MicroServices Implementation using .NET

Nagaraju B
Sr. Program Manager
Development Head - .NET

Microsoft Certified Trainer
Microsoft Certified Solution Architect
http://nbende.wordpress.com

Twitter
@nbende

FaceBook
nbende

Multi-stage builds in Dockerfile

from Docker 17.05 and higher

The core idea is that you can separate the Dockerfile execution process in stages, where a stage is an initial image followed by one or more commands, and the last stage determines the final image size.

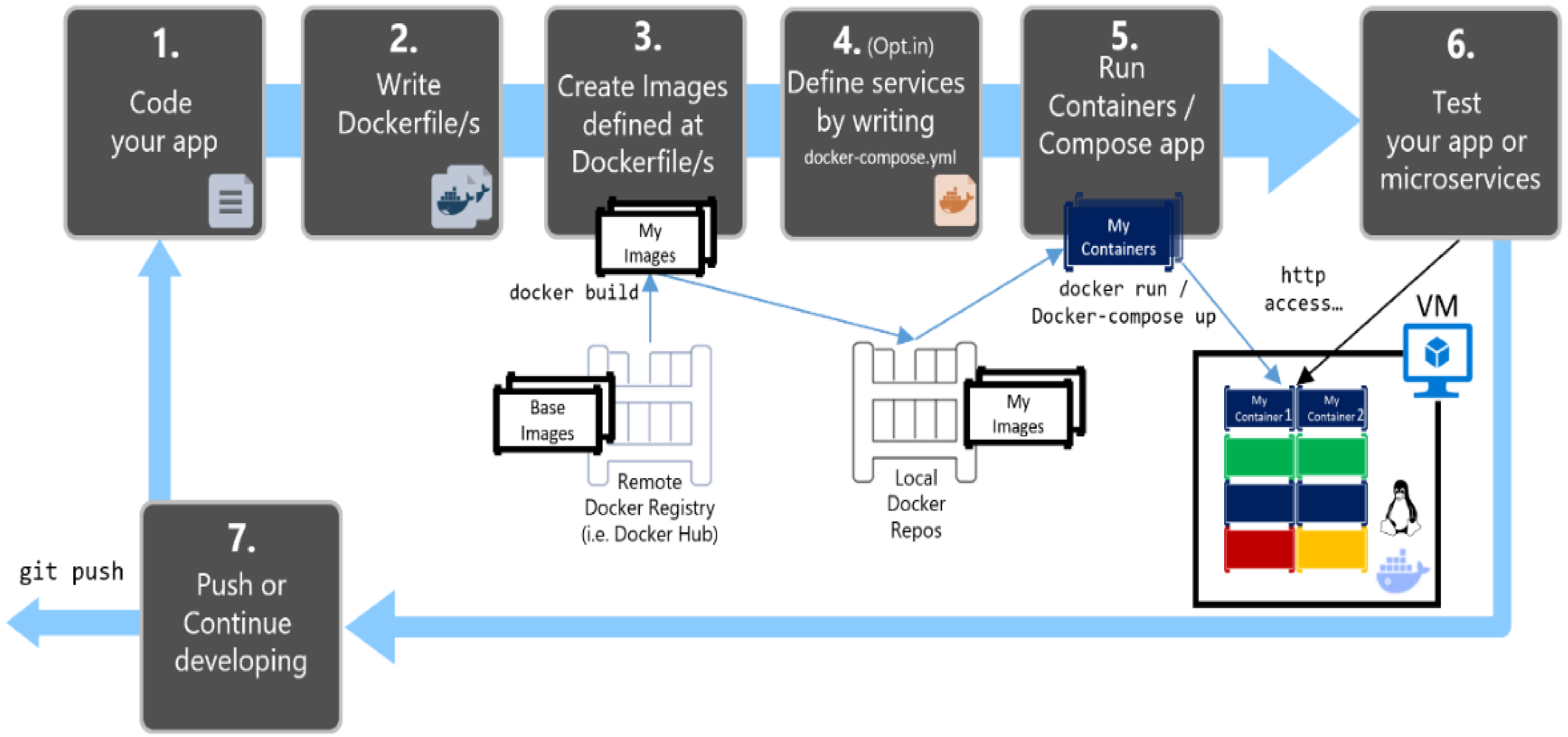
How it works

1. Use a base SDK image (doesn’t matter how large), with everything needed to build and publish the application to a folder and then

