//denver/dev/day/

Denver Dev Day
June 24, 2016

TEKsystems
Our people make IT possible.

revgen partners
DRIVING PROFITABILITY®

Microsoft

S&P Global

NEUDESIC™

Syncfusion®

ASP.NET Core 1.0 and Microservices

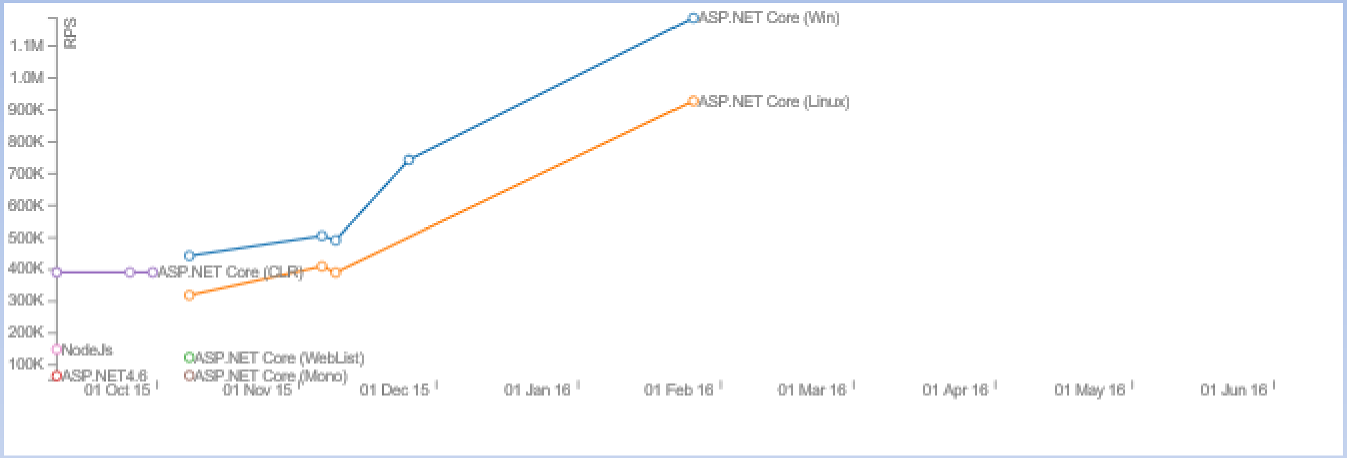
Donald Lutz
Technetronic Solutions
delutz@techsoli.com

Main Tenets

- Open Source
- Get all developers on the platform
- Increase speed of execution to compete with frameworks like Node.js
- Allow for the creation of microservices

1.15 Million represents a 2300% gain from ASP.NET 4.6! As of the end of June, 5 million RPS.

Compared to Classic ASP.NET and Node.js



- ASP.NET 4.6 and Node.js are bottom left.

- ASP.NET Core and .NET Core come with the great advantage of only including the libraries and functions you explicitly want to use in your application rather than bringing in the entire framework. So you only “pay”, programmatically speaking, for what you use.

