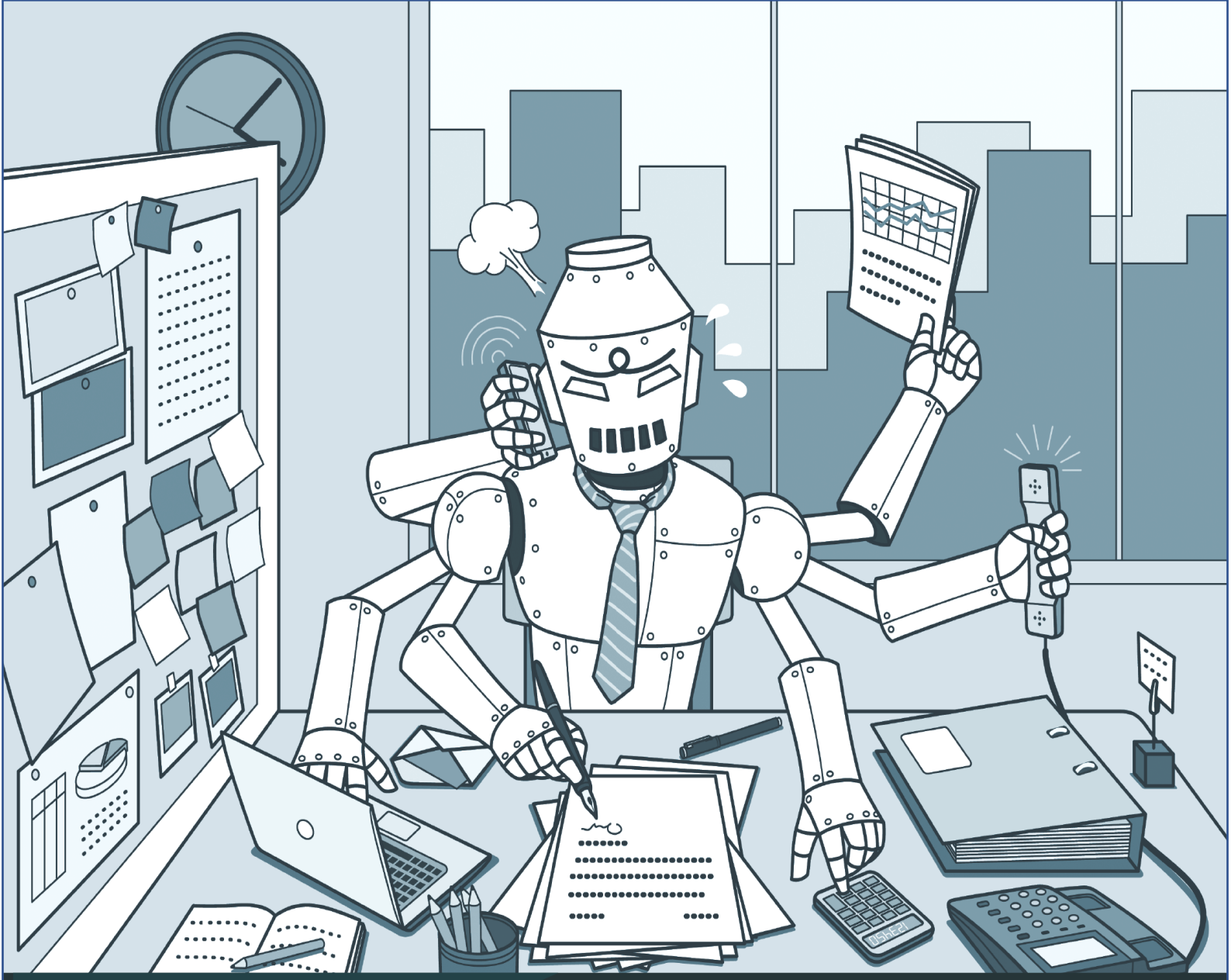

SOFTWARE ARCHITECTURE FOR BUSY DEVELOPERS

Talk and act like a software architect in one weekend

by Stéphane Eyskens

Packt

Chapter 1

Figures

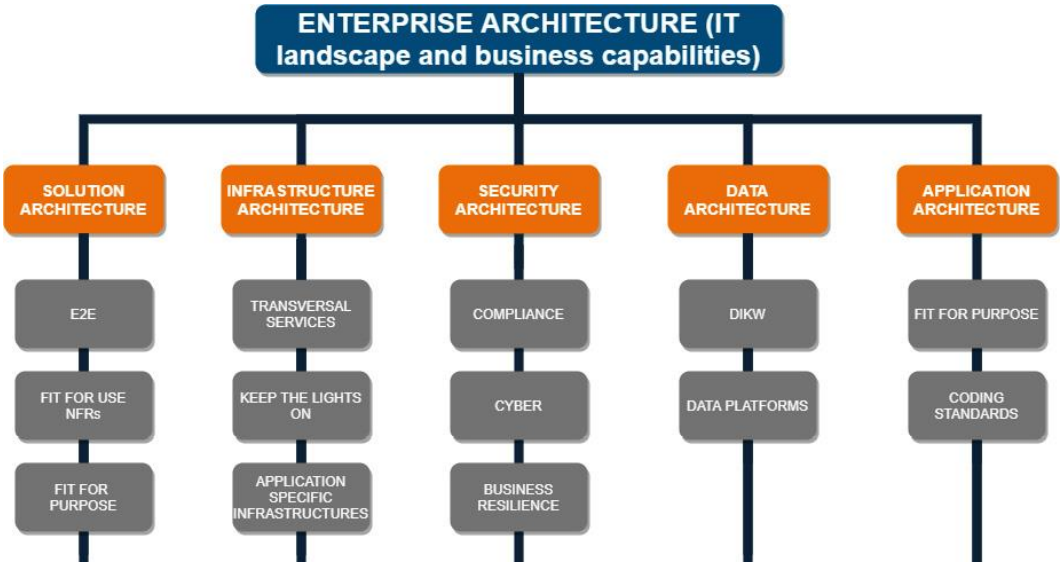

Figure 1.1 – Architecture disciplines: main duties

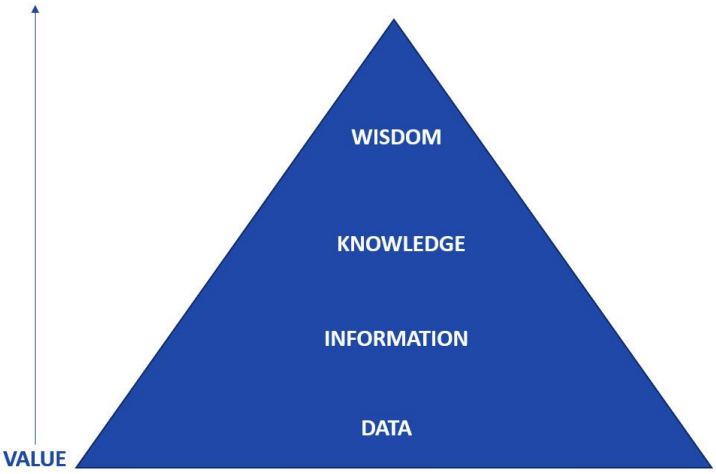

Figure 1.2 – DIKW pyramid

Data	31, 3, 3000 31, 3, 3100 31, 3, 3500
Information	Day: 31, Month: March, Concurrent Users: 3000
Knowledge	3000 concurrent users visited our web site on March 31. We know that this is way above our daily average, which is about 650 concurrent users. March 31 is always a busy day, year after year.
Wisdom	We'll make sure to restock warehouses before March 31.

Figure 1.3 – DIKW pyramid example

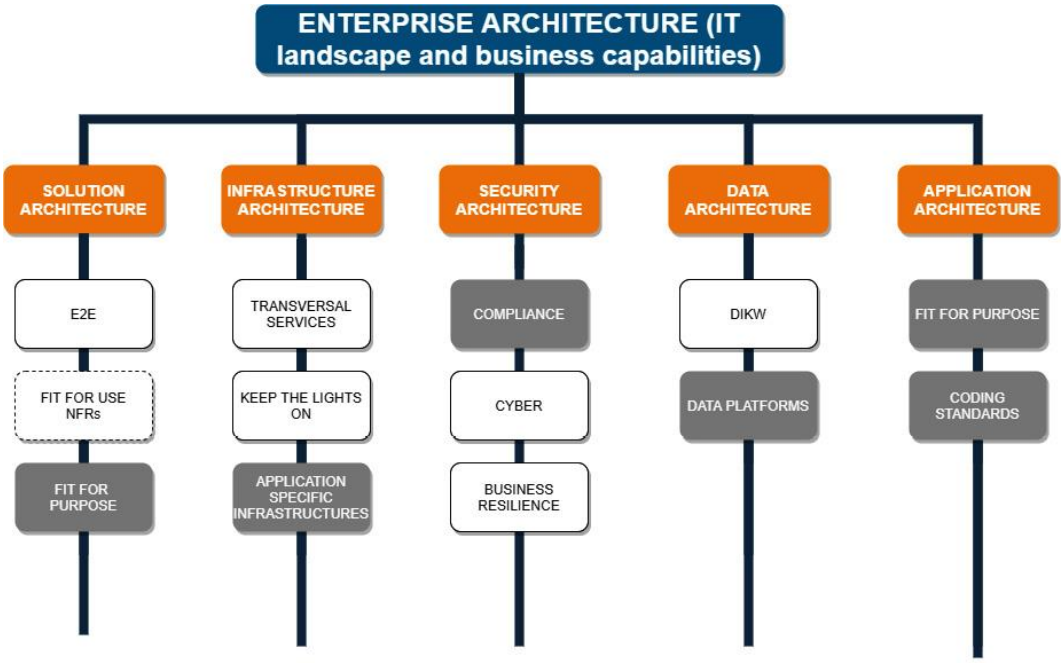

Figure 1.4 – Software architecture within the architecture landscape

Links

Definition of Software Architect: https://en.wikipedia.org/wiki/Software_architect

