamento da produção do pedido e a mensagem de ordem finalizada não é disparada pelo serviço de Produção.

A mensagem de ordem paga (somente as ordens com status pago) é coletada por um serviço de Dashboard que consolida o número de vendas concretizadas por minuto.

O Dashboard também recebe uma mensagem da loja informando sempre que um usuário utiliza o kiosk de auto-atendimento e aparecem disponíveis para ele menos de 2 produtos, o que indica uma baixa oferta de produtos para sua restrição alimentar.

Todos os IDs devem ser GUIDs (desta forma fica fácil de trocar informações entre serviços garantindo índice único de cada recurso). No contexto desta atividade, o Dashboard deve ser um projeto web api com: Api de vendas que mostra o total e valor das vendas por loja, número e valor das vendas por hora por loja. Api de usuários com menor oferta que deve retornar a quantidade de usuários e ocorrências com menos de 4 produtos na loja e suas restrições alimentares, atualizadas em tempo real cada vez que o dashboard recebe uma mensagem.

NOTA: No contexto desta atividade o tempo de produção será um timer randômico no microserviço de produção, variando de 5 à 20 segundos.

NOTA: No contexto desta atividade, a cada 5 ordens, uma deve ter o pagamento negado.

O Diagrama de componentes (Diagrama 1) abaixo mostra os diversos microserviços da solução, e os pontos de comunicação entre eles:

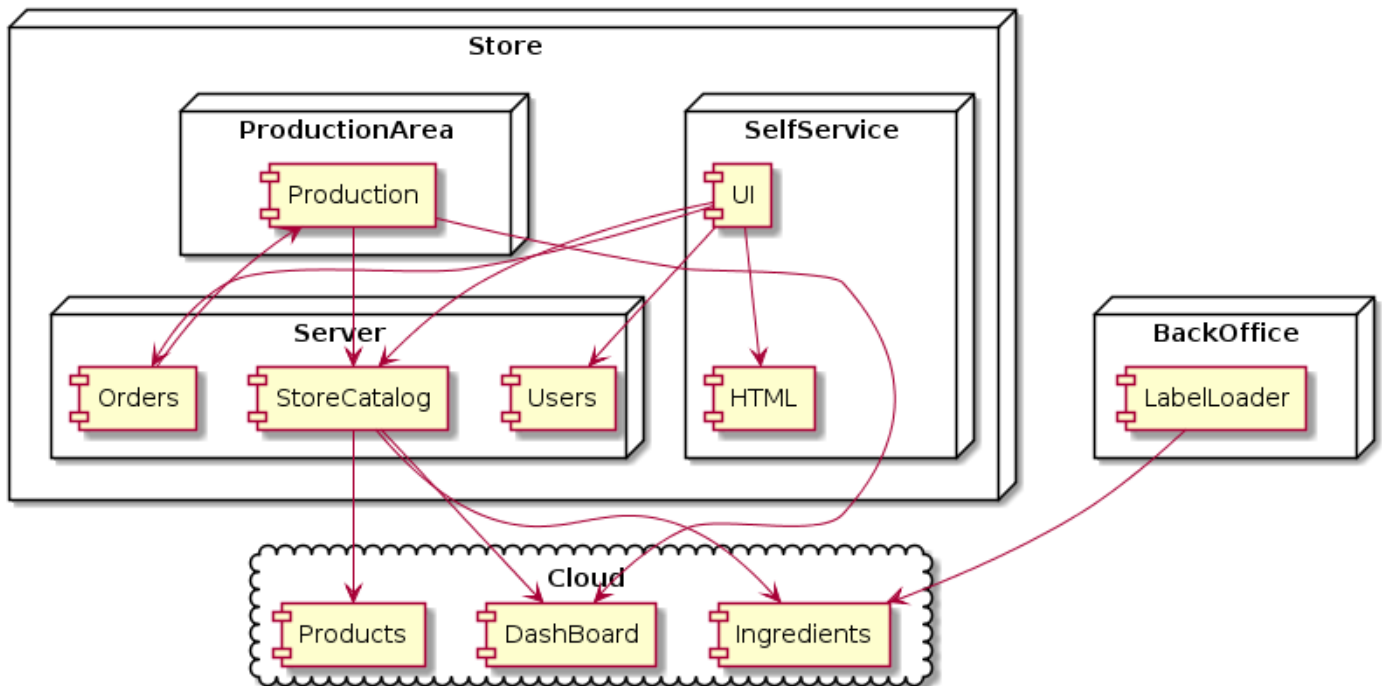


Diagrama 1

NOTA: Cada grupo será responsável por um microserviço do diagrama acima, exceto no caso dos microserviços de Dashboard e LabelLoader que por serem menos complexos, um único grupo será responsável pela criação dos dois. São 10 microserviços ao todo, mas os microserviços **HTML** e

Products já estão prontos. Os grupos podem ter no mínimo 2 pessoas e no máximo 3. Os grupos serão avaliados na mesma proporção quanto à qualidade do seu microserviço e o sucesso do projeto como um todo, portanto o bom alinhamento entre os times e o apoio entre times deve ser cultivado.

Os diagramas de sequência (Diagramas 2, 3, 4 e 5) abaixo apresentam o fluxo completo de interação entre os serviços da solução durante as etapas de inicialização, login, compra e produção.