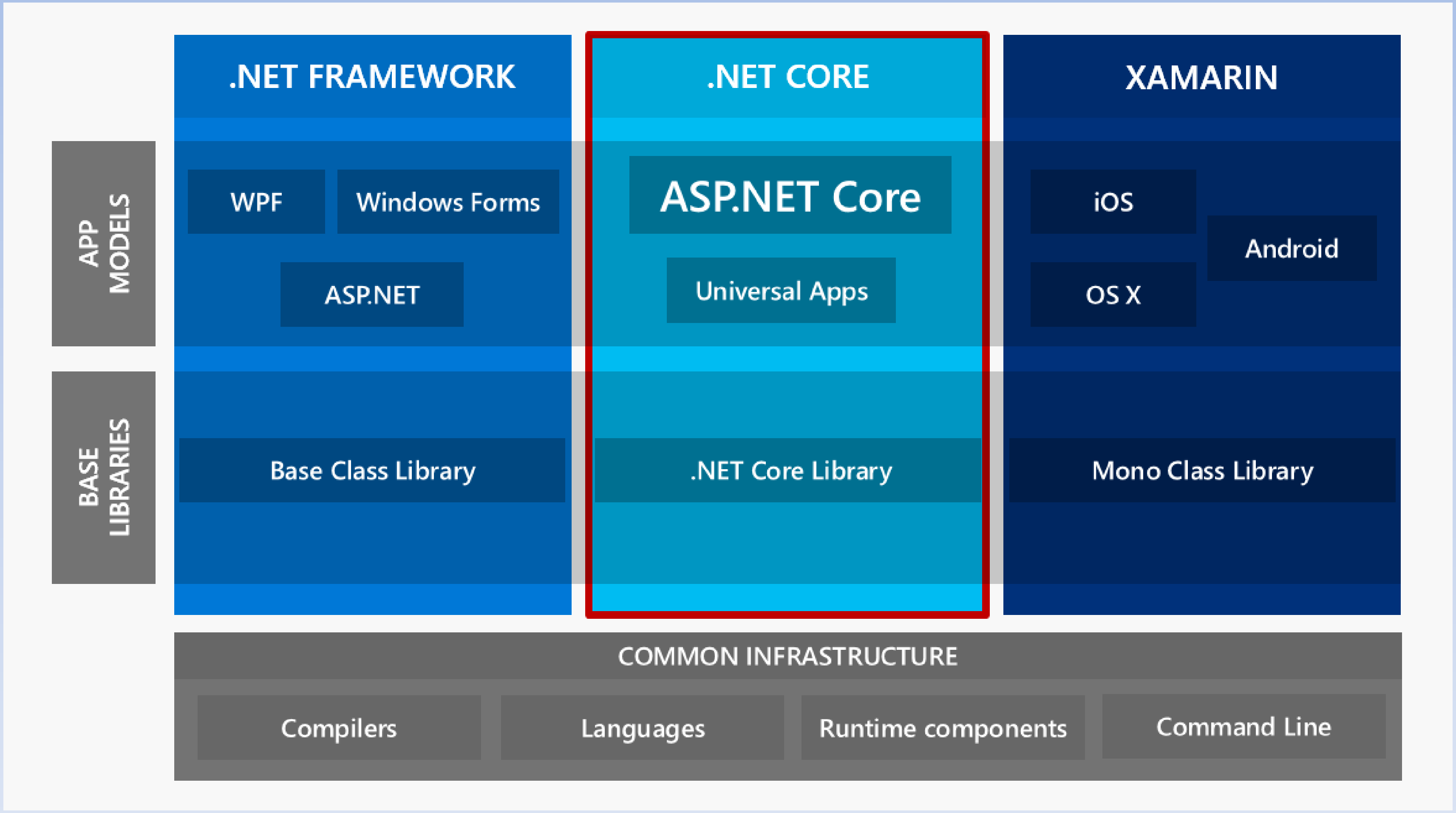
ASP.NET Core

ASP.NET Core is an open-source web application framework, and the next generation of ASP.NET, developed by Microsoft and the community. It is a modular framework that runs on both the full .NET Framework, on Windows, and the cross-platform .NET Core. It is currently is ASP.NET Core 1.0 RC2. It will be released to RTM on June 27th.

Common Cross Platform .NET

- Unified .NET base level library
- Create a new .NET standard across all runtimes
- Support Windows, Linux, and Mac
- Support mobile applications, UWP applications, game development and desktop applications through mono and Xamarian

