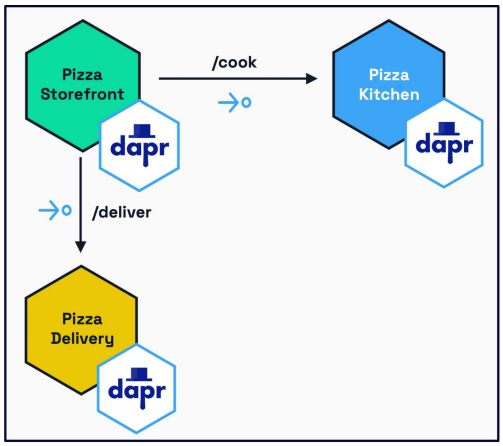
Challenge 2 - Service Invocation

Overview

In this challenge, you will create three services that will change the status of your order via service invocations. For that, you will:

- Create a new service called `pizza-storefront` with a `/order` endpoint.
- Create a new service called `pizza-kitchen` with a `/cook` endpoint.
- Create a new service called `pizza-delivery` with a `/delivery` endpoint.
- Use Dapr's Service Invocation API to call the `/cook` and `/deliver` endpoints from the `pizza-storefront` app.



To learn more about the Dapr Service Invocation building block, refer to the Dapr docs.

Project structure

All three services follow the structure below:

```
PizzaKitchen/
├── Controllers/
|   └── AppController.cs        # Where all endpoints are defined
├── Models/
|   └── OrderModels.cs          # Order model definition
└── Services/
    └── AppService.cs           # Service implementation. This is where our Dapr
APIs will be implemented.
```

Create the Kitchen service

1. Navigate to the PizzaKitchen service:

```
cd PizzaKitchen
```

2. Inside `/Controllers/CookController.cs`, analyse the endpoint `/cook` which is simply calling a service that you will implement in the next step.

```
[HttpPost]
public async Task<ActionResult<Order>> Cook(Order order)
{
    _logger.LogInformation("Starting cooking for order: {OrderId}", order.OrderId);
    var result = await _cookingService.CookPizzaAsync(order);
    return Ok(result);
}
```

3. Navigate to `/Services/CookService.cs` and update `CookPizzaAsync` with the code below:

```
public async Task<Order> CookPizzaAsync(Order order)
{
    var stages = new (string status, int duration)[]
    {
        ("cooking_preparing_ingredients", 2),
        ("cooking_making_dough", 3),
        ("cooking_adding_toppings", 2),
        ("cooking_baking", 5),
        ("cooking_quality_check", 1)
    };

    try
    {
        foreach (var (status, duration) in stages)
        {
            order.Status = status;
            _logger.LogInformation("Order {OrderId} - {Status}", order.OrderId,
status);

            await Task.Delay(TimeSpan.FromSeconds(duration));
        }

        order.Status = "cooked";
        _logger.LogInformation("Order {OrderId} - {Status}", order.OrderId,
order.Status);

        return order;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error cooking order {OrderId}", order.OrderId);
        order.Status = "cooking_failed";
        order.Error = ex.Message;
        return order;
    }
}
```

This function applies 5 different cooking states to the order and waits a certain duration to log each one of them.

Create the Delivery service

1. Navigate to the PizzaDelivery service:

```
cd PizzaDelivery
```

2. Inside `/Controllers/DeliveryController.cs`, analyse the endpoint `/deliver` which is simply calling a service that you will implement in the next step.

```
[HttpPost("deliver")]
public async Task<ActionResult<Order>> Deliver(Order order)
{
    _logger.LogInformation("Starting delivery for order: {OrderId}", order.OrderId);
    var result = await _deliveryService.DeliverPizzaAsync(order);
    return Ok(result);
}
```