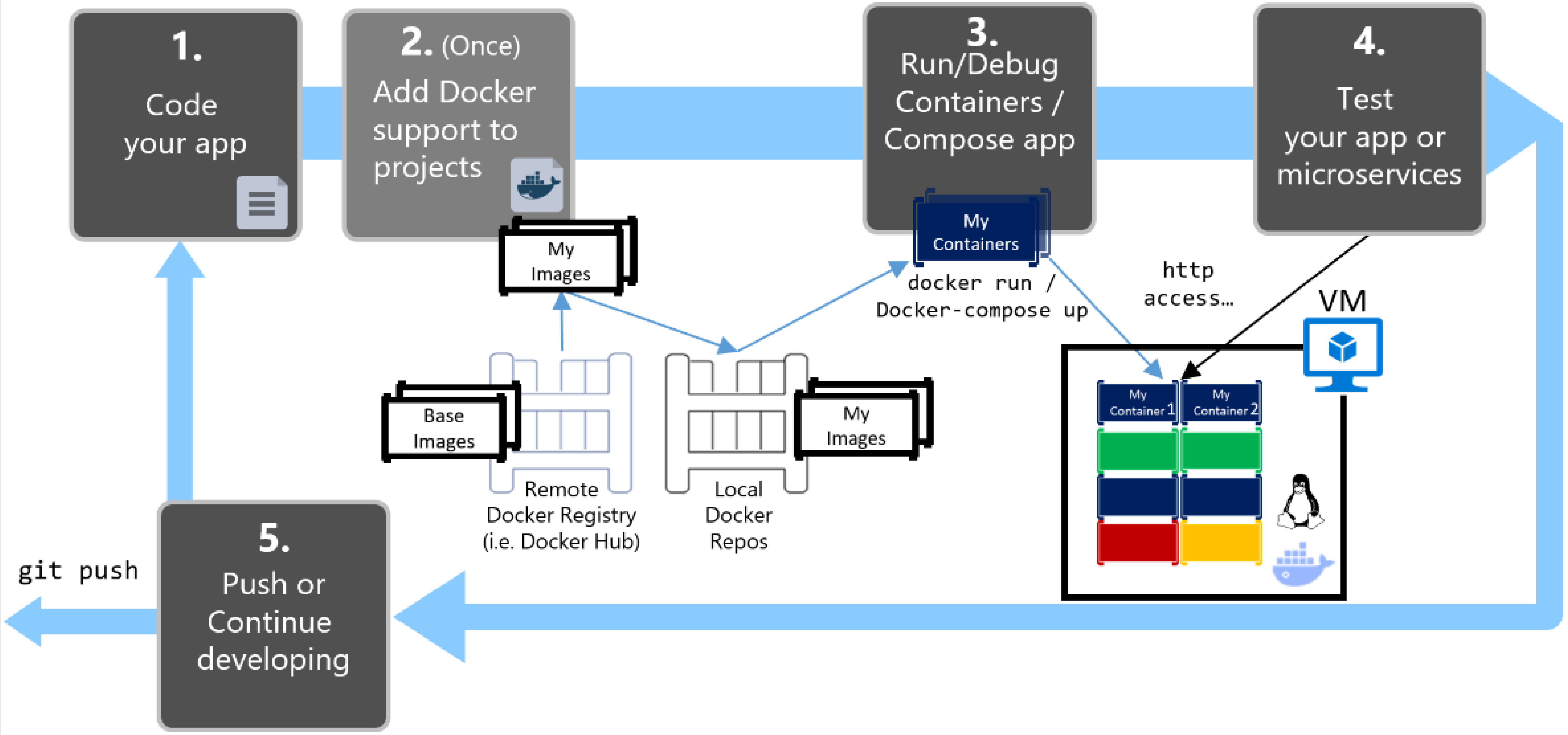
2. Use a base, small, runtime-only image and copy the publishing folder from the previous stage to produce a small final image.

VS 2019 creates this optimized multistage dockerfile only

Inner-Loop development workflow for Docker apps


1. Code your app
2. Write Dockerfile/s
3. Create Images defined at Dockerfile/s
4. (Opt.in) Define services by writing docker-compose.yml
5. Run Containers / Compose app
6. Test your app or microservices
7. Push or Continue developing
git push
My Images
docker build
Base Images
Remote Docker Registry (i.e. Docker Hub)
Local Docker Repos
My Images
My Containers
docker run / Docker-compose up
http access…
VM
My Container 1
My Container 2

VS development workflow for Docker apps


1.
Code your app
2. (Once)
Add Docker support to projects
3.
Run/Debug Containers / Compose app
4.
Test your app or microservices
My Images
My Containers
docker run / Docker-compose up
http access…
VM
Base Images
Remote Docker Registry (i.e. Docker Hub)
My Images
Local Docker Repos
My Container 1
My Container 2
git push
5.
Push or Continue developing