Chapter 2

Figures

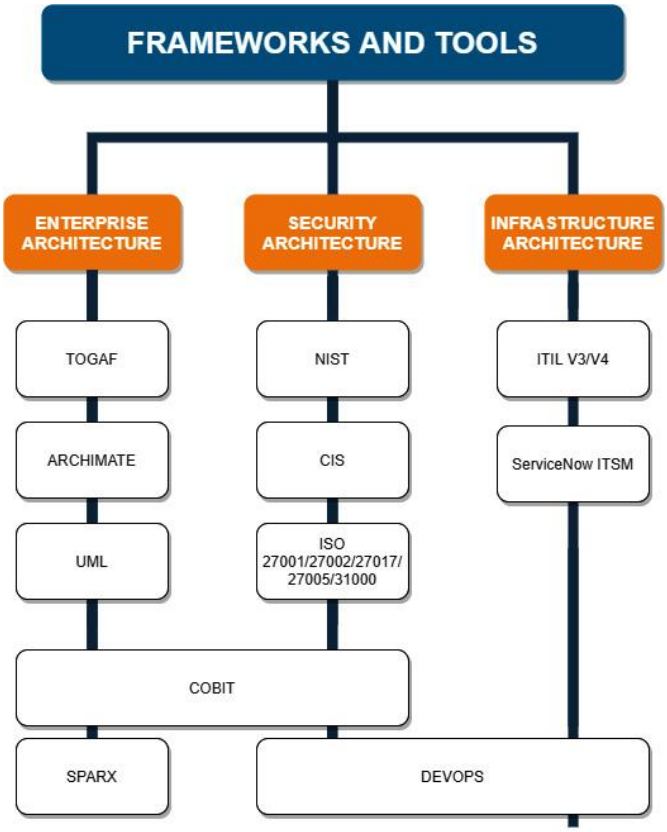

Figure 2.1 – Frameworks and tools

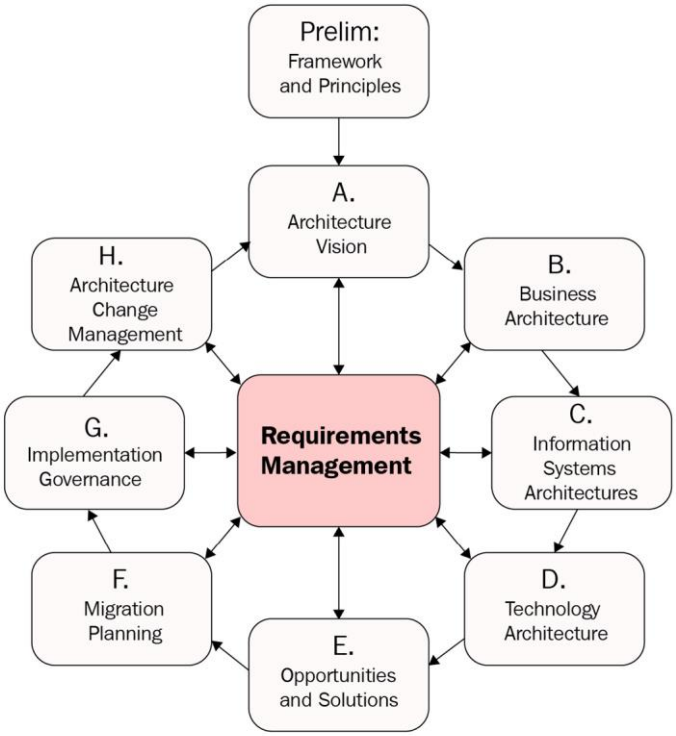

Figure 2.2 – TOGAF ADM

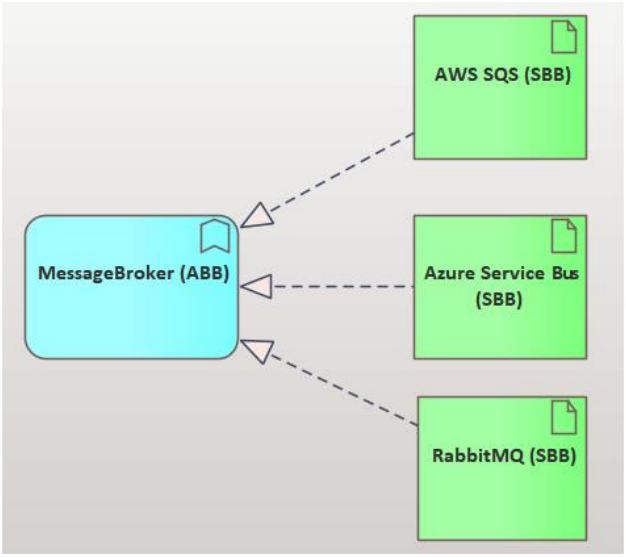

Figure 2.3 – ABB and SBB

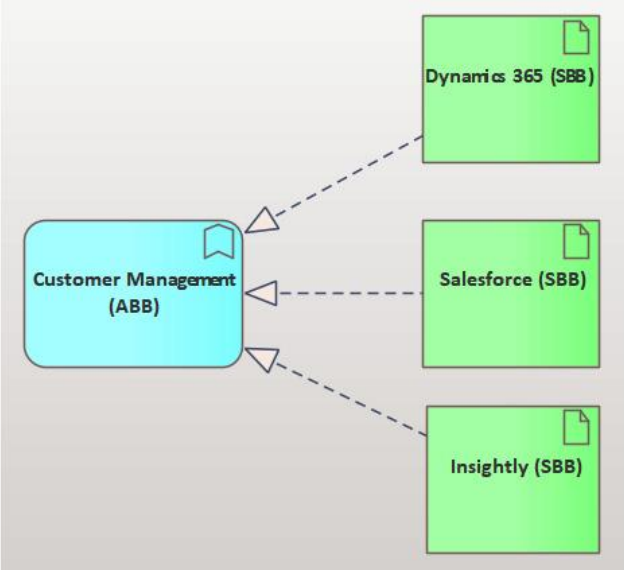

Figure 2.4 – ABBs and SBBs in a business context

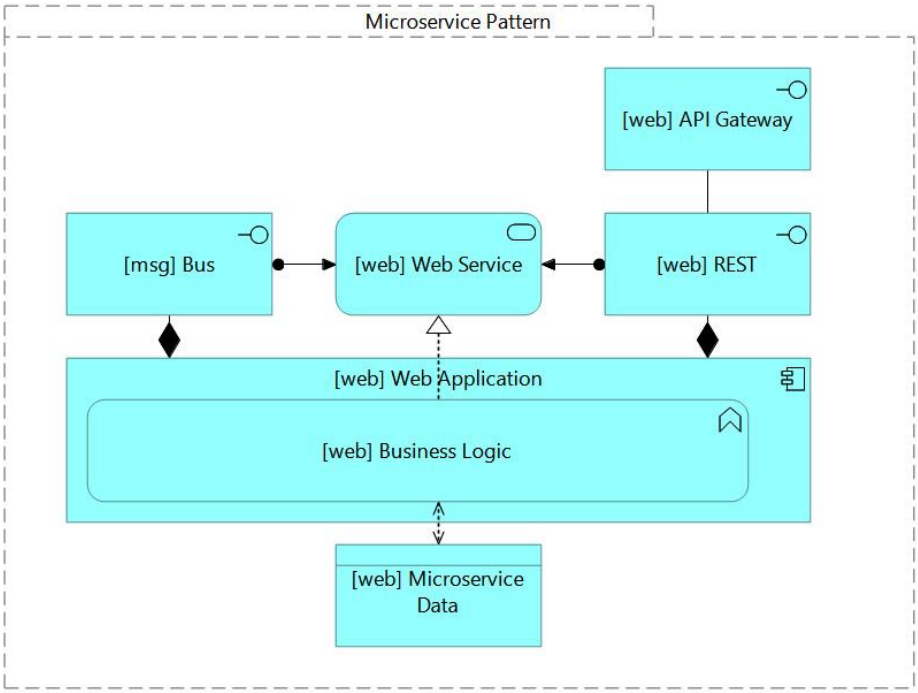

Figure 2.5 – Microservice pattern

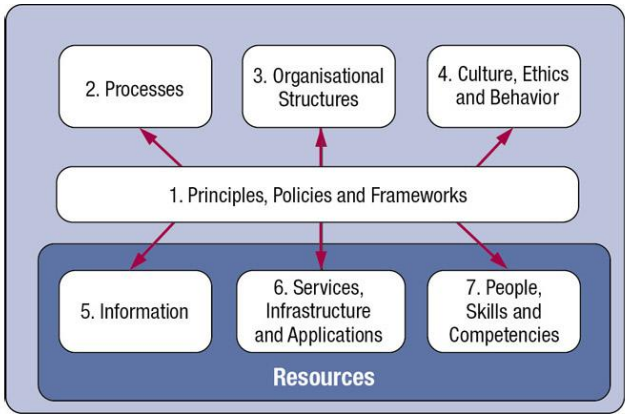

Figure 2.6 – COBIT's seven enablers

Risk Scenario Title	Resource Unavailability
Risk Scenario Category	Portfolio establishment and maintenance
Risk Scenario Description	
Resource allocation is often subject to change due to a switch of priorities across projects of program XXX.	
Risk Scenario Components	
Threat Type: Failure in resource allocation	
Actor: Internal resource allocation is managed by the program manager in cooperation with the IT leadership team.	
Event: Ineffective execution of *APO05 - Manage Portfolio process*	
Asset/Resource (Cause): Process APO05 is not well applied.	
Duration: Extended because the new system might live with important delays	
Occurrence: Timing is noncritical	
Detection: Slow as the key resources will progressively be made unavailable with new coming priorities	
Lag: Delayed as the effect of staff unavailability will be visible at a later stage.	
Possible Risk Response	
Avoid: Pre-allocate resources and do not switch later.	
Accept: N/A	
Share/Transfer: N/A	
Mitigate: Re-prioritize the program and ensure knowledge transfer in case of resource switch.	

Figure 2.7 – Non-technical risk scenario

Risk Scenario Title	Symmetric key compromised
Risk Scenario Category	Infrastructure theft or destruction
Risk Scenario Description	
Symmetric keys could be compromised if stored improperly or used maliciously, leaving the door open to unattended access to Service Bus.	
Risk Scenario Components	
Threat Type: Malicious	
Actor: Internal, malicious insiders (most likely). External, Attackers (less likely).	
Event: Theft	
Asset/Resource (Cause): Process BAI10 is not well applied.	
Duration: Extended	
Occurrence: Timing is noncritical	
Detection: Most probably slow	
Lag: Immediate as the attacker can immediately authenticate against service bus	
Possible Risk Response	
Avoid: Use Azure AD to authenticate and prevent access to *RootManageSharedAccessKey*	
Accept: N/A	
Share/Transfer: N/A	
Mitigate • Store keys in Key vault • Renew keys frequently • Monitor the *get secret* and *list key* operations	

Figure 2.8 – Technical risk scenario

Links

ArchiSurance Case Study: https://publications.opengroup.org/y163

Archimatetool download: https://www.archimatetool.com/download/

Chapter 3

Figure

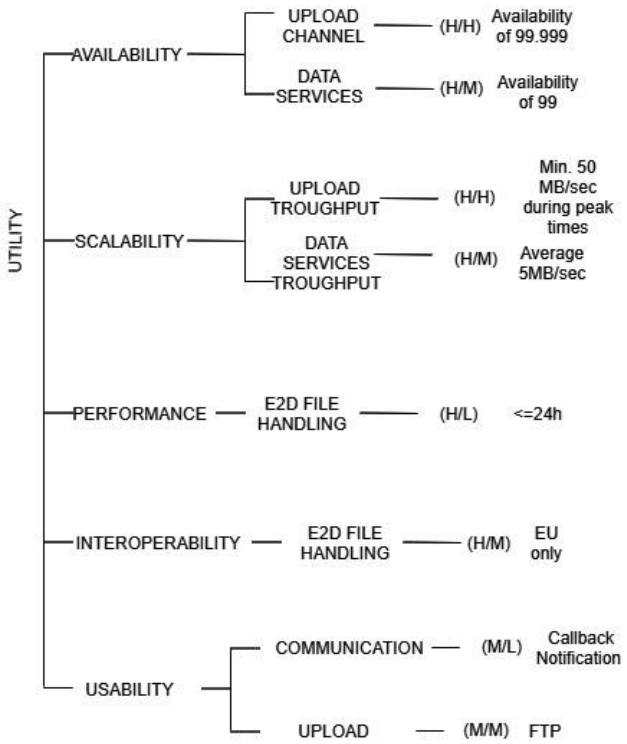

Figure 3.1 – Utility tree

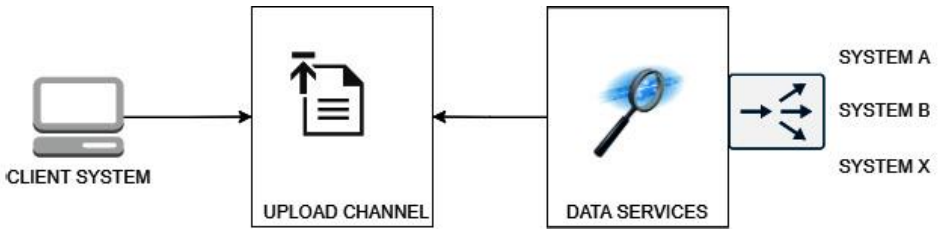

Figure 3.2 – Our simplified component diagram

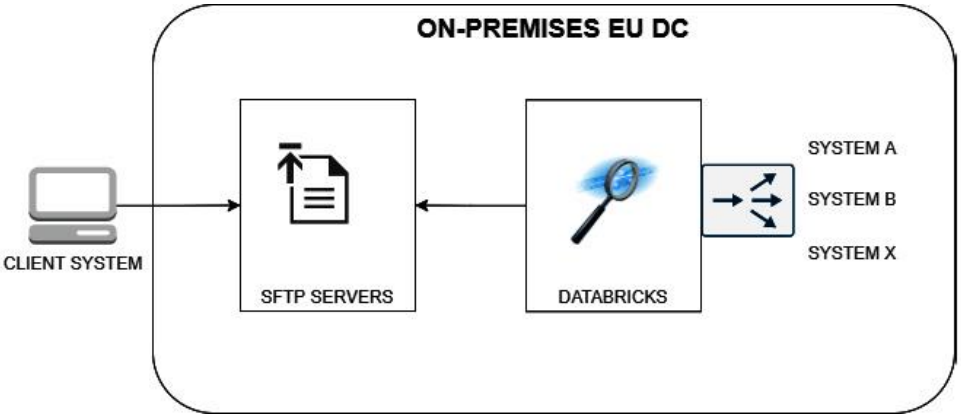

Figure 3.3 – High-level architecture proposal: first option

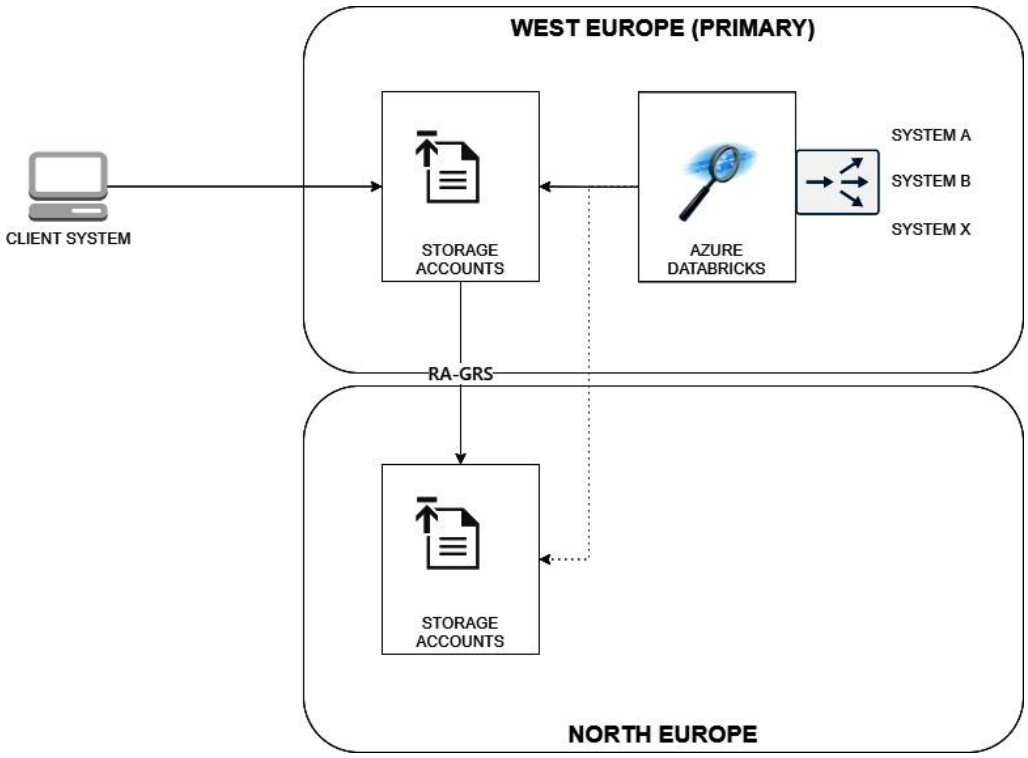

Figure 3.4 – High-level architecture proposal: second option

Tables

S1 – Description	Operations on the platform must happen during planned downtime. The impact over the upload channel(s) should be minimal and short.
Attribute	Availability.
Business value/ architectural impact	High/Medium.
Stimulus	Maintenance tasks.
Environment	Underlying upload channel infrastructure.
Artifact	Upload channels.
Response	Operations occur within an agreed timeframe and maintenance takes less than 2 hours.

Table 3.1 – First scenario

S2 – Description	Upload channel(s) outage
Attribute	Availability
Business value/ architectural impact	High/High
Stimulus	Outage
Environment	Channel hosting platform
Artifact	Upload channel
Response	Automatic restore of upload channel within < 10 minutes

Table 3.2 – Second scenario

S3 – Description	Data service(s) outage
Attribute	Availability
Business value/ architectural impact	Medium/Low
Stimulus	Outage
Environment	Data platform
Artifact	Data services
Response	Degraded user experience (UX) until service(s) is/are back up and running, with an RTO of a maximum of 6 hours

Table 3.3 – Third scenario

S4 – Description	Data sent by customers must be handled E2E within Europe.
Attribute	Interoperability.
Business value/ architectural impact	High/Medium.
Stimulus	File transfer and handling.
Environment	At runtime.
Artifact	Customer data file.
Response	Requests are treated in Europe 100% of the time.

Table 3.4 – Fourth scenario

S5 – Description	Customers prefer to send files over FTP.
Attribute	Usability.
Business value/ architectural impact	Medium/Medium.
Stimulus	File transfer.
Environment	At runtime.
Artifact	Customer data file.
Response	FTP Secure (FTPS)/Secure FTP (SFTP) channel(s) is/are exposed through port 21 or 22.

Table 3.5 – Fifth scenario

S6 – Description	During peak times, there could be up to 250 concurrent customers sending data files.
Attribute	Scalability.
Business value/ architectural impact	High/High.
Stimulus	Increase in memory, CPU, and input/output (I/O).
Environment	In normal circumstances and during peak times.
Artifact	Upload channel.
Response	Provisioned channels will be able to ingest 5 megabytes (MB)/second; additional capacity will be made available to accommodate 50 MB/second during peak times.