3. Navigate to `/Services/DeliveryService.cs` and update `DeliverPizzaAsync` with the code below:

```
public async Task<Order> DeliverPizzaAsync(Order order)
{
    var stages = new (string status, int duration)[]
    {
        ("delivery_finding_driver", 2),
        ("delivery_driver_assigned", 1),
        ("delivery_picked_up", 2),
        ("delivery_on_the_way", 5),
        ("delivery_arriving", 2),
        ("delivery_at_location", 1)
    };

    try
    {
        foreach (var (status, duration) in stages)
        {
            order.Status = status;
            _logger.LogInformation("Order {OrderId} - {Status}", order.OrderId,
status);

            await Task.Delay(TimeSpan.FromSeconds(duration));
        }

        order.Status = "delivered";
        _logger.LogInformation("Order {OrderId} - {Status}", order.OrderId,
order.Status);
        return order;
```

```
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error delivering order {OrderId}", order.OrderId);
        order.Status = "delivery_failed";
        order.Error = ex.Message;
        return order;
    }
}
```

Similar to what we did in the Kitchen Service, this function applies 7 different delivery states to the order and waits a certain duration to log each one of them to the console.

Create the Storefront service

1. Now navigate to `PizzaStorefront` and add the Dapr dependencies:

```
cd PizzaStorefront
dotnet add package Dapr.Client
dotnet add package Dapr.AspNetCore
```

2. Register the DaprClient

Open `Program.cs` and add the `DaprClient` registration to the `ServiceCollection`:

```
builder.Services.AddControllers().AddDapr();
```

This enables the dependency injection of the `DaprClient` in other classes.

3. Navigate to `/Controllers/StorefrontController.cs` and check for the endpoint implementation.

```
[HttpPost]
public async Task<ActionResult<Order>> CreateOrder(Order order)
{
    _logger.LogInformation("Received new order: {OrderId}", order.OrderId);
    var result = await _storefrontService.ProcessOrderAsync(order);
    return Ok(result);
}
```

4. Navigate to `/Services/StorefrontService.cs`. Add the Dapr import statement.

```
using Dapr.Client;
```

5. Add a Dapr client reference and update the constructor with the Dapr dependency:

```
private readonly DaprClient _daprClient;

public StorefrontService(DaprClient daprClient, ILogger<StorefrontService> logger)
{
    _daprClient = daprClient;
    _logger = logger;
}
```

5. Modify `ProcessOrderAsync` with the following:

```
public async Task<Order> ProcessOrderAsync(Order order)
{
    var stages = new (string status, int duration)[]
    {
        ("validating", 1),
        ("processing", 2),
        ("confirmed", 1)
    };

    try
    {
        // Set pizza order status
        foreach (var (status, duration) in stages)
        {
            order.Status = status;
            _logger.LogInformation("Order {OrderId} - {Status}", order.OrderId,
status);

            await Task.Delay(TimeSpan.FromSeconds(duration));
        }

        _logger.LogInformation("Starting cooking process for order {OrderId}",
order.OrderId);

        // Use the Service Invocation building block to invoke the endpoint in the
pizza-kitchen service
        var response = await _daprClient.InvokeMethodAsync<Order, Order>(
            HttpMethod.Post,
            "pizza-kitchen",
            "cook",
            order);

        _logger.LogInformation("Order {OrderId} cooked with status {Status}",
            order.OrderId, response.Status);

        // Use the Service Invocation building block to invoke the endpoint in the
pizza-delivery service
        _logger.LogInformation("Starting delivery process for order {OrderId}",
order.OrderId);

        response = await _daprClient.InvokeMethodAsync<Order, Order>(
```

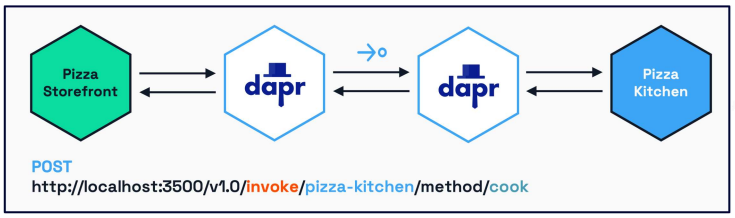
```
            HttpMethod.Post,
            "pizza-delivery",
            "delivery",
            order);

        _logger.LogInformation("Order {OrderId} delivered with status {Status}",
            order.OrderId, response.Status);

        return order;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error processing order {OrderId}", order.OrderId);
        order.Status = "failed";
        order.Error = ex.Message;

        return order;
    }
}
```