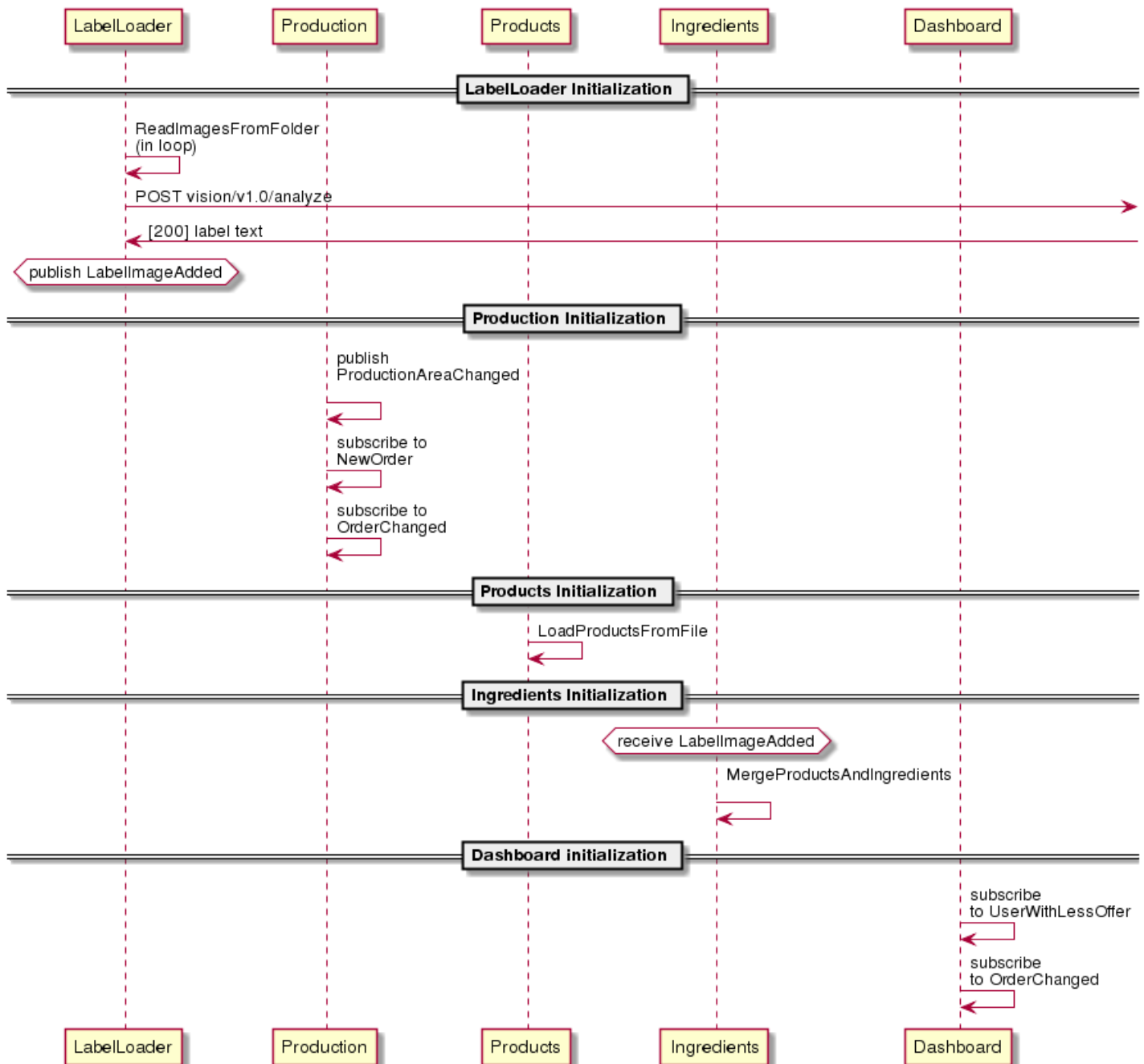


Diagrama 2 - Inicialização

FIAP

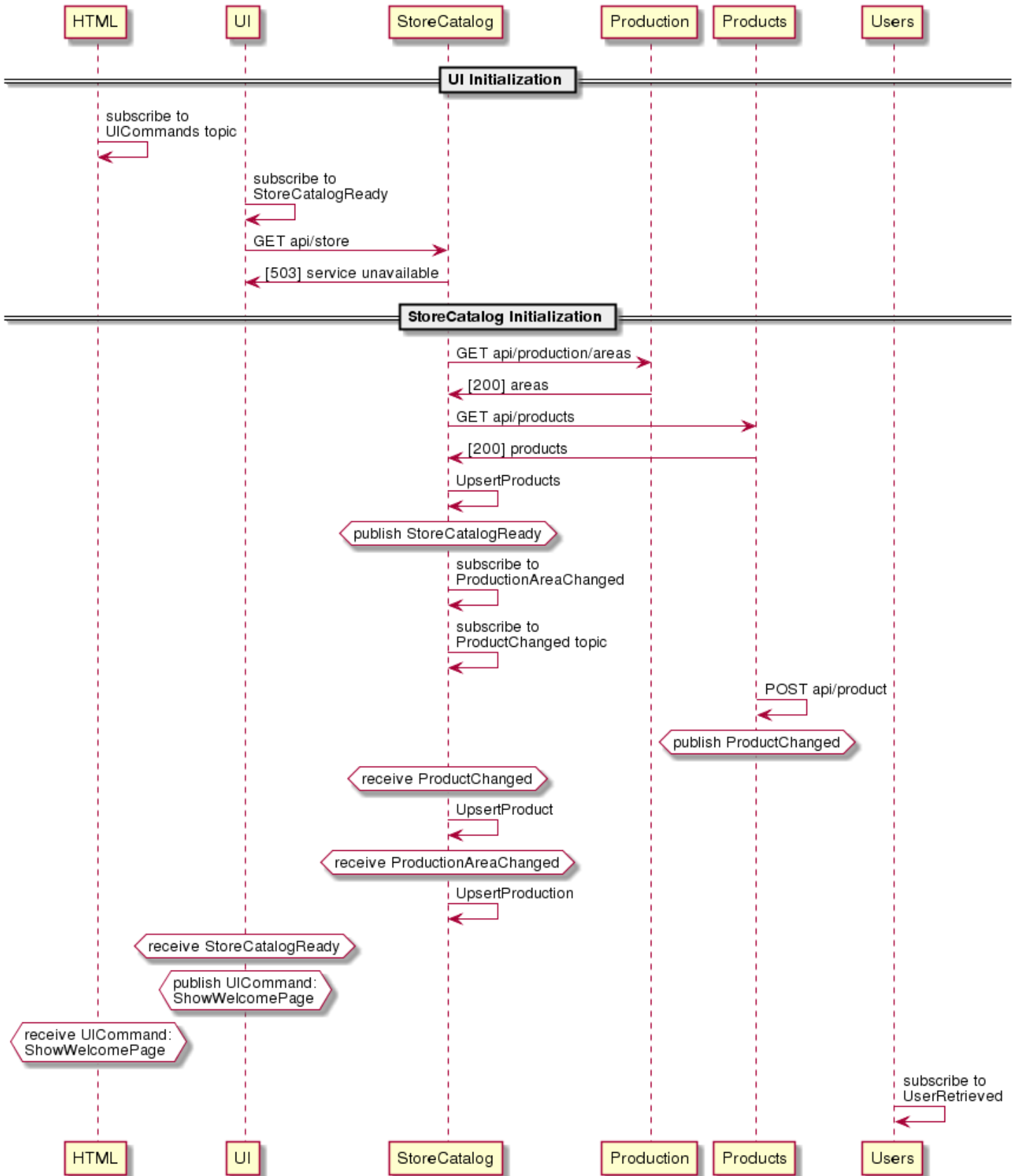


Diagrama 3 - Inicialização

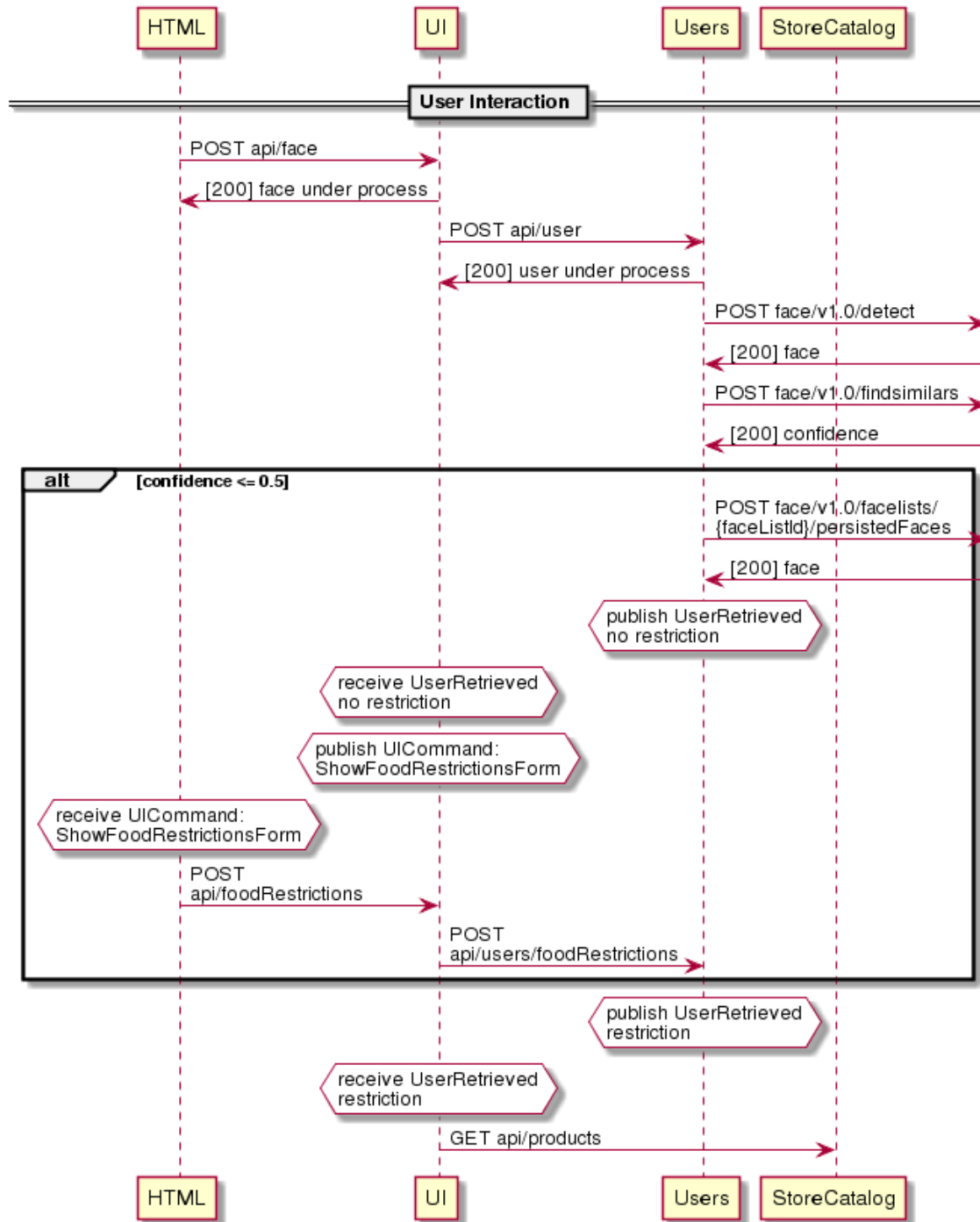


Diagrama 4 – Login

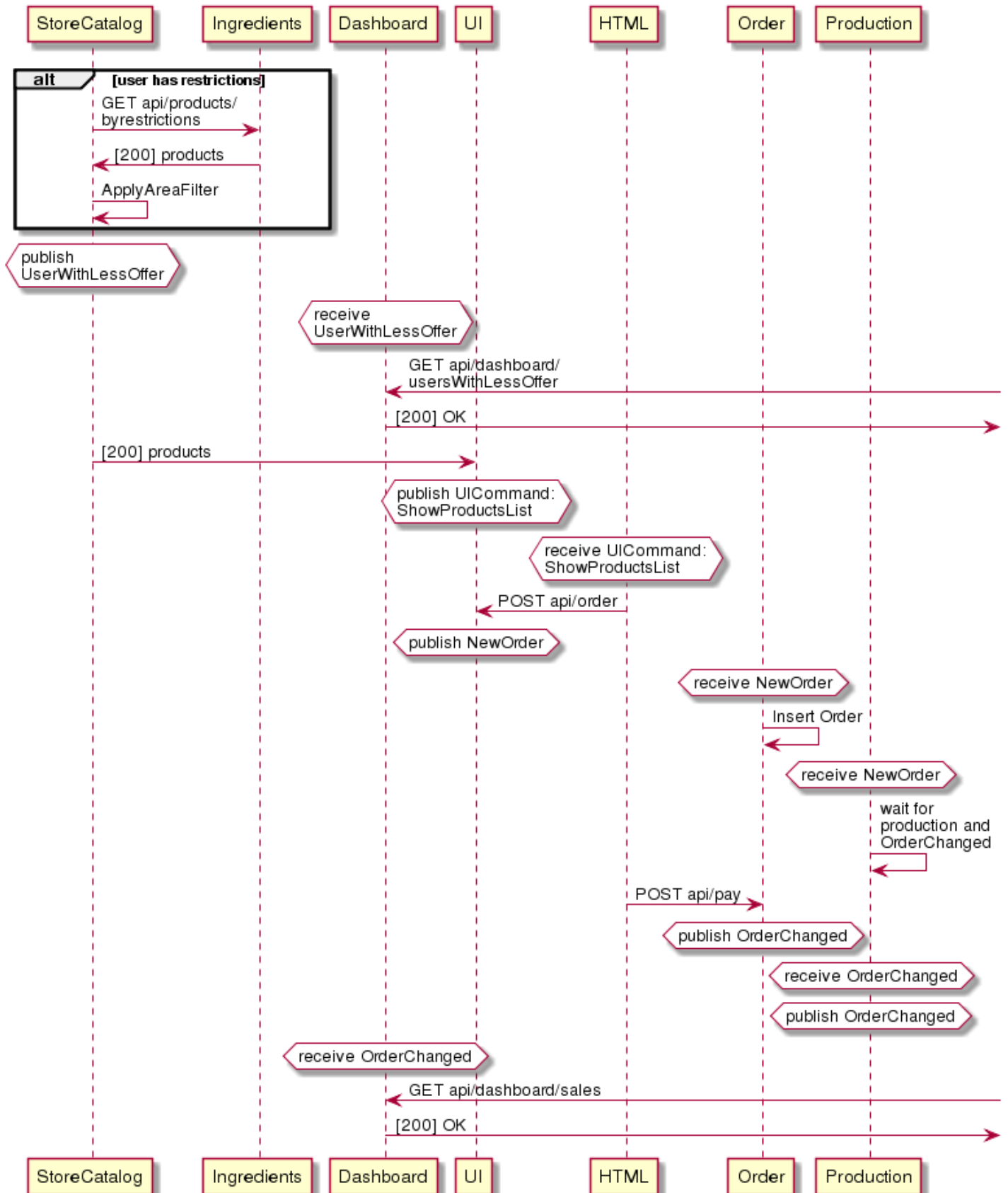


Diagrama 5 – Compra e Produção



Os diversos microserviços interagem através de mensagens (Service Bus) e APIs (Web Api). Abaixo apresento os contratos de todas as mensagens e APIs apresentadas nos diagramas acima. A definição e utilização destes contratos é fundamental para a interação entre os diversos microserviços.

```
//LabelImageAdded message
{ "ItemName": "meat",
  "Ingredients": ["diary","gluten","soy"] }
{ "ItemName": "bread",
  "Ingredients": ["peanut","gluten"] }

*** CHANGED ***

//ProductionAreaChanged message
{ "ProductionId": 1111,
  "Restrictions":
["soy","dairy","gluten","peanut"]
  "On": true }

//NewOrder message
{ "OrderId": 1111,
  "Total": "10.20",
  "Products": [{ "ProductId": 1111}]
  "ProductionIds": [1111,1112] }

//OrderChanged message
{ "OrderId": 1,
  "State": "Paid|Canceled|Finished" }

//GET api/products request
{ "StoreName": "Beverly Hills" }

//GET api/products response
[{ "StoreId": 1111,