Table 3.6 – Sixth scenario

Links

List of system quality attributes: https://en.wikipedia.org/wiki/List_of_system_quality_attributes

Chapter 4

Figures

Figure 4.1 – Literal meaning of a monolith

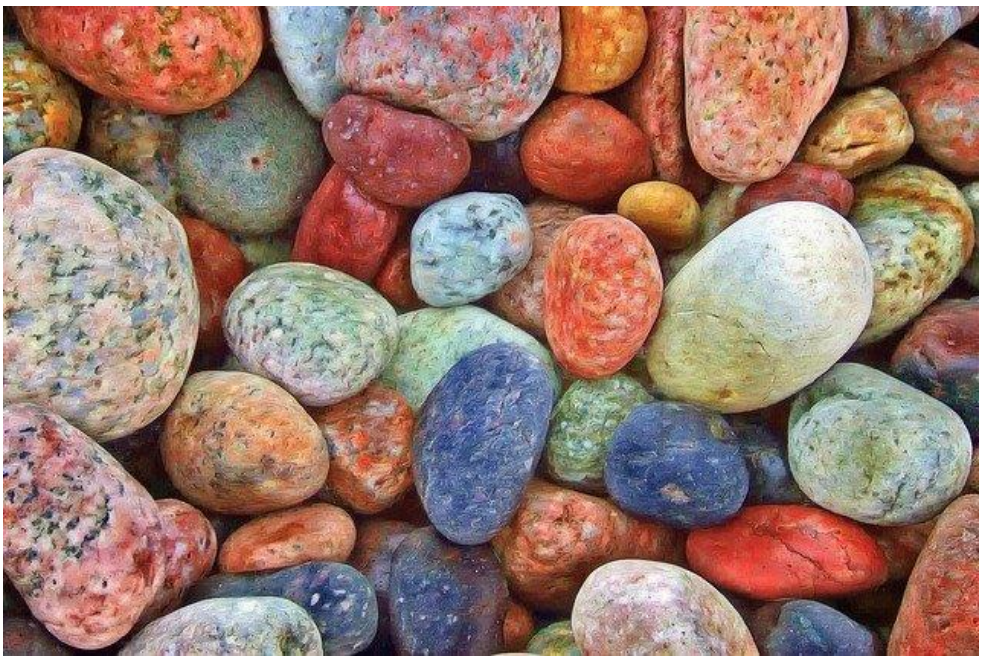

Figure 4.2 – Small monoliths

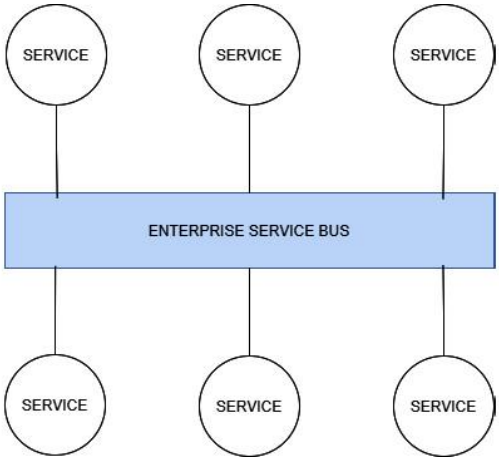

Figure 4.3 – Typical SOA implementation

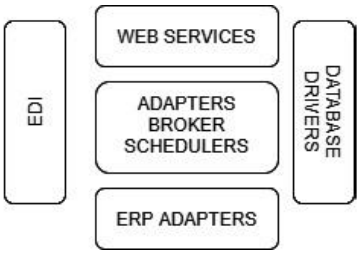

Figure 4.4 – Typical ESB duties

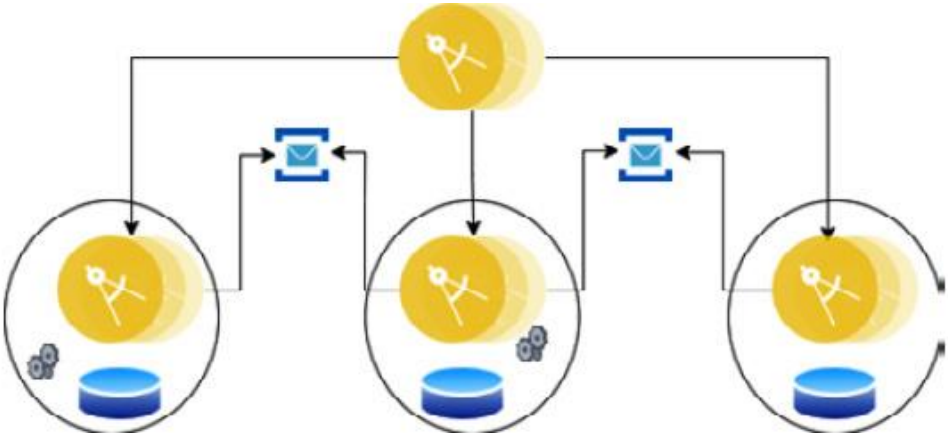

Figure 4.5 – Microservice architecture

Links

Protocol Buffers: https://developers.google.com/protocol-buffers

If you want to see microservices in action, I recommend that you look at the open source sample application at https://github.com/dotnet-architecture/eShopOnContainers or its variant at https://github.com/dotnet-architecture/eShopOnDapr, which simulates an online eShop.

Chapter 5

Technical requirements

To make some abstract concepts more concrete, some design patterns will be illustrated with .NET code samples. If you want to test the code locally, you will need **Visual Studio** or **Visual Studio Code** (**VS Code**). Both can be downloaded for free from Microsoft websites.

Note that the code is provided for illustration purposes only. All the code samples and diagrams are available at https://github.com/PacktPublishing/Software-Architecture-for-Humans.

Figures

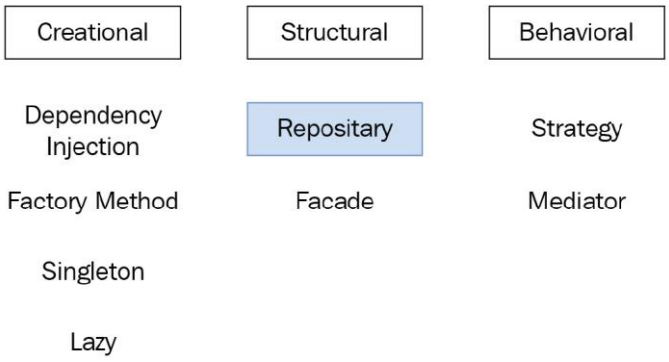

Figure 5.1 – Design patterns we will focus on

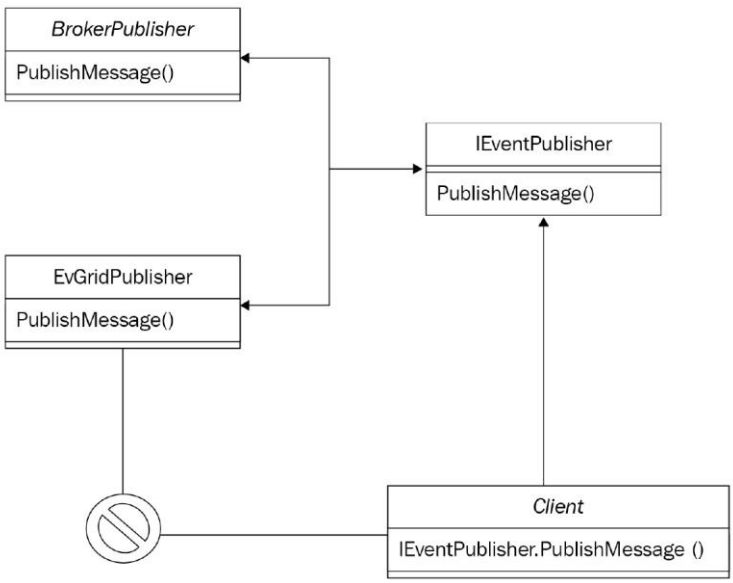

Figure 5.2 – DI diagram

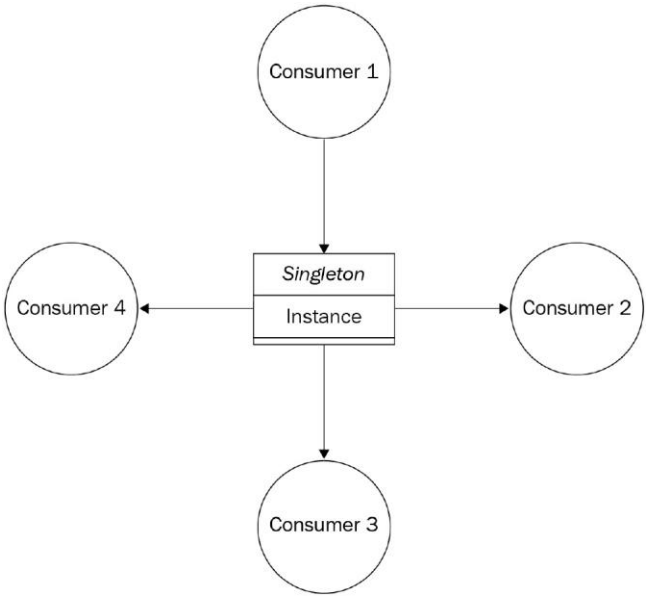

Figure 5.11 – Singleton design pattern

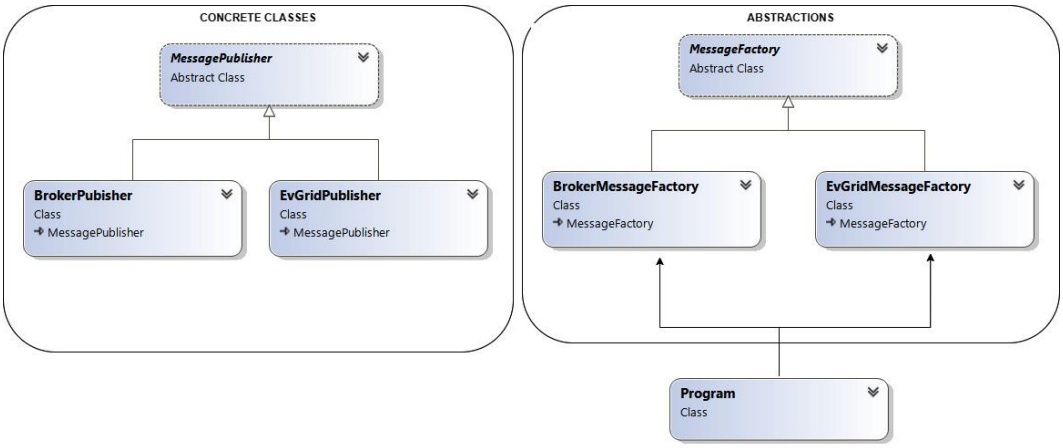

Figure 5.16 – Factory method pattern

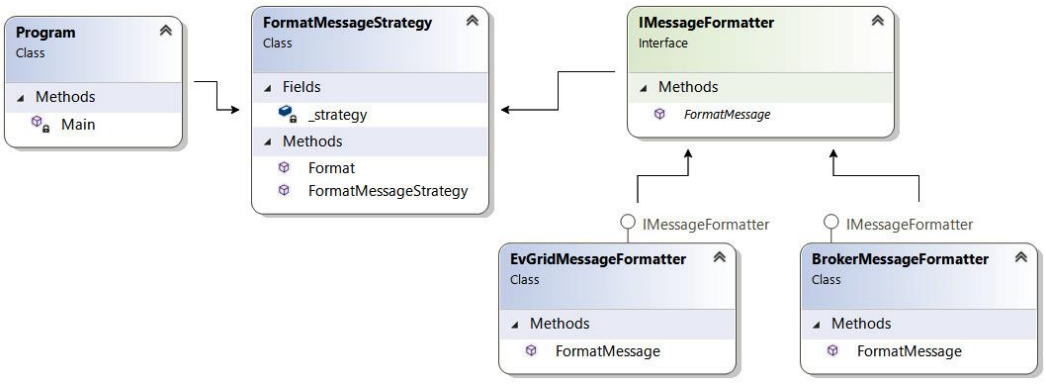

Figure 5.27 – Strategy pattern

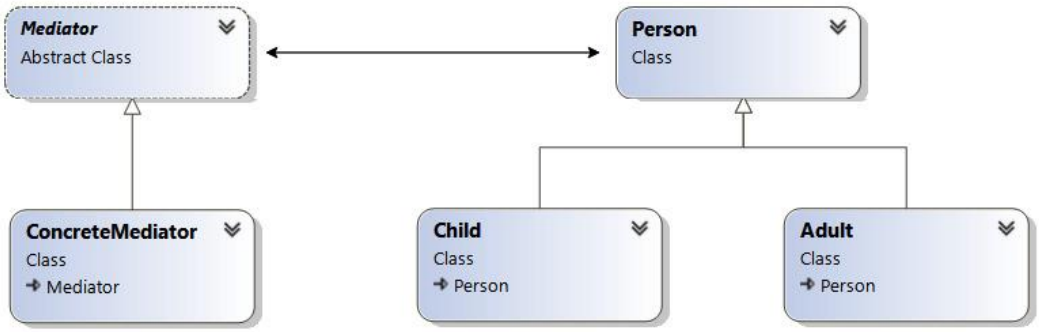

Figure 5.34 – Mediator pattern diagram

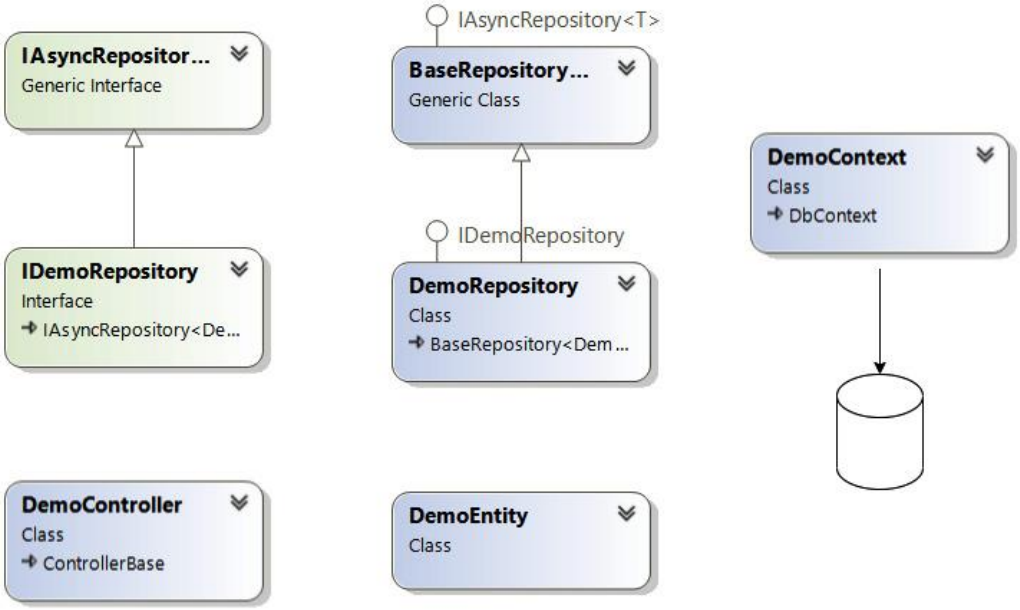

Figure 5.41 – Repository pattern diagram

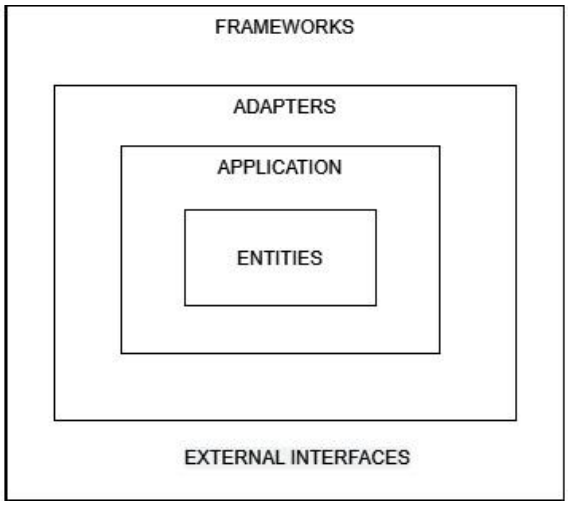

Figure 5.46 – Clean architecture

Code and commands

Code 5.1

```
Rectangle rect = new Square();

rect.setWidth(10);

rect.setHeight(5);

Assert.Equal(50, CalculateArea(rect));
```

