Building real world production-ready Web APIs with ASP.NET Core

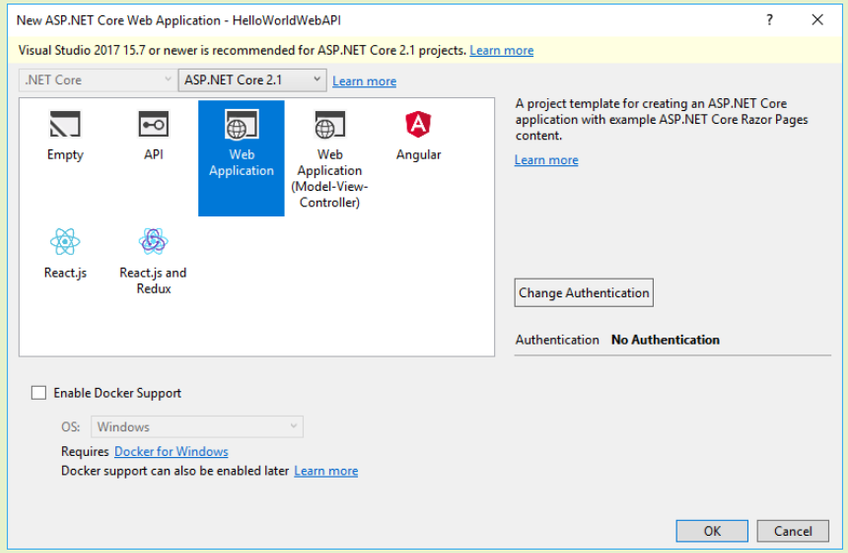
Alex Thissen
Cloud architect at Xpirit, The Netherlands
@alexthissen

DOTNEXT
.NET КОНФЕРЕНЦИЯ

Xpirit
Think ahead. Act now.

MVP Microsoft® Most Valuable Professional

Journey to a modern cloud-ready Web API



File, New Project...

Documentation

Monitoring

Resilient

Secure

Scalable

Instrumentation

Versioning

Service contract tested

Health checks

ASP.NET Core 2.1

Many useful new features for building Web APIs

- Convention based assumptions for action methods
- `ActionResult<T>` return type for predictable API surface
- API Behavior options: Model validation errors
- `HttpClientFactory`
- Resiliency policies
- Global tools for HTTPS support
- HTTP Strict Transport Security (HSTS)
- OpenAPI support

Design your modern Web API

Choose REST or POX

RESTful or less REST

Check Richardson maturity model

Take a look at:

Microsoft's API guidelines

Azure Architecture API design

Follow Web API best practices

Conform to HTTP semantics

Shape of URIs and meaningful verbs

Idempotent and stateless

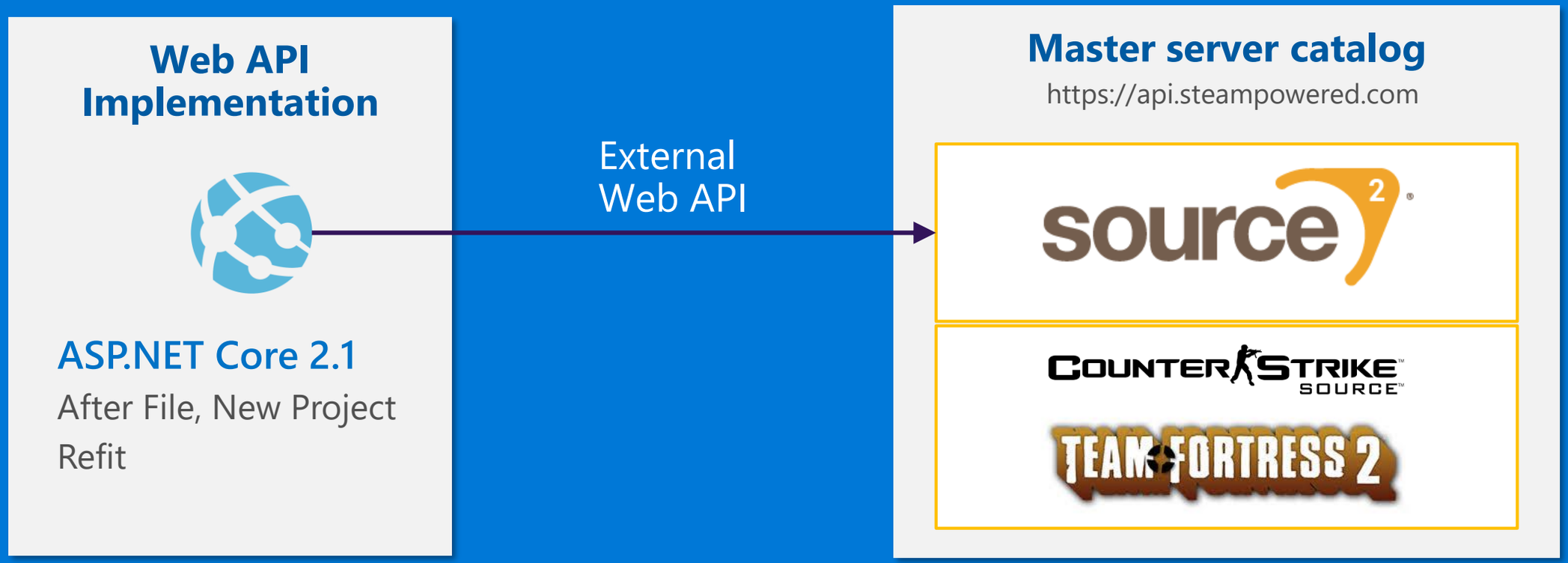
Asynchronous support for long-running API operations

HATEOAS for resource navigation

Versioning of your API

Demo Web API

Quick walkthrough of ASP.NET Core 2.1 project



Versioning Web APIs

Allow your APIs to evolve over time

Versioning of your API

Dealing with types of change

Incompatible changes	**Compatible**
Breaking of contract Violation of 'Principle of Least Astonishment' → Increase major and/or minor version	Careful with behavior changes → Keep current version

Version options

Advertise versions

Group-name and/or version number	**2018-22-04** or **v1.0**
URI includes version in path or query string	/api/**v1.2**/gameserver or /api/gameserver**?api-version=1.2**
ACCEPT header or custom media type	ACCEPT: application/vnd.gameserver.app-**v1.2**+json

Implementing versioning

Startup options

Attributes for version metadata

Conventions for runtime flexibility

Microsoft.AspNetCore.Mvc.Versioning
Version 2.1.0

Can keep implementation of versions together

Separating by namespace is convenient

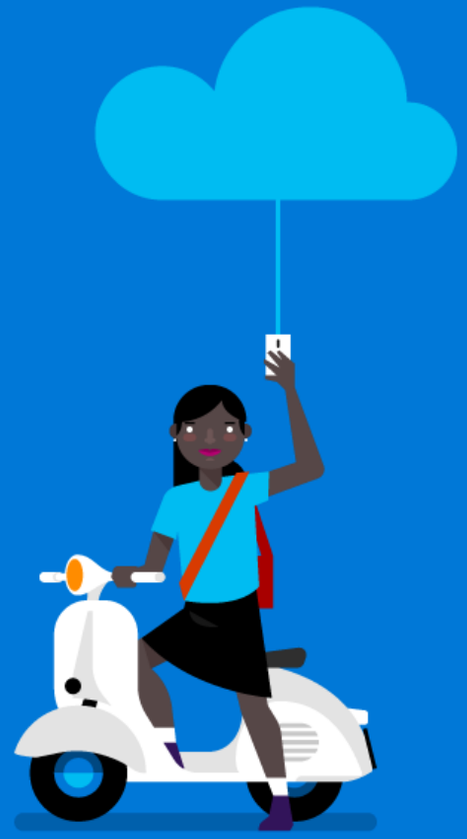
But might want to create separate solutions depending on complexity

```
namespace GameServerWebAPI.Controllers.V1
{
  [Route("api/v{version:apiVersion}/[controller]")]
  [ApiVersion("1.0")]
  public class GameServerController : ControllerBase
```