Breaking down the code above:

1. First the `DaprClient.InvokeMethodAsync` is used to create an invocation object with the Dapr `app-id` of the service you want to invokea and its wndpoint:

```
var response = await _daprClient.InvokeMethodAsync<Order, Order>(
                HttpMethod.Post,  # HTTP method to be used
                "pizza-delivery", # app-id of the service we want to discover
                "delivery",       # endpoint to the called
                order);           # body of the request
```

The code above wraps an HTTP call to the host `localhost` with the port `3504`. This is not calling the *pizza-delivery* service directly, but rather the sidecar of the *pizza-storefront* service. The responsibility of making the service invocation call is then passed to the sidecar, as the picture below illustrates:



This way, services only need to communicate to their associated sidecar over localhost and the sidecar handles the service discovery and invocation capabilities.

Run the application

It's now time to run all three applications.

1. In your terminal, ensure you are in the `/PizzaStorefront` folder and run the command below:

```
dapr run --app-id pizza-storefront --app-protocol http --app-port 8002 --dapr-http-port 3502 -- dotnet run
```

2. Open a new terminal window and navigate to `/PizzaKitchen` folder. Run the command below:

```
dapr run --app-id pizza-kitchen --app-protocol http --app-port 8003 --dapr-http-port 3503  -- dotnet run
```

3. Open a third terminal window and navigate to `/PizzaDelivery` folder. Run the command below:

```
dapr run --app-id pizza-delivery --app-protocol http --app-port 8004 --dapr-http-port 3504  -- dotnet run
```

> **Important**
> If you are using Consul as a naming resolution service, add `--config ../resources/config/config.yaml` before `-- dotnet run` on your Dapr run command.

Test the service

Use VS Code REST Client

Open `Endpoints.http` and find the `Direct Pizza Store Endpoint (for testing)` endpoint call. Click on `Send request`

Navigate to the `pizza-storefront` terminal, where you should see the following logs:

```
== APP == info: PizzaStorefront.Controllers.StorefrontController[0]
== APP ==       Received new order: 123
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Order 123 - validating
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Order 123 - processing
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Order 123 - confirmed
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Starting cooking process for order 123
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Order 123 cooked with status cooked
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Starting delivery process for order 123
== APP == info: PizzaStorefront.Services.StorefrontService[0]
== APP ==       Order 123 delivered with status delivered
```

The logs for `pizza-kitchen` should read:

```
== APP == info: PizzaKitchen.Controllers.CookController[0]
== APP ==       Starting cooking for order: 123
```

```
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooking_preparing_ingredients
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooking_making_dough
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooking_adding_toppings
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooking_baking
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooking_quality_check
== APP == info: PizzaKitchen.Services.CookService[0]
== APP ==       Order 123 - cooked
```

Finally, on `pizza-delivery`:

```
== APP == info: PizzaDelivery.Controllers.DeliveryController[0]
== APP ==       Starting delivery for order: 123
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_finding_driver
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_driver_assigned
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_picked_up
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_on_the_way
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_arriving
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivery_at_location
== APP == info: PizzaDelivery.Services.DeliveryService[0]
== APP ==       Order 123 - delivered
```

Use *cURL*

Alternatively, open a third terminal window and create a new order via cURL:

```
curl -H 'Content-Type: application/json' \
    -d '{ "orderId": "1", "pizzaType": "pepperoni", "size": "large", "customer": {
"name": "John Doe", "address": "123 Main St", "phone": "555-0123" } }' \
    -X POST \
    http://localhost:8002/storefront/order
```

Next steps

Now that the services are updating the event information for every order step, you need to make sure that this information is being updated in the Redis state store. You will do this in the next challenge using Dapr Pub/Sub!