Example 1

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2
RUN mkdir app
COPY dockr-poc/dist/* /app/
EXPOSE 80
ENTRYPOINT ["dotnet", "/app/ dockr-poc.dll"]
```

From top to bottom, this file instructs Docker to:

Use the ASP.NET Core 2.2 image as the base image

Execute a command to create a folder app in the image

Copy all files from the subfolder dockr-poc/dist of the host to the app folder inside the image

Expose port 80

Execute dotnet /app/ dockr-poc.dll when the container is started

Example 2

```dockerfile
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY src/Services/Catalog/Catalog.API/Catalog.API.csproj …
COPY src/BuildingBlocks/HealthChecks/src/Microsoft.AspNetCore.HealthChecks …
COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions.HealthChecks …
COPY src/BuildingBlocks/EventBus/IntegrationEventLogEF/ …
COPY src/BuildingBlocks/EventBus/EventBus/EventBus.csproj …
COPY src/BuildingBlocks/EventBus/EventBusRabbitMQ/EventBusRabbitMQ.csproj …
COPY src/BuildingBlocks/EventBus/EventBusServiceBus/EventBusServiceBus.csproj …
COPY src/BuildingBlocks/WebHostCustomization/WebHost.Customization …
COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions …
COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions …
RUN dotnet restore src/Services/Catalog/Catalog.API/Catalog.API.csproj
COPY . .
WORKDIR /src/src/Services/Catalog/Catalog.API
RUN dotnet build Catalog.API.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish Catalog.API.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Catalog.API.dll"]
```

Swagger

standard for the APIs description metadata domain

should include Swagger description metadata with any kind of microservice, either data-driven microservices or more advanced domain-driven microservices