Demo: **Versioning**

Evolving your web API

Attributes for versions
Changed routing
Fluent API for runtime versioning



Resiliency

Design for failure, because things will break

Resiliency: design for failure

Cloud:
Transient errors
are a fact,
not a possibility

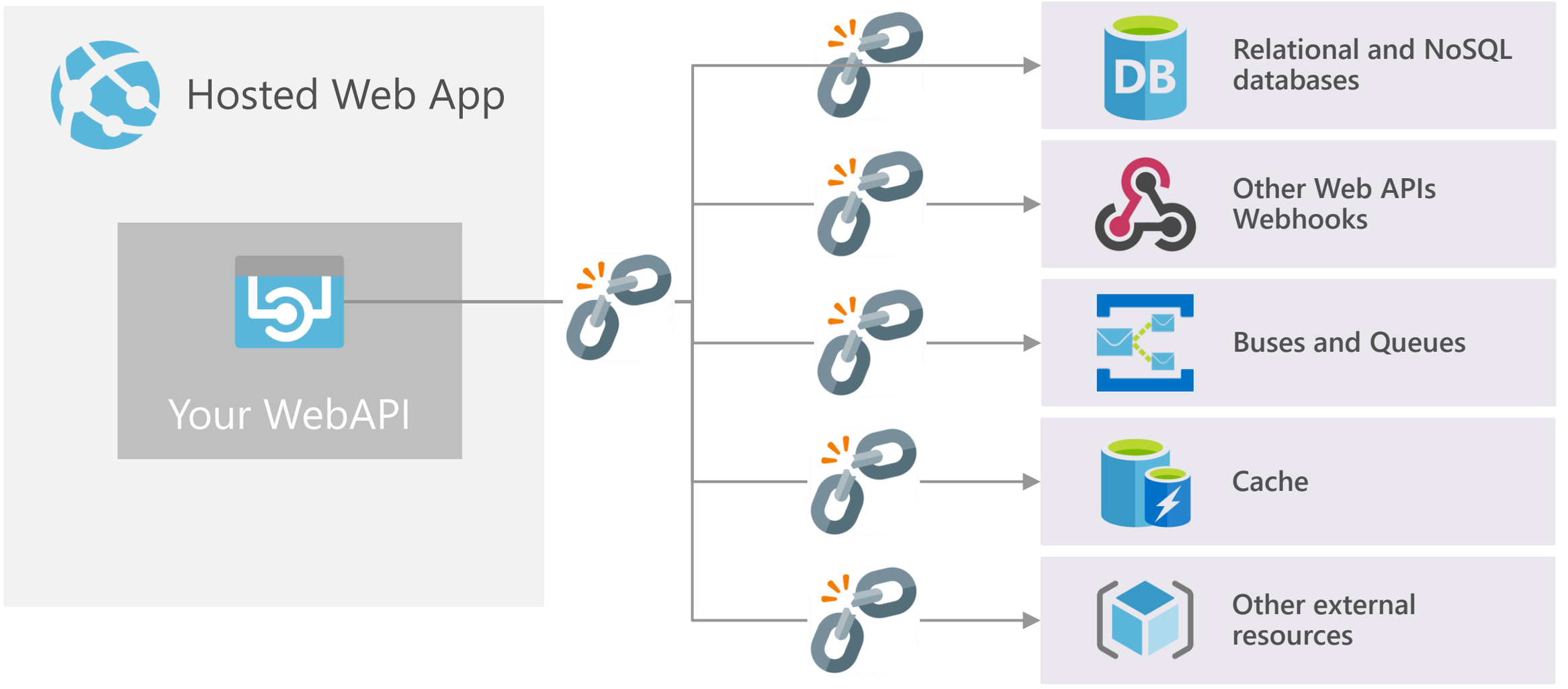
Determine your strategy for resiliency

- Which failures can you expect?
- Will you be able to recover in time? Is transient?
- How do you handle outages?

Focuses on dependencies outside of your code

Proper exception handling in your implementation is a prerequisite

Depending on external resources



Cloud patterns for handling transient faults

Retry
Retry a number of times.

Circuit breaker
If you know you are likely to fail, fail fast

Bulkhead isolation
Isolate parts of implementation to contain failures

Timeout
Avoid waiting too long. Fail within a certain amount of time

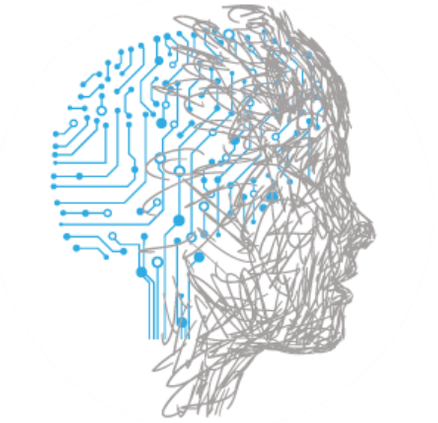
Fallback
Use an alternative when all attempts have failed

Careful retries

Not too aggressive to avoid contention:

Exponential back-off, regular intervals, increasing intervals, immediate, random

Some Azure services have built-in retry logic



Built-in Retry mechanisms

Most Azure services and client SDKs include retry mechanism

Service	Retry capabilities
Azure Storage	Native in client
SQL Database with Entity Framework	Native in client
SQL Database with Entity Framework Core	Native in client

```
services.AddDbContext<GameServerContext>(options => {
    string connectionString =
        Configuration.GetConnectionString("GameServerContext");
    options.UseSqlServer(connectionString, sqlOptions =>
      {
        sqlOptions.EnableRetryOnFailure(
          maxRetryCount: 5,
          maxRetryDelay: TimeSpan.FromSeconds(30),
          errorNumbersToAdd: null);
      });
  });
```

Fault handling policies

Use policy configurations

Create pro-active handling strategies

Add instrumentation and logging where necessary

Centralize policies in registry

Reuse existing policies


```
Policy.Handle<HttpRequestException>().WaitAndRetryAsync(
  6, // Number of retries
  retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
  (exception, timeSpan, retryCount, context) => { // On retry
    var msg = $"Retry {retryCount} at {context.ExecutionKey} : {exception}.";
      logger.LogWarning(msg);
    })
```

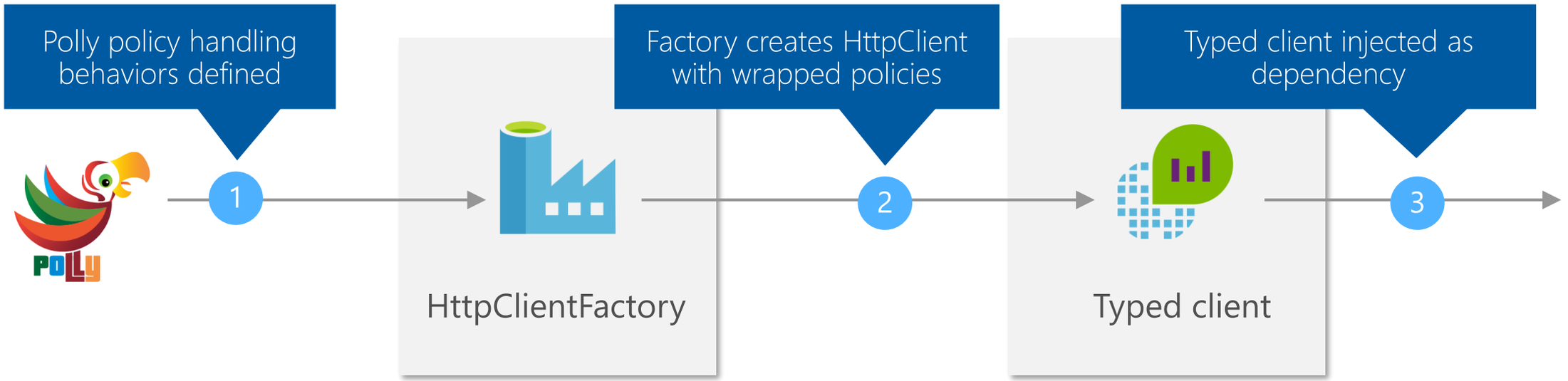
Creating resilient HTTP clients with Polly

Leverage HttpClientFactory

Configure HttpClient objects and create typed clients that contains these clients

Built-in Polly support

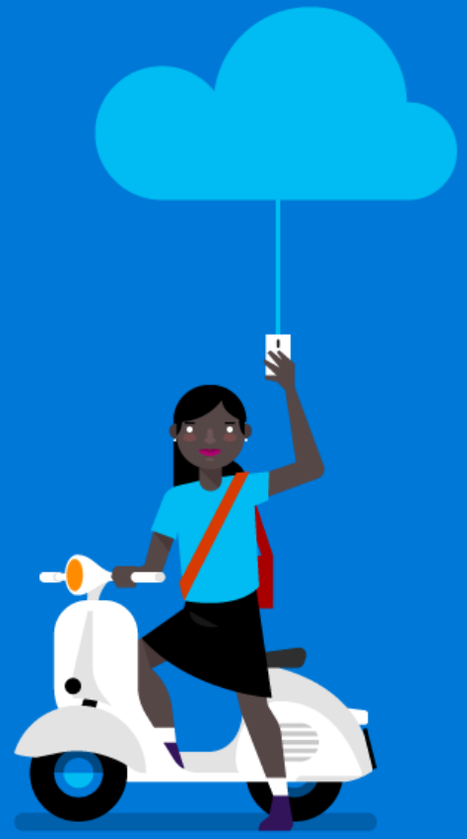
Microsoft.Extensions.Http combined with **Microsoft.Extensions.Http.Polly**