  "ProductId": 1111, "Name": "Darth Bacon",
  "Image" "img_db.jpg",
  "Items": [
    { "ItemId": 1111, "Name": "bread" },
    { "ItemId": 2222, "Name": "meat" }
  ],
  "Price": "10.20" }]

//GenerateIngredientsList model
[{ "ItemId": 1111,
  "Ingredients": ["diary","gluten","soy"] }]

//GET api/production/areas request
{}

*** CHANGED ***

//GET api/production/areas response
{ "ProductionId": 1111,
  "Restrictions":
["soy","dairy","gluten","peanut"]
  "On": true }

//StoreCatalogReady and GET api/store
{ "StoreId": 1111, "Ready": true }

//ProductChanged
{ "State": "Deleted|Modified|Added",
  "Product": {
    "StoreId": 1111,
    "ProductId": 1111,
    "Name": "Darth Bacon",
    "Image" "img_db.jpg",
    "Items": [
      { "ItemId": 1111, "Name": "bread" },
      { "ItemId": 2222, "Name": "meat" }
    ],
    "Price": "10.20"
  } }

//MergeProductsAndIngredients
{ "ProductId" : 1111,
  "Ingredients" :
["diary","gluten","soy","peanut"],
  "StoreId": 1111 }

//UserWithLessOffer
{ "UserId" : 1111,
  "Restrictions": ["soy","diary","peanut"] }

//GET api/face request
{ "Face": "XXXXXXXXXXXX" //base64 string,
  "RequesterId": 1111 }

//GET api/user request
{ "Face": "XXXXXXXXXXXX" //base64 string }
```


