ASP.NET Core Architecture overview

David Fowler

Software Architect

What is ASP.NET Core?

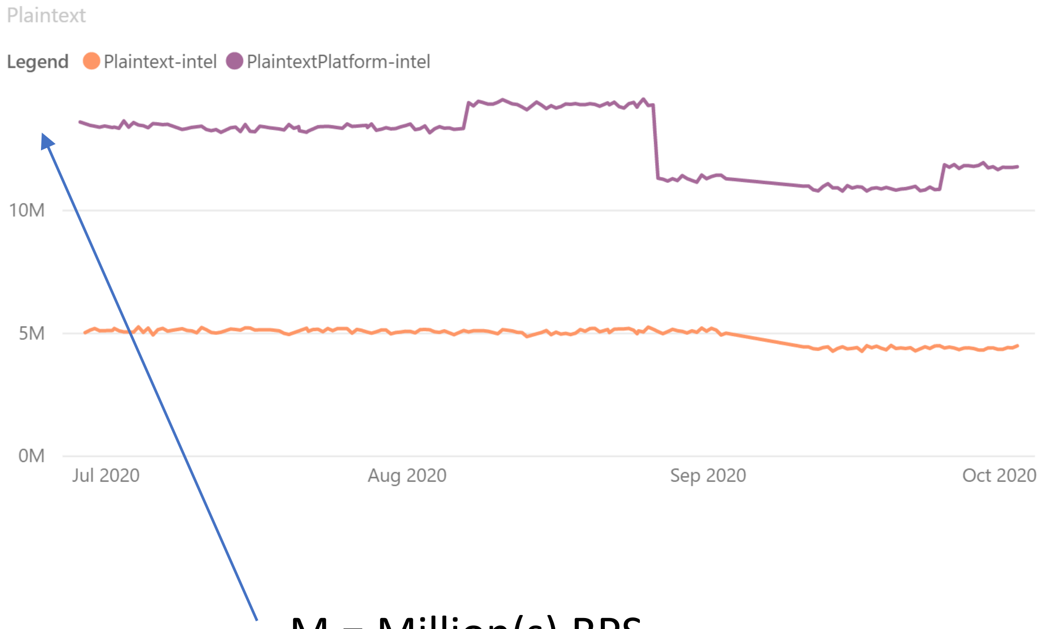
- Cross platform web platform from the .NET team
- Similar concepts as previous versions of ASP.NET but not binary compatible
- Can build a wide range of application types
  - REST APIs or gRPC services
  - Single Page Applications with Blazor
  - Server Rendered web applications with Razor Pages
  - Real time web applications with SignalR

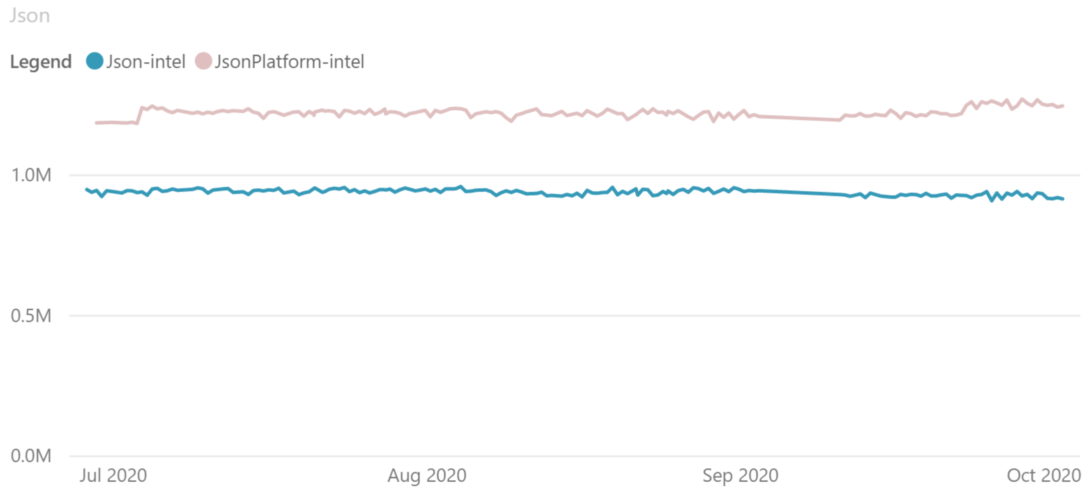
ASP.NET Core Design principles

- Various components of the stack should be independently usable
- Only pay for things you that you use
- Performance is paramount
- Code over configuration
- Composition via dependency injection
- Extensibility points used by the framework should be usable by anyone
- No statics, it should be possible to run multiple applications in the same process and not conflict.