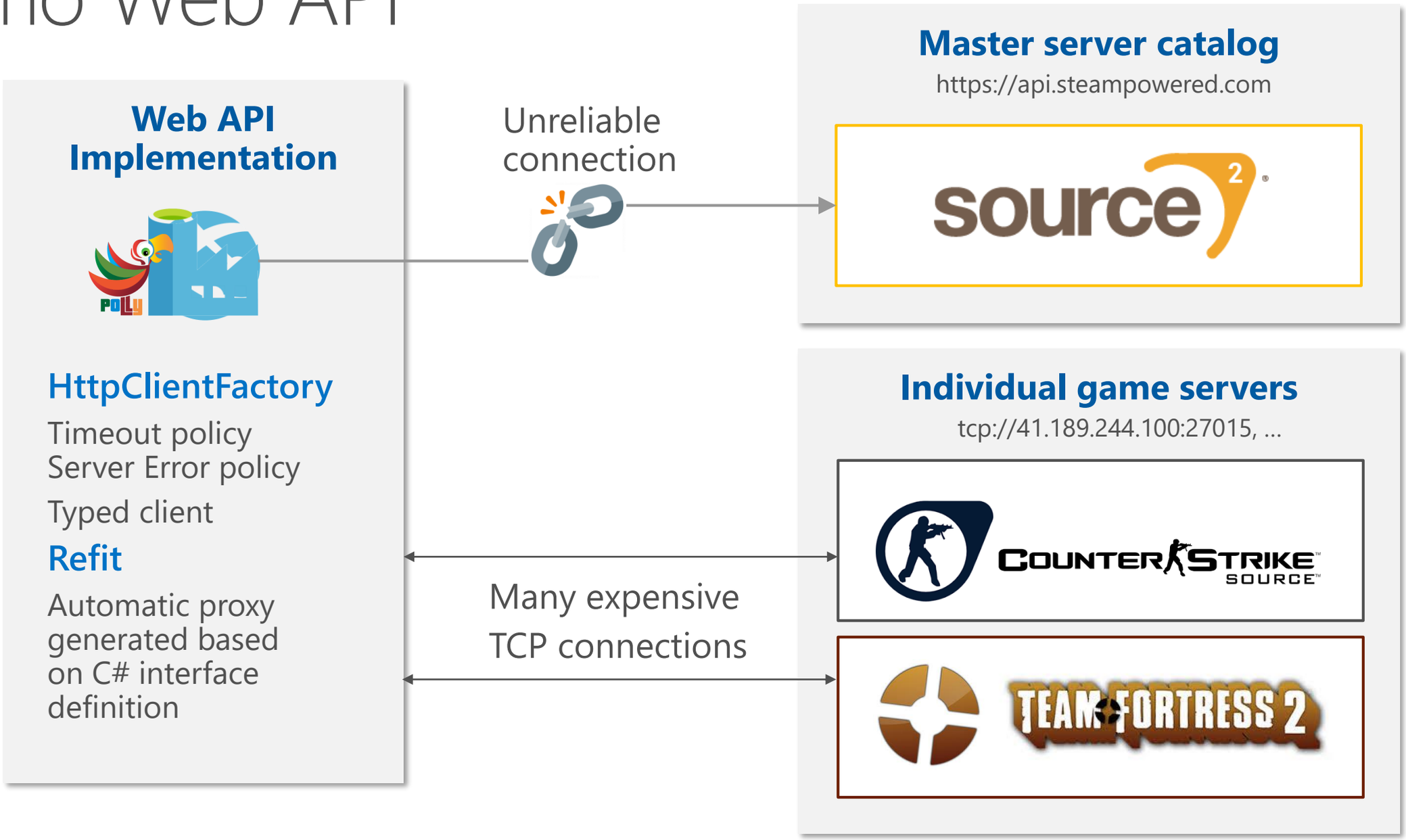
Demo: **Resiliency**

Dealing with failures

HttpClientFactory
Typed clients
Polly fault handling policies

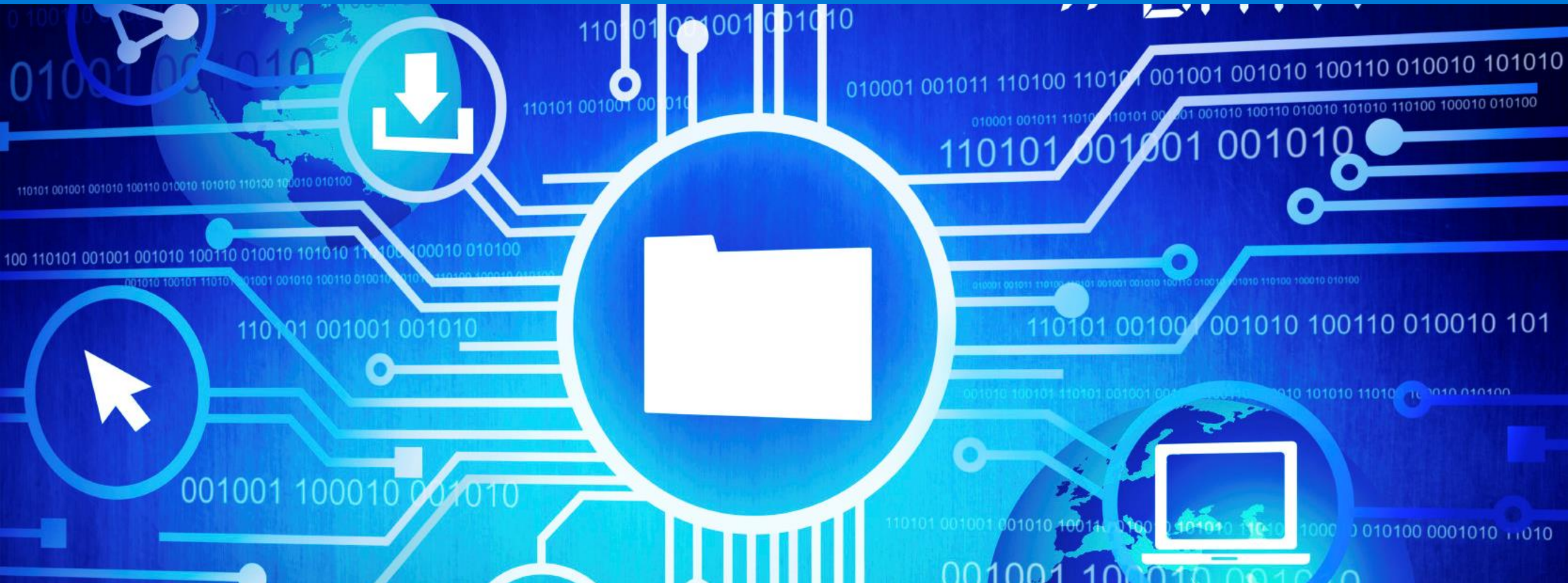


Demo Web API


Web API Implementation
HttpClientFactory
Timeout policy
Server Error policy
Typed client
Refit
Automatic proxy generated based on C# interface definition
Unreliable connection
Many expensive TCP connections
Master server catalog
https://api.steampowered.com
source 2
Individual game servers
tcp://41.189.244.100:27015, ...
COUNTER-STRIKE SOURCE
TEAM FORTRESS 2

Documentation

Describe your Web API



OpenAPI as documentation

OPENAPI INITIATIVE

- Extendable
- Mocking
- Automated testing
- Generate documentation
- Data ingestion
- Lots of libraries
- Lots of integrations