FIAP

```
//GET api/face & api/user response &
api/foodRestrictions &
api/users/foodRestrictions
{ "Processing": true , "UserId": 1111 }

//UserRetrieved
{ "AreRestrictionsSet": false,
  "UserId": 1111 }

//UICommand:ShowFoodRestrictionsForm
{ "UserId": 1111, "RequesterId": 1111 }

//api/foodRestrictions &
api/users/foodRestrictions
{ "Restrictions": ["soy","gluten"],
  "Others": "brocolis",
  "UserId": 1111,
  "RequesterId": 1111 }

//api/products/byrestrictions request
{ "Restrictions": ["soy","gluten"],
  "StoreId": 1111 }

*** CHANGED ***

//api/products/byrestrictions response
[{ "ProductId": 1111,
  "Ingredients": ["soy","gluten"] },
 { "ProductId": 1112,
  "Ingredients": ["peanut"] }]

//UICommand:\nShowProductsList
[{ "StoreId": 1111,
  "ProductId": "XXXX",
  "Name": "Darth Bacon",
  "Image": "img1.png",
  "Items": [{ "ItemId": 1111, "Name": "bread"
},
  { "ItemId": 1112, "Name": "meat" }
], "Price": 4,
  "RequesterId": 1111 }]

//GET api/dashboard/sales request
{}

//GET api/dashboard/sales response
{ "StoreId": 1111,
  "Total": 1000,
  "Value": "59385.00" }

//GET api/dashboard/sales request
{ "Per": "hour", "Value": 18 }

//GET api/dashboard/sales response
{ "StoreId": 1111,
  "Total": 10, "Value": "4092.00" }

//GET api/dashboard/usersWithLessOffer request
{}

//GET api/dashboard/usersWithLessOffer
response
{ "users": 10,
  "restrictions": [{ "type": "soy,diary",
"users": 2 },
  { "type": "gluten", "users" : 8 }
], "usage": 50 }

//POST api/order request
{ "OrderId": 1111,
  "Products": [{ "ProductId": 1111, "Price":
"10.20" }]
  "ProductionIds": [1111,1112] }

//POST api/order response
{ "OrderId": 1111,
  "Total": "10.20" }

//POST api/pay request
{ "OrderId": 1111,
  "StoreId" : 1111,
  "PayType" : "credit",
  "CardNumber": "XXXXXXXXXX",
  "CardOwnerName": "XXXXXXXXXX",
  "SecurityCode": "XXX",
  "ExpirationDate": "05/10/2022",
  "RequesterId": 1111 }

//POST api/pay response
{}
```



Quanto aos dados de lojas, produtos, itens e áreas de produção das 2 lojas, devemos seguir a lista base a seguir. São 2 lojas e 6 produtos distintos:

```
[{ "storename": "Los Angeles - Pasadena",
  "name": "Darth Bacon",
  "image": "hamb1.png",
  "price": 2.5,
  "items": [ { "name": "beef" }, { "name":
"ketchup" }, { "name": "bread" } ]
},
{ "storename": "Los Angeles - Pasadena",
  "name": "Cap. Spork",
  "image": "hamb2.png",
  "price": 3.5,
  "items": [ { "name": "pork" }, { "name":
"mustard" }, { "name": "whole bread" } ]
},
{ "storename": "Los Angeles - Pasadena",
  "name": "Beef Turner",
  "image": "hamb3.png",
  "price": 4.0,
  "items": [ { "name": "beef" }, { "name":
"mustard" }, { "name": "bread" } ]
},
{ "storename": "Los Angeles - Pasadena",
  "name": "The Big Bang Meal",
  "image": "hamb4.png",
  "price": 6.5,
  "items": [ { "name": "beef" }, { "name":
"mustard" }, { "name": "bread" }, { "name": "beef"
}, { "name": "ketchup" }, { "name": "bread" } ]
},
{ "storename": "Los Angeles - Pasadena",
  "name": "Out of this World",
  "image": "hamb5.png",
  "price": 4.0,
  "items": [ { "name": "salmon" }, { "name":
"onion wings" }, { "name": "whole bread" } ]
},
{ "storename": "Los Angeles - Beverly Hills",
  "name": "Darth Bacon",
  "image": "hamb1.png",
  "price": 2.5,
  "items": [ { "name": "beef" }, { "name":
"ketchup" }, { "name": "bread" } ]
},
{ "storename": "Los Angeles - Beverly Hills",
  "name": "Cap. Spork",
  "image": "hamb2.png",
  "price": 5.0,
  "items": [ { "name": "pork" }, { "name":
"mustard" }, { "name": "whole bread" } ]
},
{ "storename": "Los Angeles - Beverly Hills",
  "name": "Beef Turner",
  "image": "hamb3.png",
  "price": 5.5,
  "items": [ { "name": "beef" }, { "name":
"mustard" }, { "name": "bread" } ]
},
{ "storename": "Los Angeles - Beverly Hills",
  "name": "The Big Bang Meal",
  "image": "hamb4.png",
  "price": 8.5,
  "items": [ { "name": "beef" }, { "name":
"mustard" }, { "name": "bread" }, { "name": "beef"
}, { "name": "ketchup" }, { "name": "bread" } ]
},
{
  "storename": "Los Angeles - Beverly Hills",
  "name": "Out of this World",
  "image": "hamb5.png",
  "price": 5.5,
  "items": [ { "name": "salmon" }, { "name":
"onion wings" }, { "name": "whole bread" }
]
},
{
  "storename": "Los Angeles - Beverly Hills",
  "name": "Run to the Hills",
  "image": "hamb6.png",
  "price": 4.5,
  "items": [ { "name": "chicken" }, { "name":
"tomato" }, { "name": "whole bread" }
]
}
```

Áreas de Produção:

```
{ "ProductionId": 1111, "Restrictions": ["soy","dairy","gluten","sugar"] "On": true }
{ "ProductionId": 1112, "Restrictions": ["soy"] "On": true }
{ "ProductionId": 1113, "Restrictions": ["gluten"] "On": true }
{ "ProductionId": 1114, "Restrictions": [] "On": true }
```

3. Atividade

Após a divisão dos grupos, siga os seguintes passos para declarar em código as APIs e mensagens do microserviço do seu grupo. Por fim, compartilhem suas APIs via NuGet, para que os outros grupos possam utilizar. O objetivo final de cada grupo nesta atividade é implementar (ainda de forma hard-coded) todas as APIs de seus microserviços, publicar os projetos de contrato no NuGet e publicar a API no Azure, e comitar o código no GitHub.

Para criar um projeto .NET Core 2.0, abra o Visual Studio 2017 e clique em **File > New Project**. Será apresentada uma janela (Figura 1) com todos os templates disponíveis. No canto esquerdo, localize e clique no item **Visual C# > .NET Core**. Ao lado selecione a opção **ASP.NET Core Web Application**.

Você pode ignorar a caixa de seleção de Framework .NET na parte superior da janela para templates .NET Core. Ainda na janela novo projeto, informe o nome do projeto, nesta atividade, usaremos o nome do *sistema.microserviço* de cada grupo, ex: GeekBurger.Products e informe a localização de sua preferência. Em seguida, clique no botão **OK**.

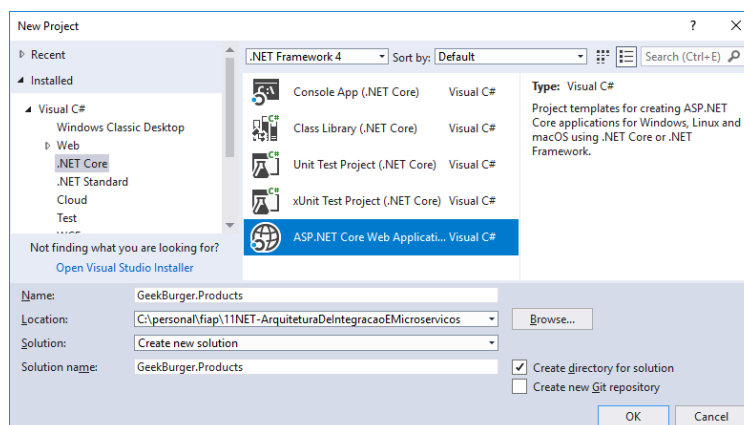


Figura 1

Uma nova janela será apresentada (Figura 2). Certifique-se que no topo da janela a opção **.NET Core** e **ASP.NET Core 2.0** estejam selecionadas. Logo abaixo, nas opções de template, selecione a opção **Empty** e clique em **OK**.