ASP.NET and .NET Core



Provided by Scott Hunter and Jeff Fritz

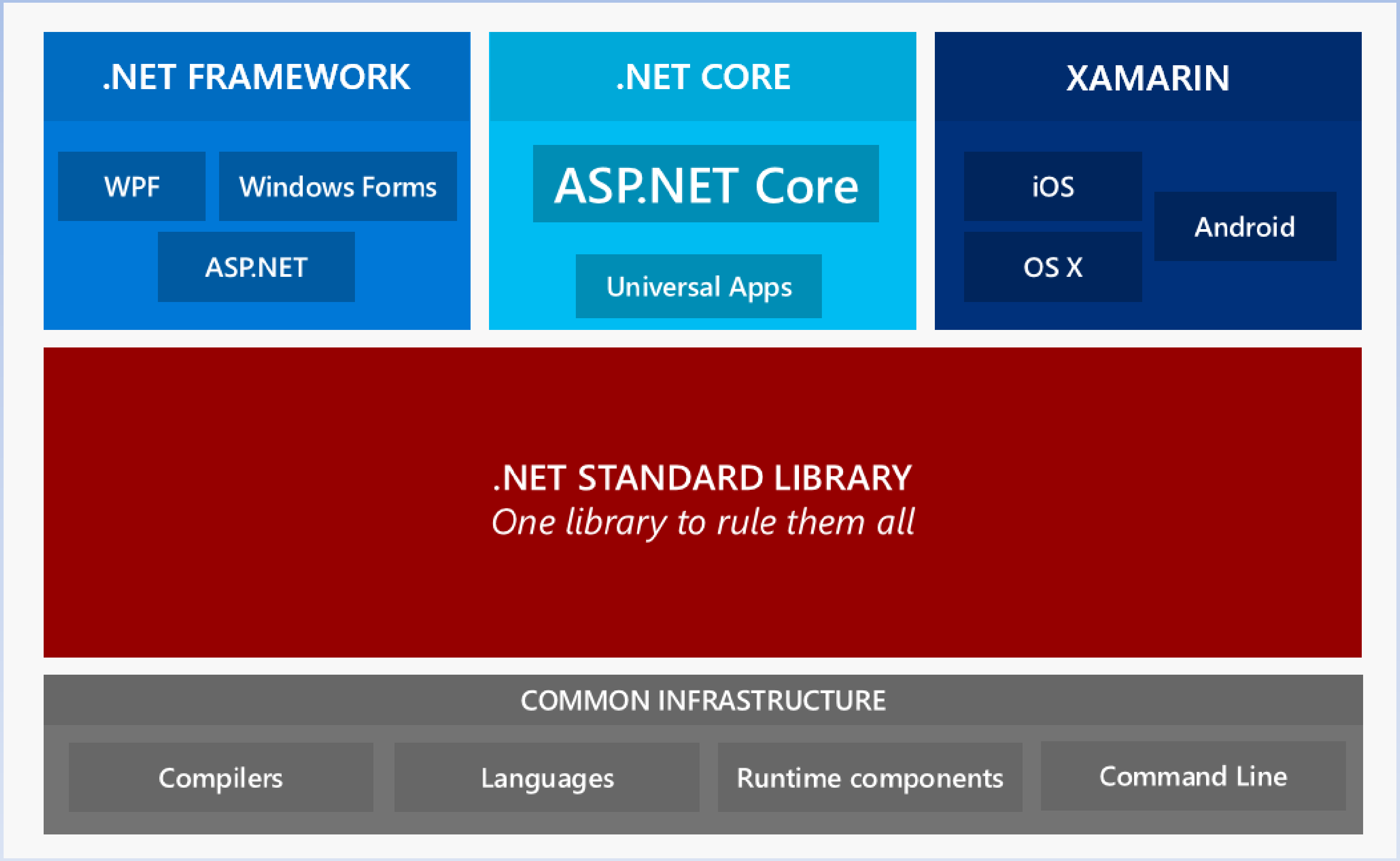
.NET Common Infrastructure

- .NET SDK – no requirement on Visual Studio
- No requirement for assemblies to be in the GAC
- Roslyn compiler platform
- Languages – C#, F#, VB.NET
- Runtime loaders
- Dotnet based command line tools
- Support for cross platform usage

.NET Class Libraries

- Unify all the different .NET runtimes
- Create .NET standard library
- Have the .NET standard library provide the core system services in mscorlib or the BCL (Base Class Library)
- FCL (Function class library) that provide System assemblies will not live in the .NET standard
- Allow other features to be developed and live above the .NET standard

.NET Baselibrary



Reason for ASP.NET Core

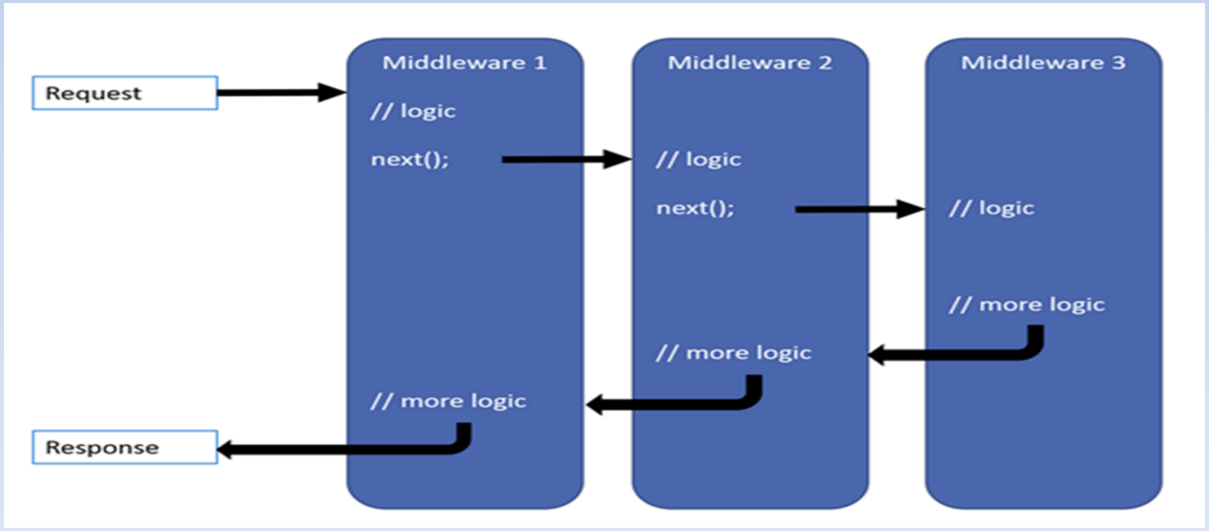
- Reduces the memory footprint of requests to a few kilobytes
- This framework is a tenth of the size of the full .NET framework
- Easier to configure and deploy
- Easier to deploy to the cloud both AWS and Azure
- Can be deployed into a Docker container
- Able to create and deploy microservices