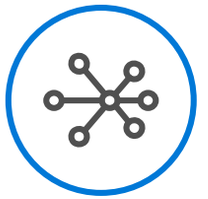
Specification of REST services

Schema comparable to WSDL for SOAP Services

JSON format

Previously known as Swagger

Service contract
Controller based grouping of operations

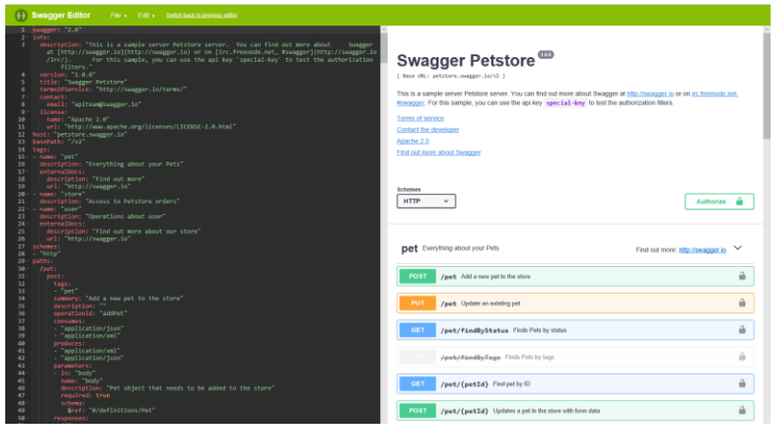


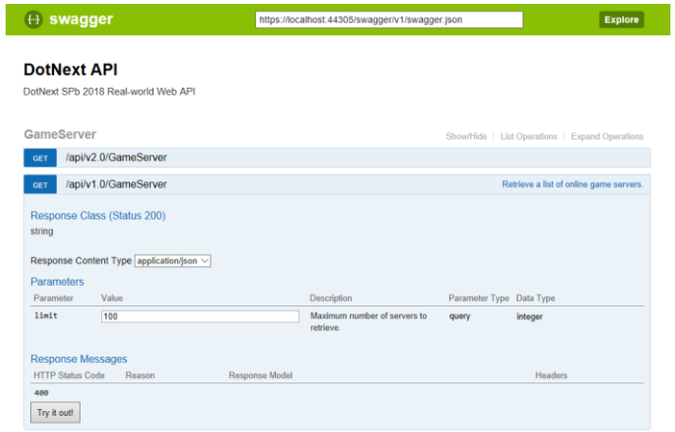
Message contract
Individual API actions

Data contract
Object graphs

Swagger

Tooling to implement OpenAPI





.NET Core compatible

- Swashbuckle
- NSwag
- QSwag

Swagger Editor

Visual editor for creating Swagger files

Swagger UI

Help pages based on API surface and Swagger document

Libraries

Generate OpenAPI documentation, client libraries, server stubs

Describe your Web API

Metadata: Annotations in your code

Actions

```
[HttpGet("~/api/gameserver/status")]
[ProducesResponseType(typeof(GameServerStatus[]), 200)]
[ProducesResponseType(400)]
[UsedImplicitly]
```

Data contract

```
[Required]
[DefaultValue(false)]
```

API Controller

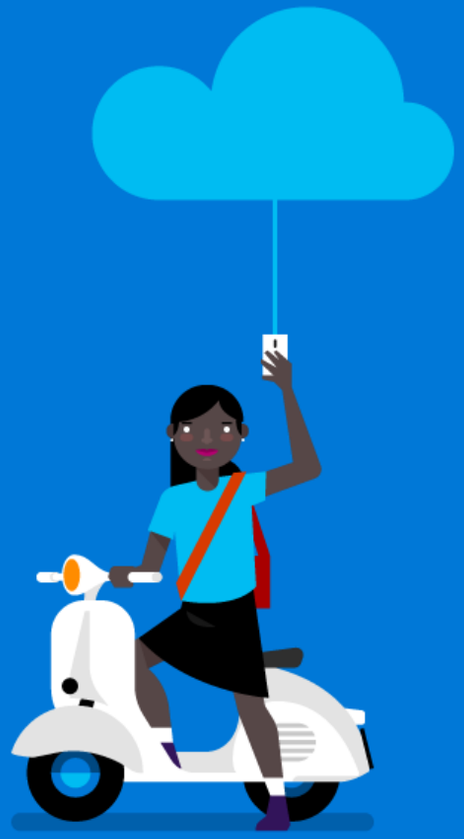
```
[Produces("application/json")]
public class GameServerController
```

Additional metadata

```
/// <response code="201">Returns the newly-created item</response>
/// <response code="400">If the item is null</response>
```

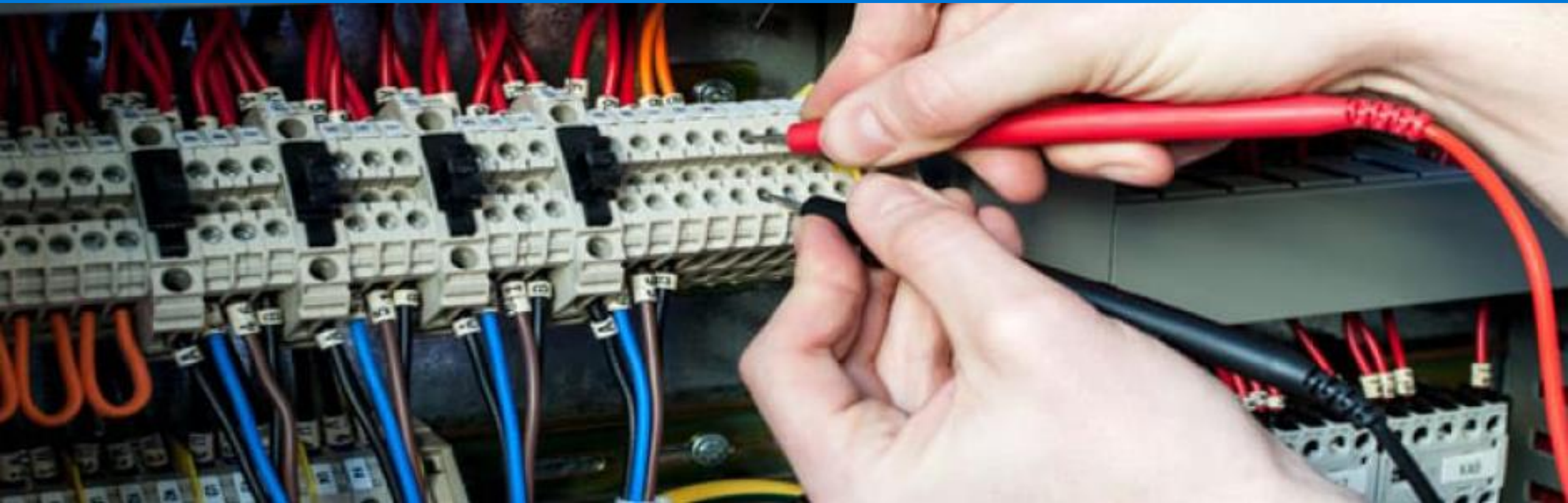
Demo: **Documentation**

Attributes and XML comments
NSwag generated OpenAPI specification
Multiple API versions



Consuming and testing Web APIs

Putting your Web API through its paces



Web API Client libraries

Provide a client SDK

Facilitates consuming your Web API
Typed proxies and client-side data model

Generate from specs

Use open source tools and libraries

Refit Version 4.3.0
Microsoft.Rest.ClientRuntime Version 2.1.0

AutoRest

- Uses OpenAPI spec
- Client libraries
- Used by Microsoft Azure SDKs

NSwag

- Server and client libraries
- NSwagStudio tool

Refit

- Dynamically generated
- Based on attributed C# interface

Service contract integration testing

Find out whether your Web API is still compatible

Did your changes break any clients?

Use versioned client proxies

Special ASP.NET Core test server

Microsoft.AspNetCore.TestHost
Version 2.1.0

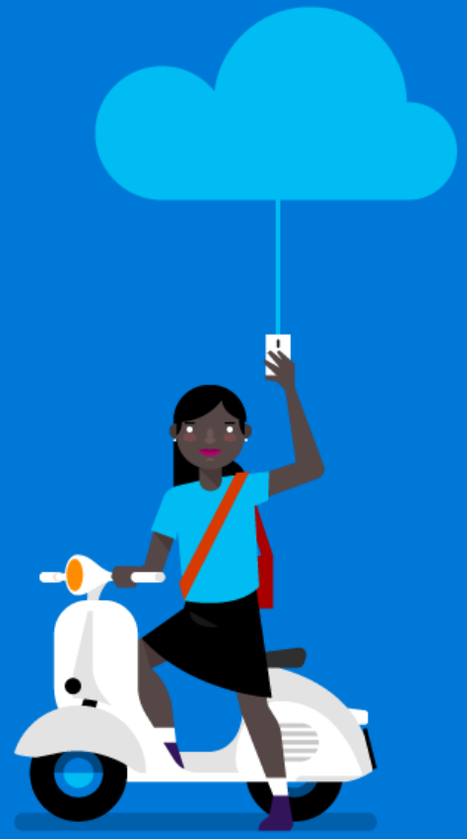
Integration testing from unit tests

Full Web API calls without network overhead

```
var builder = new WebHostBuilder() …
// Create test stack
TestServer server = new TestServer(builder);
HttpClient client = server.CreateClient();
DotNextAPI proxy = new DotNextAPI(client);
```