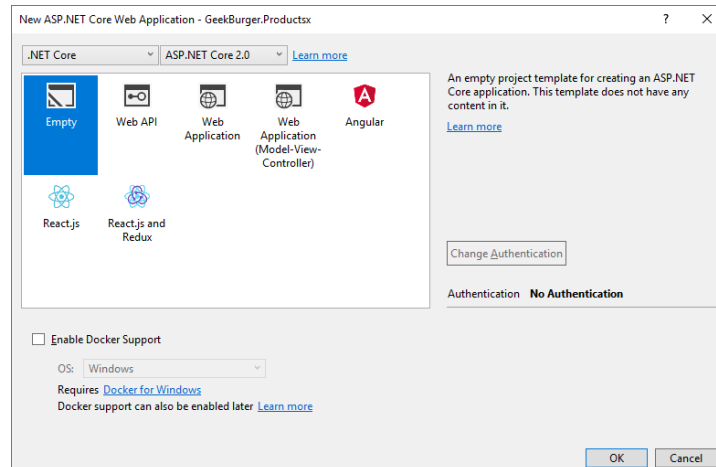


Figura 2

A partir deste momento o projeto em questão é criado e seus arquivos ficam visíveis na janela de **Solution Explorer**. Abra o arquivo **Startup.cs** localizado na pasta raiz do projeto. Altere o conteúdo dos métodos **ConfigureServices** e **Configure** conforme instruções da Tabela 1.

<pre>public void ConfigureServices(IServiceCollection services) { var mvcCoreBuilder = services.AddMvcCore(); mvcCoreBuilder .AddFormatterMappings() .AddJsonFormatters() .AddCors(); }</pre>	<p>AddMvcCore retorna um objeto de builder através do qual é possível adicionar serviços essenciais de MVC no IServiceCollection que é a coleção de serviços que ficam disponíveis para serem injetados nas classes do .NET Core. Saiba mais em: Injeção de Dependência ASP.NET Core</p> <p>Os serviços essenciais que precisamos são:</p> <ul style="list-style-type: none"> - FormatterMappings: formata o content-type dos retornos de APIs; - JsonFormatters: permite serialização dos retornos em Json; - Cors: permite que páginas web de outras origens (ex: http://mysite.com) solicitem dados desta API;
<pre>public void Configure(IApplicationBuilder app, IHostingEnvironment env) { if (env.IsDevelopment()) { app.UseDeveloperExceptionPage(); } app.UseMvc(); }</pre>	<p>UseDeveloperExceptionPage permite que qualquer exception não tratada gerada na chamada de qualquer API apareça na página web de retorno, somente se o projeto for compilado em Debug.</p> <p>Adiciona a engine de MVC no construtor de aplicação do ASP.NET core. Saiba mais sobre MVC em: Visão geral do ASP.NET MVC</p>

Tabela 1

NOTA: Neste exemplo vamos abordar a criação do Microserviço de Produtos, no entanto **cada grupo deve criar o seu microserviço descrito no Case, atribuído a cada grupo durante a aula.**

Agora vamos criar a primeira API do microserviço de Produtos em questão. Crie a pasta **Controllers**, clique com o botão direito na pasta e selecione **Add > New From Template > Controller**. Informe o nome **ProductsController**. Será criado um arquivo com o nome **ProductsController.cs** na pasta **Controllers**. Inicialmente o arquivo terá somente o código que define a classe **ProductsController**, herdando da classe **Controller**, e um método chamado **Index** que retorna um objeto do tipo **ActionResult**. O retorno de uma **View()** é mais adequado em aplicações .NET MVC onde deseja-se retornar um HTML para um usuário. No caso de APIs, o retorno não é para um usuário final, portanto trata-se de uma estrutura de dados serializada para um formato comum, ex: JSON. A Tabela 2 mostra a classe de **ProductsController.cs** com a primeira API do microserviço de produtos.

```
[Route("api/products")]
public class ProductsController : Controller
{
    [HttpGet("{storeid}")]
    public IActionResult
    GetProductsByStoreId(Guid storeId)
    {
        return Ok();
    }
}
```

O atributo **Route**, definido na **Controller**, permite definir a URL base das APIs disponíveis nesta classe.

Segundo o diagrama, a primeira API do microserviço de Produto é a **api/products/{storeid}**, que é utilizada pelo **StoreCatalog**. Através desta URI o método **GetProductsByStoreId** é invocado e retorna todos os produtos da loja informada na requisição.

Com o atributo **HttpGet** definimos que esta API fará somente consulta de dados, sem alterar informações do recurso a ser pesquisado e que esta API receberá o parâmetro **storeid** que utilizará para obter somente os produtos filtrados pelo id da loja.

Definimos também que o retorno será o status HTTP 200 que significa OK, sem nenhum conteúdo, pois ainda não criamos a lista de produtos.

O próximo passo é definir os contratos das API e mensagens do microserviço de Produtos. Para que estes modelos sejam compartilhados entre os times, é necessário criar um projeto separado específico de contratos, onde estes modelos serão disponibilizados. Para isso clique em **File > Add > New Project**. Será apresentada uma janela (Figura 3) com todos os templates disponíveis. No canto esquerdo, localize e clique no item **Visual C# > .NET Core**. Ao lado selecione a opção **Class Library (.NET Core)** e clique **OK**.

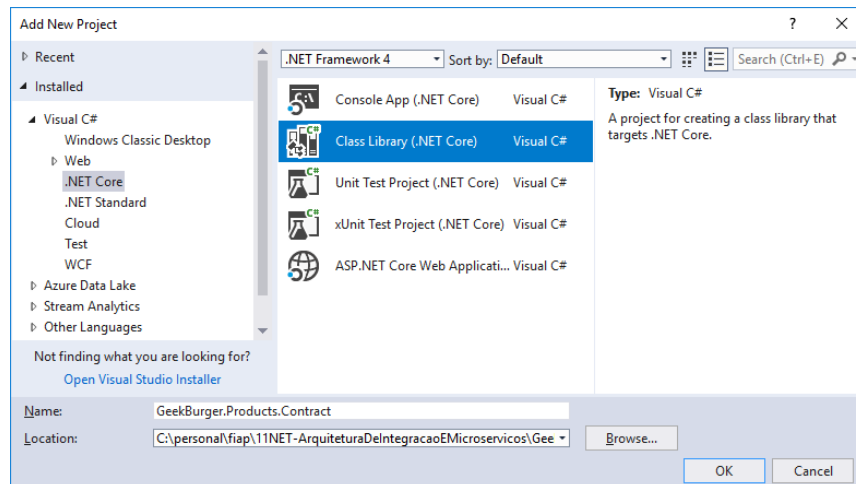


Figura 3

A Tabela 3 mostra as classes de **Products**, **Item** e **ProductsChanges**, classes que fazem parte do contrato do microserviço de Produtos.

<pre> public class Product { public Guid StoreId { get; set; } public Guid ProductId { get; set; } public string Name { get; set; } public string Image { get; set; } public List<Item> Items { get; set; } } public class Item { public Guid ItemId { get; set; } public string Name { get; set; } } </pre>	<p>Para o retorno da API GetProductsByStoreId criaremos os modelos ProductDto e ItemDto.</p> <p>É importante notar que os modelos que devemos criar inicialmente possuem um único objetivo que é retornar os dados da API. A este tipo de objeto dá-se o nome de DTO (Data Transfer Object).</p> <p>Estes modelos não são os mesmos que serão utilizados para armazenamento no banco de dados, caso exista um banco de dados.</p> <p>O tipo Guid é usado para identificar de forma única o recurso. Isso facilita a identificação única do recurso em todos os outros microserviços.</p>
<pre> public class ProductChangedMessage { public ProductState State { get; set; } public ProductToGet Product { get; set; } } public enum ProductState { Deleted = 2, Modified = 3, Added = 4 } </pre>	<p>Conforme o diagrama, caso haja alguma mudança nos produtos, o microserviço de Produtos deve disparar uma mensagem na rede para que assinantes (subscribers) possam receber a notificação. Para isso criaremos a mensagem ProductChangedMessage com o Produto que foi alterado.</p>

Tabela 3

Após criar as classes de contrato, vamos utilizá-las para retornar uma primeira lista de produtos (inicialmente hard-coded) na API `api/products/{storeid}` do projeto `GeekBurger.Products`. O código em questão é apresentado na Tabela 4.