New Programming Model

- Azure is the center of the Microsoft development universe
- ASP.NET Core framework is the first framework to support the .NET framework
- The programming model is based on ASP.NET MVC and should look similar to the previous model
- A new web programming framework that supports a webserver, kestrel, and a new HTTP pipeline process using Middleware and changes how common practices for modules and handlers
- The new infrastructure supports built-in IoC (Inversion of Control) for components into other components
- Supports the CoreCLR as well as the same set of tools in the full .NET framework
- Supports the various Node tools such as bower, gulp, and npm

Middleware in ASP.NET CORE 1.0

- One of the new features from ASP.NET Core 1.0 is the idea of Middleware. Middleware provides components to an application that examine the requests responses coming in to and going out from an ASP.NET Core application or microservice and gives developers total control over the HTTP pipeline.
- Middleware, as explained above, are components that live on the HTTP pipeline and examine requests and responses. This means that they are effectively classes that can decide whether or not to continue allowing the request to be processed by other Middleware components further down the stream. The pipeline looks like this:



Defining a Custom Middleware Component

A Middleware component, in an ASP.NET Core project, is a class like any other. The difference is that the Middleware component needs to have a private property of type RequestDelegate, like so:

```
public class AuthorizationMiddleware
{
    private readonly RequestDelegate _next;

    public AuthorizationMiddleware(RequestDelegate next)
    {
        _next = next;
    }
}
```

The _next property represents a delegate for the next component in the pipeline. Each component must also implement an async task called Invoke:

```
public async Task Invoke(HttpContext context)
{
    await _next.Invoke(context);
}
```

Middleware Tasks

Since each piece of Middleware can examine the incoming request and the corresponding response, one possible task Middleware could accomplish is that of authorization.

For a basic example, let's say that each request must NOT have a header "X-Not-Authorized", and if it does, the response must be returned immediately with a 401 Unauthorized status code. Our Middleware component's Invoke method would now look like this:

```
public async Task Invoke(HttpContext context)
{
    if (context.Request.Headers.Keys.Contains("X-Not-Authorized"))
    {
        context.Response.StatusCode = 401; //Unauthorized
        return;
    }

    await _next.Invoke(context);
}
```

Adding Middleware to the HTTP Pipeline

The environment for an ASP.NET Core 1.0 is set up in that app's Startup.cs file. In order to use our newly-created Middleware, we need to register them with the environment created by Startup class:

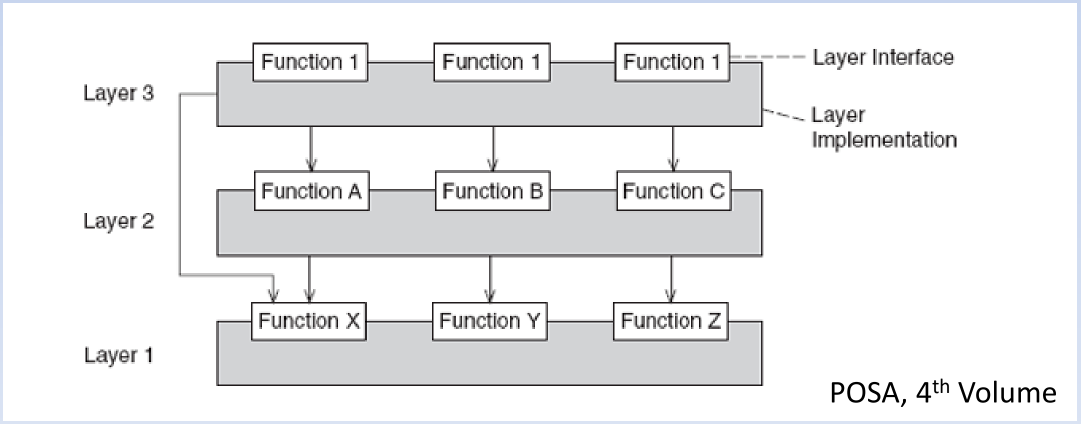
```
 // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
     public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
     {
        loggerFactory.AddConsole(Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();
        app.UseRequestHeaderMiddleware();
        app.UseProcessingTimeMiddleware();
        app.UseMvc();

     }
  }
```