Demo: **Testing clients**

Generate client proxies with AutoRest
Service contract integration tests



Monitoring

Using telemetry, logging and instrumentation

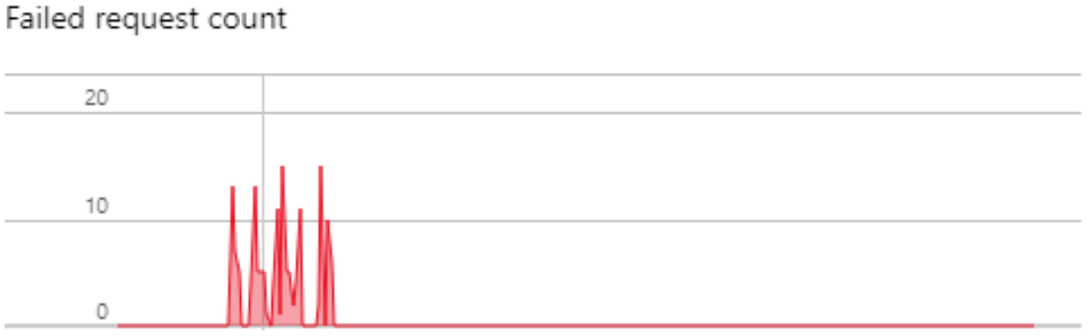
Instrumentation not optional

Who wants to fly in the blind?

Instrumentation, health and telemetry

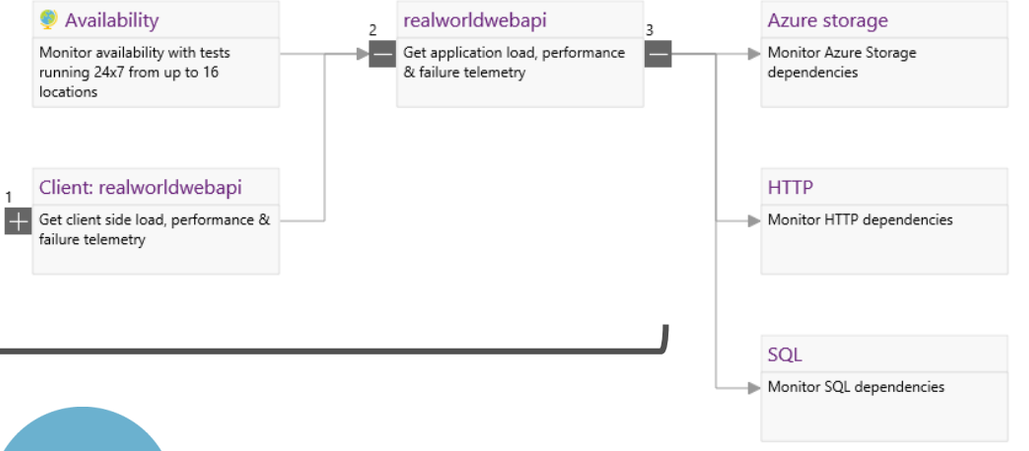
Individual applications

Health endpoints
Metrics (performance and other statistics)
Tracing calls



Entire landscape

End-to-end tracing
Application maps
Bottlenecks and failure locations

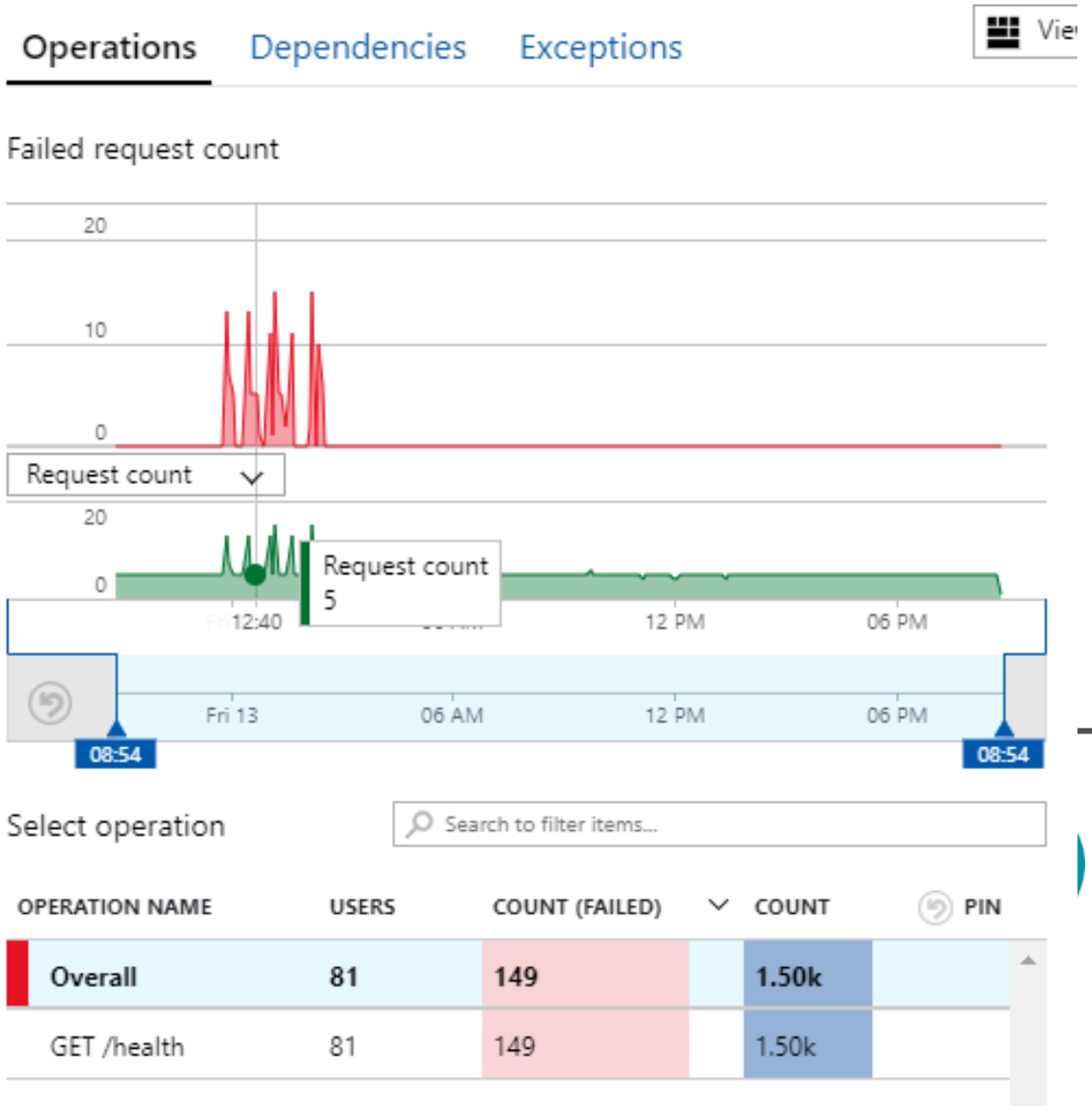


Application Insights

New Relic

AppMetrics

Instrumentation, health and telemetry



Health endpoints: how are you doing?