ASP.NET Core is fast

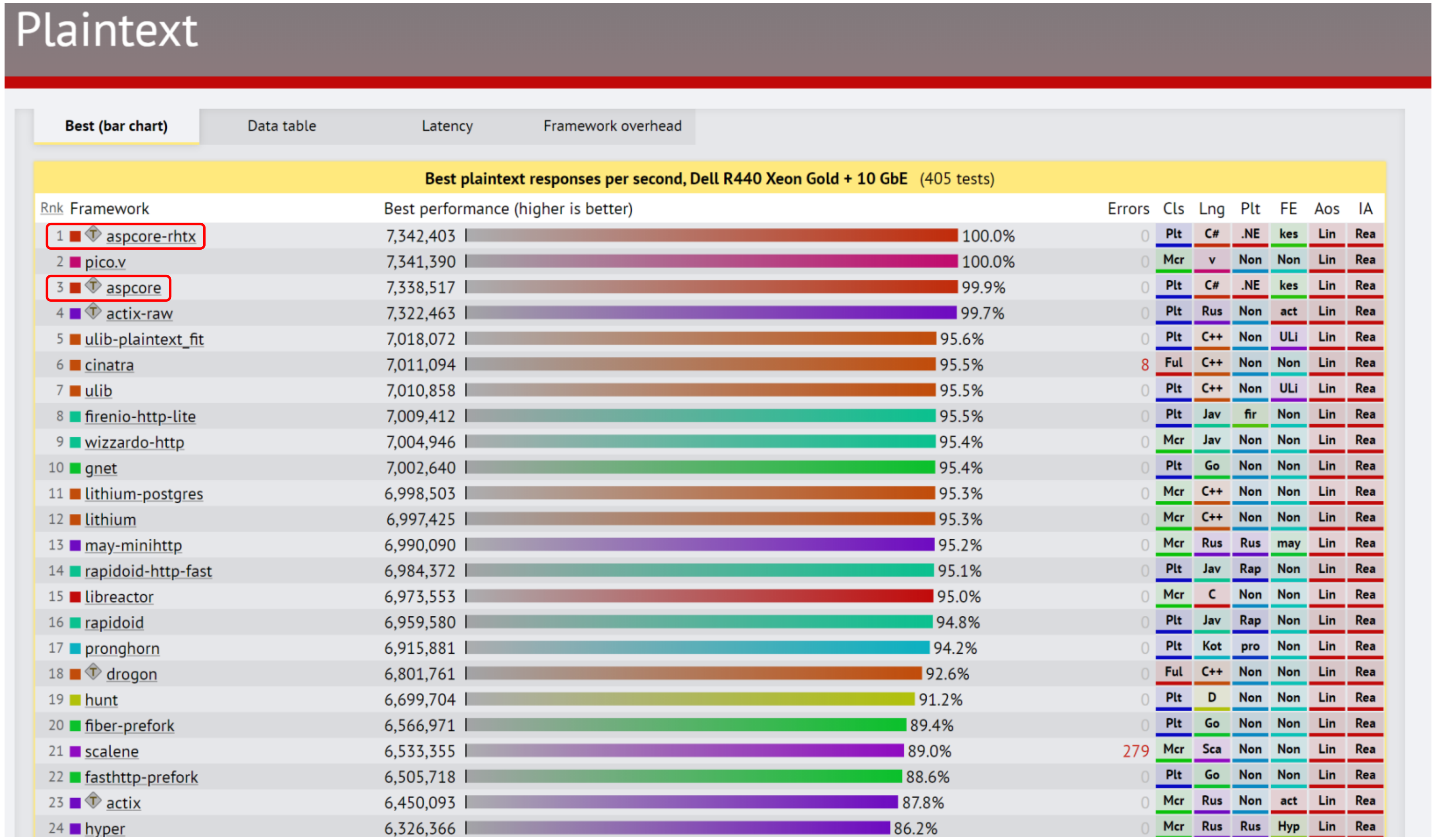
http://aka.ms/aspnet/benchmarks





M = Million(s) RPS

ASP.NET Core: TechEmpower benchmarks



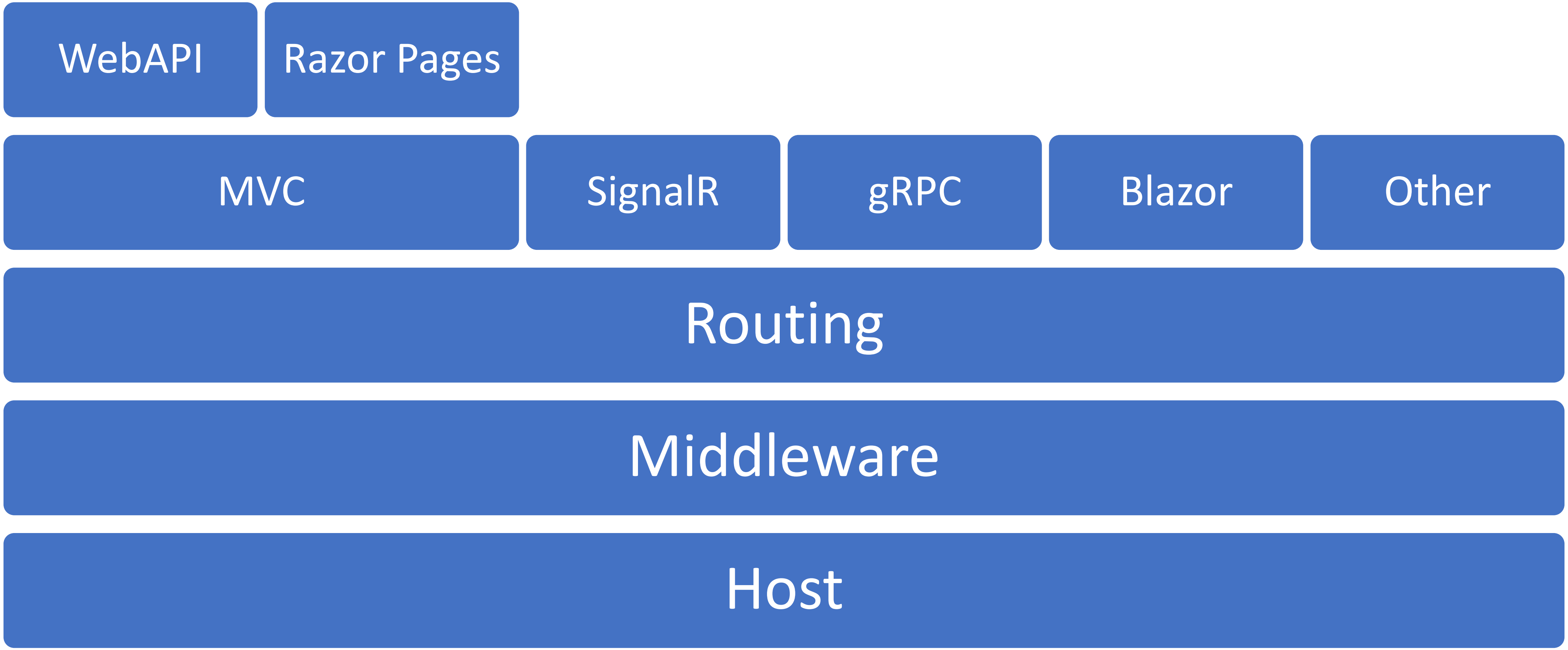
Plaintext

Best (bar chart) | Data table | Latency | Framework overhead

Best plaintext responses per second, Dell R440 Xeon Gold + 10 GbE (405 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	aspcore-rhtx	7,342,403	100.0%	0	Plt	C#	.NE	kes	Lin	Rea
2	pico.v	7,341,390	100.0%	0	Mcr	v	Non	Non	Lin	Rea
3	aspcore	7,338,517	99.9%	0	Plt	C#	.NE	kes	Lin	Rea
4	actix-raw	7,322,463	99.7%	0	Plt	Rus	Non	act	Lin	Rea
5	ulib-plaintext_fit	7,018,072	95.6%	0	Plt	C++	Non	ULi	Lin	Rea
6	cinatra	7,011,094	95.5%	8	Ful	C++	Non	Non	Lin	Rea
7	ulib	7,010,858	95.5%	0	Plt	C++	Non	ULi	Lin	Rea
8	firenio-http-lite	7,009,412	95.5%	0	Plt	Jav	fir	Non	Lin	Rea
9	wizzardo-http	7,004,946	95.4%	0	Mcr	Jav	Non	Non	Lin	Rea
10	gnet	7,002,640	95.4%	0	Plt	Go	Non	Non	Lin	Rea
11	lithium-postgres	6,998,503	95.3%	0	Mcr	C++	Non	Non	Lin	Rea
12	lithium	6,997,425	95.3%	0	Mcr	C++	Non	Non	Lin	Rea
13	may-minihttp	6,990,090	95.2%	0	Mcr	Rus	Rus	may	Lin	Rea
14	rapidoid-http-fast	6,984,372	95.1%	0	Plt	Jav	Rap	Non	Lin	Rea
15	libreactor	6,973,553	95.0%	0	Mcr	C	Non	Non	Lin	Rea
16	rapidoid	6,959,580	94.8%	0	Plt	Jav	Rap	Non	Lin	Rea
17	pronghorn	6,915,881	94.2%	0	Plt	Kot	pro	Non	Lin	Rea
18	drogon	6,801,761	92.6%	0	Ful	C++	Non	Non	Lin	Rea
19	hunt	6,699,704	91.2%	0	Plt	D	Non	Non	Lin	Rea
20	fiber-prefork	6,566,971	89.4%	0	Plt	Go	Non	Non	Lin	Rea
21	scalene	6,533,355	89.0%	279	Mcr	Sca	Non	Non	Lin	Rea
22	fasthttp-prefork	6,505,718	88.6%	0	Plt	Go	Non	Non	Lin	Rea
23	actix	6,450,093	87.8%	0	Mcr	Rus	Non	act	Lin	Rea
24	hyper	6,326,366	86.2%	0	Mcr	Rus	Rus	Hyp	Lin	Rea

ASP.NET Core Architecture



WebAPI

Razor Pages

MVC

SignalR

gRPC

Blazor

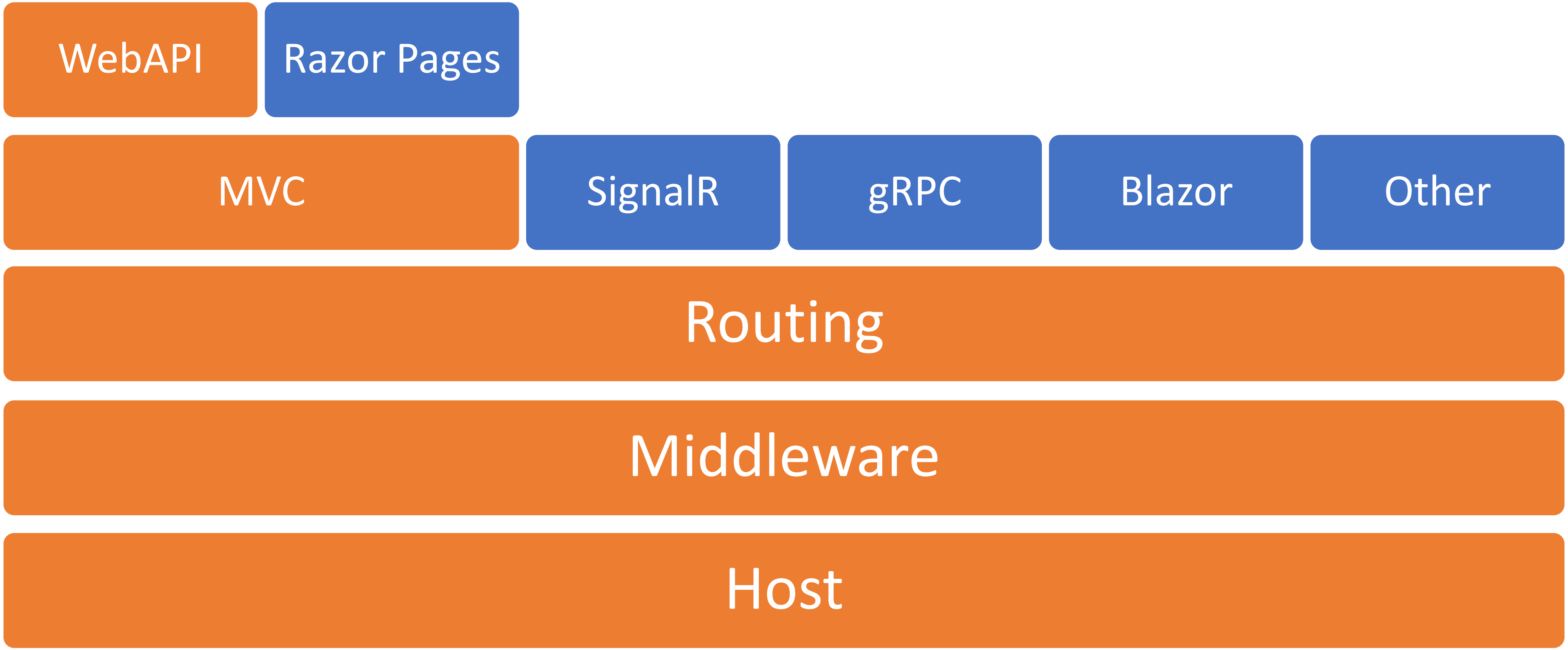
Other

Routing

Middleware

Host

The focus of this talk



WebAPI

Razor Pages

MVC

SignalR

gRPC

Blazor

Other

Routing

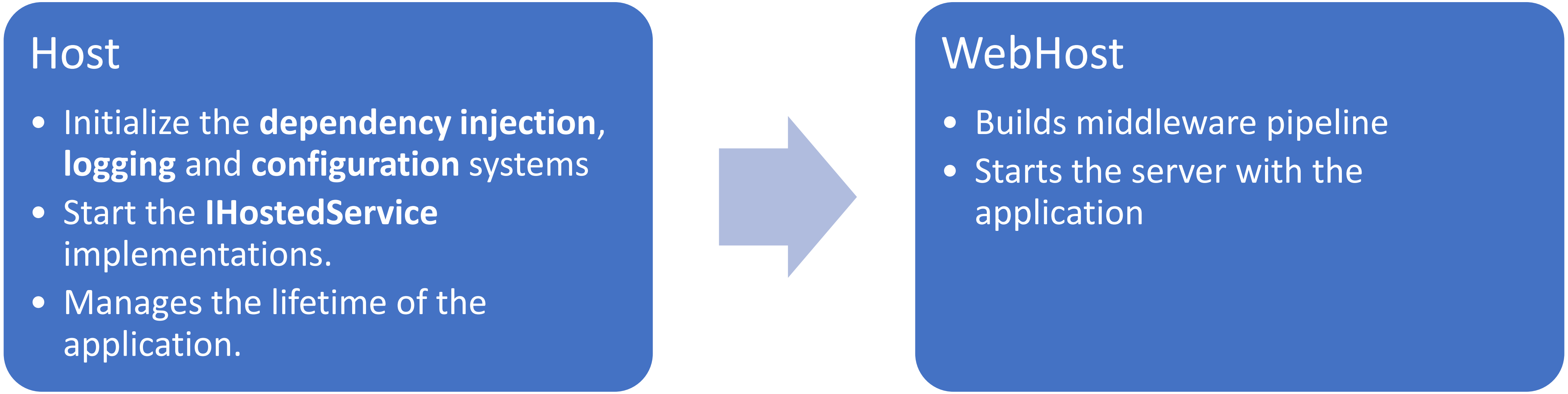
Middleware

Host

Overview

1. Application bootstrapping
2. Anatomy of a request

Application Bootstrapping



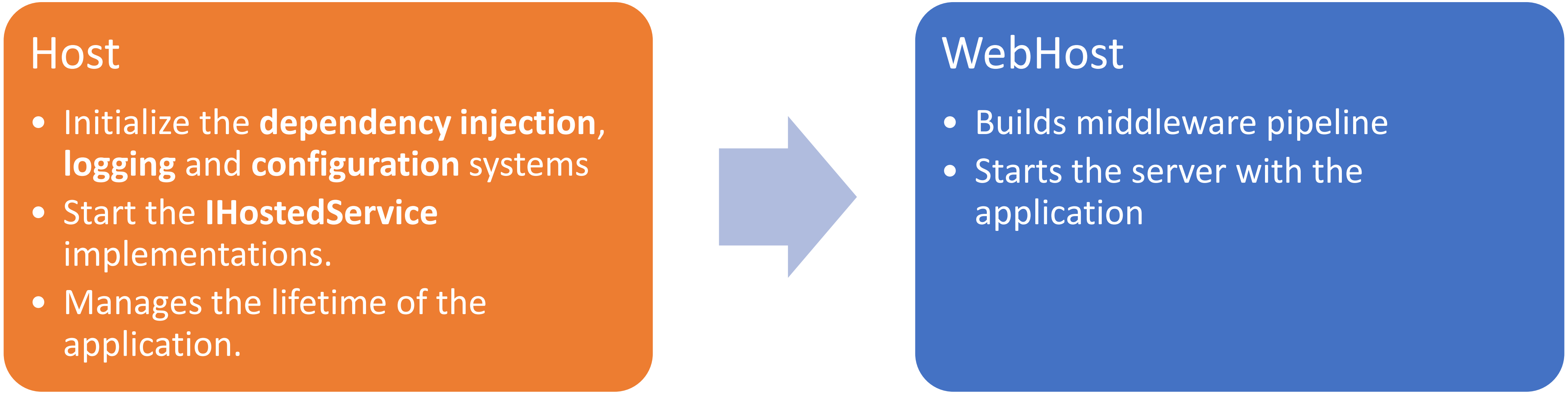
Host

- Initialize the **dependency injection**, **logging** and **configuration** systems
- Start the **IHostedService** implementations.
- Manages the lifetime of the application.

WebHost

- Builds middleware pipeline
- Starts the server with the application

Application Bootstrapping: Host



Host

- Initialize the **dependency injection**, **logging** and **configuration** systems
- Start the **IHostedService** implementations.
- Manages the lifetime of the application.

