Building a RESTful API with ASP.NET Core Web API

Gill Cleeren

@gillcleeren

Covered topics

- Creating an API from scratch
- Entity Framework Core
- Content Negotiation
- Tooling
- Azure
- And much more!

Why do we need API's in the first place?

- Regular app is a synchronous web application
  - POST request goes off to the server
  - We wait
  - We wait
  - And wait some more
  - Browser will then render the HTML coming back from the server

Speed problem can be large

- A lot of data is travelling over the wire

Slow… Why?

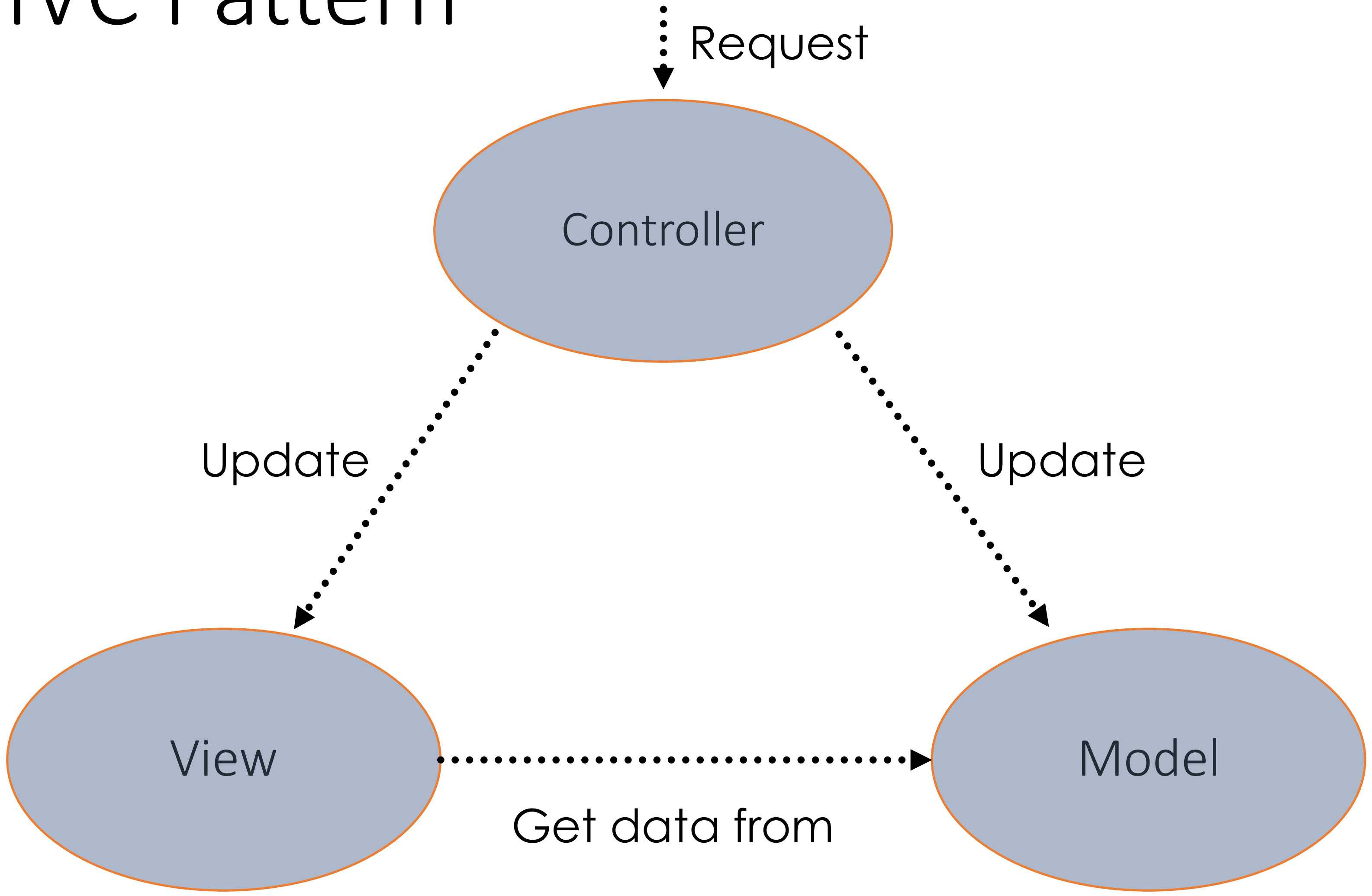
- The browser is nothing more than a renderer of HTML
- Initial download will contain
  - Page with actual content
  - Posted content from entering data
  - CSS (may be cached)
- Adding data to the page will increase the page size
  - All “old” data is discarded
  - Entire view needs to be replaced with fresh copy
- Most of the data is useless anyway