Specific endpoints to query status of Web API

Usually /health or /ping

Microsoft.AspNetCore.Diagnostics.HealthChecks
Version 2.2.0 (preview 1)
App.Metrics.AspNetCore.Health
Version 2.0.0

Middleware exposes health details

Bootstrap specific health checks and information to determine status

```
services.AddHealthChecks(checks => {
  checks
    .AddUrlCheck("https://api.steam.com")    // Web based dependencies
    .AddPrivateMemorySizeCheck(10000000);    // Maximum private memory
});
```

Combine with monitoring solution to alert ops

Structured logging

Message templates

Strings with named placeholders semantics

Avoid string interpolation for messages

```
// Placeholders in template become queryable custom data
logger.LogInformation("Searching with {SearchLimit} results.", limit);
```

Log providers supporting semantic logging

Azure Application Insights

Serilog

A better logging

Get your log levels right

Choose your level regardless of providers
Too much information makes log hard to read
Different log filter level per environment

Log Levels

5 – Critical
4 – Error
3 – Warning
2 – Information
1 – Debug
0 – Trace

Register logging providers (sinks)

```
loggerFactory.AddApplicationInsights(…);
loggerFactory.AddAzureWebAppDiagnostics(…)
loggerFactory.AddEventSourceLogger();
```

Tips and tweaks
High performance logging with LogMessage.Define
Use scopes for bundling logs messages
Apply filters per category prefix to reduce noise

Centralized logging

Leverage existing resources

Increases traceability and easy of access
Plug into facilities already available
Harvest across entire landscape

Appropriate filters for each sink

Minimum log level required
Special filter expressions

Your Web API
Instrumented with log output

≥ Trace

≥ Warning

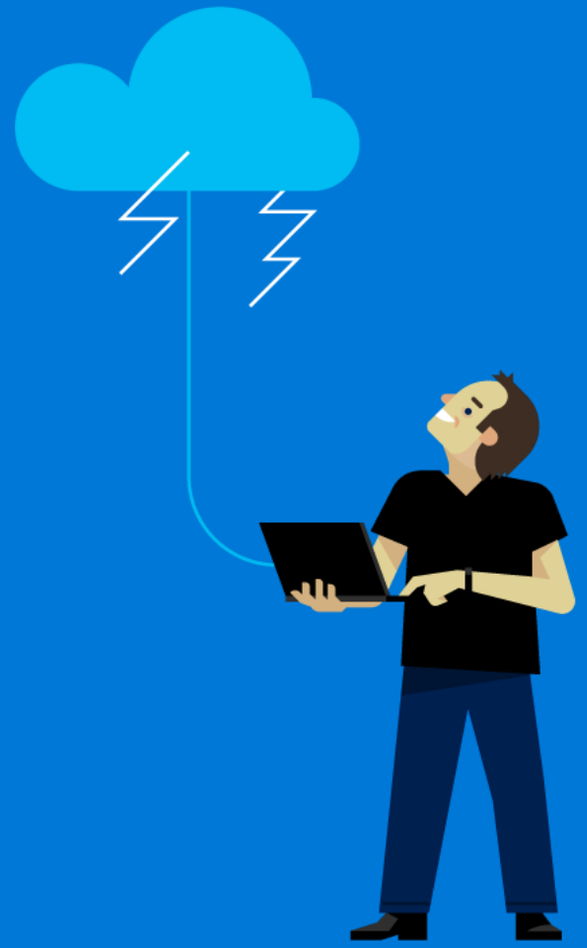
Metrics and monitoring
e.g. Application Insights
AppMetrics

Log indexers
e.g. OMS Log Analytics,
Splunk, ELK stack

Application logs and traces
File or blob based logs
ETW, IntelliTrace,
Azure App Service logging

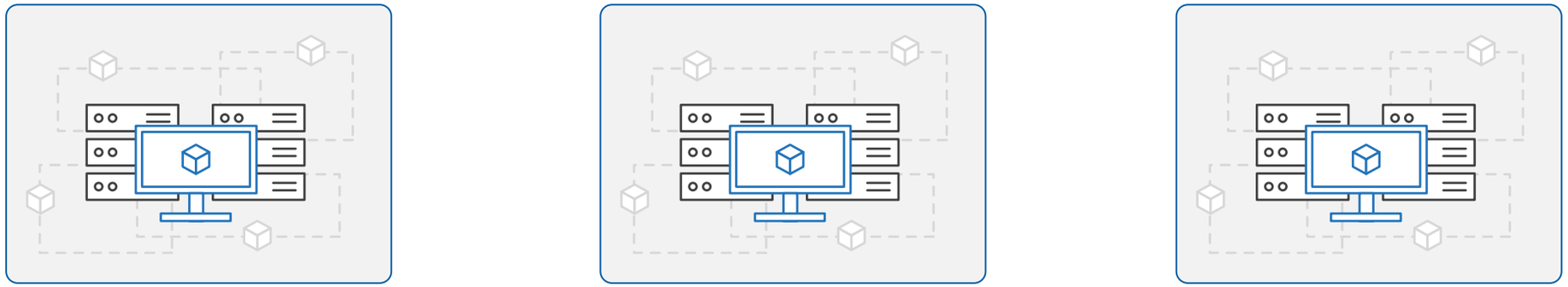
Demo: **Monitoring**

Logging
Monitoring and instrumentation
Health checks



Moving into production

Environments



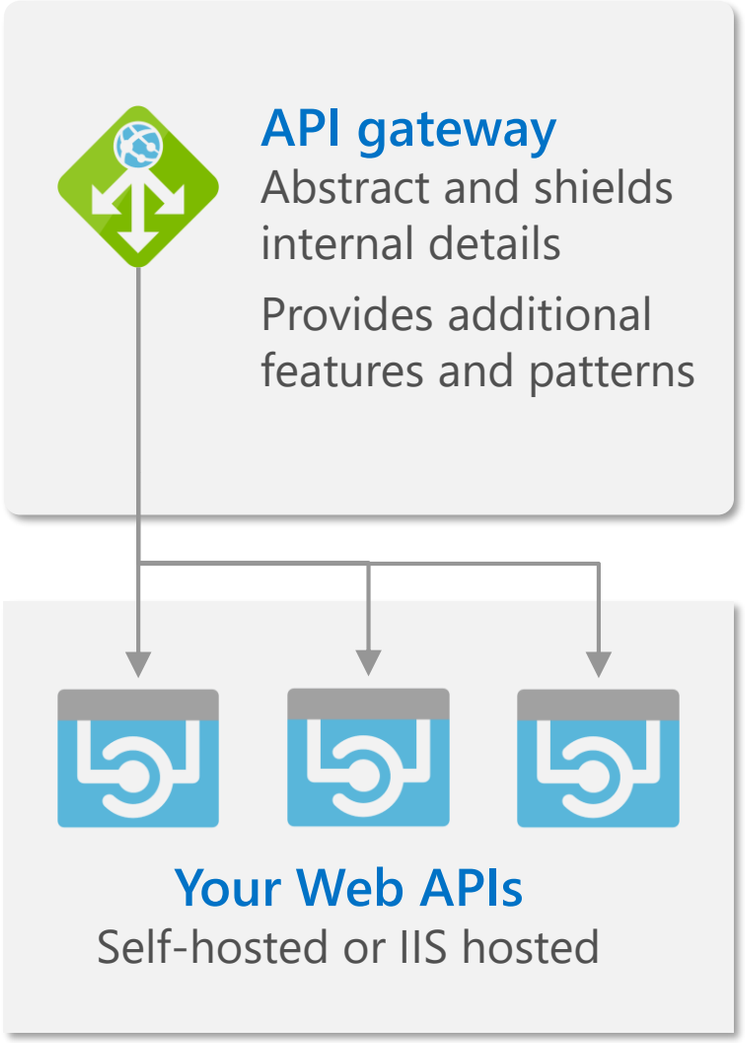
Test Staging Production

Keep environments same as much as possible

Environment specific configurations:

Files `appsettings.*.json`
Environment variables

Hosting in production



Integrate with an API gateway

E.g. Azure API Management or Amazon API Gateway

Leverage features from gateway

Throttling and client tracking
Monetization (if needed)
Developer registration and access of API keys

Do not expose Kestrel to Internet

Not security hardened like IIS
At least use reverse proxy, such as NGINX proxy

Take a look at:

Microsoft.AspNetCore.Proxy
Version 0.2.0

A new lifecycle

Production is everywhere

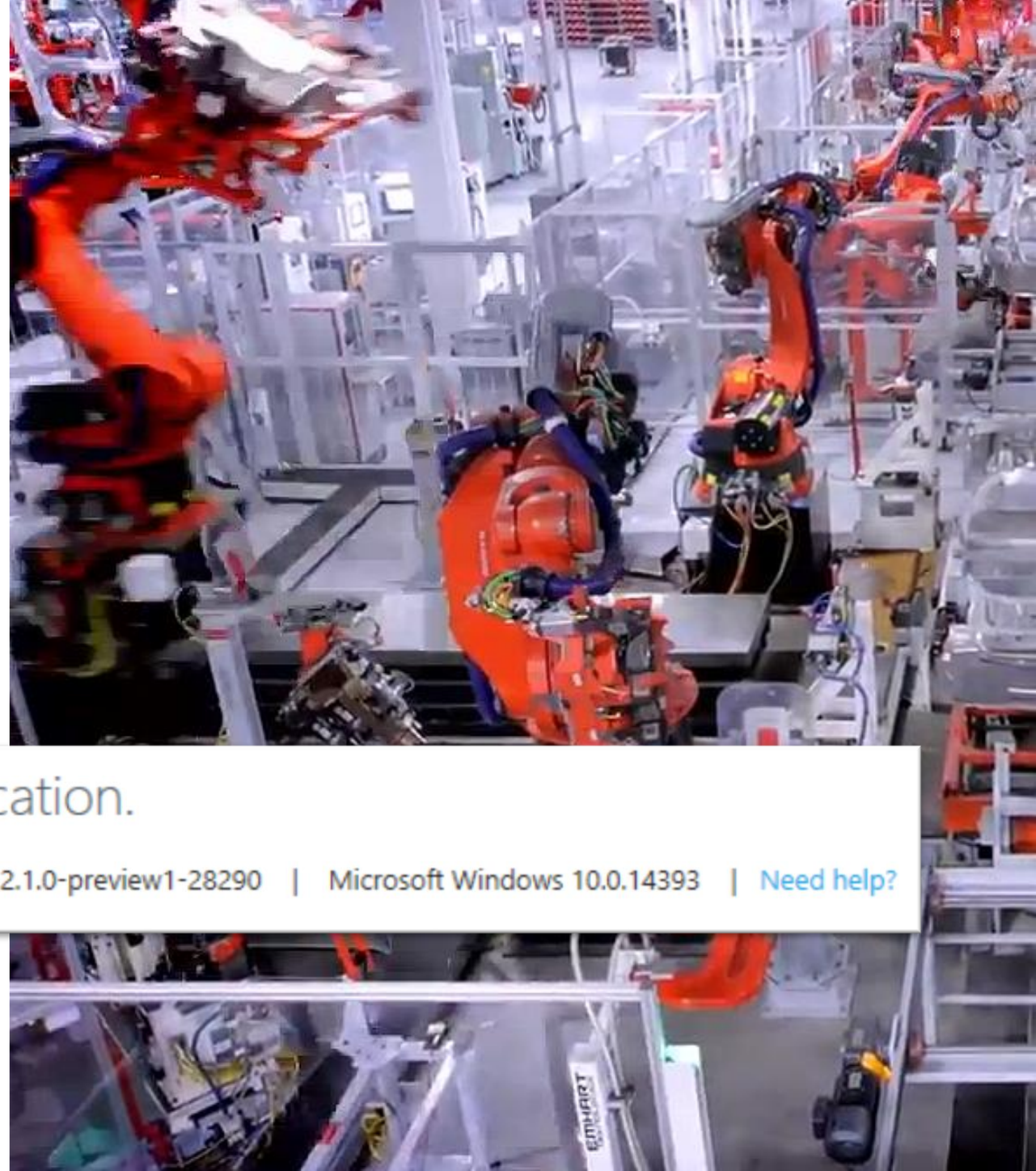
Automate everything

One-click deployments

Pipelines for build, release and infrastructure

Fixing bugs

Roll forward, never go back



An error occurred while starting the application.

.NET Core 4.6.26216.04 X86 v4.0.0.0 | Microsoft.AspNetCore.Hosting version 2.1.0-preview1-28290 | Microsoft Windows 10.0.14393 | Need help?

Testing in production

Works well with CI/CD

Breaks with traditional OTAP

No dependency on different environments

Traffic management

Run different versions side by side and observe

Release gates based on runtime behavior of new functionality

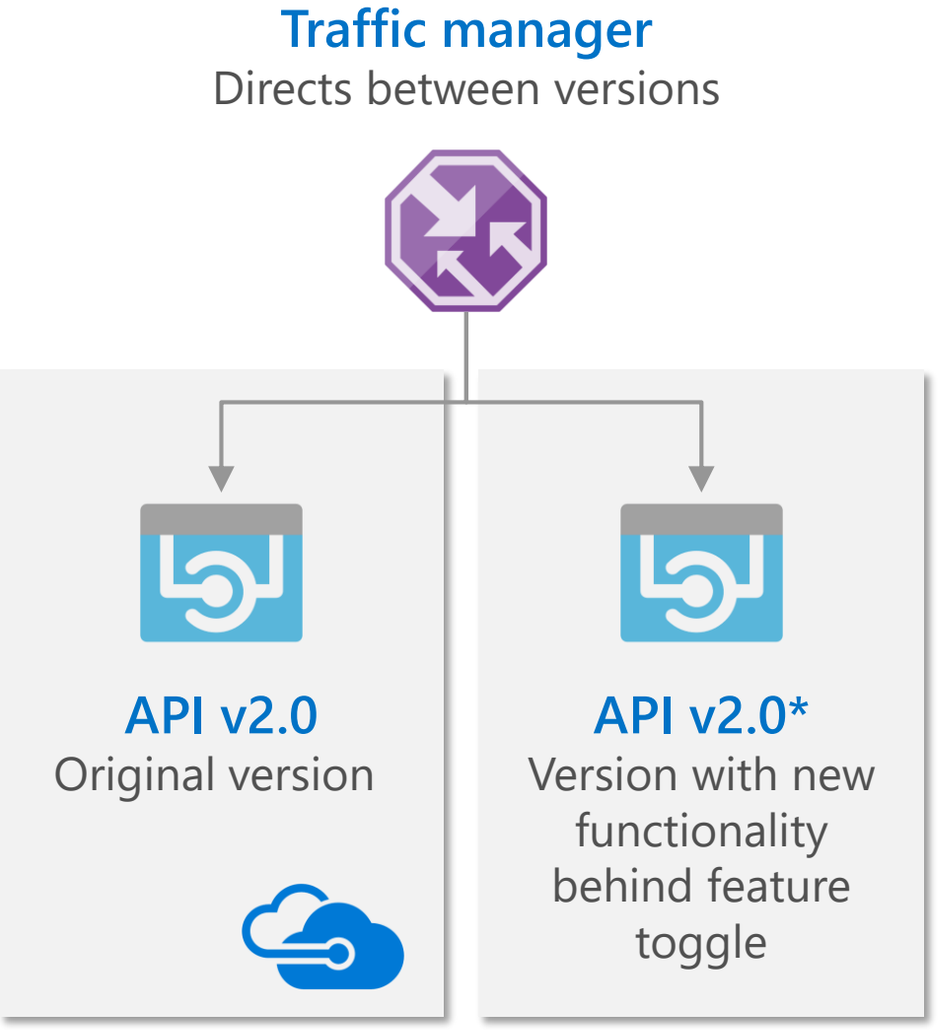
Feature toggles

New functionality behind a switch

Also works for Web APIs

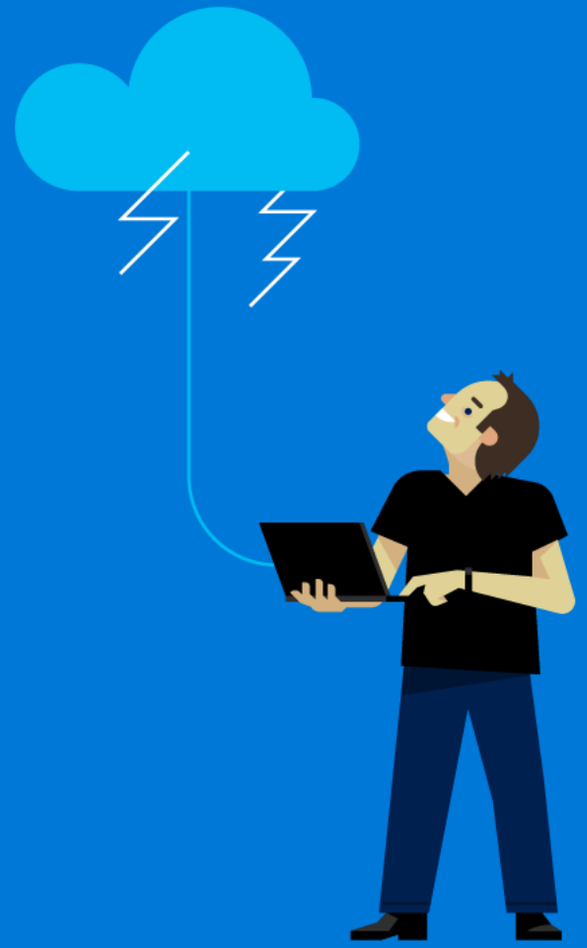
Switch on and off

FeatureToggle
Version 4.0.2



Demo: **Feature toggles**

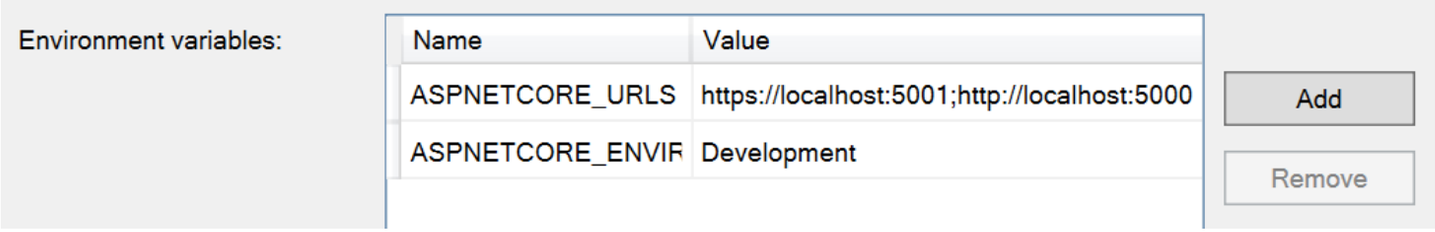
Defining and applying features



Secure it

Use HTTPS protocol

Listeners for HTTP and HTTPS by default in ASP.NET 2.1
Consider redirecting HTTP to HTTPS