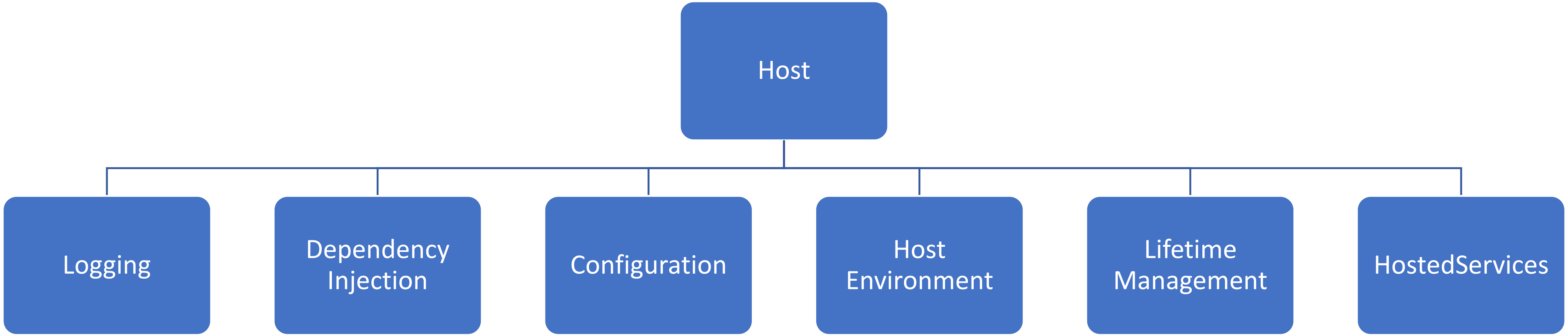
WebHost

- Builds middleware pipeline
- Starts the server with the application

Host: Microsoft.Extensions.*

- Responsible for bootstrapping the dependency injection, logging and configuration systems.
- Abstracts how the underlying platform manages lifetime for startup and shutdown (e.g. windows services, cloud services)
- Provides abstractions for getting environment information.
- Notifies hosted services on start up and shutdown.

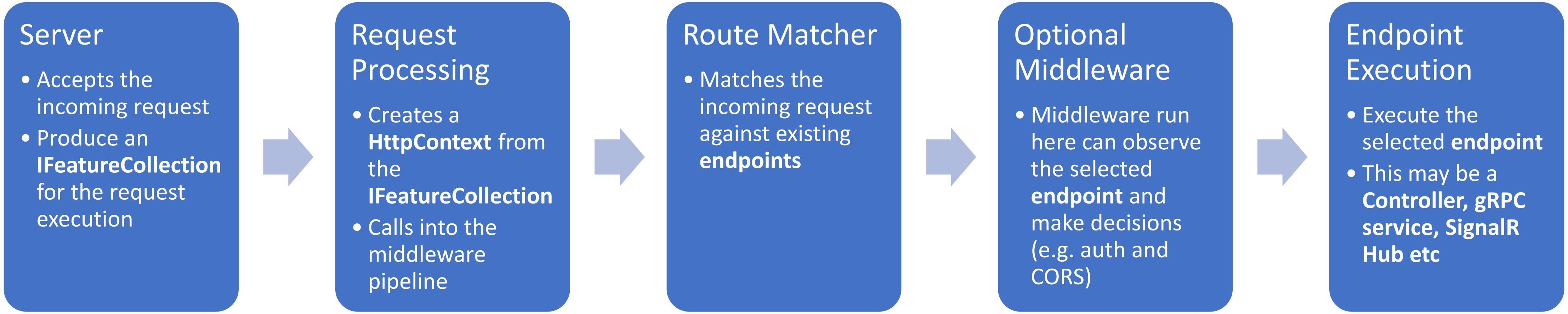
Host Architecture: Microsoft.Extensions.*



Microsoft.Extensions.* Design principles

- Decoupled from ASP.NET Core
- Built with dependency injection in mind
- netstandard2.0 compatible for the widest adoption
- Explicitly designed around provider model to allow extensibility (e.g. configuration, logging)

Anatomy of a request



Server

- Accepts the incoming request
- Produce an **IFeatureCollection** for the request execution

Request Processing

- Creates a **HttpContext** from the **IFeatureCollection**
- Calls into the middleware pipeline

Route Matcher

- Matches the incoming request against existing **endpoints**

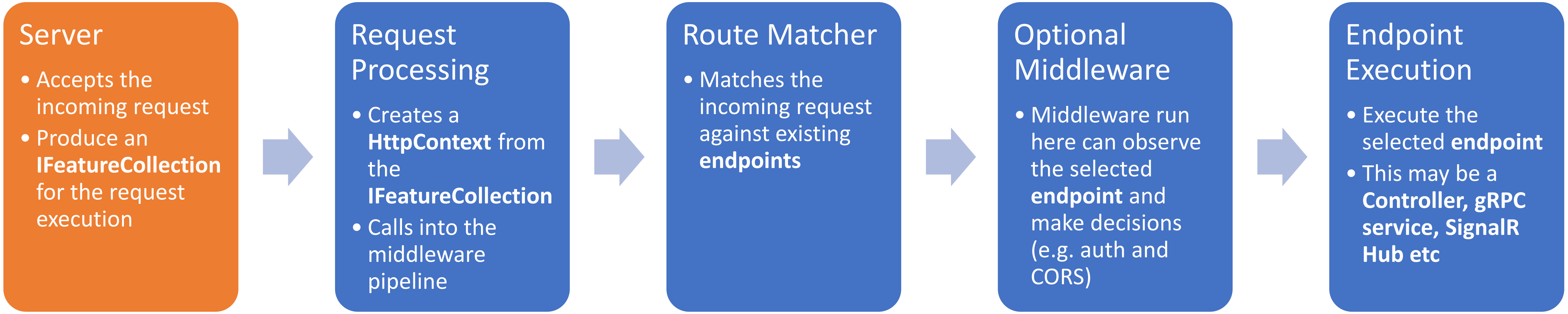
Optional Middleware

- Middleware run here can observe the selected **endpoint** and make decisions (e.g. auth and CORS)

Endpoint Execution

- Execute the selected **endpoint**
- This may be a **Controller, gRPC service, SignalR Hub etc**

Anatomy of a request: Server



Server

- Accepts the incoming request
- Produce an **IFeatureCollection** for the request execution

Request Processing

- Creates a **HttpContext** from the **IFeatureCollection**
- Calls into the middleware pipeline

Route Matcher

- Matches the incoming request against existing **endpoints**

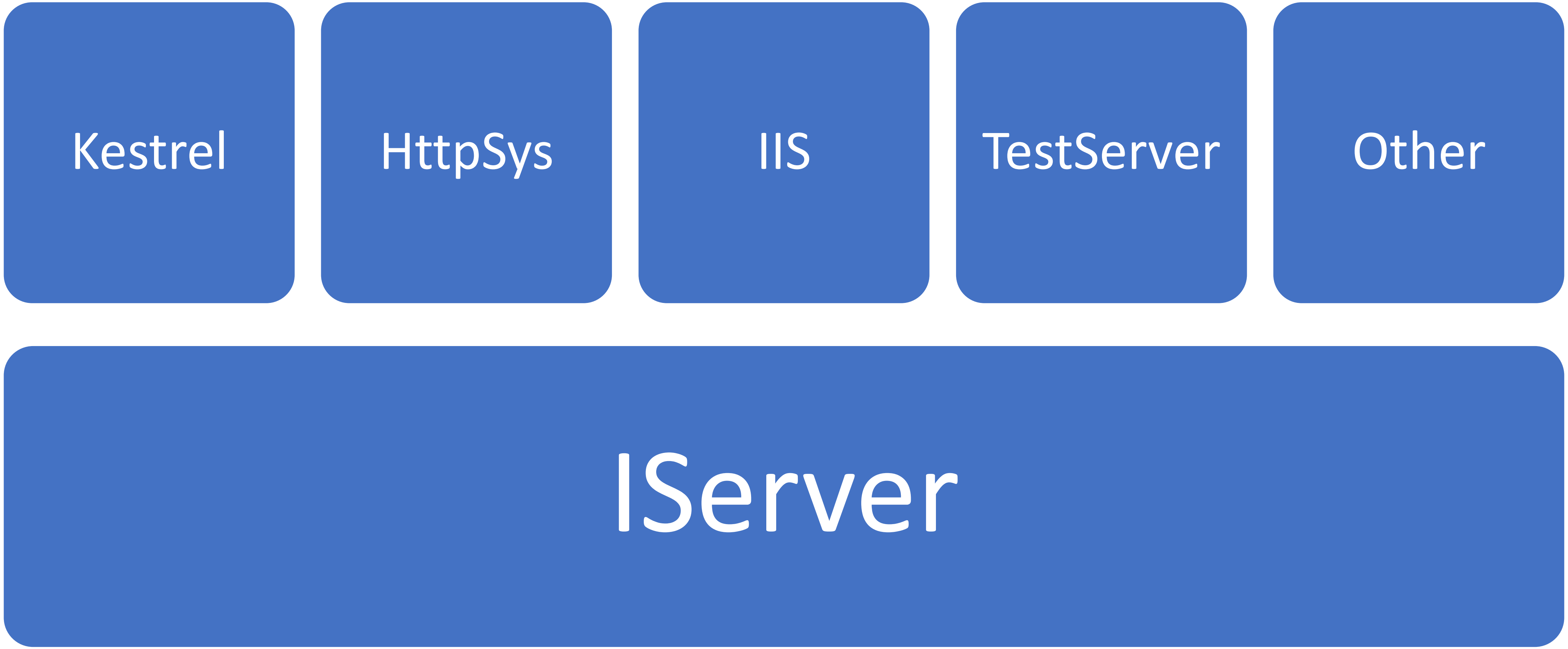
Optional Middleware

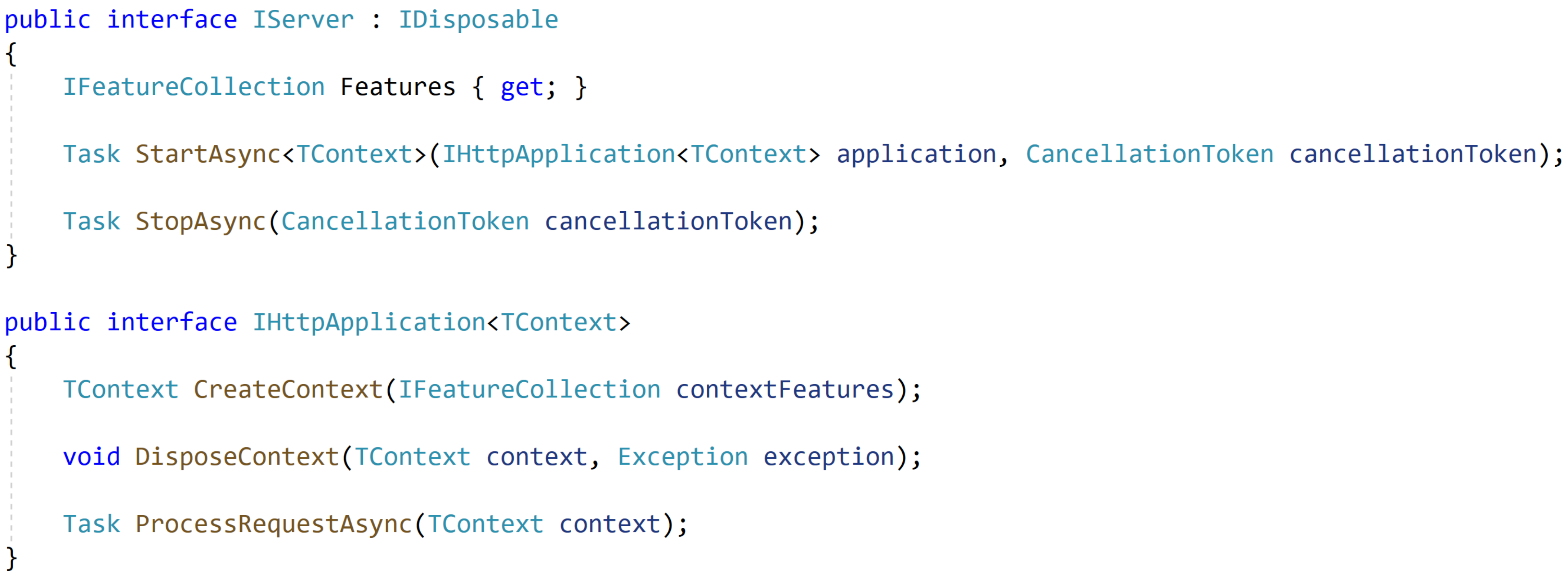
- Middleware run here can observe the selected **endpoint** and make decisions (e.g. auth and CORS)