Patterns in action

DI code – indirect mode

Indirect injection is provided by a DI container. There are numerous container frameworks available. By default, ASP.NET Core comes with its own default implementation. ASP.NET Core's startup class, and—more specifically—the `ConfigureServices` method, is where most of the DI plumbing happens. Here is an extract of that class:

```
public void ConfigureServices(IServiceCollection services){
    services.AddControllers();
    services.AddScoped<IEventPublisher, BrokerPublisher>();
}
```

Figure 5.3 – ConfigureServices

We can clearly see that the `IEventPublisher` contract is mapped to a concrete implementation represented by `BrokerPublisher`. In the case of ASP.NET Core, the built-in DI container is in charge of injecting consumers with a concrete implementation of the contract whenever an instance of a given contract is encountered. Here is our contract:

```
public interface IEventPublisher{
    4 references
    Task PublishMessage(string message);
}
```

Figure 5.4 – IEventPublisher contract

Our `BrokerPublisher` class is shown here:

```
public class BrokerPublisher : IEventPublisher{
    4 references
    public async Task PublishMessage(string message)
    {
        await Task.Run(() => {

            //SendMessageToBus()
        });
    }
}
```

Figure 5.5 – BrokerPublisher class

We pretend to send a message to a bus in the actual implementation of the `SendMessage` method, which is part of our contract. Similarly, another implementation of `IEventPublisher` could be this:

```
public class EvGridPublisher: IEventPublisher{
    4 references
    public async Task PublishMessage(string message){
        await Task.Run(() => {
            //SendMessageToGrid()
        });
    }
}
```

Figure 5.6 – Another implementation of IEventPublisher

This time, we pretend to send a message to an event manager. The two implementations show how flexible it is to work with interfaces. Now, in our **application programming interface** (**API**) controller, we let the DI container inject the relevant mapped concrete classes, as follows:

```
[ApiController]
[Route("[controller]")]
3 references
public class DemoController : ControllerBase{

    private readonly IEventPublisher _evPublisher;

    2 references
    public DemoController(IEventPublisher evPublisher){
        _evPublisher = evPublisher;
    }

    [HttpPost]
    2 references
    public async Task<IActionResult> PublishMessage(string message){
        await _evPublisher.PublishMessage(message);
        return new AcceptedResult();
    }
}
```

Figure 5.7 – DI container injecting classes

The constructor gets an instance of `IEventPublisher` as an argument, which in this case is replaced by an instance of `BrokerPublisher`, as defined in the startup class. It is possible to have multiple concrete classes implementing the same contract at the same time and add them to collections. What is key to remember in this example is that the controller is unaware of the implementation details. It will receive a concrete instance of an object that implements a certain contract. This makes it easier to write unit tests, as we will see in our next section.

DI code – direct mode

One of the biggest advantages of DI is its positive impact on the *testability* quality attribute. Let's see two different ways of testing the controller shown in the previous section. Still in ASP.NET Core, the following method makes use of a mocking framework:

```
[Fact]
0 references
public async Task PublishMessageTest()
{
    var _mockEventPub = new Mock<IEventPublisher>();
    var _demoController = new DemoController(_mockEventPub.Object);
    var actionResult = await _demoController.PublishMessage("test") ;
    Assert.IsType<AcceptedResult>(actionResult);
}
```

Figure 5.8 – Mocking framework

As you can see, we directly instantiate `DemoController` with a mocked representation of our `IEventPublisher` contract. This allows us to inject a fake object instead of the actual `BrokerPublisher` class we used previously. The reason you want to do this is that you do not want the actual actions to be taken during a unit test. Note that the usage of a mocking framework is not required, but it just makes your life easier. Here is an alternative implementation:

```
[Fact]
0 references
public async Task PublishMessageTestWithBasicDirectInjection(){
    var _demoController = new DemoController(new TestPublisher());
    var actionResult = await _demoController.PublishMessage("test");
    Assert.IsType<AcceptedResult>(actionResult);
}
```

Figure 5.9 – Alternative mocking framework

This time, we inject an instance of `TestPublisher`, whose implementation is shown here:

```
public class TestPublisher : IEventPublisher{
    4 references
    public async Task PublishMessage(string message)
    {
        await Task.Run(() => {
            //test use case()
        });
    }
}
```

Figure 5.10 – TestPublisher instance

This is just another variant of `IEventPublisher`, and this is where we can simulate whatever we want, thanks to the DI pattern. As you can see, in the preceding example, we performed a *direct* injection of the dependency via the constructor injection technique.

Singleton pattern in action

The following code is the default startup method of a console application:

```
class Program{
    0 references
    static void Main(string[] args){
        Parallel.For(0, 10, i =>{
                Console.WriteLine("Iteration {0} - Identifier {1}",i,
                    ThreadSafeSingletonExample.instance.identitier);
                Console.WriteLine("Iteration {0} - Identifier {1}", i,
                    NotThreadSafeSingletonExample.instance.identitier);
            });

        Console.Read();
    }
}
```

Figure 5.12 – Singleton main method

As you can see from the preceding code snippet, I use a parallel `for` loop to simulate a high concurrency. In the loop, I get an instance of `ThreadSafeSingletonExample` and `NotThreadSafeSingletonExample` objects. Here is the implementation of `NotThreadSafeSingletonExample`:

```
class NotThreadSafeSingletonExample{
    static NotThreadSafeSingletonExample _instance = null;
    static Guid _id = Guid.Empty;

    1 reference
    private NotThreadSafeSingletonExample(){
        Thread.Sleep(100);
        _id = Guid.NewGuid();
    }
    1 reference
    public static NotThreadSafeSingletonExample instance{
        get{
            if (_instance == null)
                _instance = new NotThreadSafeSingletonExample()
            return _instance;
        }
    }
    1 reference
    public string identitier{get{return _id.ToString();}}
}
```

Figure 5.13 – Non-thread-safe singleton implementation

As their names indicate, one implementation is thread-safe while the other is not. With this example, I want to illustrate one of the major drawbacks of singletons—namely, thread safety.

Under a high load, you may end up with multiple instances of a singleton if you do not pay attention to concurrency, which leads to unexpected outcomes because the main purpose of a singleton is to have a single instance in all circumstances. Moreover, such beginner errors may not be visible directly and might show up later once an application is already in production. This is not that easy to troubleshoot, so you'd better flush such issues out soon enough.

Coming back to the example, let's first start with the non-thread-safe implementation, the `NotThreadSafeSingletonExample` class. First, we see that `_instance` is a private class member, which is set when the private constructor is kicked off by the public `instance` property. In the `get` accessor, we set the value of `_instance` if it is `null`, and then we return it.

I have added a `sleep` statement to simulate a slow-initializing object. This code is supposed to create a single instance in all circumstances. The problem is that multiple threads can access the same resource at the same time. Under a race condition, when there is no synchronization mechanism in place, multiple threads could resolve the `if` statement to `true` at the same time and create an instance that results in breaking the concept of a singleton. A singleton is, by the way, not the only thing that is subject to concurrency issues. Every built-in collection may be subject to thread-safety issues as well.

To prevent concurrency issues, you must ensure multiple threads cannot create an instance at the same time. The way you achieve this is often specific to the programming language you work with. Some manual locking techniques are possible (single locking and double locking) but they come with their own caveats, such as having a negative impact on performance. In C# (specific to C# here), there is an easy way to overcome this issue, as shown in the following example:

```
class ThreadSafeSingletonExample{
    static readonly ThreadSafeSingletonExample _instance =
        new ThreadSafeSingletonExample();
    static Guid _id = Guid.Empty;
    0 references
    static ThreadSafeSingletonExample(){
        _id = Guid.NewGuid();
        Thread.Sleep(100);
    }
    1 reference
    public static ThreadSafeSingletonExample instance{
        get{
            return _instance;
        }
    }
    1 reference
    public string identitier { get { return _id.ToString(); } }
}
```

Figure 5.14 – Thread-safe singleton implementation

At first glance, this code looks very similar to that shown in *Figure 5.13*, except for one noticeable difference: the constructor is not only private, but it has also become **static**. C# ensures that static constructors are only called once per application domain. This small change makes a hell of a difference.

The following screenshot shows the output of the console program when executed:

```
Iteration 0 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 2 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 4 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 6 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 8 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 8 - Identifier 60fe4df0-a1da-4d90-806e-703dc9095f0f
Iteration 4 - Identifier 60fe4df0-a1da-4d90-806e-703dc9095f0f
Iteration 6 - Identifier 60fe4df0-a1da-4d90-806e-703dc9095f0f
Iteration 2 - Identifier 60fe4df0-a1da-4d90-806e-703dc9095f0f
Iteration 0 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
Iteration 1 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 1 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
Iteration 3 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 7 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 7 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
Iteration 3 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
Iteration 9 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 9 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
Iteration 5 - Identifier fc71afb7-604d-4cbd-9b20-0c604565ed9a
Iteration 5 - Identifier f6189dd1-e279-45cc-a8eb-20b7de562d59
```

Figure 5.15 – Singleton and concurrency

You can count if you do not believe it, but there are exactly *10 lines* reporting the `fc71afb7-604d-4cbd-9b20-0c604565ed9a` **globally unique identifier** (**GUID**), which corresponds to our thread-safe implementation. For the non-thread-safe implementation, we count two different GUIDs: one starting with `60fe4df0` and another one starting with `f6189dd1`. This means that we indeed got two different instances out of our poor implementation.

Factory method in action

To illustrate the factory method pattern, we will reuse the message publishers of the previous section. Remember that we had two different concrete implementations that were implementing the same contract in a different way. This time, our contract is represented by the following abstract class:

```
public abstract class MessagePublisher{
    4 references
    public abstract string ProviderName { get; }
    2 references
    public abstract Task PublishMessage(string message);
}
```

Figure 5.17 – The factory contract

This abstract class declares two abstract members: `PublishMessage` and `ProviderName`. Here are the concrete implementations of that abstract class:

```
public class BrokerPubisher : MessagePublisher
{
    4 references
    public override string ProviderName => "BusMessage";
    2 references
    public override async Task PublishMessage(string message){
        await Task.Run(() => {
            //SendMessageToBus()
        });
    }
}
1 reference
public class EvGridPublisher : MessagePublisher{
    4 references
    public override string ProviderName => "EvGridMessage";
    2 references
    public override async Task PublishMessage(string message)
    {
        await Task.Run(() => {
            //SendToGrid()
        });
    }
}
```

Figure 5.18 – Concrete implementations of the MessagePublisher class

This is very similar to what we did in the DI example. The difference lies in how these concrete classes get indirectly used by the consumer class. This is done by declaring a few extra classes. The first one is the abstract creator class, which is illustrated here:

```
public abstract class MessageFactory{
    4 references
    public abstract MessagePublisher GetPublisher();
}
```

Figure 5.19 – Factory abstract creator class

The `MessageFactory` abstract class forces any inheriting class to implement the `GetPublisher` method, which returns a type of `MessagePublisher`, our contract. `GetPublisher` is our **factory method**. Now come the concrete creator classes that implement `MessageFactory`, as illustrated here:

```
public class BrokerMessageFactory : MessageFactory{
    4 references
    public override MessagePublisher GetPublisher(){
        return new BrokerPubisher();
    }
}
1 reference
public class EvGridMessageFactory : MessageFactory{
    4 references
    public override MessagePublisher GetPublisher(){
        return new EvGridPublisher();
    }
}
```

Figure 5.20 – Concrete creator classes

As you might have guessed, these concrete creator classes are the ones used by our consumer class, which in this case is a console program, as illustrated here:

```
static void Main(string[] args){
    Console.Write("Bus (1) or Event Grid (2): ");
    var choice = Console.ReadLine().Trim();
    switch(choice){
        case "1":
            Console.WriteLine(new BrokerMessageFactory().GetPublisher().ProviderName);
            break;
        case "2":
            Console.WriteLine(new EvGridMessageFactory().GetPublisher().ProviderName);
            break;
        default:
            Console.WriteLine("Oops, invalid choice");
            break;
    }
}
```

Figure 5.21 – Factory consumer program

We prompt the user to choose between options `1` and `2`. Option `1` causes the client class to get an instance of `MessagePublisher` using `BrokerMessageFactory`. The following screenshot shows the output of the program when executed:

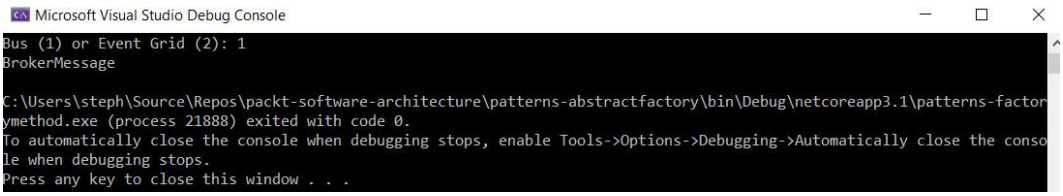

Figure 5.22 – Factory method program output

Because we took option `1`, the returned concrete implementation of the `MessagePublisher` contract is an instance of `BrokerMessage`. Let's now wrap up the section.