<pre>public ProductsController() { var losAngelesStore = new Guid("8048e9ec-80fe-4bad-bc2a-e4f4a75c834e"); var beverlyHillsStore = new Guid("8d618778-85d7-411e-878b-846a8eef30c0"); var beef = new Item { ItemId = Guid.NewGuid(), Name = "beef" }; var mustard = new Item { ItemId = Guid.NewGuid(), Name = "mustard" }; var bread = new Item { ItemId = Guid.NewGuid(), Name = "bread" }; return new List<Product>() { new Product { ProductId = Guid.NewGuid(), Name = "Darth Bacon", Image = "hamb1.png", StoreId = losAngelesStore, Items = new List<Item> {beef, new Item { ItemId = Guid.NewGuid(), Name = "ketchup" }, bread } }, new Product { ProductId = Guid.NewGuid(), Name = "Cap. Spork", Image = "hamb2.png", StoreId = losAngelesStore, Items = new List<Item> { new Item { ItemId = Guid.NewGuid(), Name = "pork"}, mustard, new Item { ItemId = Guid.NewGuid(), Name = "whole bread" } } }, new Product { ProductId = Guid.NewGuid(), Name = "Beef Turner", Image = "hamb3.png", StoreId = beverlyHillsStore, Items = new List<Item> {beef, mustard, bread } } }; }</pre>	<p>Inicialmente é importante adicionar uma referência local à <code>GeekBurger.Products.Contract</code> no projeto <code>GeekBurger.Products</code>.</p> <p>No construtor a lista de produtos inicial é criada (inicialmente hard-coded com três produtos)</p>
<pre>[HttpGet("{storeid}")] public IActionResult GetProductsByStoreId(Guid storeId) { var productsByStore = Products.Where(product => product.StoreId == storeId).ToList(); if(productsByStore.Count <= 0) return NotFound(); return Ok(productsByStore); }</pre>	<p>Em seguida o método de <code>GetProductsByStoreId</code> é atualizado para obter todos os produtos da loja solicitante e retorná-los junto com o HTTP status 200 (OK). Caso nenhum produto seja encontrado, deve-se retornar 404 (NotFound).</p> <p>Por padrão, as APIs de controllers MVC retornam conteúdo no formato JSON.</p>

Tabela 4

Com este código já podemos hospedar esta API na Azure. Na aba de **Solution Explorer**, clique com o botão direito no projeto e clique em **Publish** (Figura 4).

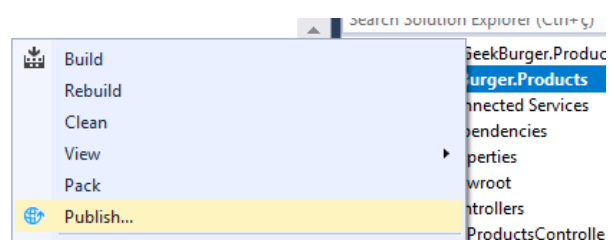


Figura 4

Será apresentada a página de Publish (Figura 5). Nesta página você deve selecionar **Microsoft Azure App Service**, manter selecionada a opção **Create New** logo abaixo, e clicar em **Create Profile**.

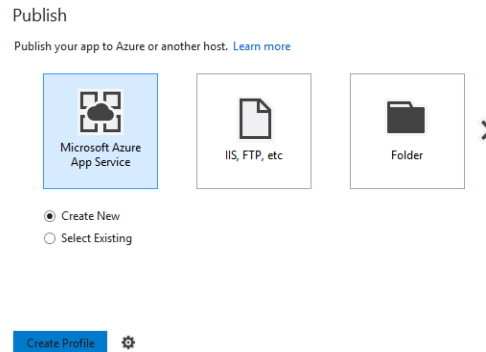


Figura 5

Será apresentada uma janela (Figura 6) chamada **Create App Service** que deve ser utilizada para a criação de uma nova Aplicação na Azure. Clique em **Change Type** e escolha o tipo **Api App**. Defina um **App Name** que deve ser o nome do seu microserviço, ex: GeekBurger-Products . Selecione a assinatura (**Subscription**) que você está utilizando, o grupo de recursos (Resource Group), que deve ser um grupo de recursos de uma localidade próxima e o plano de serviços do aplicativo (**App Service Plan**) que disponibiliza os recursos necessários para o funcionamento da sua aplicação web, portanto também deve estar em uma localidade próxima. Caso o plano de serviços de aplicativo não tenha sido criado, clique no botão de **new** ao lado da lista de opções de **App Service Plan** e na nova janela que abrir defina o nome, localidade e tamanho provisionado para este serviço e clique **OK**. Por fim, selecione o plano e clique em **Create** na janela de **Create App Service**.

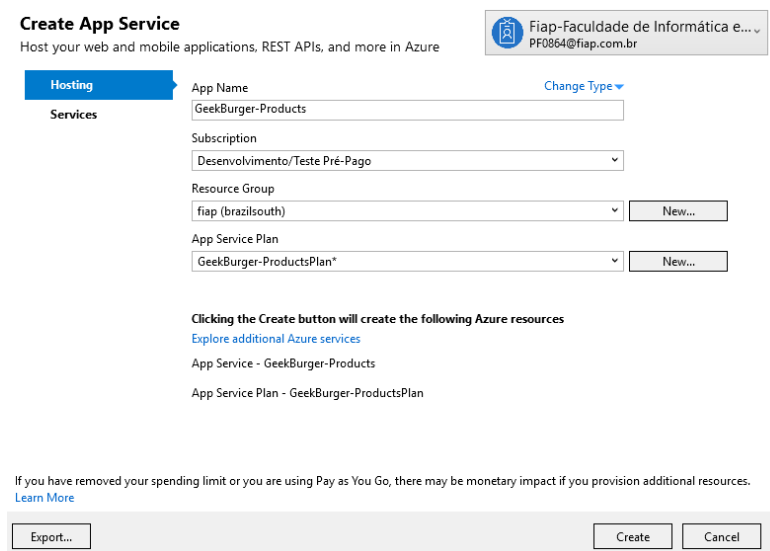


Figura 6

Na página de Publish o perfil recém criado aparecerá numa lista de seleção. Ao lado desta lista haverá o botão de Publish (Figura 7). Clique no botão de publish e aguarde alguns segundos.

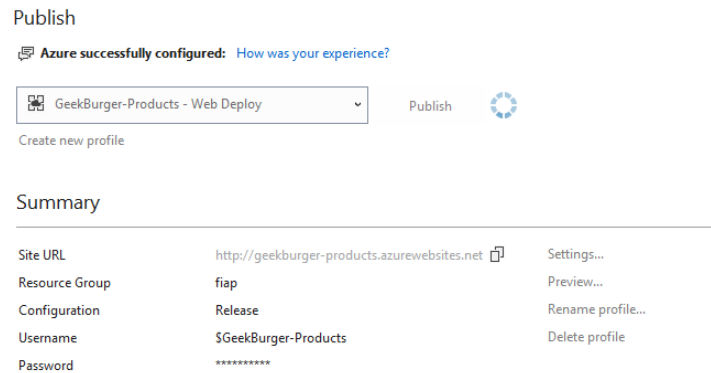


Figura 7

Uma janela de navegador abrirá automaticamente na URL raiz do seu aplicativo, ex: `http://geekburger-products.azurewebsites.net/`, no entanto a página que aparecerá no navegador será uma página de erro (Figura 8). Isso está correto pois não definimos nenhum conteúdo para a URL raiz do aplicativo. Para testar a API devemos utilizar Postman, uma ferramenta de apoio ao desenvolvimento de APIs.