Layered Systems

Reason

The major problem in software development is the management of complexity. Separation of concerns - the decomposition of the program into small components enables breaking a large complex system into simpler smaller ones.
Complex components may in turn require further decomposition.
Interfaces should be stable – they may even be an external contract for the system.
Changes to implementation details should not ripple through the system. Components should be replaceable with alternative implementations without affecting the rest of the system.
Similar responsibilities should have high cohesion – be grouped together – to aid reasoning about the software.
The dominant characteristic is a mix of higher and lower order operations, where high level operations depend on lower-level ones.



POSA, 4th Volume

Core Layer Tenets

Invariants

Layers can only interact with adjacent layers.

A layer can provide services to a layer above, and consume services provided by the layer below.

A layer depends only on lower layers and has no knowledge of the higher layers.

The structure can be compared with a stack or onion.

Properties

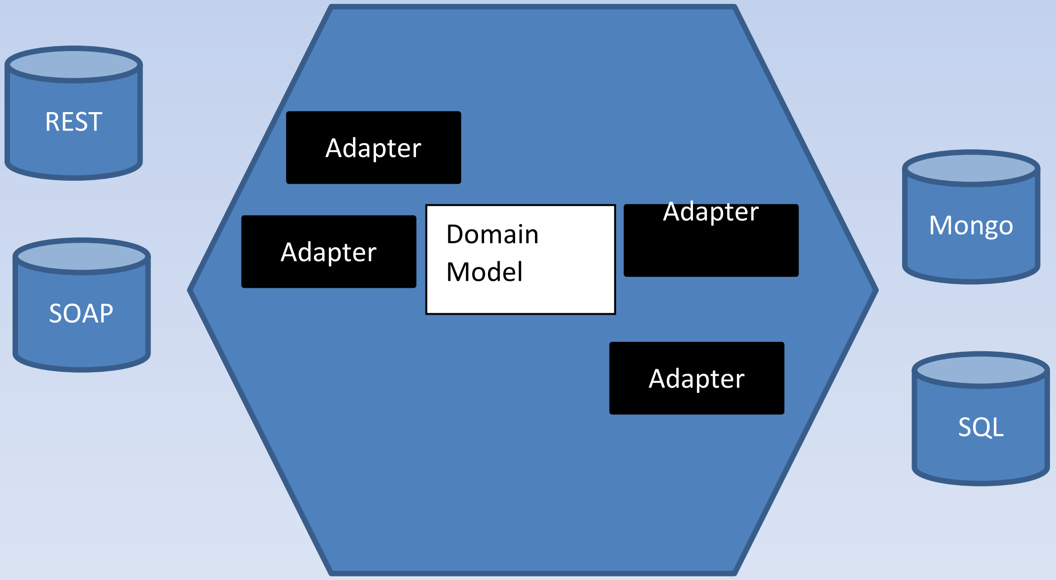
Designs are based on increasing levels of abstraction. This allows portioning of a complex problem into a series of steps.

Changes to one layer impact at most two layers – below and above, supporting change.

Different implementations of the same layer can be reused interchangeably provided they support the same interface

Hexagonal Architecture

The Hexagonal Architectural style is a variation of the layered architectural style which makes clear the separation between the domain model which contains the rules of our application and the adapters, which abstract the inputs to the system and outputs to client applications. The advantage of this style is that the application is decoupled from the nature of the input or output access, and any frameworks used to implement them.



What is a Monolith?

- By Monolith we mean an application that is integrated and deployed as a whole.
  - This can include distribution from a Monolith
  - A web application with browser, server, database, and message queues may be a monolith, if it must be integrated and deployed as a complete unit.

Microservice characteristics

- A microservice is responsible for one single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

Provided by
Martin Fowler

Why Microservices – not a Monolith

- Enable continuous delivery
- Allow for easy maintenance
- Provide robust design
- Individual microservices can scale up and down independent of other microservices
- Enables continuous delivery

Distributed Application Issues

- The network is unreliable.
- Latency is high.
- Bandwidth is limited.
- The network is insecure.
- Topology changes.
- There is no centralized administration.
- Transport cost is variable.
- The network is heterogeneous.