The heart of Swagger is the Swagger specification, which is API description metadata in a JSON or YAML file

The specification creates the RESTful contract for your API, detailing all its resources and operations in both a human- and machine-readable format for easy development, discovery, and integration.

Swashbuckle

One good option to automate Swagger metadata generation for ASP.NET Core REST API applications

automatically generates Swagger metadata for your ASP.NET Web API projects

Swashbuckle.AspNetCore (Nuget Package)

ORM’s

Simple and Faster to build

Don't use raw ADO - Use an ORM

Entity Framework – Full ORM

- Pros
  - Developer Productivity
  - Compile-time safety with LINQ Queries
  - Extremely quick to add new CRUD operations
  - Built in Unit of Work
  - Migration support
- Cons
  - Less performant – need proper techniques
  - Less control over the queries generated
  - Heavier in – some cases

Dapper – Micro ORM

- Pros
  - Performance near ADO
  - More control over the queries
  - Extremely simple to setup
  - Stack Overflow beta tests
- Cons
  - SQL strings = Big column name refactorings are harder
  - Less features than EF

DDD and CQRS Architectural Patterns

Simple and Faster to build

DDD

Domain-Driven Design is a method and a process for designing complex systems.

objective of domain design is to understand the exact domain problems and then draft a model that can be written in any language or set of technologies

characteristics of the DDD

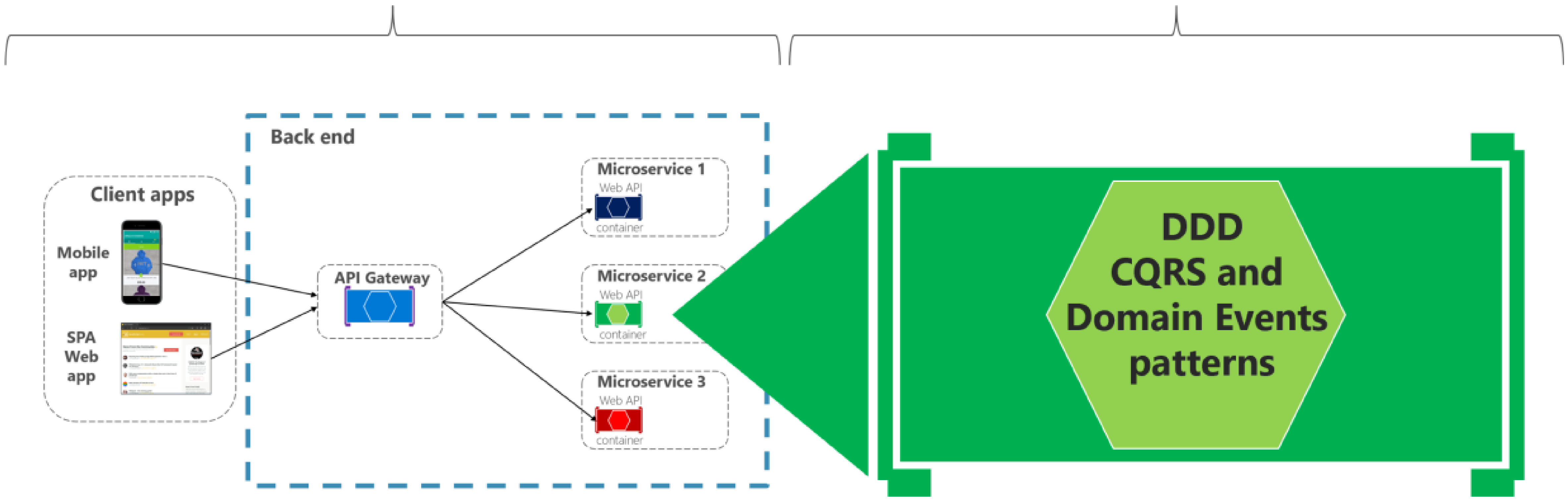
- A domain model should focus on a specific business model and not multiple business models.
- It should be reusable
- It should be designed so that it can be called in a loosely coupled way, unlike the