Lazy loading in action

Here is a very simple example of the lazy loading pattern in a C# console program:

```
static void Main(string[] args){
    Stopwatch sw = new Stopwatch();
    sw.Start();
    //does not cause a delay
    Lazy<DemoClass> _lazy = new Lazy<DemoClass>();
    Console.WriteLine(sw.ElapsedMilliseconds);
    //calling the object property causes its lazy loading
    Console.WriteLine(_lazy.Value.DemoProperty);
    Console.WriteLine(sw.ElapsedMilliseconds);
    Console.Read();
}

3 references
public class DemoClass{
    public readonly string DemoProperty = "a value";
    0 references
    public DemoClass(){
        Thread.Sleep(5000);
    }
}
```

Figure 5.23 – Lazy loading in action

In our `DemoClass` object, we simulate a slow initialization by adding a `Thread.Sleep` statement. In the `Main` method of the console program, we make use of a `Stopwatch` object to calculate the elapsed time between the different instructions. The output of the preceding program looks like this:

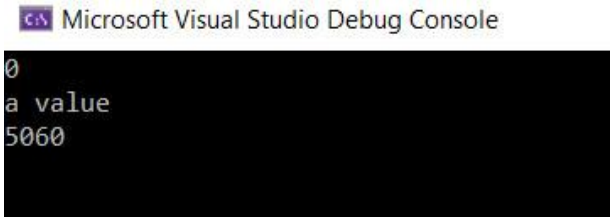

Figure 5.24 – Lazy loading in action (continued)

As you can see from *Figure 5.24*, no time elapsed between the start of our `Stopwatch` object and the creation of our `lazy` object. The lazy behavior is ensured by the `Lazy` keyword, which is a built-in C# instruction. However, when calling explicitly the property of our `DemoClass` object through `_lazy.Value.DemoProperty`, we can see that our second printout of the elapsed time shows `5060` milliseconds. This proves that the actual initialization of the `DemoClass` object was delayed until we called one of its properties (it is the same with methods, of course). The same program without the lazy pattern looks like this:

```
static void Main(string[] args){
    Stopwatch sw = new Stopwatch();
    sw.Start();
    DemoClass _notLazy = new DemoClass();
    Console.WriteLine(sw.ElapsedMilliseconds);
    Console.Read();
}
```

Figure 5.25 – Non-lazy initialization

Now, we instantiate the same `DemoClass` object the normal way. The following screenshot shows the outcome:

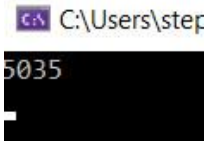

Figure 5.26 – Non-lazy DemoClass object

Strategy pattern in action

The strategy pattern helps prevent the following code constructs in the consumer class:

```
if (msg.MessageType == Message.MessageTypes.Bus){
    //write code to format bus messages
}
else if (msg.MessageType == Message.MessageTypes.Grid){
    //write code to format grid messages
}
```

Figure 5.28 – Strategy pattern prevents inline decisions

Here, you would add some formatting logic directly into the client. The goal of the strategy pattern is instead to delegate this logic to dedicated strategies and choose the appropriate one at runtime. Here is the code of the strategy contract:

```
public interface IMessageFormatter{
    3 references
    string FormatMessage(string message);
}
```

Figure 5.29 – Strategy contract

This is a very simple contract that specifies that every strategy should implement the `FormatMessage` method. Here, we can see two strategies that implement our contract:

```
public class EvGridMessageFormatter : IMessageFormatter{
    3 references
    public string FormatMessage(string message){
        Console.WriteLine("in EvGridMessageFormatter");
        //let's pretend we formatted the message for evengrid
        return message;
    }
}
1 reference
public class BrokerMessageFormatter : IMessageFormatter{
    3 references
    public string FormatMessage(string message){
        Console.WriteLine("in BrokerMessageFormatter");
        //let's pretend we formatted the message for a bus
        return message;
    }
}
```

Figure 5.30 – Concrete strategies

We pretend to format the message differently according to the strategy. Now comes our context class, the one that is used by our consumer class, as follows:

```
public class FormatMessageStrategy{
    private IMessageFormatter _strategy;
    2 references
    public FormatMessageStrategy(IMessageFormatter strategy){
        _strategy = strategy;
    }
    2 references
    public string Format(string message){
        return _strategy.FormatMessage(message);
    }
}
```

Figure 5.31 – Context class

The context class has a private member of the `IMessageFormatter` type. Its constructor takes a strategy as input and assigns it to the private member. This lets the consumer class specify which strategy to use at runtime. The `Format` method can be called by the consumer class and, in turn, calls the `FormatMessage` method of the strategy that was passed in. Now, the next code block shows how that context class can be used from within the consumer class:

```
static async Task Main(string[] args){
    IEventPublisher brokerPublisher = new BrokerPublisher();
    await brokerPublisher.PublishMessage(
        new FormatMessageStrategy(new BrokerMessageFormatter()).Format("a message"));
    IEventPublisher gridPublisher = new EvGridPublisher();
    await gridPublisher.PublishMessage(
        new FormatMessageStrategy(new EvGridMessageFormatter()).Format("a message"));
    Console.Read();
}
```

Figure 5.32 – Strategy consumer code

We subsequently call both strategies. The following screenshot shows the output of the console program when executed:

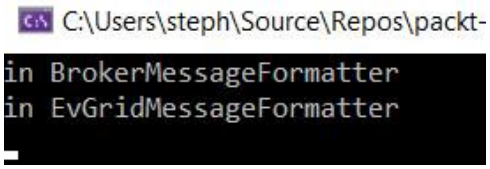

Figure 5.33 – Strategy pattern console output

Mediator in action

As always, we start by defining our contract, as follows:

```
interface IMediator{
    4 references
    public abstract void Register(Person p);
    2 references
    public abstract void Send(string from, string message, audience to);
}
```

Figure 5.35 – Mediator contract

The contract is an interface or an abstract class. We then implement the concrete mediator, as illustrated here:

```
class ConcreteMediator : IMediator{
    private Dictionary<string, Person> _persons =
        new Dictionary<string, Person>();
    4 references
    public void Register(Person p){
        _persons.TryAdd(p.Name, p);
        p.concreteMediator = this;
    }
    2 references
    public void Send(
        string from,string message, audience to){

        var persons = (audience.adult == to) ? _persons.Values.Where(
            p=>p.Name != from && p.GetType().Equals(typeof(Adult))) :
            _persons.Values.Where(p=>p.Name != from);
        if(persons.Count()>0)
            foreach (var person in persons)
                person.Receive(from, message);
    }
}
```

Figure 5.36 – Concrete mediator

In the `Register` method of the concrete mediator, we add the person to our dictionary or replace them. In the `Send` method, we handle the business logic to forward messages to the relevant recipients. Admittedly, the filtering logic is not robust since the name is not a good ID, but this is irrelevant for the pattern demonstration. We also pass the current concrete mediator to the `Person` object, as follows:

```
class Person{
    6 references
    public string Name { get; }
    2 references
    public Person(string name) => this.Name = name;
    2 references
    public ConcreteMediator concreteMediator { set; get; }
    2 references
    public void Send(string message, audience to = audience.everyone){
        concreteMediator.Send(Name,message, to);
    }
    1 reference
    public void Receive(
        string from, string message){
        Console.WriteLine("{0} - received {1}: from {2}",
            Name,message,from);
    }
}
```

Figure 5.37 – Person object

The important bits of the preceding code block are in the `Send` method, where the `Person` class calls back the concrete mediator to handle the sending of messages. The `Send` method also has an optional `audience` parameter, which allows us to filter out target recipients when sending messages. The following code shows two persons' flavors:

```
class Adult : Person{
    2 references
    public Adult(string name): base(name){}
}
2 references
class Child : Person{
    1 reference
    public Child(string name): base(name){}
}
```

Figure 5.38 – Person variants

Now, from the main program, we can start using our classes, as follows:

```
static void Main(string[] args){
    ConcreteMediator cm = new ConcreteMediator();
    Person a1 = new Adult("adult 1");
    Person a2 = new Adult("adult 2");
    Person c1 = new Child("child 1");
    cm.Register(a1);
    cm.Register(a2);
    cm.Register(c1);
    a1.Send("Hello adults",audience.adult);
    a1.Send("Hello everyone");
    Console.Read();
}
```

Figure 5.39 – Consumer code

Notice how we register all person instances to the concrete mediator. Here is the program's output when executed:

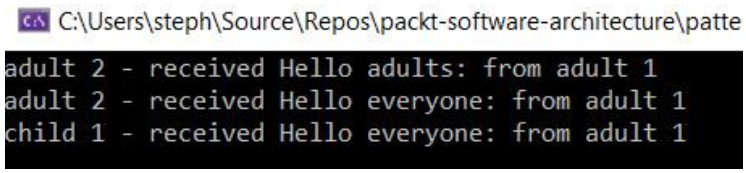

Figure 5.40 – Mediator pattern program output

Figure 5.40 shows that only `adult 2` received the message from `adult 1` because they explicitly targeted adults. Their second message is for everyone, hence the reason why `child 1` also received it.

Repository pattern in action

Let's first start with our generic repository, as follows:

```
public interface IAsyncRepository<T> where T : class{
    0 references
    Task<T> GetByIdAsync(Guid id);
    0 references
    Task<IReadOnlyList<T>> ListAllAsync();
    0 references
    Task<T> AddAsync(T entity);
    0 references
    Task UpdateAsync(T entity);
    0 references
    Task DeleteAsync(T entity);
}
```

Figure 5.42 – Generic repository

The repository is generic because it takes an input of `T`, which is the .NET way to handle generics. All the methods declared in the interface are also generic because they do not serve any specific business purpose. They simply represent the typical **create, read, update, and delete** (**CRUD**) operations.

This generic repository is particularly the reason why the repository pattern is subject to controversy. If you only stick to this implementation, you simply add redundant code because, by default, ORMs also come with such default CRUD operations over collections of objects.

Some may argue that a generic repository also gives you an opportunity to abstract away the ORM, should you change it in the future. Such levels of abstraction are also recommended by *clean architecture*, which we will talk about in our next section. This is where you must exercise good judgment and identify trade-offs to make sure you do not overengineer the solution and that it makes sense in your own context.

But of course, there's more to it, and that is the specific business domain repository, whose contract is listed here:

```
public interface IDemoRepository : IAsyncRepository<DemoEntity>{
    2 references
    Task<IEnumerable<DemoEntity>> ListOnlyOddEntities();
}
```

Figure 5.43 – Domain-level repository

The reason why it is business-specific is that it specifies an extra non-generic method looking like a specific query. This type of repository starts to bring value because it holds business logic. Let's now see a truncated (for brevity) version of the `BaseRepository` class, as follows:

```
public class BaseRepository<T> : IAsyncRepository<T> where T : class{
    protected readonly DemoContext _dbContext;

    1 reference
    public BaseRepository(DemoContext dbContext) {
        _dbContext = dbContext;
    }
    2 references
    public async Task<IReadOnlyList<T>> ListAllAsync(){
        return await _dbContext.Set<T>().ToListAsync();
    }
    other methods
}
```

Figure 5.44 – Base repository class

Our `BaseRepository` class is also generic and can take any entity. It implements `IAsyncRepository<T>`, our generic repository contract. As you can see, it takes an instance of the `DemoContext` object, which is nothing other than our ORM's entry point. Now comes our tailor-made repository, as follows:

```
public class DemoRepository : BaseRepository<DemoEntity>, IDemoRepository
{
    0 references
    public DemoRepository(DemoContext dbContext) : base(dbContext) { }
    2 references
    public async Task<IEnumerable<DemoEntity>> ListOnlyOddEntities(){
        return await Task<IEnumerable<DemoEntity>>.Run(() => {
            return _dbContext.DemoEntities.ToList().Where((c, i) => i % 2 != 0)
                as IEnumerable<DemoEntity>;
        });
    }
}
```

Figure 5.45 – Tailor-made repository

`DemoRepository` derives from `BaseRepository` and passes in the `DemoEntity` type. It also implements `IDemoRepository` through the concrete implementation of the `ListOnlyOddEntities` method. If you know a little bit about .NET, you have probably identified a *code smell* here. The odd/even check is done in memory and causes the ORM to produce a `SELECT * FROM …` statement, which is never good performance-wise. I did this on purpose to show you how easy it is to write very extendable code but leave such poor constructs in it. By the end of this chapter, I will let you know what I consider to be the most important things to catch in a code-review round.

Links

https://en.wikipedia.org/wiki/Software_design_pattern

Clean architecture regroups the best of **hexagonal architecture** (https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)) and **onion architecture** (https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/).

Robert C. Martin's original drawing is available on his website at https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Clean architecture: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

DDD & Microservices: At Last, Some Boundaries! • Eric Evans • GOTO 2015: https://www.youtube.com/watch?v=yPvef9R3k-M

Chapter 6

Technical requirements

If you want to practice implementing the explanations provided in this chapter, you will need **Visual Studio 2019** to open the solution provided on GitHub.

All the code samples and diagrams for this chapter are available at https://github.com/PacktPublishing/Software-Architecture-for-Humans/tree/master/CHAPTER%206/.

Figures

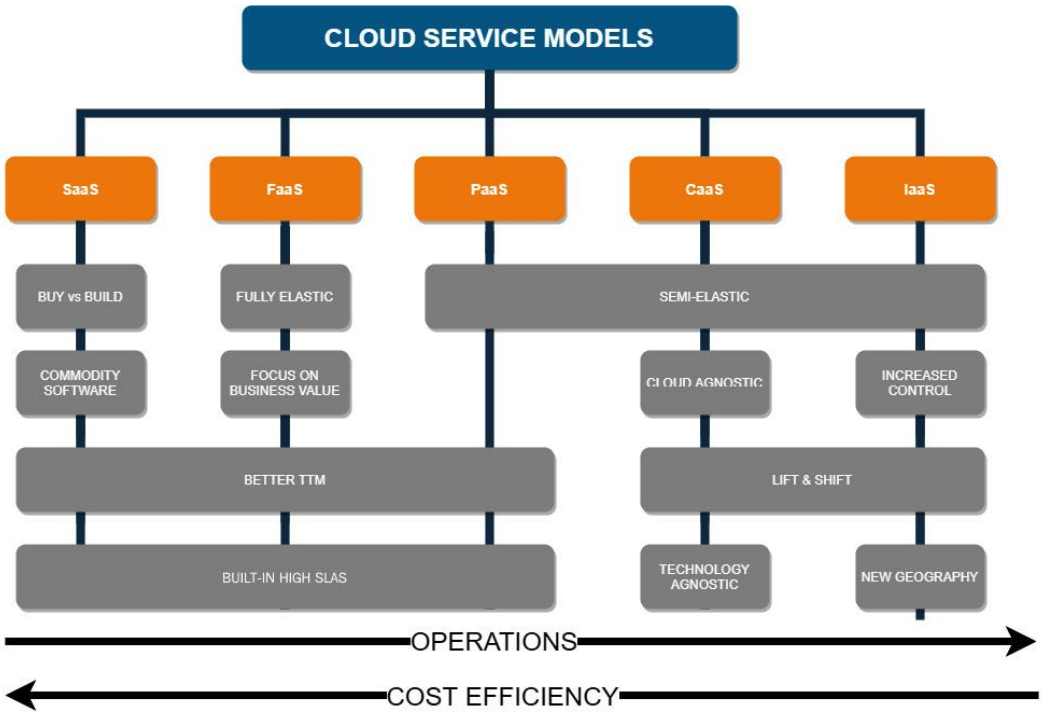

Figure 6.1 – Cloud service models

	FaaS	PaaS	CaaS
Scalability	+++	++	+++
Availability	+++	++	++
Deployability	+	+	+++
Portability	-	-	+++
Testability	+	+	++
Security (control)	-	+	++
Auditability	+	+	+
Reliability	++	++	++
Manageability	+++	+++	+