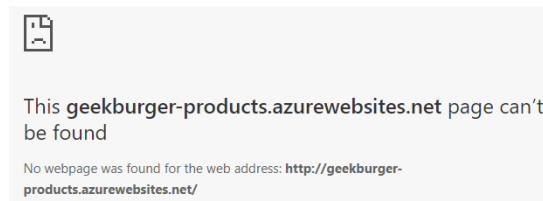


Figura 8

Na tela principal do Postman (Figura 9), clique na opção **Request**. Na tela subsequente defina uma nome para a requisição e uma breve descrição, crie uma nova collection clicando no botão **+ Create Collection** e clique em **Save**.

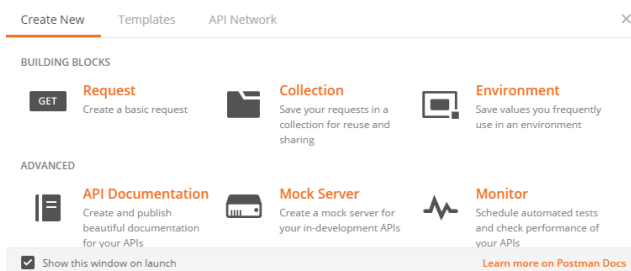


Figura 9

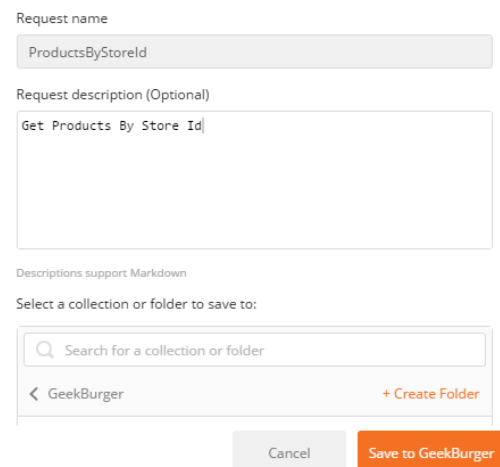


Figura 10

Informe a URL da sua API, ex: `http://geekburger-products.azurewebsites.net/api/products/8048e9ec-80fe-4bad-bc2a-e4f4a75c834e` (este GUID é da loja de Los Angeles) e clique em **Send**. O resultado é apresentado abaixo no formato JSON: uma lista de produtos e itens da loja em questão (Figura 11).

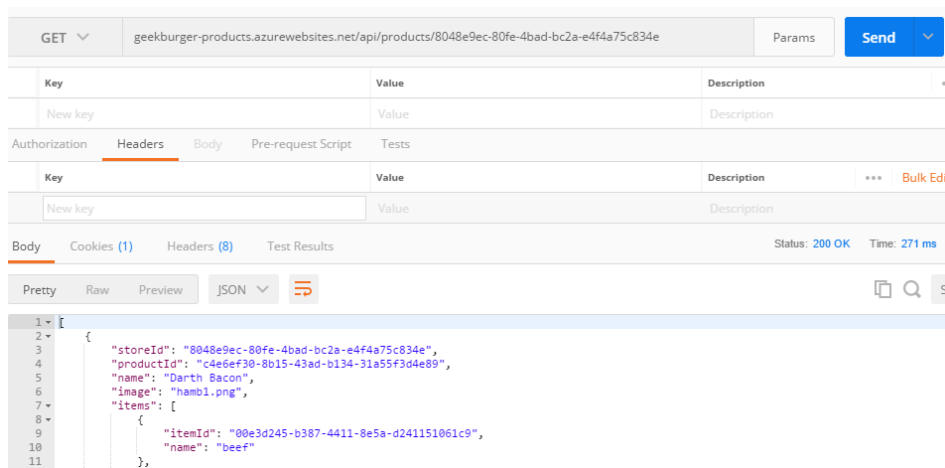


Figura 11

O NuGet é um gerenciador de pacotes da plataforma .NET. Através dele os times podem criar, compartilhar e consumir pacotes de código. Um pacote NuGet é um arquivo ZIP com a extensão `.nupkg` que contém o código compilado, outros arquivos relacionados a esse código (suas dependências nas versões corretas) e um manifesto descritivo que inclui informações como o número de versão de um pacote.

Os desenvolvedores de uma biblioteca criam arquivos de pacote e os publicam em um host. Os consumidores de pacote recebem esses pacotes, os adicionam aos seus projetos e chamam a funcionalidade da biblioteca no código de projeto. Saiba mais sobre NuGet em: [Uma Introdução ao NuGet](#).

Para publicar pacotes no NuGet, você precisa ter o NuGet.exe na sua máquina. Caso você não tenha o NuGet.exe na sua máquina, faça o download do executável em [aqui](#) e registre o caminho do executável na variável PATH da sua máquina.

NOTA: Neste exemplo vamos abordar a criação do pacote NuGet do projeto de contrato de Produtos, no entanto **cada grupo deve criar o pacote do seu microserviço descrito no Case, atribuído a cada grupo durante a aula.**

Vamos ao próximo passo que é gerar um pacote NuGet do projeto de contrato de Produtos. Localize a janela de **Solution Explorer** (Figura 12). Nesta janela encontre o projeto **GeekBurger.Products.Contract** e clique com o botão direito no projeto e clique na opção **Properties** do menu. Após o clique, a página de propriedades do projeto será apresentada (Figura 5).

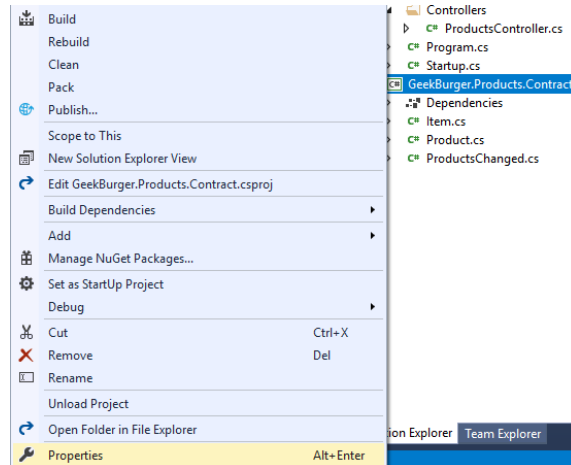


Figura 12

Na página de propriedades do projeto **GeekBurger.Products.Contract** (Figura 13), clique na aba **Package** e marque a opção **Generate NuGet package on build**.

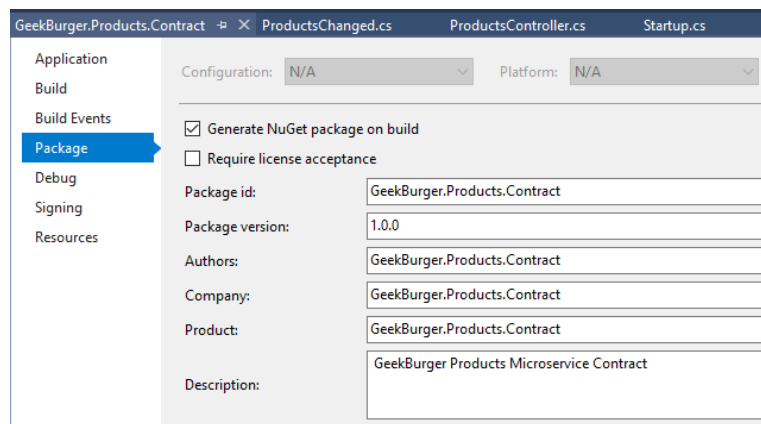


Figura 13


Recompile a solução (ctrl + shift + b) e identifique na janela de output do visual studio a linha:

```
Successfully created package
'C:\...\GeekBurger.Products.Contract\bin\Debug\GeekBurger.Products.Contract.1.0.0.nupkg'
```