Quality of Service

Any call to remote components, such as a database, message broker, or REST service can fail. Code that makes assumption that they will succeed is subject to the Distributed Application Issues.

For this reason we need to set a limit on the time we wait for a response from the a service or other remote resource.

Otherwise we risk tying up a thread waiting for a response that will never come, potentially causing a cascade failure where once all the resources on the machine are tied up processing requests to which no response will ever come, we cannot process new requests.

Thus, if those requests do not have a timeout, that initial failure may now bubble to our caller, and continue on and on.

The Circuit Breaker Pattern

In a distributed environment where a microservice performs operations that access remote resources and services, it is possible for these operations to fail due to transient faults such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable.

However, there may also be situations where faults are due to unexpected events that are less easily anticipated, and that may take much longer to rectify. These faults can range in severity from a partial loss of connectivity to the complete failure of a microservice. In these situations it may be pointless for a microservice to continually retry performing an operation that is unlikely to succeed, and instead the microservice should quickly accept that the operation has failed and handle this failure accordingly.

The Circuit Breaker pattern prevents a microservice from executing an operation that is likely to fail, thus freeing up resources that would otherwise be consumed waiting for a timeout and retry cycle to occur.

Once normal microservice has been restored to the server or resource pool the circuit breaker pattern allows detection of the resumption of service.

A Circuit Breaker acts as a proxy to operations that can fail. It has one of three states:

Closed: Requests are routed as normal, on a failure a counter is incremented and if the threshold is exceeded within a time limit, the circuit breaker opens.

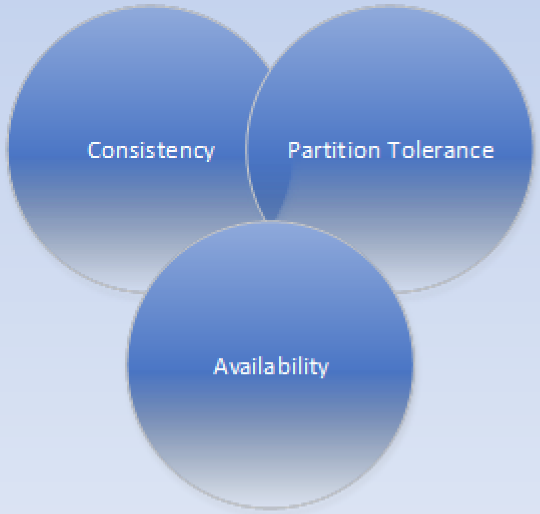
Open: No calls are allowed, and are failed automatically by the proxy. After a specified time interval the circuit breaker is moved to Half-Open state.

Half-Open: A call is allowed. On a failure the breaker moves to Open state, on a success it moves to Closed state.

CAP Theorem- Microservice Operation

Created by Eric Brewer.

- In a distributed system, of these three properties, you can only pick can two.
- Consistency (all nodes agree on the state)
- Availability (you get a response from one of the nodes when you make a request)
- Partition tolerance (if nodes cannot communicate, or communicate at too high a latency, the system continues to offer service)
- You can choose CP, or AP or CA if P indicates catastrophic failure.



Virtual Machines Versus Containers

- Small microservice calculates the rating of a stock portfolio on a daily – runs periodically.
  - Virtual machines take time to start.
  - The size of the virtual machine can be large.
  - Scaling virtual machines requires new larger machine with more CPU, memory, and storage.
  - Virtual machines use considerable resources and can be expensive.

The Twelve Factor App

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
- And can **scale up** without significant changes to tooling, architecture, or development practices.