Figure 6.2 – Impact of service models on quality attributes

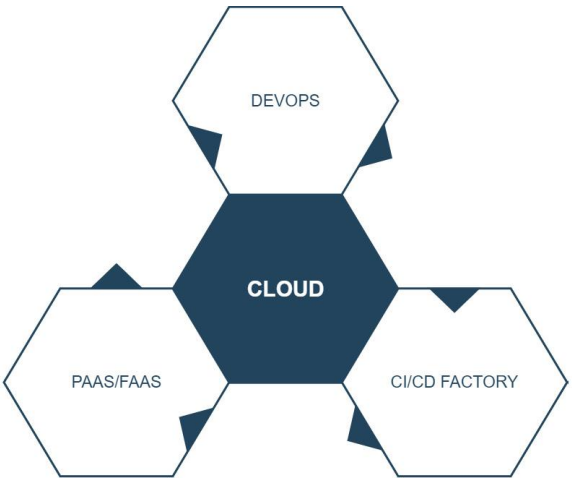

Figure 6.3 – Cloud development approach

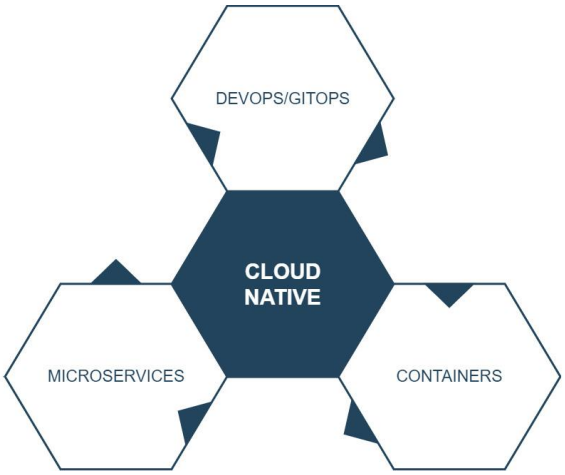

Figure 6.4 – Cloud-native development

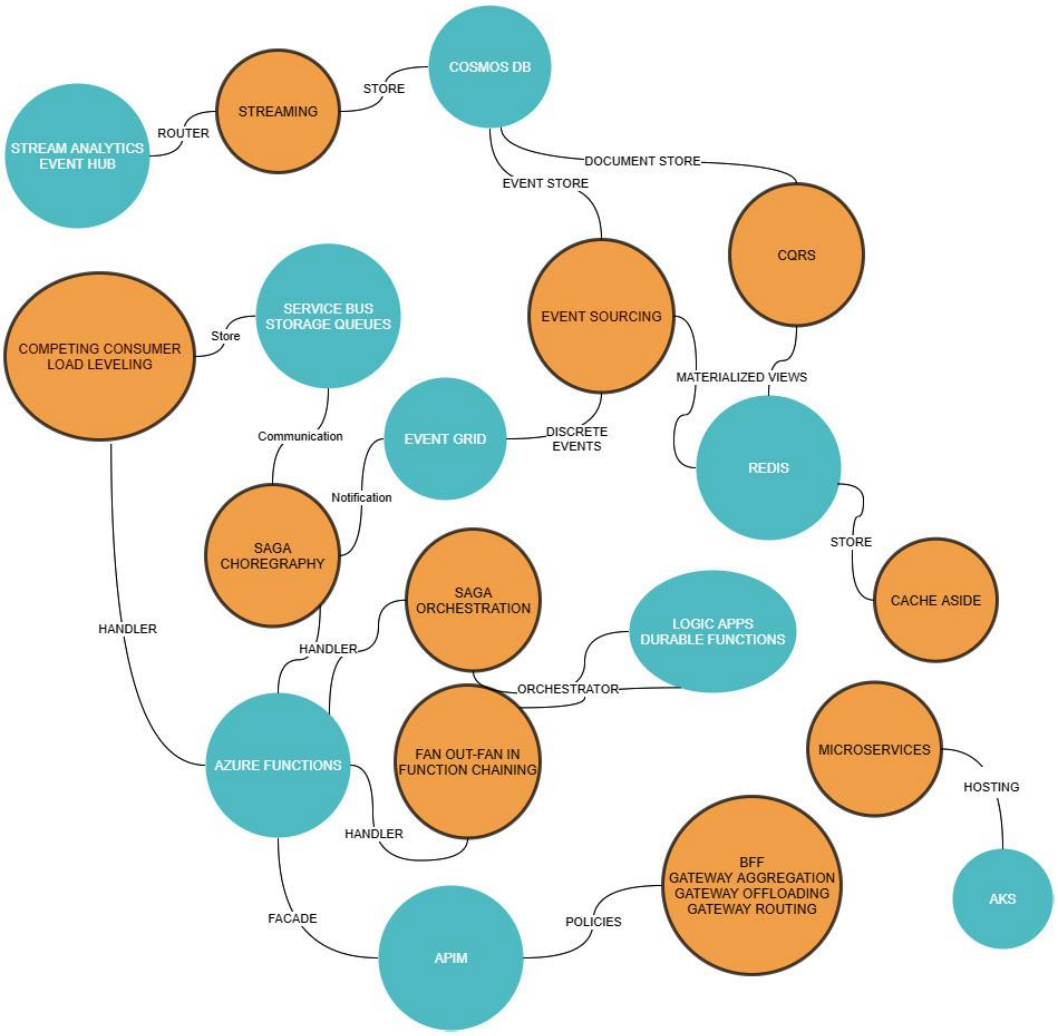

Figure 6.5 – Azure services mapped to patterns

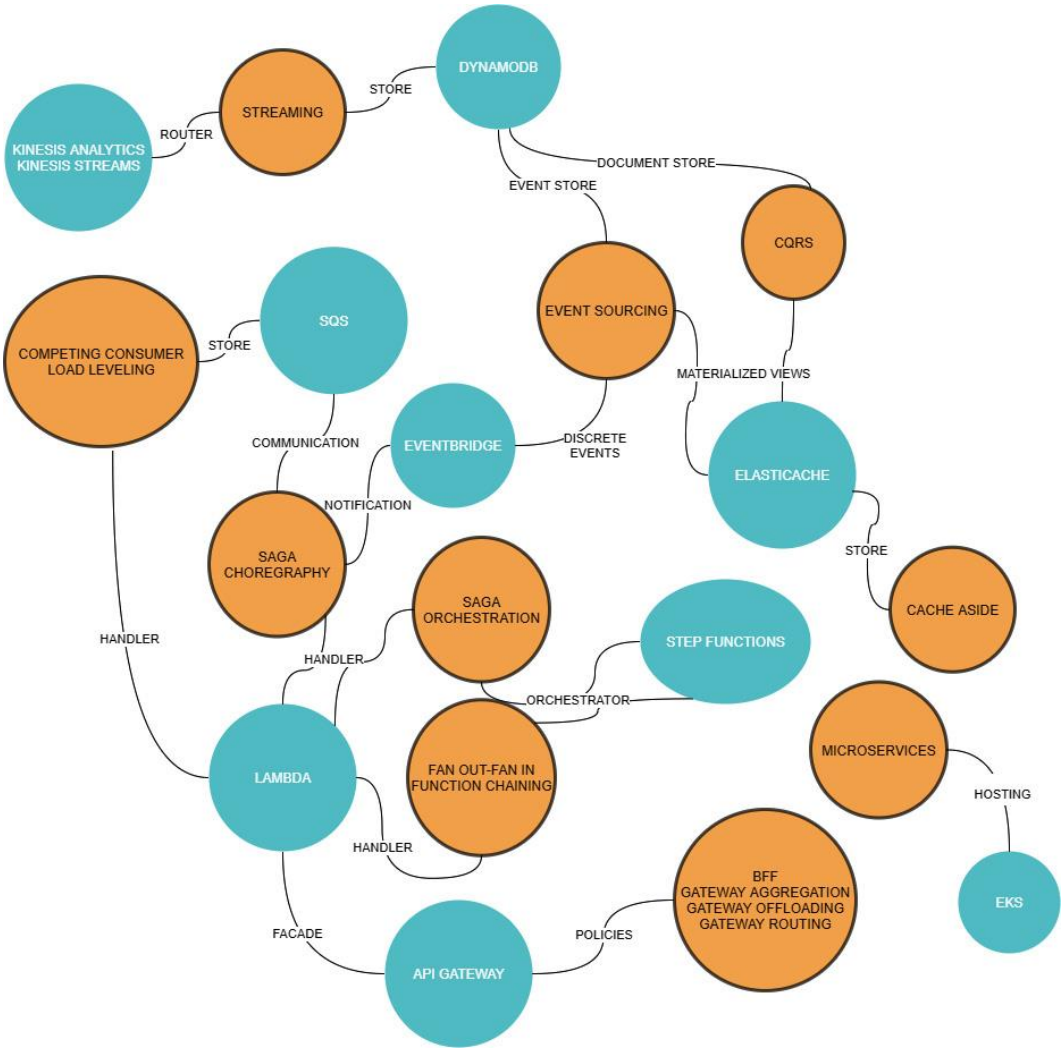

Figure 6.6 – AWS services mapped to patterns

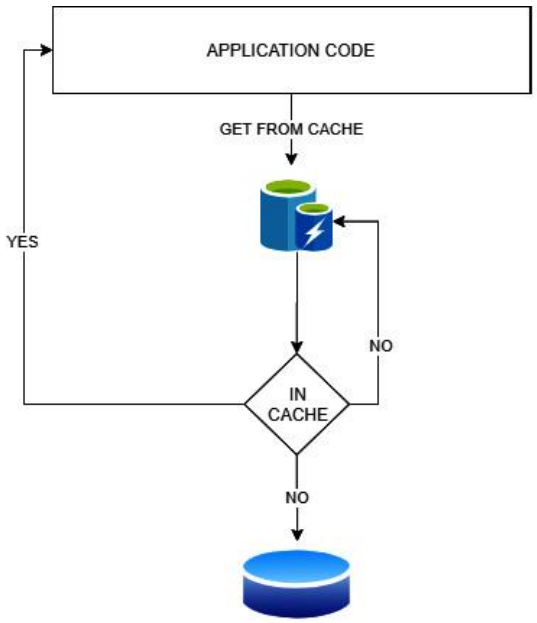

Figure 6.7 – Cache-Aside pattern

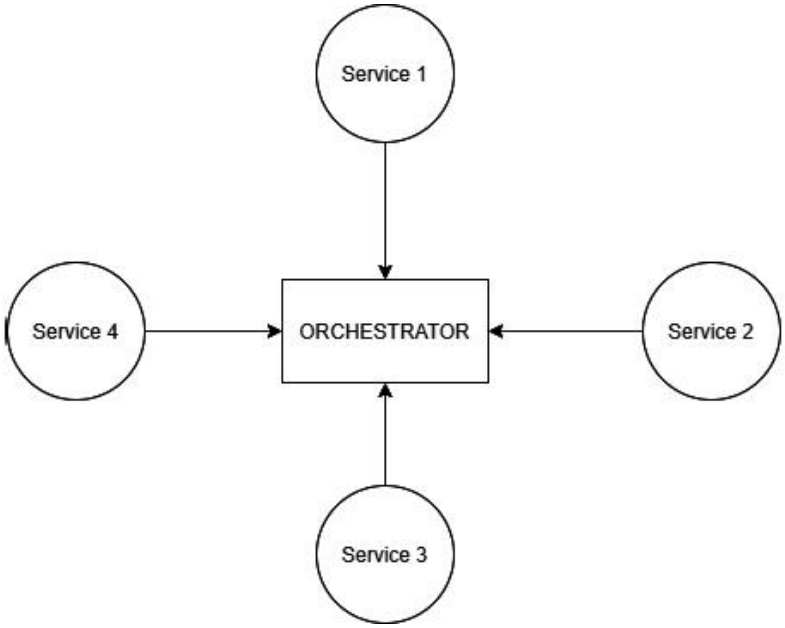

Figure 6.8 – SAGA orchestrator-based pattern

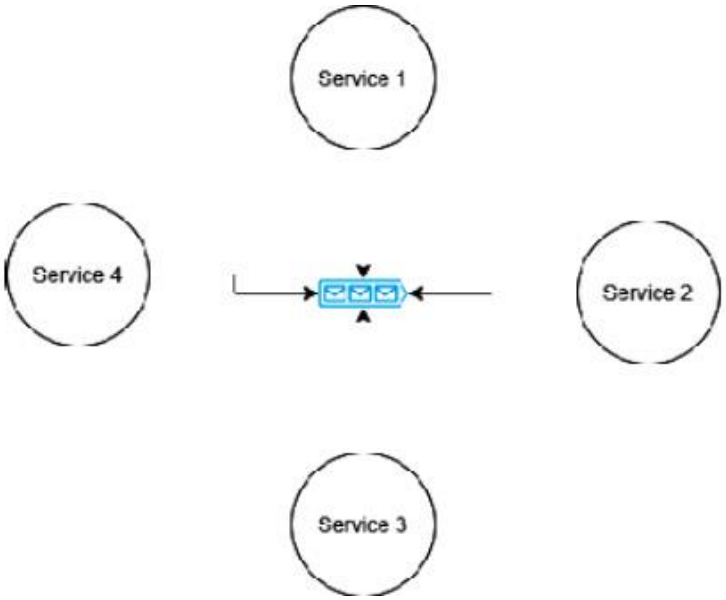

Figure 6.9 – SAGA choreography-based pattern

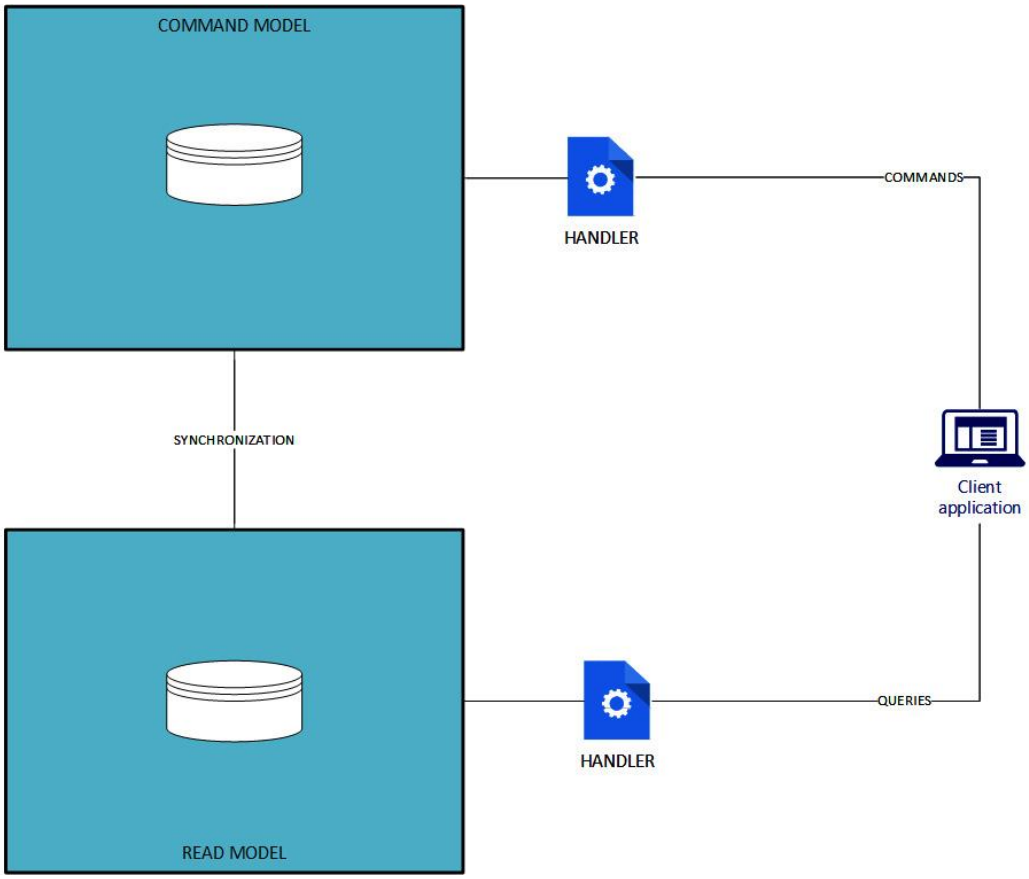

Figure 6.10 – The CQRS pattern

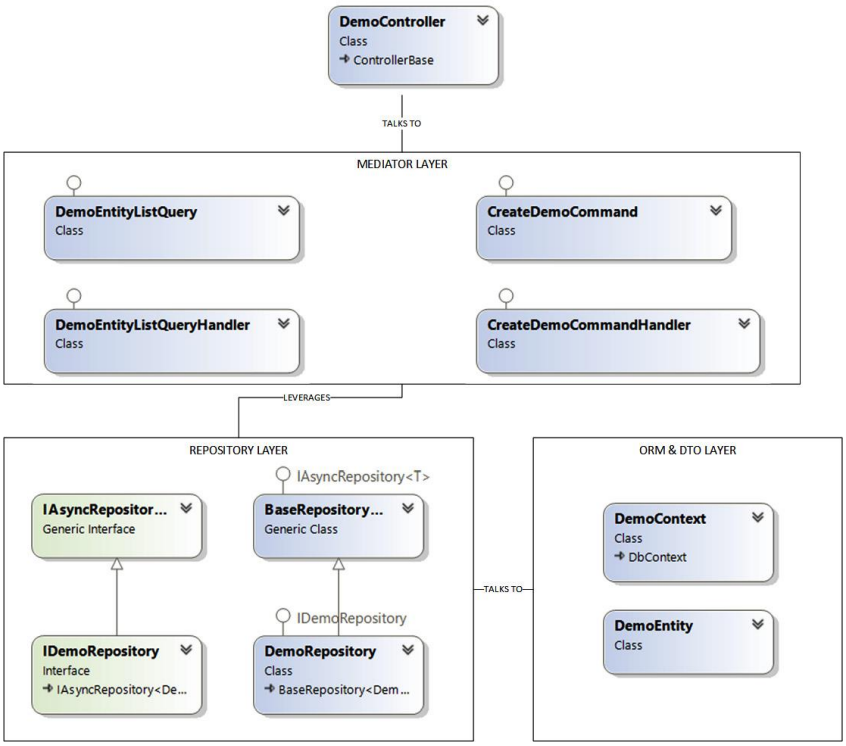

Figure 6.11 – Mediator combined with repositories to achieve CQS or CQRS served by a single API

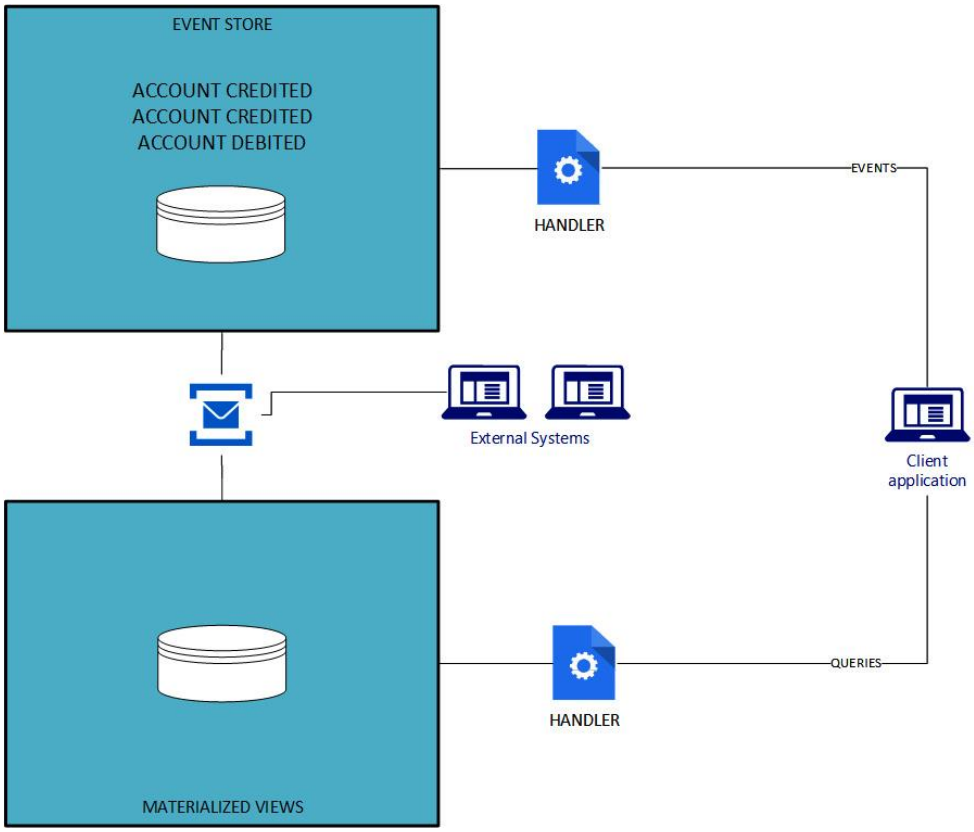

Figure 6.12 – Event sourcing diagram

Links

You can see an example of SAGA using Azure services at https://github.com/Azure-Samples/saga-orchestration-serverless

Chapter 7

Technical requirements

If you want to practice implementing the explanations provided in this chapter, you will need the following:

- **Visual Studio 2019**: To open the solution provided on GitHub.
- **Kubernetes**: You will need a vanilla cluster such as **MiniKube** or **Docker Desktop** with K8s embedded. You can also use any cloud-provided cluster (Azure, AWS, or GCP). I used Azure Kubernetes Service to host my demo solution. Whatever solution you choose, make sure that the cluster has access to the internet so that it can pull the Docker images that I published to **Docker Hub**.
- **An Azure subscription**: I used Azure for the serverless sample application. To create your own free Azure account, follow the steps explained at https://azure.microsoft.com/free/.

All the code samples and diagrams for this chapter are available at https://github.com/PacktPublishing/Software-Architecture-for-Humans/tree/master/CHAPTER%207/.

Figure

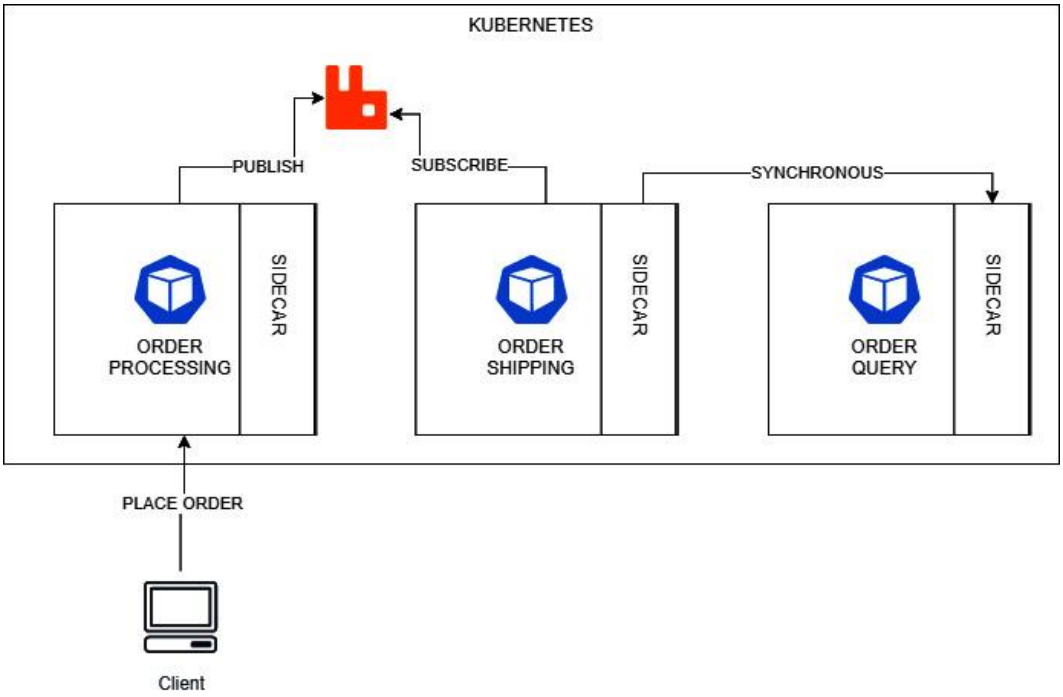

Figure 7.1 – Microservice application example

```
[ApiController]
3 references
public class OrderController : ControllerBase{
    private readonly DaprClient _dapr;
    private readonly ILogger<OrderController> _logger;

    0 references
    public OrderController(ILogger<OrderController> logger, DaprClient dapr){
        _logger = logger;
        _dapr = dapr;
    }

    [HttpPost]
    [Route("order")]
    0 references
    public async Task<IActionResult> Order([FromBody] Order order, [FromServices] DaprClient daprClient){
        //we pretend to create an order
        _logger.LogInformation($"Order with id {order.Id} created!");
        await _dapr.PublishEventAsync<Order>("bus", "order", order);
        return Ok();
    }
    0 references
    async Task<IActionResult> PublishOrderEvent(Guid OrderId, OrderEvent.EventType type){
        //we publish the order created event
        var ev = new OrderEvent
        {
            id = OrderId,
            name = "OrderEvent",
            type = type
        };
        await _dapr.PublishEventAsync<OrderEvent>("bus", "order", ev);
        return Ok();
    }
}
```

Figure 7.2 – Order processing backend service

```
[Topic("bus", "order")]
[HttpPost]
[Route("dapr")]
0 references
public async Task<IActionResult> ProcessOrderEvent([FromBody] OrderEvent ev){
    _logger.LogInformation($"Received new event");
    _logger.LogInformation("{0} {1} {2}", ev.id, ev.name, ev.type);
    switch (ev.type){
        case OrderEvent.EventType.Created:
            var order = await GetOrder(ev.id);