Abra uma instância de power shell e digite o seguinte commando para adicionar o pacote de contrato ao servidor NuGet da nossa disciplina:

```
nuget push C:\...\GeekBurger.Products.Contract\bin\Debug\GeekBurger.Products.Contract.1.0.0.nupkg
-Source http://nugetfiap12.azurewebsites.net/nuget 8db3cebeef8fd6fb85c4c187160b53d1

Pushing GeekBurger.Products.Contract.1.0.0.nupkg to 'http://nugetfiap12.azurewebsites.net/nuget'...
PUT http://nugetfiap12.azurewebsites.net/nuget/
Created http://nugetfiap12.azurewebsites.net/nuget/ 1961ms
Your package was pushed.
```

Para visualizar o contrato adicionado, vá até o Visual Studio, navegue no menu **Tools > NuGet Package Manager** e clique em **Package Manager Settings** (Figura 14). Uma nova janela será apresentada. No menu lateral esquerdo da janela (Figura 15) escolha a opção **NuGet Package Manager > Package Sources**, e clique no botão de  no canto superior direito. Na parte inferior direita preencha os campos de **Name** com o nome do repositório: **NuGet Server Fiap** e o campo **Source** com a URL do servidor: **https://nugetfiap12.azurewebsites.net/nuget**. Verifique que o checkbox da nova fonte de pacotes está marcado, clique em **Update** e logo em seguida em **OK**.

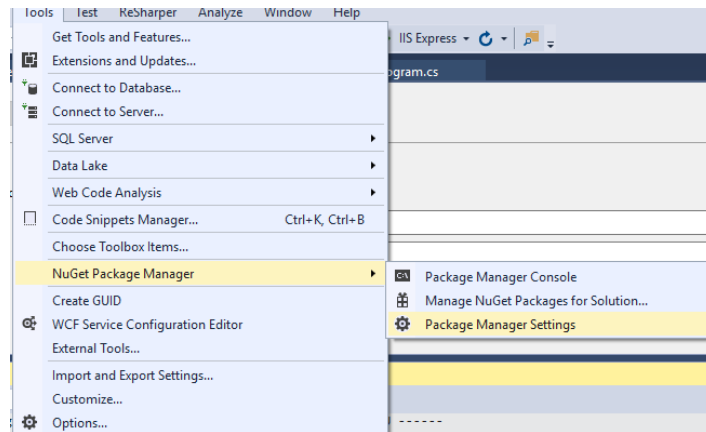


Figura 14

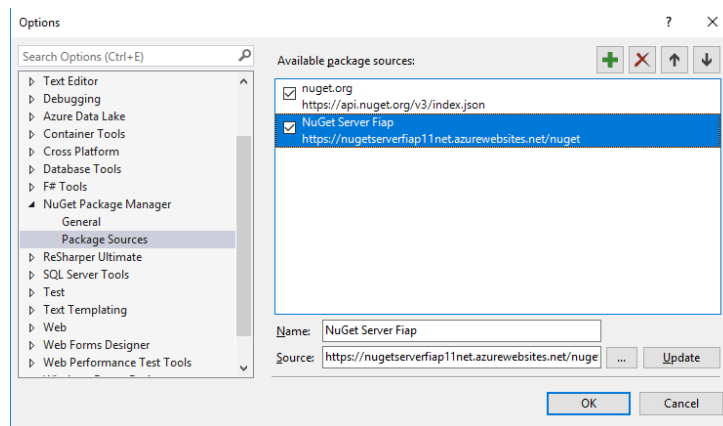


Figura 15

Localize novamente a janela de Solution Explorer e clique com botão direito na solução (Figura 16). Selecione a opção **Manage NuGet Packages for Solution**. Na página do NuGet que é aberta ao clicar nesta opção (Figura 17) localize no canto a caixa de seleção intitulada **Package Source**. Selecione a opção **NuGet Server Fiap** e clique na aba **Browse** no canto superior esquerdo. Veja que o pacote **GeekBurger.Products.Contract** aparece listado como pacote disponível neste servidor.

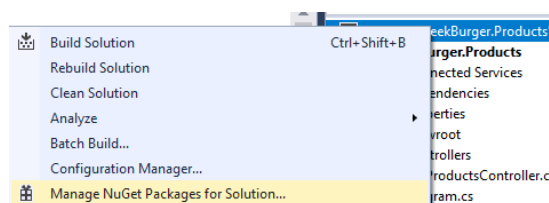


Figura 16

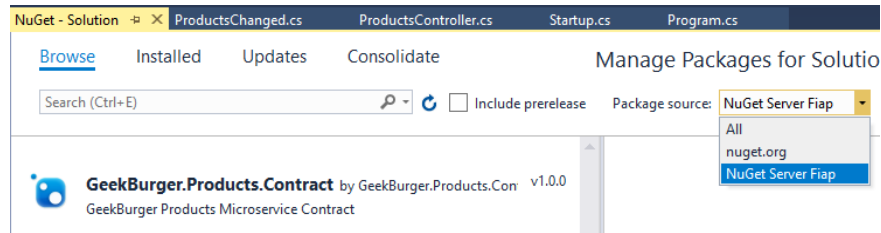


Figura 17

Dica para o Microserviço de Store Catalog. Criei um linqpad que mostra como a lista já filtrada de produtos por restrição deve ser em seguida filtrada pela lista de áreas disponíveis para produção. Veja que mesmo que o produto não tenha o ingrediente restrito ele só pode ser produzido em áreas que restringem este ingrediente para evitar a contaminação cruzada. Por sua vez, as áreas que restringem certos ingredientes não estão disponíveis para produzir alimentos que contenham estes ingredientes. É para lidar com estas duas restrições cruzadas que o Store Controller deve fazer o filtro apresentado abaixo:

```
void Main() {
    var areas = new List<Production>() {
        //comment this list and no product will be available for the user, why?
        new Production(){ ProductionId = 1, Restrictions = new string[]{"soy", "dairy", "gluten", "sugar"}},
        new Production(){ ProductionId = 2, Restrictions = new string[]{} },
        //try to add gluten to the restriction list of this area. All products will be available for the user, why?
        new Production(){ ProductionId = 3, Restrictions = new string[] { "soy" } } };

    //products already filtered by user restriction
    var filteredProducts = new List<Product>(){
        new Product(){ ProductId = 1111, Ingredients = new string[]{"sugar"} },
        new Product(){ ProductId = 1112, Ingredients = new string[]{"egg"} },
        new Product(){ ProductId = 1113, Ingredients = new string[]{"egg", "peanut"} },
        new Product(){ ProductId = 1114, Ingredients = new string[]{"egg", "sugar"} } };

    var user = new User(){ Restrictions = new string[] { "soy", "gluten" } };

    //any area allowed for this user
    var allowedAreas =
        areas.Where(area => user.Restrictions.All(restriction => area.Restrictions.Contains(restriction) ) );

    //products that can be produced in allowed areas for this user
    var allowedProducts = filteredProducts.Where(product => product
        //all ingredients from this product are not in the restriction list of at least one allowed area
        .Ingredients.All(ingredient =>
            allowedAreas.Any(area => !area.Restrictions.Contains(ingredient) ) ));

    allowedProducts.Dump();
}

public class User { public string[] Restrictions {get;set;} }
public class Product { public int ProductId {get;set;} public string[] Ingredients {get;set;} }
public class Production { public int ProductionId {get;set;} public string[] Restrictions {get;set;} }
```

Relembrando, o objetivo final de cada grupo nesta atividade é declarar os contratos e implementar ainda de forma hard-coded todas as APIs de seus microserviços, publicar os projetos de contrato no NuGet e publicar a API no Azure, e comitar o código no GitHub. Os seguintes microserviços possuem APIs: Users, Ingredients, StoreCatalog, UI, Production, Orders e Dashboard.