Endpoint Execution

- Execute the selected **endpoint**
- This may be a **Controller, gRPC service, SignalR Hub etc**

Server Architecture

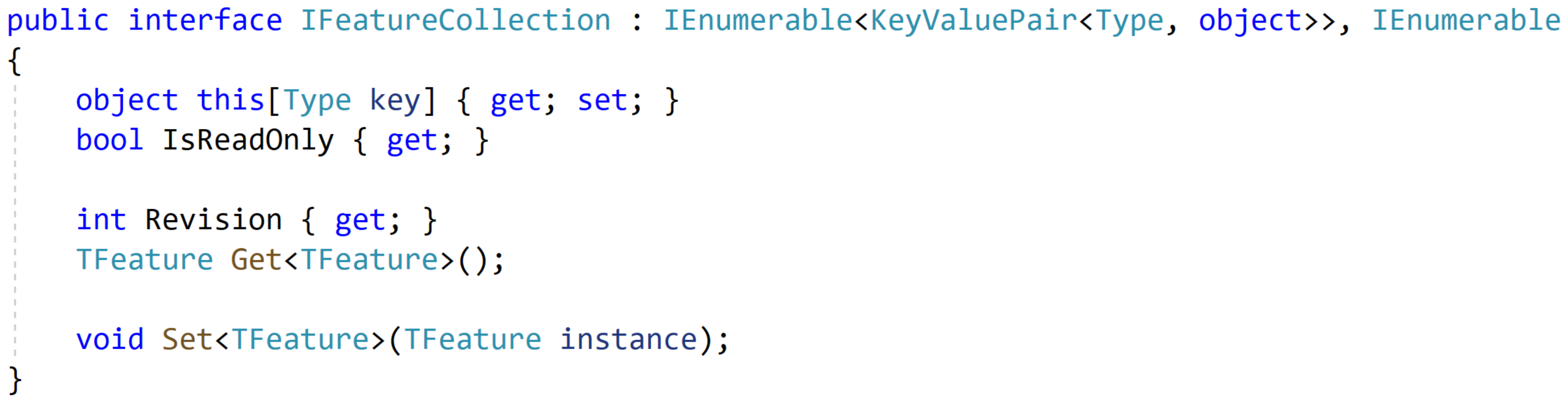




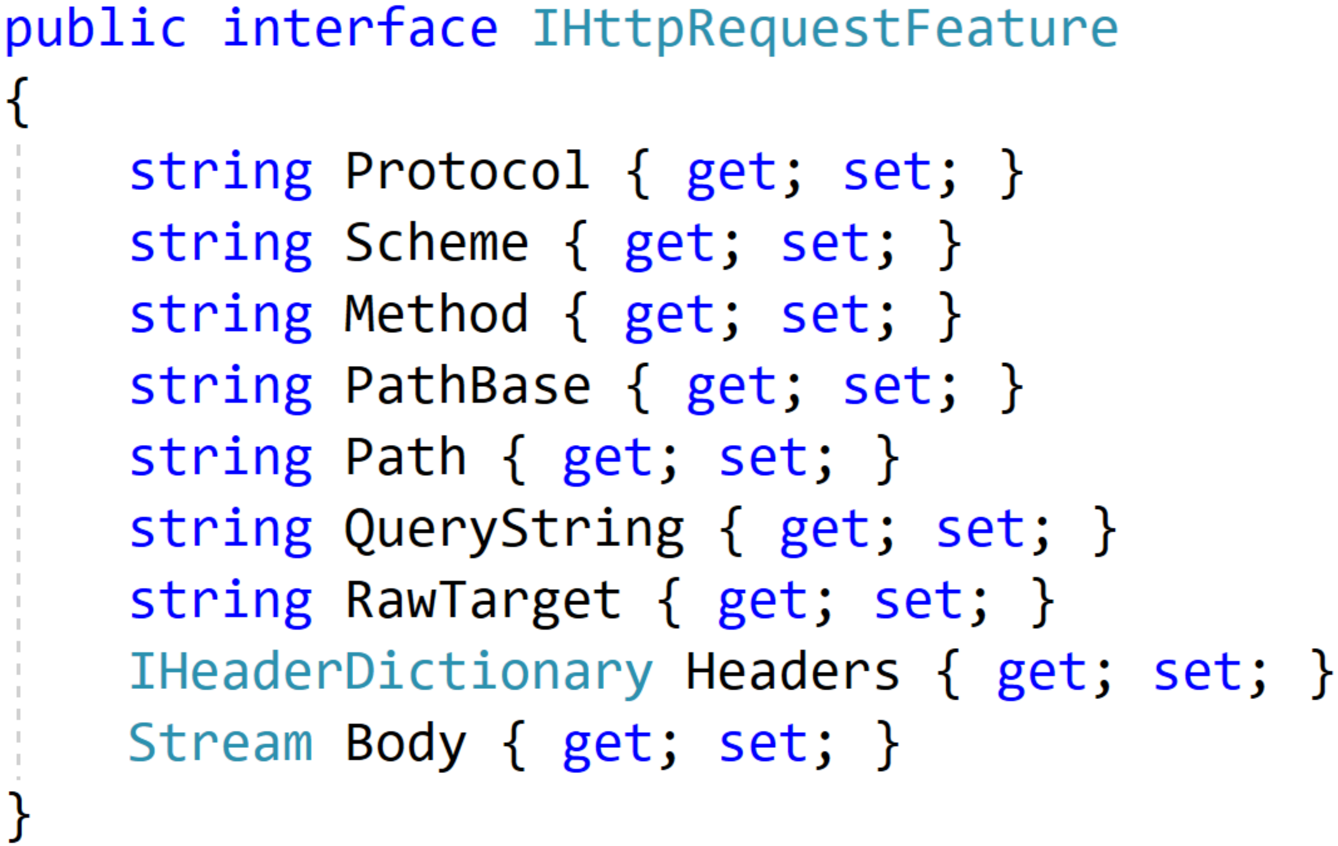
```
public interface IServer : IDisposable
{
    IFeatureCollection Features { get; }

    Task StartAsync<TContext>(IHttpApplication<TContext> application, CancellationToken cancellationToken);

    Task StopAsync(CancellationToken cancellationToken);
}

public interface IHttpApplication<TContext>
{
    TContext CreateContext(IFeatureCollection contextFeatures);

    void DisposeContext(TContext context, Exception exception);

    Task ProcessRequestAsync(TContext context);
}
```

Server

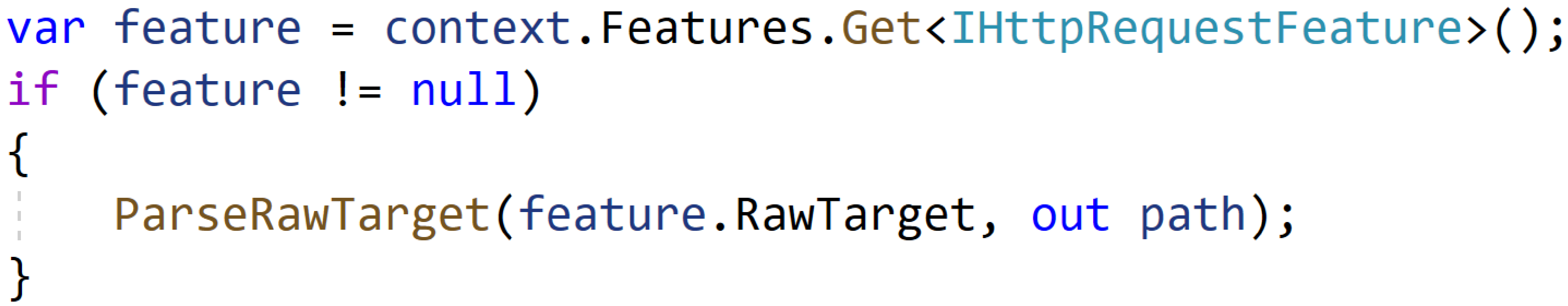
- Listen for incoming requests
- Responsible for the core request handling logic
- Produces an IFeatureCollection for request execution
  - ASP.NET Core has a minimum set of features it expects all IServers to implement.
  - May expose additional server specific functionality (e.g. server variables in IIS)
- Call into the IHttpApplication registered with the server when requests arrive



```csharp
public interface IFeatureCollection : IEnumerable<KeyValuePair<Type, object>>, IEnumerable
{
    object this[Type key] { get; set; }
    bool IsReadOnly { get; }

    int Revision { get; }
    TFeature Get<TFeature>();

    void Set<TFeature>(TFeature instance);
}
```



```
public interface IHttpRequestFeature
{
    string Protocol { get; set; }
    string Scheme { get; set; }
    string Method { get; set; }
    string PathBase { get; set; }
    string Path { get; set; }
    string QueryString { get; set; }
    string RawTarget { get; set; }
    IHeaderDictionary Headers { get; set; }
    Stream Body { get; set; }
}
```

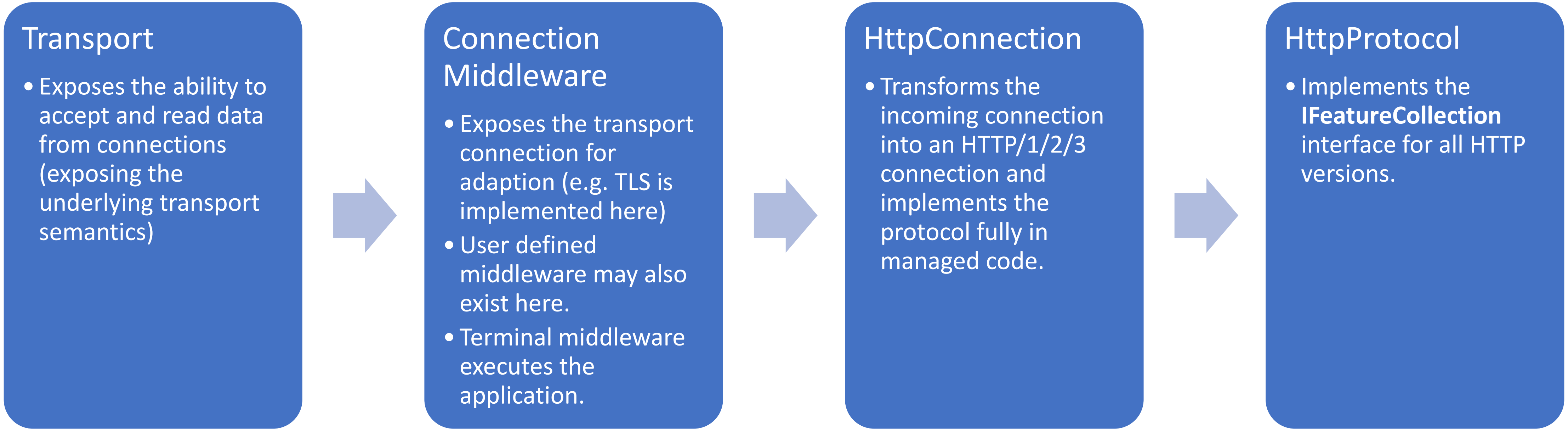


```
var feature = context.Features.Get<IHttpRequestFeature>();
if (feature != null)
{
    ParseRawTarget(feature.RawTarget, out path);
}
```

Server: Kestrel

- Cross platform
- Written entirely in managed code
- Extremely optimized
- Supports HTTP/1, HTTP/2, HTTP/3 (preview)
- Pluggable transports (see project bedrock)
    - Default transport uses Sockets
- Other protocols are possible on top of transport abstraction

Server: Kestrel Architecture



Transport

- Exposes the ability to accept and read data from connections (exposing the underlying transport semantics)

Connection Middleware

- Exposes the transport connection for adaption (e.g. TLS is implemented here)
- User defined middleware may also exist here.
- Terminal middleware executes the application.