            if (order!=null){
                _logger.LogInformation($"Starting shipping process for order {ev.id} with " +
                    $"{order.Products.Count} " +$"products!");
            }
            else{
                _logger.LogInformation($"order {ev.id} could not be retrieved, suspending shipping process!");
            }
            break;
        other cases
    }
    return Accepted();
}
2 references
async Task<Order> GetOrder(Guid id){
    try{
        return await _dapr.InvokeMethodAsync<object, Order>(
            HttpMethod.Get,
            "orderquery",
            id.ToString(),
            null);
    }
    catch (Exception ex){//should be more specific
        _logger.LogError(ex.Message);
        return null;
    }
}
```

Figure 7.3 – Shipping service implementation

```yaml
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: bus
  namespace: microserviceapp
spec:
  type: pubsub.rabbitmq
  version: v1
  metadata:
  - name: host
    value: "amqp://user:pwd@rabbitmq.default.svc.cluster.local:5672"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orderprocessing
  namespace: microserviceapp
  labels:
    app: orderprocessing
spec:
  replicas: 1
  selector:
    matchLabels:
      app: orderprocessing
  template:
    metadata:
      labels:
        app: orderprocessing
      annotations:
        dapr.io/enabled: "true"
        dapr.io/app-id: "orderprocessing"
        dapr.io/app-port: "80"
    spec:
      containers:
      - name: orderprocessing
        image: stephaneey/orderprocessing:dev
        imagePullPolicy: Always
```

Figure 7.4 – YAML deployment file

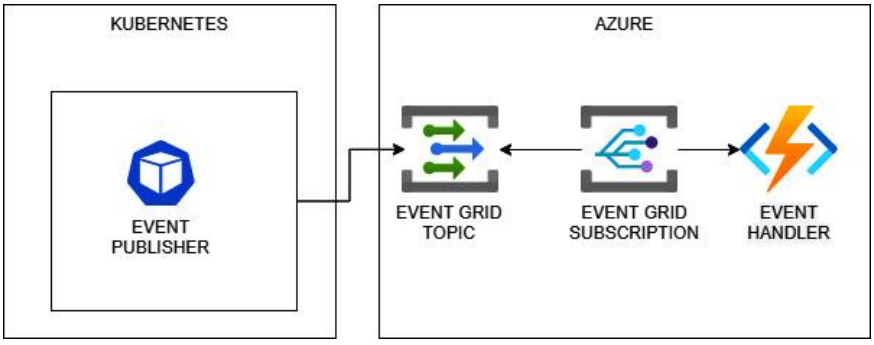

Figure 7.10 – Diagram of the serverless sample application

```
class Program{
    0 references
    static async Task Main(string[] args){
        EventGridPublisherClient client = new EventGridPublisherClient(
            new Uri(Environment.GetEnvironmentVariable("EvGridEndpoint")),
            new AzureKeyCredential(Environment.GetEnvironmentVariable("EvGridAccessKey")));
        while(true){
            await client.SendEventAsync(new EventGridEvent(
           "Serverless",
           "Serverless.OrderEvent",
           "1.0",
           Guid.NewGuid().ToString())
            );
            Thread.Sleep(100);
        }
    }
}
```

Figure 7.11 – Event publisher code

Code and commands

Code 7.1:

```
dapr init --runtime-version 1.0.1 -k -
```

Testing the microservices application

Follow these steps to test the app:

1. Verify that all the pods are running:

```
Windows PowerShell
PS C:\> kubectl get pods -n microserviceapp
NAME                                READY   STATUS    RESTARTS   AGE
orderprocessing-ff6f5644-5sjzb      2/2     Running   0          87s
orderquery-5557767d84-mcwcc         2/2     Running   0          87s
shippingprocessing-7bcfb96bfb-q6kbq   2/2     Running   0          87s
PS C:\>
```

Figure 7.5 – Listing pods

2. You should see that each pod runs two containers – the service and the Dapr sidecar. To place an order, we need to forward the order processing (or any other Dapr-injected pod) traffic to the host:

```
Windows PowerShell
PS C:\> kubectl -n microserviceapp port-forward orderprocessing-ff6f5644-5sjzb 3500:3500
Forwarding from 127.0.0.1:3500 -> 3500
Forwarding from [::1]:3500 -> 3500
```

Figure 7.6 – Forwarding traffic to the host

3. Make sure to replace the pod name with your own. Once done, we can start making calls to our localhost endpoint.

4. Using your preferred tool (Postman, Fiddler, or whichever you like), run the following HTTP POST request (a request sample is also available on GitHub):