http://12factor.net/

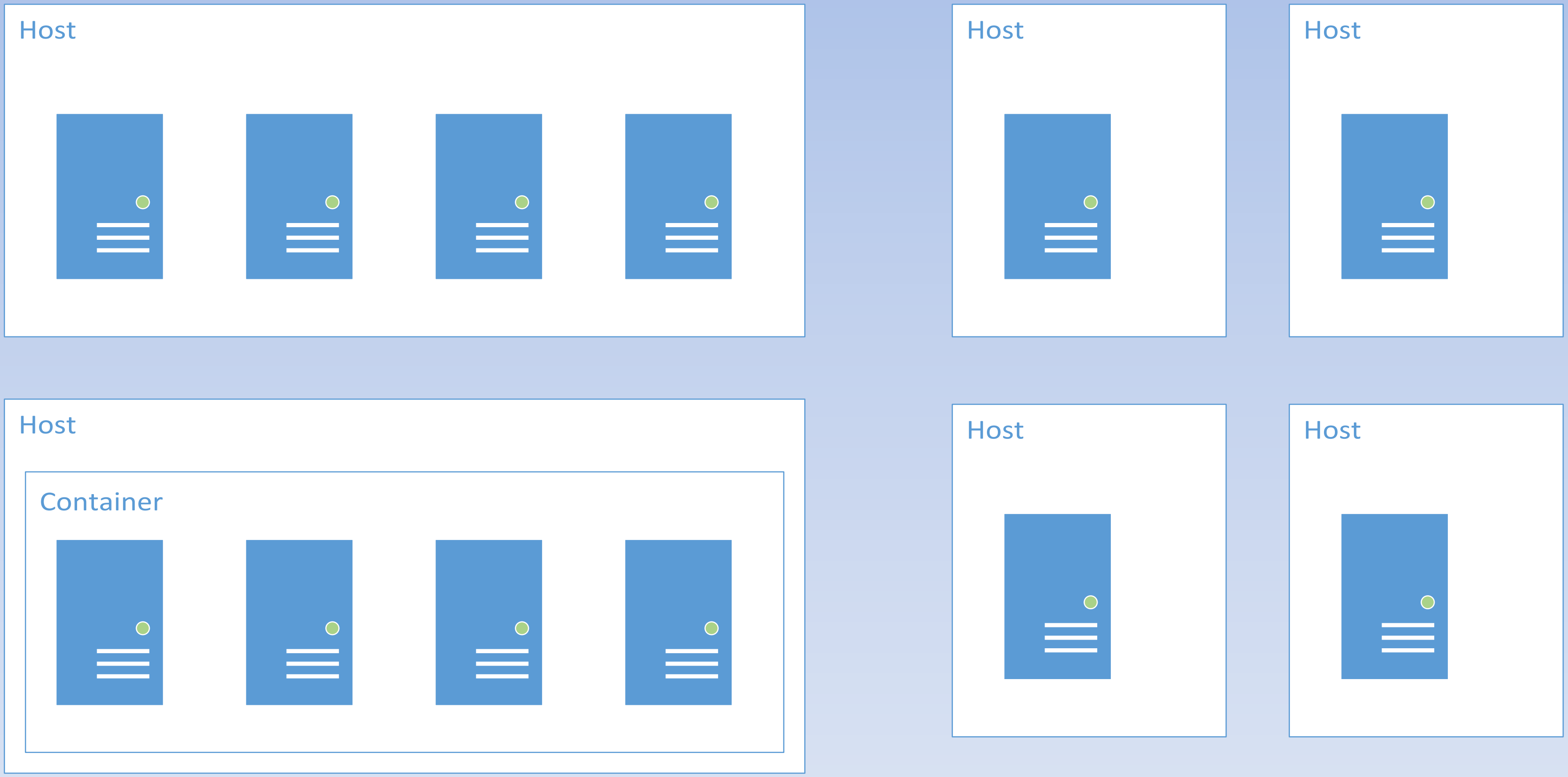
The Twelve Factors

- I. Codebase
- One codebase tracked in revision control, many deploys
- II. Dependencies
- Explicitly declare and isolate dependencies
- III. Config
- Store config in the environment
- IV. Backing services
- Treat backing services as attached resources
- V. Build, release, run
- Strictly separate build and run stages
- VI. Processes
- Execute the app as one or more stateless processes
- VII. Port binding
- Export services via port binding
- VIII. Concurrency
- Scale out via the process model
- IX. Disposability
- Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity
- Keep development, staging, and production as similar as possible
- XI. Logs
- Treat logs as event streams
- XII. Admin processes
- Run admin/management tasks as one-off processes