HttpConnection

- Transforms the incoming connection into an HTTP/1/2/3 connection and implements the protocol fully in managed code.

HttpProtocol

- Implements the **IFeatureCollection** interface for all HTTP versions.

Server: Kestrel Optimizations

- Buffering pooling at all the layers
- Uses the pinned object heap to reduce fragmentation
- Non-allocating HTTP parsers using Span
- Headers are dictionaries optimized for known header access
  - We never allocate known header keys
  - We can reuse header values
- Pooled HttpContext objects and associated state across requests
- Low level knobs exposed to optimize threading

Server: HttpSys (Windows Only)

- Managed wrapper over HTTP.sys
- Supports advanced HTTP.sys features
    - Port sharing
    - Request queue creation
    - Kernel caching
    - Sendfile
    - Windows Auth (NTLM, Kerberos)
    - Request queue delegation

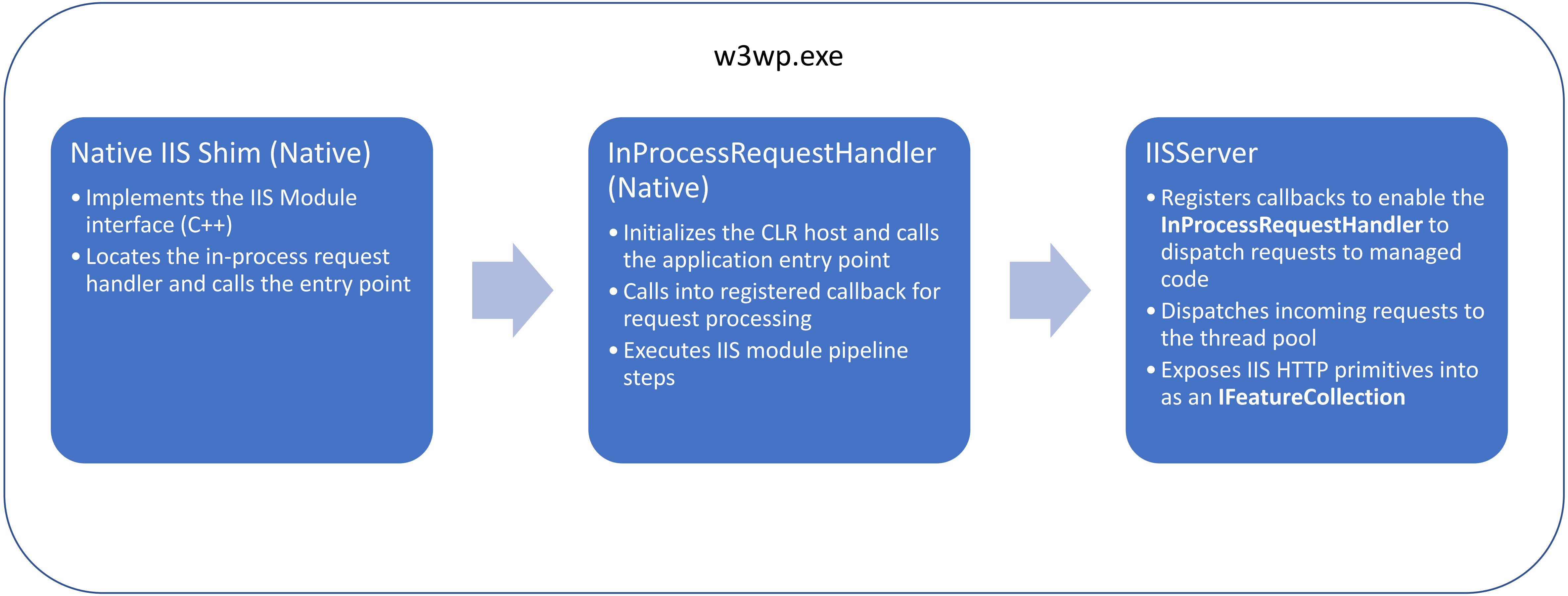
Server: HttpSys Architecture

- Dequeue request from HTTP.sys request queue
- Dispatches to the request to the ThreadPool
- Wraps the HTTP API primitives in an **IFeatureCollection**

Server: IIS (Windows Only)

- Managed wrapper around native IIS Module
- Runs in 2 modes
  - In process – Application code runs in the IIS worker process
    - Does not support running multiple applications in a single worker process
    - Does not support handling IIS module events
  - Out of process – Application code runs in a separate process
- Deployed as 2 separate components
  - A native shim installed globally
  - A request handler that ships with ASP.NET or the application
- Not built into Windows (it’s a separate installer)

Server: IIS Architecture In-Process



w3wp.exe

Native IIS Shim (Native)

- Implements the IIS Module interface (C++)
- Locates the in-process request handler and calls the entry point

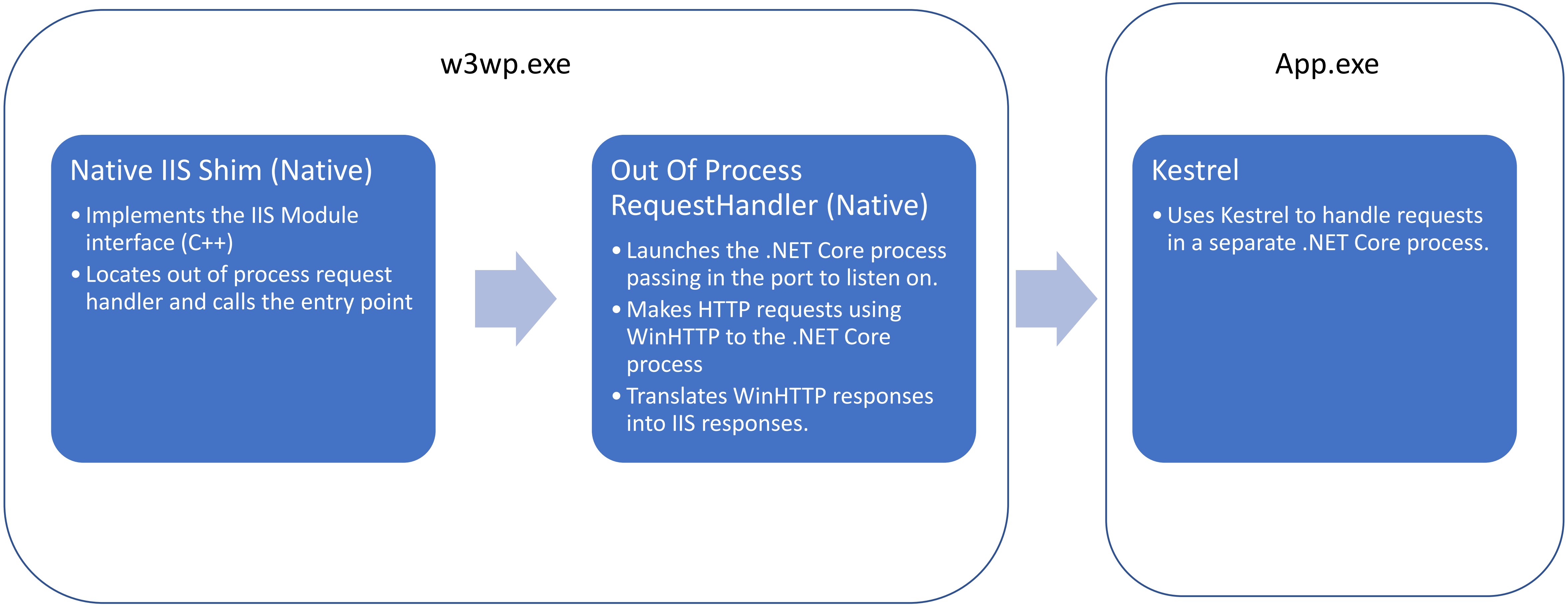
InProcessRequestHandler (Native)

- Initializes the CLR host and calls the application entry point
- Calls into registered callback for request processing
- Executes IIS module pipeline steps

IISServer

- Registers callbacks to enable the **InProcessRequestHandler** to dispatch requests to managed code
- Dispatches incoming requests to the thread pool
- Exposes IIS HTTP primitives into as an **IFeatureCollection**

Server: IIS Architecture Out-Of-Process



w3wp.exe

Native IIS Shim (Native)

- Implements the IIS Module interface (C++)
- Locates out of process request handler and calls the entry point

Out Of Process RequestHandler (Native)

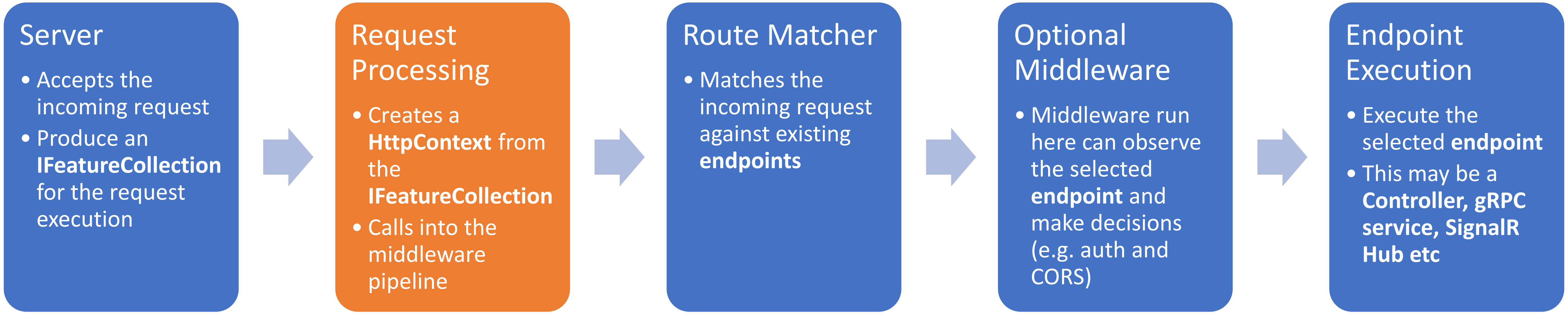
- Launches the .NET Core process passing in the port to listen on.
- Makes HTTP requests using WinHTTP to the .NET Core process
- Translates WinHTTP responses into IIS responses.

App.exe

Kestrel

- Uses Kestrel to handle requests in a separate .NET Core process.