- rest of the system.
- It should be designed independently of persistence implementations.
- It should be pulled out from a project to another location, so it should not be
- based on any infrastructure framework.

Importance for microservices

DDD is the blueprint and can be implemented by microservices. In other words, once DDD is done, we can implement it using microservices

External architecture per application

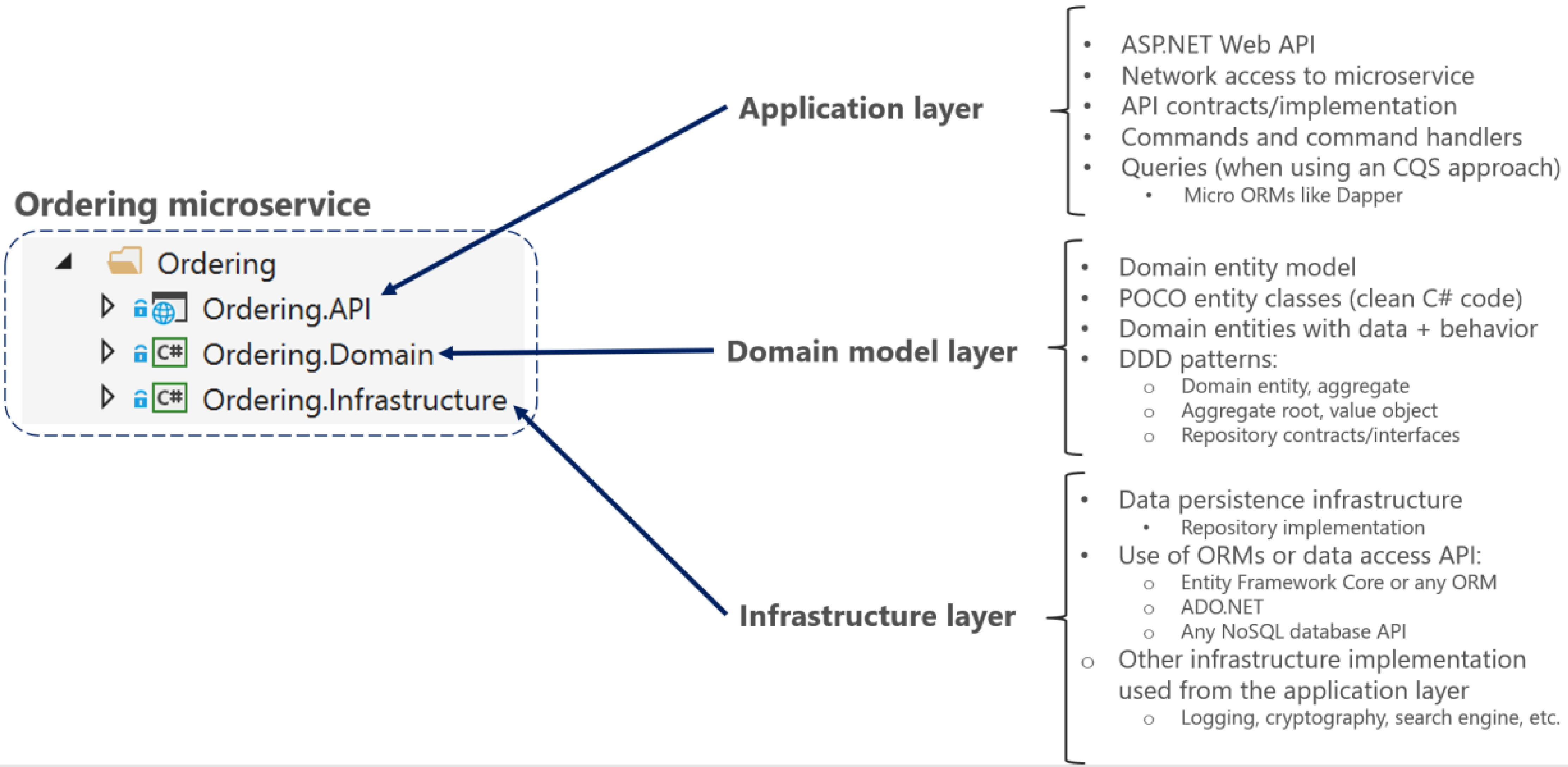
Internal architecture per microservice



- External microservice patterns
- API Gateway
- Resilient communication
- Pub/Sub and event driven

Internal DDD patterns in addition to SOLID principles and Dependency Injection

Layers in a Domain-Driven Design Microservice



Application layer

- ASP.NET Web API
- Network access to microservice
- API contracts/implementation
- Commands and command handlers
- Queries (when using an CQS approach)
  - Micro ORMs like Dapper

Domain model layer

- Domain entity model
- POCO entity classes (clean C# code)
- Domain entities with data + behavior
- DDD patterns:
  - Domain entity, aggregate
  - Aggregate root, value object
  - Repository contracts/interfaces

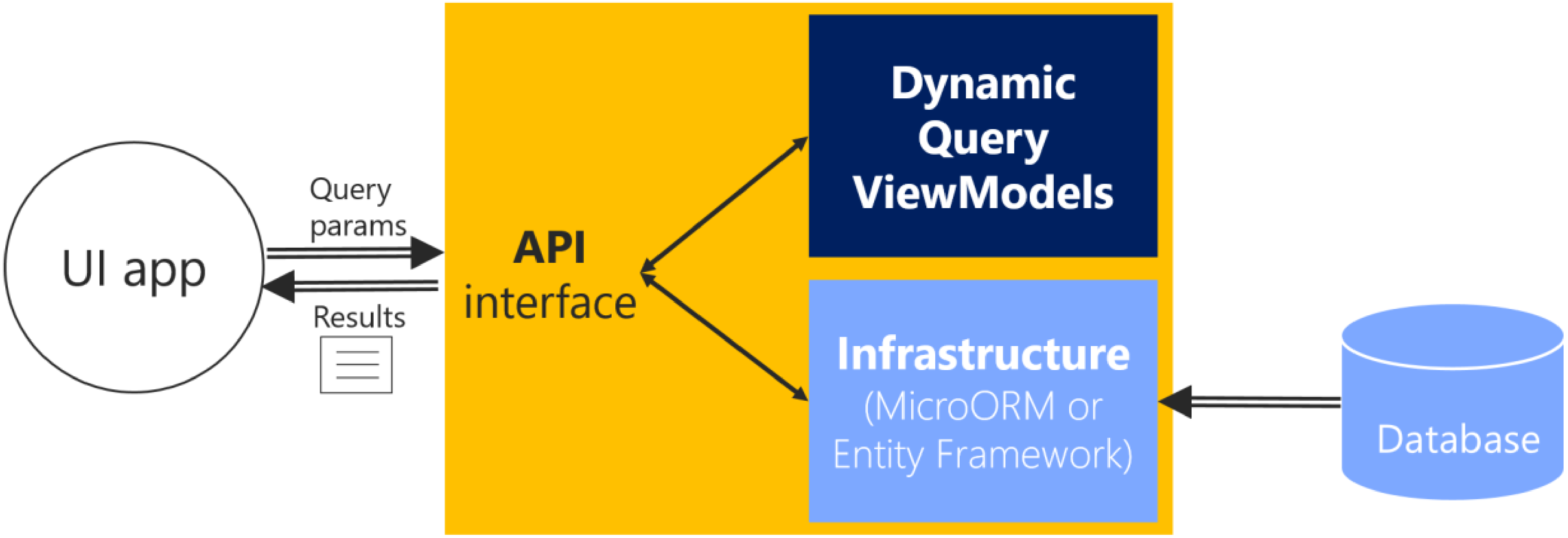
Infrastructure layer

- Data persistence infrastructure
  - Repository implementation
- Use of ORMs or data access API:
  - Entity Framework Core or any ORM
  - ADO.NET
  - Any NoSQL database API
- Other infrastructure implementation used from the application layer
  - Logging, cryptography, search engine, etc.

CQRS - Command and Query Responsibility Segregation

is an architectural pattern that separates the models for reading and writing data

High level “Queries-side” in a simplified CQRS



Thank You