Register server certificate for local development with HTTPS

Enable HSTS for production

Instructs browsers to avoid HTTP traffic on specified URL
Make sure everything is set up correctly before enabling this

Don't expose errors in production

Exclude exception details

Code snippets

```
app.UseHttpsRedirection();
```

```
dotnet install tool dotnet-dev-certs -g --version 2.1.0-preview1-final

dotnet dev-certs https --trust
```

```
if (env.IsProduction())
{
  app.UseHsts();
}
```

Hiding secrets in Azure Key Vault

Getting access to the vault

Bootstrap in Startup.cs when building Configuration

Requires:

- Application registration in Azure AD
- Service Principal with Get and List rights

If possible, use **Managed Service Identity** for Azure App Service

Microsoft.Azure.Services.AppAuthentication
Version 4.0.2

Key and secret names

Simple names work OOTB

Section names with double dash --

User secrets for local deve

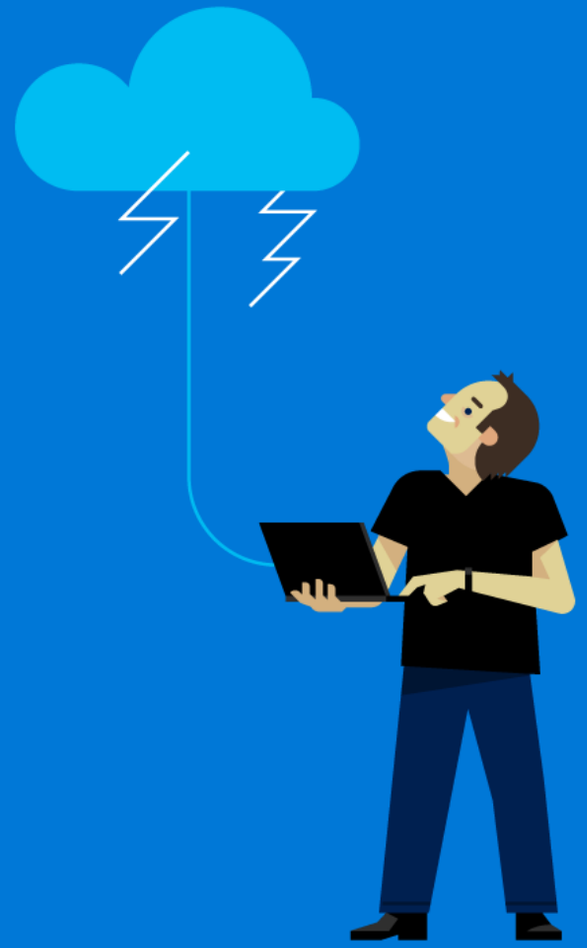
Secrets.json file, excluded from repository

GUID added to .csproj file

NAME
ApplicationInsights--InstrumentationKey
SteamApiOptions--DeveloperApiKey

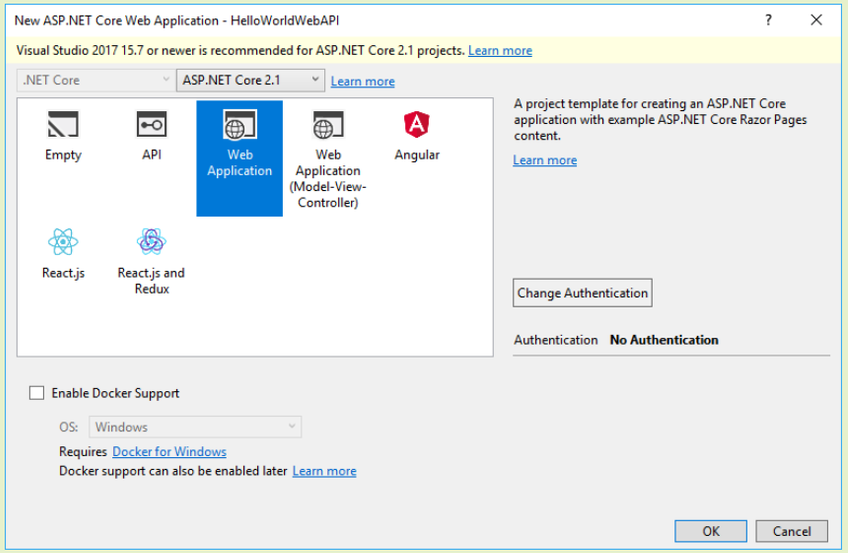
Demo: **Security features**

HTTPS
Key Vault and user secrets

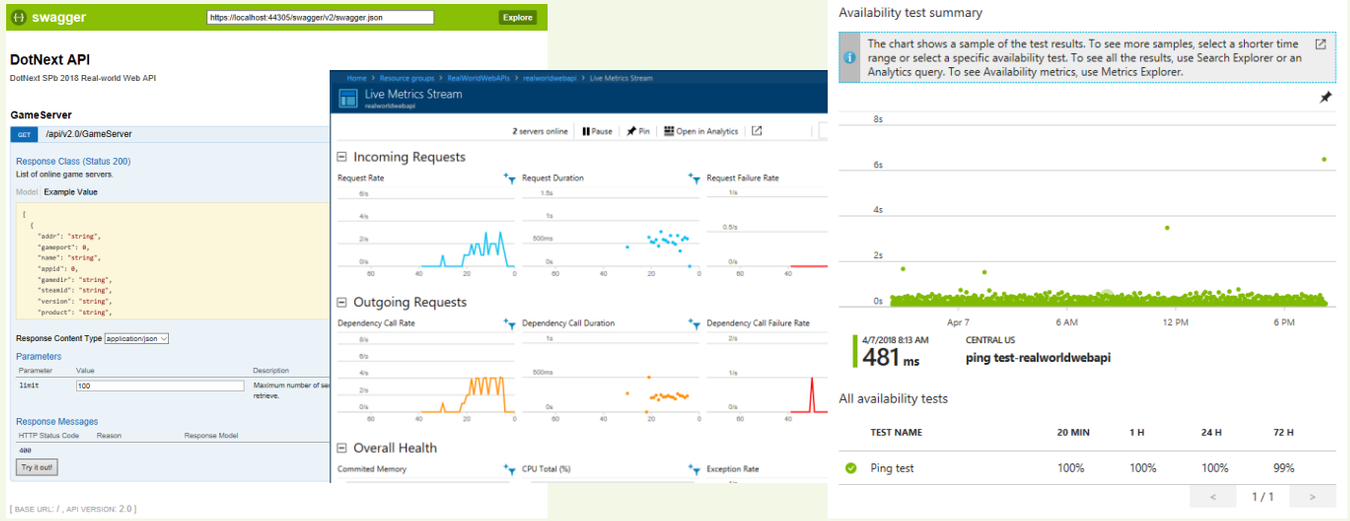


Looking back at our journey

to a modern cloud-ready Web API



File, New Project...



OpenAPI
Swagger
Polly

Logging
Versioning
HTTPS & HSTS

Health diagnostics
App Insights
TestHost

Questions and Answers

Maybe later?

@alexthissen
athissen@xpirit.com

Resources

Microsoft API guidelines https://github.com/Microsoft/api-guidelines

OpenAPI Initiative https://www.openapis.org

ASP.NET Core 2.1 Roadmap

https://blogs.msdn.microsoft.com/webdev/2018/02/02/asp-net-core-2-1-roadmap/

Dealing with wrong input

Better handling failed model validation

Follows RFC408 specification


```
services.Configure<ApiBehaviorOptions>(options => {
    options.InvalidModelStateResponseFactory = context => {
      // …
      return new BadRequestObjectResult(problemDetails) {
            ContentTypes = {
           "application/problem+json",
           "application/problem+xml" }
        };
    };
```