Anatomy of a request: Request Processing



Server

- Accepts the incoming request
- Produce an **IFeatureCollection** for the request execution

Request Processing

- Creates a **HttpContext** from the **IFeatureCollection**
- Calls into the middleware pipeline

Route Matcher

- Matches the incoming request against existing **endpoints**

Optional Middleware

- Middleware run here can observe the selected **endpoint** and make decisions (e.g. auth and CORS)

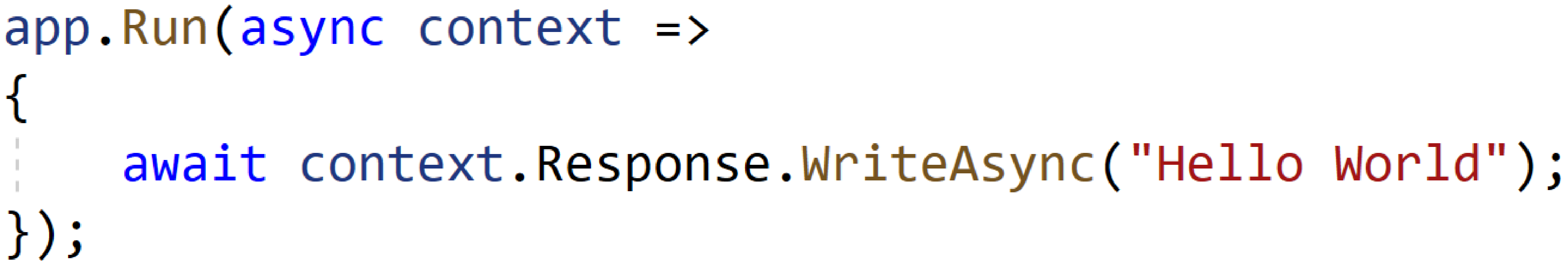
Endpoint Execution

- Execute the selected **endpoint**
- This may be a **Controller, gRPC service, SignalR Hub etc**

Request Processing

- Creates a HttpContext from the IFeatureCollection
- The HttpContext wraps the server’s IFeatureCollection and exposes a convenience layer on top
- Application code is written against this layer and is **server agnostic**
- Everything is asynchronous!

```
public delegate Task RequestDelegate(HttpContext context);
```

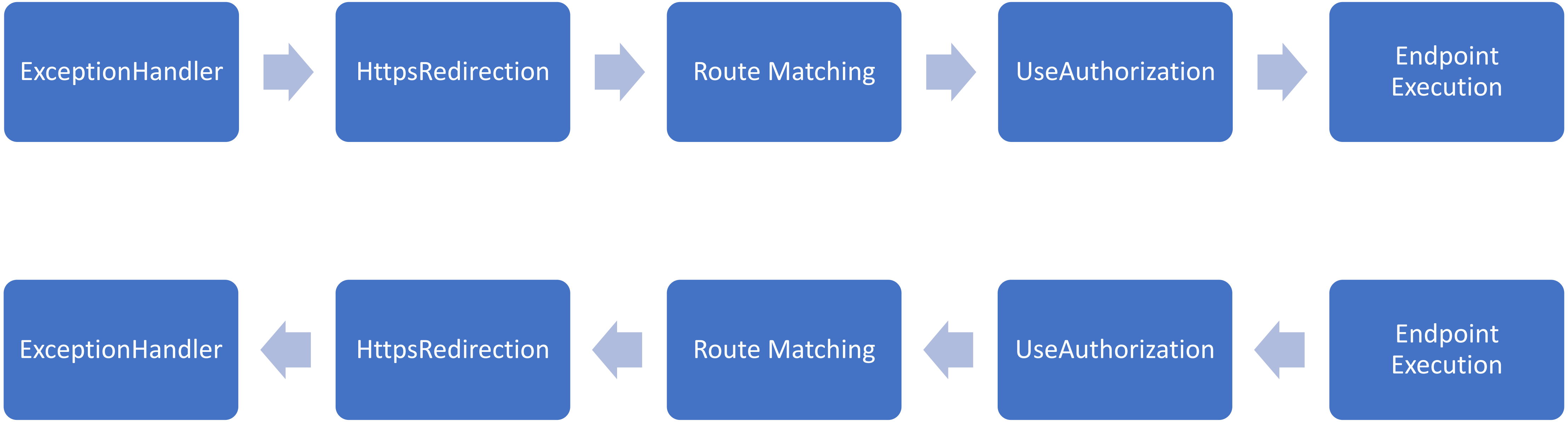


```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello World");
});
```

Middleware

- Central extensibility point of request processing.
- Execute cross cutting concerns that apply to either request and response.
- Russian doll pattern
- Exposes a wide variety of options for modifying the request and pipeline
    - Branching the pipeline
    - Short circuiting the incoming requests
    - Decorate state on the HttpContext
    - Wrapping the entire pipeline for exception handling
    - …

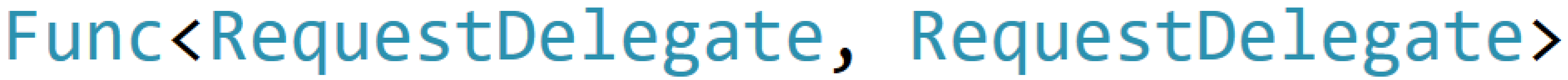
Request Processing: Middleware Architecture



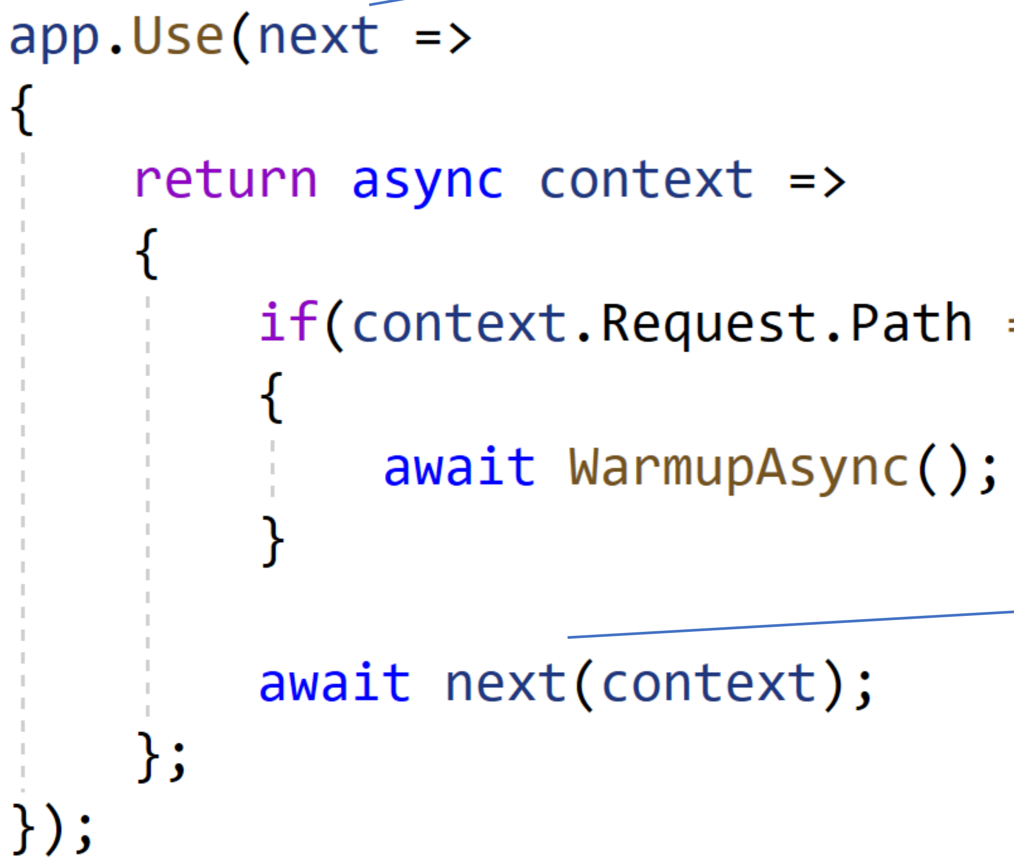
ExceptionHandler → HttpsRedirection → Route Matching → UseAuthorization → Endpoint Execution

ExceptionHandler ← HttpsRedirection ← Route Matching ← UseAuthorization ← Endpoint Execution

Middleware definition



```
Func<RequestDelegate, RequestDelegate>
```

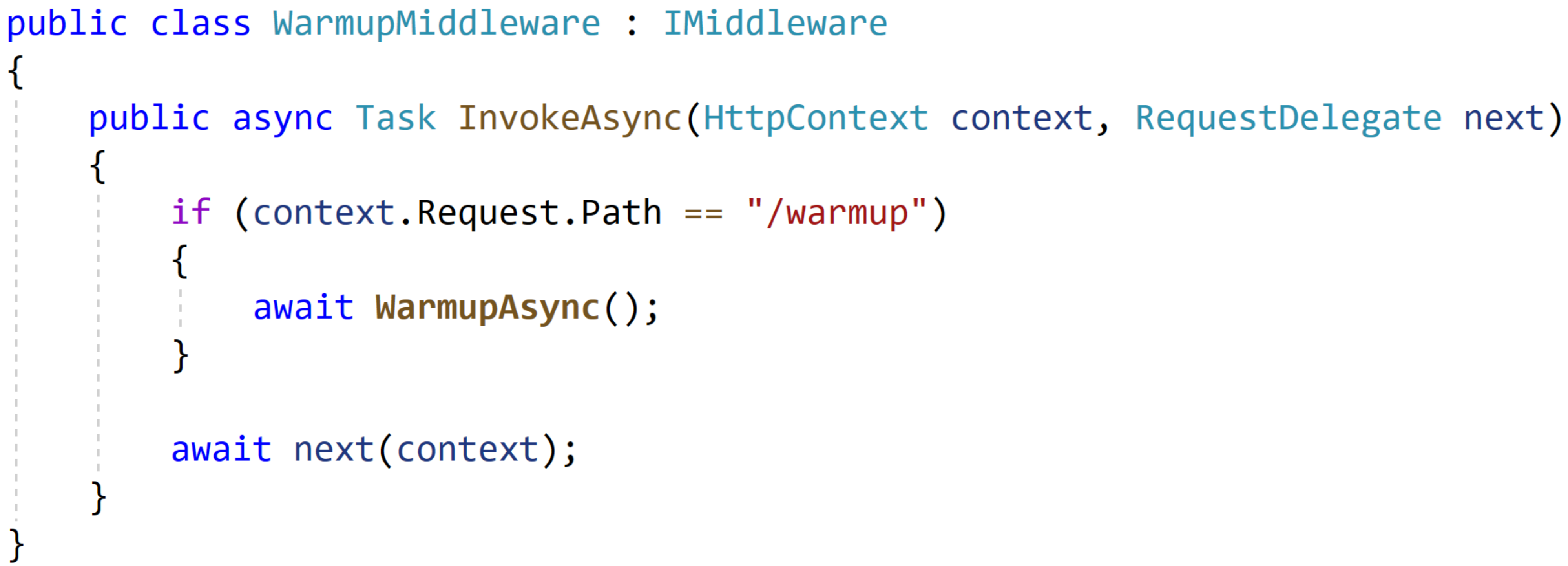


```
app.Use(next =>
{
    return async context =>
    {
        if(context.Request.Path == "/Warmup")
        {
            await WarmupAsync();
        }

        await next(context);
    };
});
```