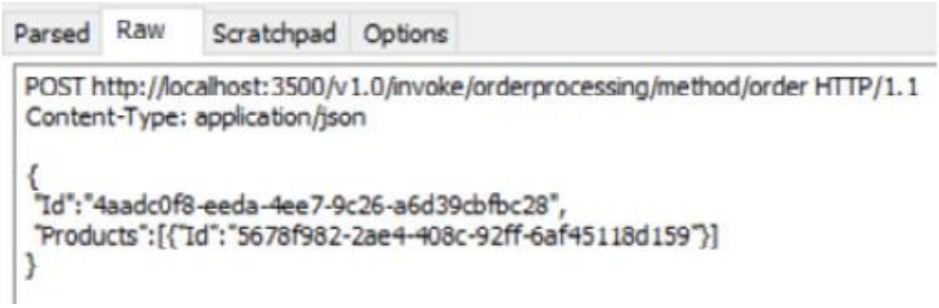

Figure 7.7 – Sample HTTP POST request against the order processing service

5. Note that we could also use gRPC but, for the sake of simplicity, I simply performed an HTTP call.

6. To see whether the request was handled properly, inspect the service logs:

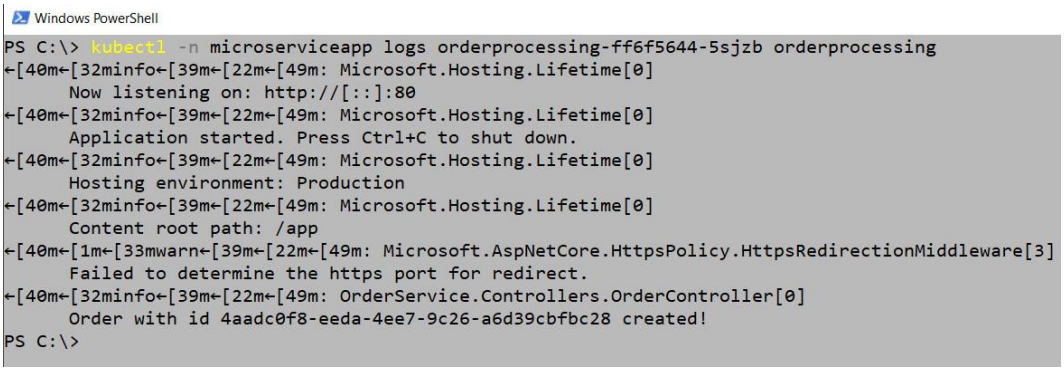

Figure 7.8 – Inspecting the order processing logs

7. The last line shows that the order was created. It's now time to look at the shipping service to see if it pulled the order created event from the RabbitMQ broker:

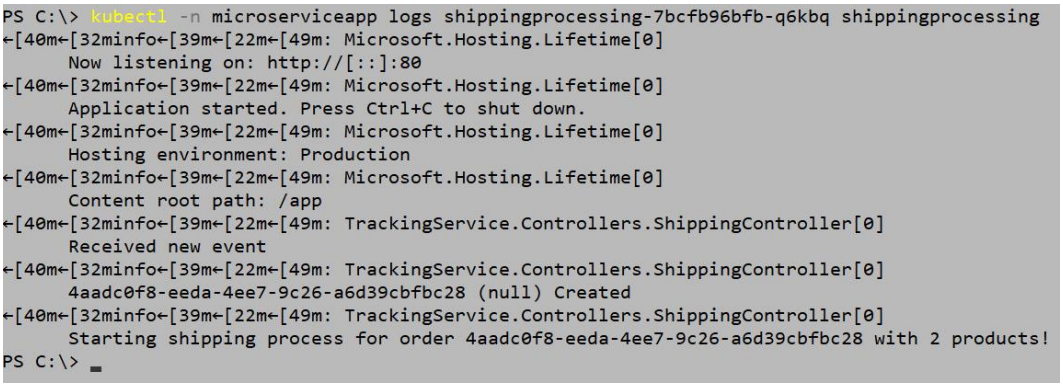

Figure 7.9 – Inspecting shipping logs

The last line also shows that the shipping process was started and that the order could be retrieved correctly from the order query service, because it found that two products were attached to it.

Deploying and testing the Azure Infrastructure

Deploying the required infrastructure

The time has come to deploy the Azure infrastructure and our event publisher within K8s. To deploy the Azure infrastructure, make sure to use an account that is at least a contributor to the subscription that will host the Azure resources. For your convenience, I have prepared a few IaC script templates, which are available in this book's GitHub repository. Let's go through the deployment steps.

1. First, you must log into https://shell.azure.com/ using your trial or paid subscription.
2. Download the IaC files and the `.zip` package that contains the function code. Alternatively, you can clone the repository. Once downloaded, upload the files to Cloud Shell:

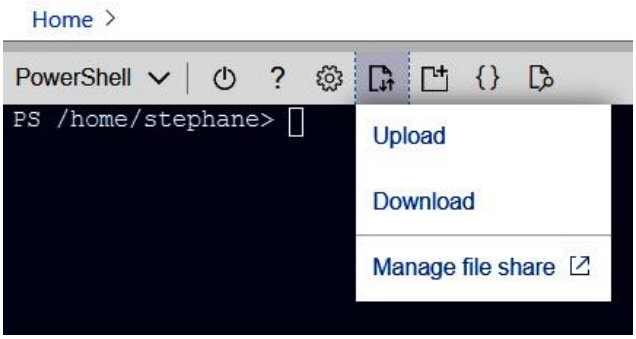

Figure 7.12 – Uploading files to Cloud Shell

3. You will have to upload files one by one.
4. Next, type the following command to create the resource group that will host the resources:

```
az group create -l westeurope -n packtserverless
```

5. Now, it is time to deploy the different services. Run the following command to deploy the first template:

```
az deployment group create --template-file serverless1.json --
resource-group packtserverless --parameters
appName="YOURVALUE"
```

6. Make sure you replace `YOURVALUE` with something unique. This will be used by Azure to connect to the function. In my case, I used `packtsrvlessarch`. Make sure not to use any fancy characters. At this stage, you should find the following resources in your resource group:

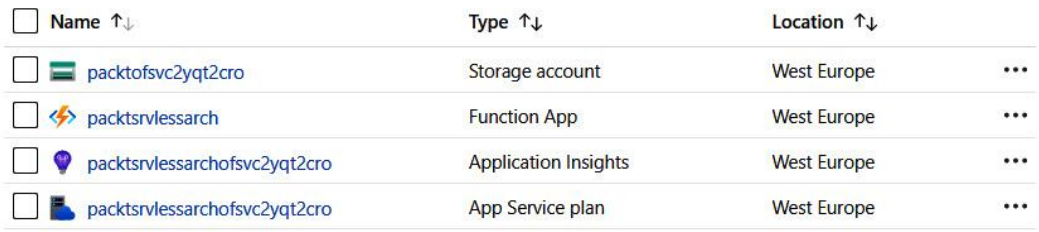

Figure 7.13 – Checking the deployed resources

7. Azure functions require a **Storage Account** and an **App Service Plan** (bound to the dynamic pricing tier for serverless functions) to function. There is also an **Application Insights** instance that is used to monitor the function.

8. Now, you can deploy the function code that sits in the `.zip` file:

```
az webapp deployment source config-zip --resource-group
packtserverless --name YOURVALUE --src event-consumer.zip
```

9. Now comes the last part of the deployment, which is creating the event grid's topic and subscription. This last part will subscribe the previously deployed function to the topic:

```
az deployment group create --template-file serverless2.json --
resource-group packtserverless --parameters
eventGridTopicName='YOURVALUEtp'
eventGridSubscriptionName='YOURVALUEtpsub'
functionAppName='YOURVALUE'
```

10. Note that `YOURVALUE` still represents the name of your function. At this stage, you should have all resources deployed and linked together:

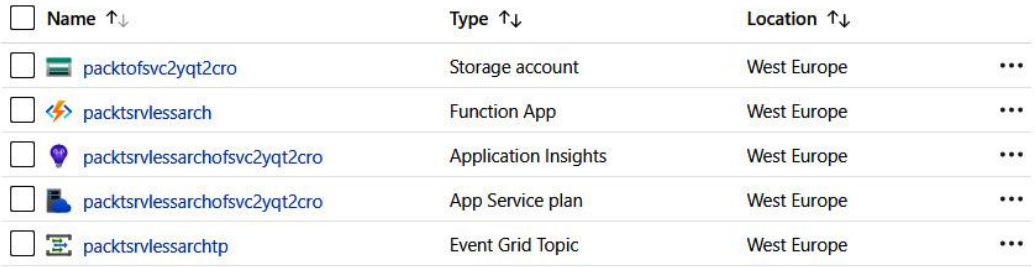

Figure 7.14 – Checking if all the resources were deployed

11. The last item is the event grid topic. Upon clicking on it, you should see that a subscription for the Azure function was indeed created:

Figure 7.15 – The Azure function subscription

12. The name will be *YOURVALUE*`tpsub`. While you are looking at your event grid topic, take note of the topic's endpoint and access keys. The topic endpoint can be grabbed from the overview page, and the access keys (two keys) are accessible through the left menu. Copy only one of the keys. We need both the endpoint and one of the keys for our event publisher.

The Azure infrastructure has been completely deployed. However, we still need to deploy the event publisher to our K8s cluster before we can test the application. So, we are almost done. Let's go through the final remaining steps:

13. Edit `serverless.yml` (available on GitHub) and replace `YOURENDPOINT` and `YOURKEY` with your own values, taken from the preceding step:

```
spec:
  containers:
  - name: eventpublisher
    image: stephaneey/eventpublisher:dev
    env:
    - name: EvGridEndpoint
      value: "YOURENDPOINT"
    - name: EvGridAccessKey
      value: "YOURKEY"
    imagePullPolicy: Always
```

Figure 7.16 – YAML spec of the event publisher container

14. Deploy the YML file to your cluster:

```
kubectl apply -f .\serverless.yml
```

Congratulations, you are done!

Testing the application

Now that everything has been deployed, you should have at least one instance of the event publisher pod running. You can verify this as follows:

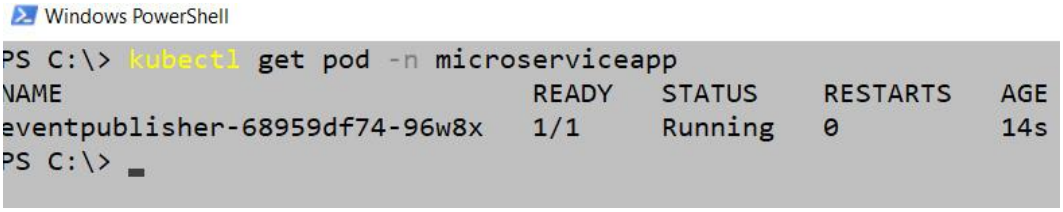

Figure 7.17 – Checking that the event publisher pod is running

This should already publish events to our grid, and notifications should be pushed to our function. Before we scale out to more instances, navigate to your Application Insights resource and click on the **live metrics** menu on the left. Once the metrics start showing up, scale out the event publisher, as shown in the following screenshot:

```
Windows PowerShell
PS C:\> kubectl scale deploy/eventpublisher --replicas=50 -n microserviceapp
deployment.apps/eventpublisher scaled
PS C:\> kubectl get pod -n microserviceapp
NAME                              READY   STATUS              RESTARTS   AGE
eventpublisher-68959df74-25kfs    0/1     ContainerCreating   0          6s
eventpublisher-68959df74-294vj    0/1     Pending             0          6s
eventpublisher-68959df74-4v4ld    0/1     Pending             0          6s
eventpublisher-68959df74-5ph4z    0/1     Pending             0          6s
eventpublisher-68959df74-5szv6    0/1     ContainerCreating   0          6s
eventpublisher-68959df74-6qwd7    0/1     ContainerCreating   0          6s
eventpublisher-68959df74-72psp    0/1     Pending             0          6s
eventpublisher-68959df74-7hmvs    0/1     Pending             0          6s
eventpublisher-68959df74-7wjxl    0/1     Pending             0          6s
```

Figure 7.18 – Scaling out the event publisher

I scaled the event publisher for 50 instances. This may or may not work in your own environment, depending on your cluster capacity. If 50 is too much, just reduce it to 5. The goal is to create more events and see how Azure functions scale accordingly. The live metrics screen should show something similar to the following:

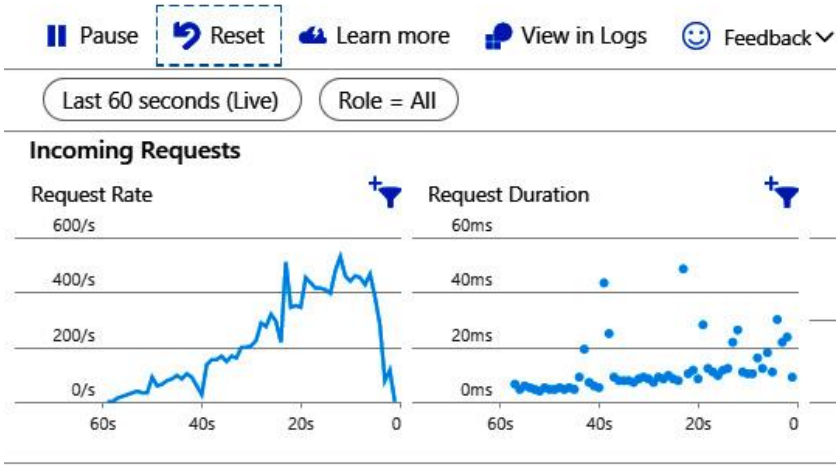

Figure 7.19 – Live metrics showing the request rate and duration

No matter how you scaled the event publisher, leave it running for about a minute and then scale the deployment back to zero.

As shown in the preceding screenshot, at peak time, the system was handling about 525 requests per second and most executions took less than 20 milliseconds. We did not have to configure anything; we just let the cloud provider adjust the computing power according to the demand. The following screenshot shows that the system scaled out, up to five instances:

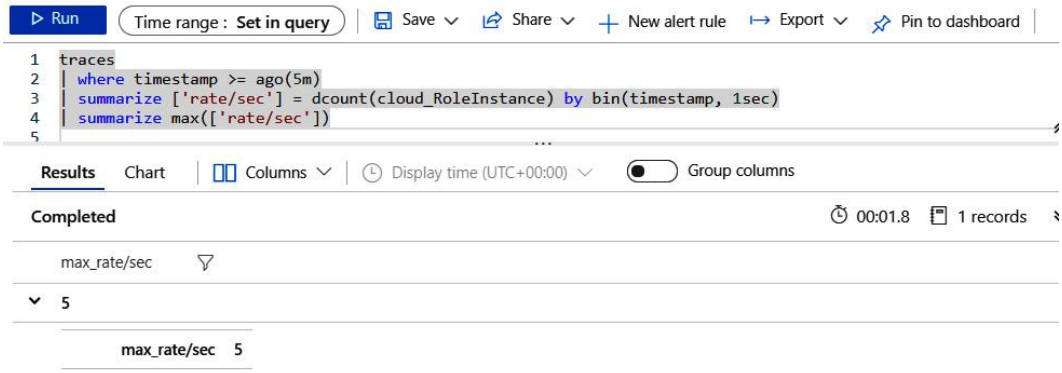

Figure 7.20 – Azure function max instance count

Running the same query after stopping the event publisher returns no results, meaning that an instance is no longer running. You do not need to run the query yourself. So, this very small serverless application is indeed based on a system that scales in and out according to demand. The cherry on the cake is that in Azure, you can have 1 million executions per month for free. You could force Azure functions to scale even more, should you have a powerful cluster or if you can run the event publisher from different systems at the same time. In my case, I simply used a single-node lab cluster, which is, by itself, a limiting factor, to really produce a high workload.

I hope that you enjoyed this short journey into the magical world of serverless architecture. Now, let's recap this chapter.

Links

Dapr CLI tool: https://docs.dapr.io/getting-started/install-dapr-cli/