Containers - Docker

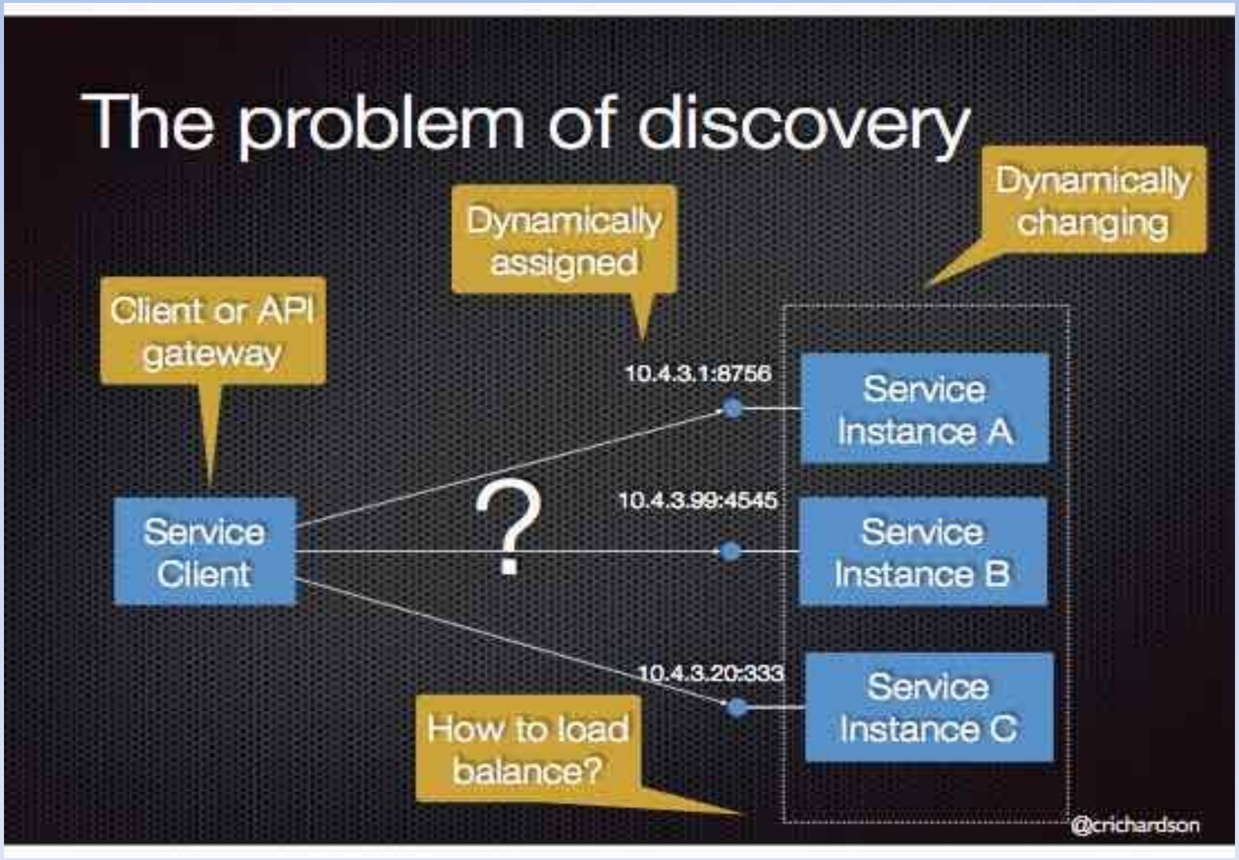
- The container allows for the separation of an operating system in order to run multiple applications in the system.
- Each container has an isolated view of network stack, file system, and process tree.
- Each container is unaware of other containers sharing the operating system.
- Containers are encapsulated, individually deployable components running as isolated instances on the same kernel leveraging operating system virtualization.
- Each container isolates an application or microservice with its runtime, dependencies, libraries with access to its required resources.

Service To Host / Container Mapping



Service Discovery

- Microservices need to be able to register themselves.
- Microservices need to be find other microservices.
- Microservices need to be load balanced.



Docker - Why Developers Care

- Build once run anywhere
  - A clean, safe, and portable runtime environment for your application or microservice.
  - No worries about missing dependencies, packages and other pain dependencies during subsequent deployments.
  - Run each app in its own isolated container, so you can run various versions of libraries and other dependencies for each app without worrying.
  - Automate testing, integration, packaging, and scripting.
  - Reduce/eliminate concerns about compatibility on different platforms, either your own or your customers.
  - Cheap containers to deploy microservices. A VM without the overhead of a VM. Instant replay and reset of images.

Docker- Why DevOps Care

- Configure once run anywhere
  - Make the entire lifecycle more efficient, consistent, and repeatable.
  - Increase the quality of code produced by developers.
  - Eliminate inconsistencies between development, test, production, and customer environments.
  - Support segregation of duties.
  - Significantly improves the speed and reliability of continuous deployment and continuous integration systems.
  - Because the containers are so lightweight, address significant performance, costs, deployment, and portability issues normally associated with VMs.

Building the Starship Enterprise – ASP.NET Core

- With all application infrastructure you have two choices:
- Shut the application infrastructure down while you replace it.
- Keep the application infrastructure working, while you replace it.
- Always avoid the first.