Reference to the next middleware

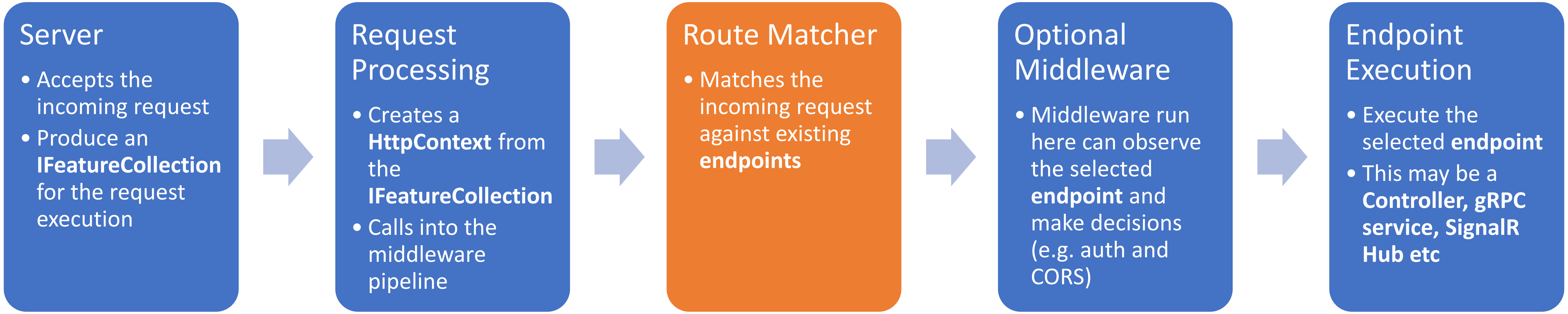
Calling the next middleware in the pipeline

Middleware interface



```
public class WarmupMiddleware : IMiddleware
{
    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        if (context.Request.Path == "/warmup")
        {
            await WarmupAsync();
        }

        await next(context);
    }
}
```

Anatomy of a request: Routing



Server

- Accepts the incoming request
- Produce an **IFeatureCollection** for the request execution

Request Processing

- Creates a **HttpContext** from the **IFeatureCollection**
- Calls into the middleware pipeline

Route Matcher

- Matches the incoming request against existing **endpoints**

Optional Middleware

- Middleware run here can observe the selected **endpoint** and make decisions (e.g. auth and CORS)

Endpoint Execution

- Execute the selected **endpoint**
- This may be a **Controller, gRPC service, SignalR Hub etc**

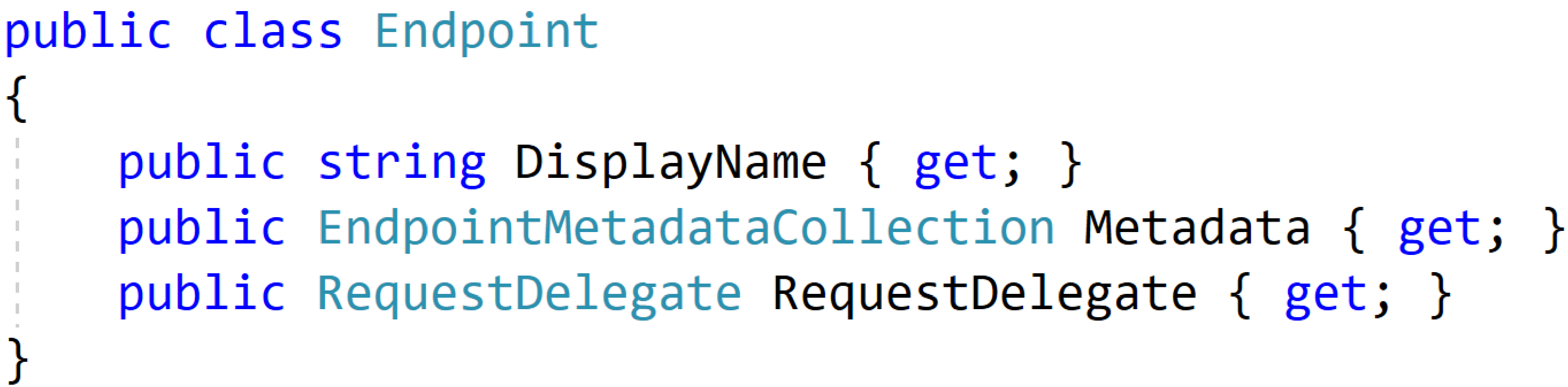
Routing

- Matches the incoming request against a set of **endpoints** and their criteria
- Supports Link/URL generation for registered routes
- Highly optimized route table and matching algorithm

Endpoints

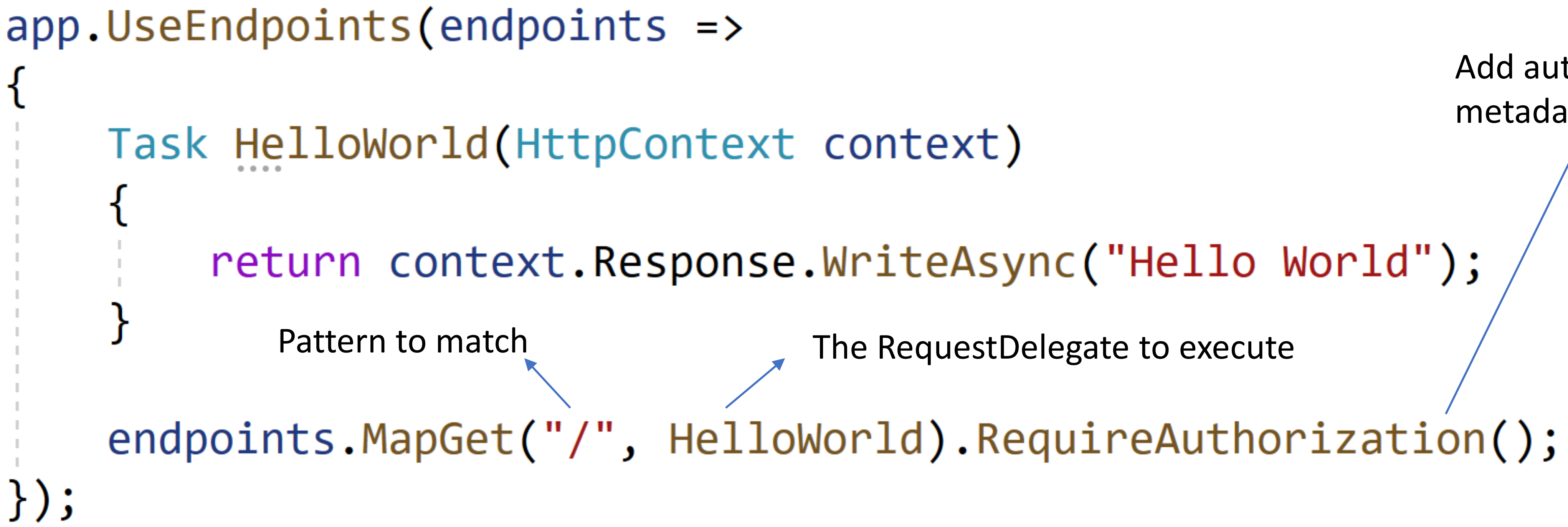
- A RequestDelegate to execute
- Metadata about the code to execute
  - For example, authorization metadata
- Decoupled from routing
- Middleware can be “endpoint aware”
  - CORS
  - Authorization

Endpoints



```
public class Endpoint
{
    public string DisplayName { get; }
    public EndpointMetadataCollection Metadata { get; }
    public RequestDelegate RequestDelegate { get; }
}
```

Endpoints



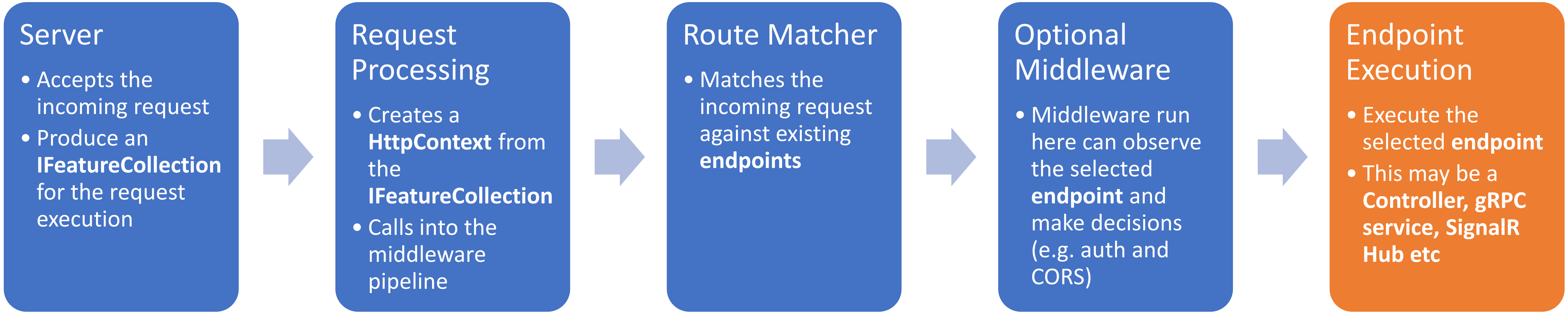
```
app.UseEndpoints(endpoints =>
{
    Task HelloWorld(HttpContext context)
    {
        return context.Response.WriteAsync("Hello World");
    }

    endpoints.MapGet("/", HelloWorld).RequireAuthorization();
});
```

Pattern to match

The RequestDelegate to execute

Add authorization metadata

Anatomy of a request: Endpoint Execution MVC



Server

- Accepts the incoming request
- Produce an **IFeatureCollection** for the request execution

Request Processing

- Creates a **HttpContext** from the **IFeatureCollection**
- Calls into the middleware pipeline

Route Matcher

- Matches the incoming request against existing **endpoints**

Optional Middleware

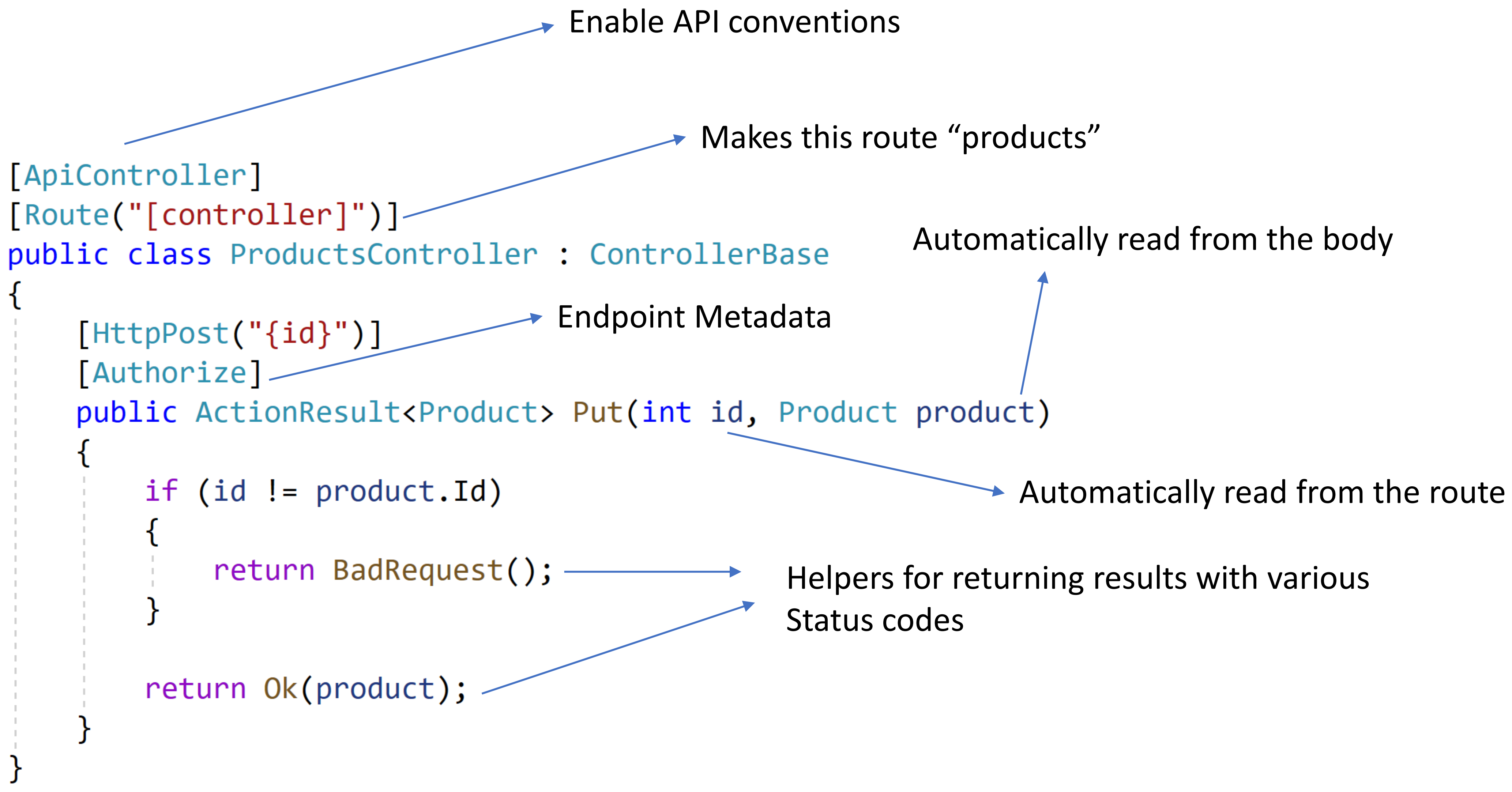
- Middleware run here can observe the selected **endpoint** and make decisions (e.g. auth and CORS)

Endpoint Execution

- Execute the selected **endpoint**
- This may be a **Controller, gRPC service, SignalR Hub etc**

MVC for APIs

- Declarative programming model that provides productivity features for writing APIs
  - **Model binding** – Convert incoming request objects into strongly typed models
  - **Model validation** – Makes sure the bound models are valid
  - **Formatters** – Read and write objects from/to the request/response
  - **Filters** – Run custom logic on code that runs before/after application logic
  - **Content negotiation**
  - **OpenAPI** support via Swashbuckle
- Built on top of routing



```
[ApiController]
[Route("[controller]")]
public class ProductsController : ControllerBase
{
    [HttpPost("{id}")]
    [Authorize]
    public ActionResult<Product> Put(int id, Product product)
    {
        if (id != product.Id)
        {
            return BadRequest();
        }

        return Ok(product);
    }
}
```

Enable API conventions

Makes this route “products”

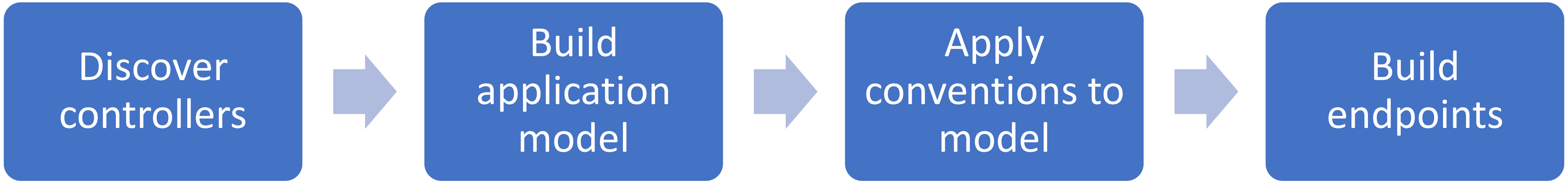
Automatically read from the body

Endpoint Metadata

Automatically read from the route

Helpers for returning results with various Status codes

MVC : Bootstrapping



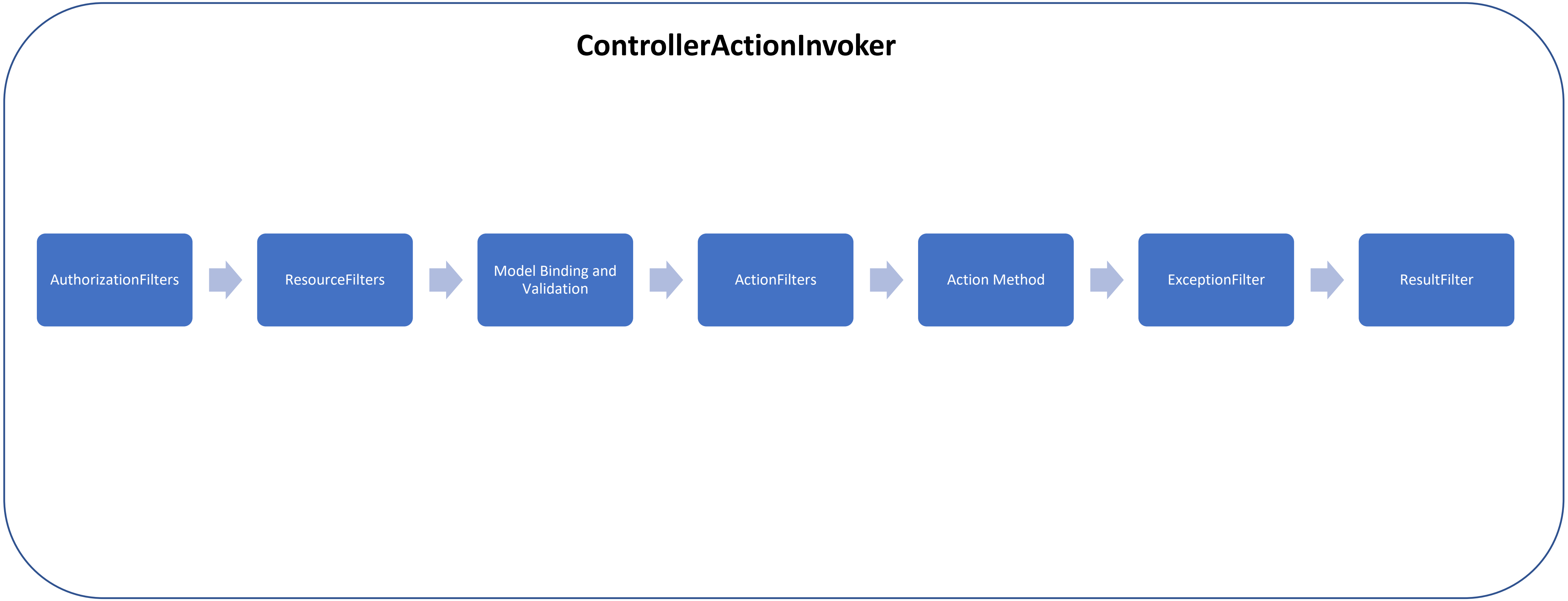
Discover controllers

Build application model

Apply conventions to model

Build endpoints

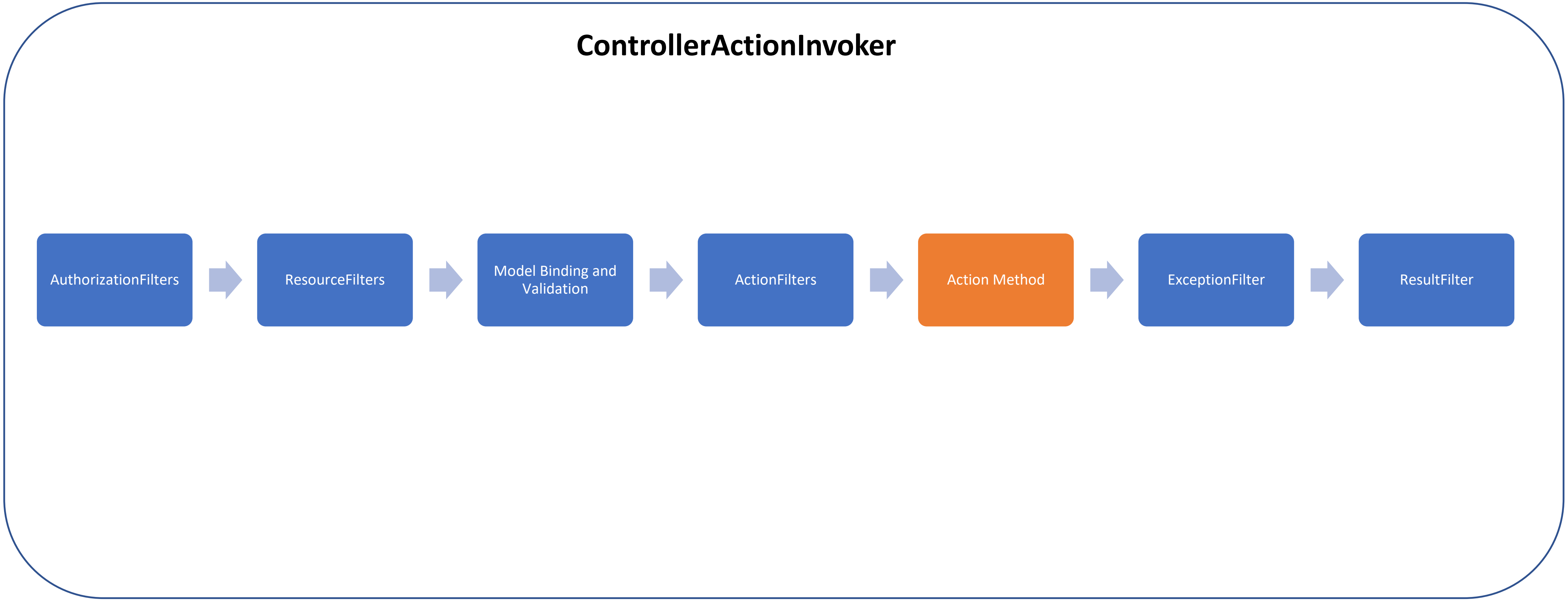
MVC : Request Processing



ControllerActionInvoker

AuthorizationFilters → ResourceFilters → Model Binding and Validation → ActionFilters → Action Method → ExceptionFilter → ResultFilter

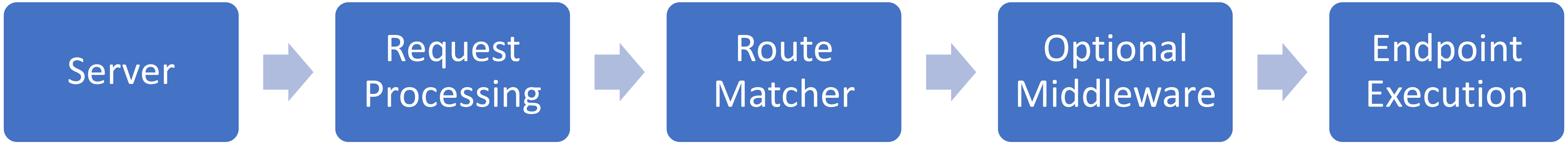
MVC : Request Processing



ControllerActionInvoker

AuthorizationFilters → ResourceFilters → Model Binding and Validation → ActionFilters → Action Method → ExceptionFilter → ResultFilter

Anatomy of a request: Review



Server

Request Processing

Route Matcher

Optional Middleware

Endpoint Execution

There’s more…

- ASP.NET Core is *huge*
- I couldn’t fit it all into this talk
- Hopefully, this gives you a good idea where to look for more details
- Read the source on https://github.com/dotnet/aspnetcore

Future: Houdini

- Project to make MVC’s disappear (hence the name)
  - Push productivity features into the core of the stack
- Make the jump from imperative routing to declarative MVC smaller
  - Improving the performance along the way

Questions?

Twitter: davidfowl