Sorry, we’re closed

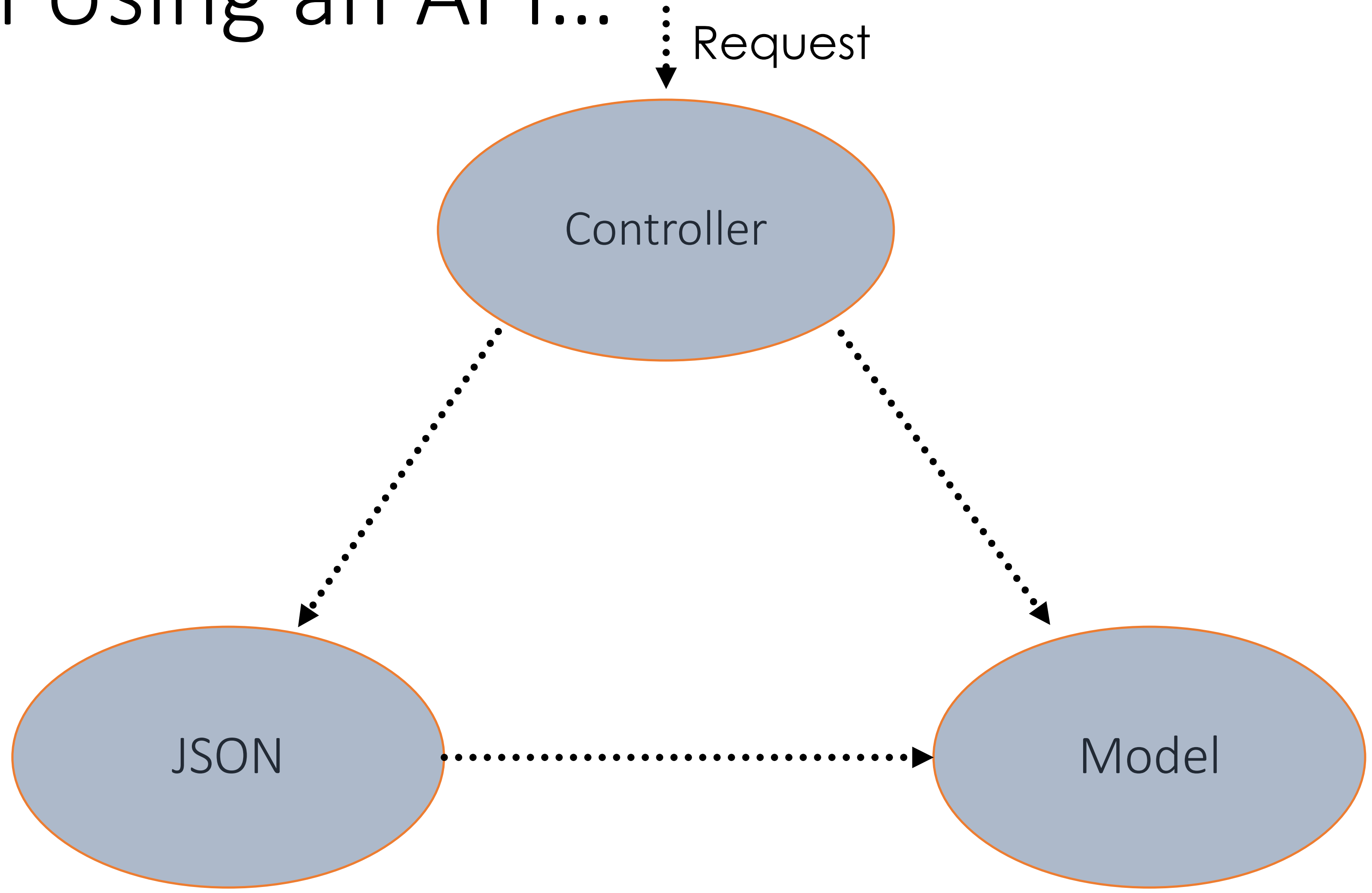
- HTML is basically also closed
- Model is only available through the Controller
- If we want the data in another application, we’re stuck
    - Something we tend to forget quite often
    - Many apps start out with just one UI but will get extra ones over time

Creating an API with ASP.NET Core *Web API*

The MVC Pattern



When Using an API…



Creating an API Controller

- API controller == MVC controller that gives access to data without HTML
- Data delivery is most often done using REST
    - Representational State Transfer
    - Format is JSON (preferred) or XML
- REST is based on HTTP verbs and URLs/resources
    - HTTP verb specifies what we want to do
    - Resource represent what we want to use, which objects

A very simple Controller

```
public class PieController : Controller
{
 ...
}
```

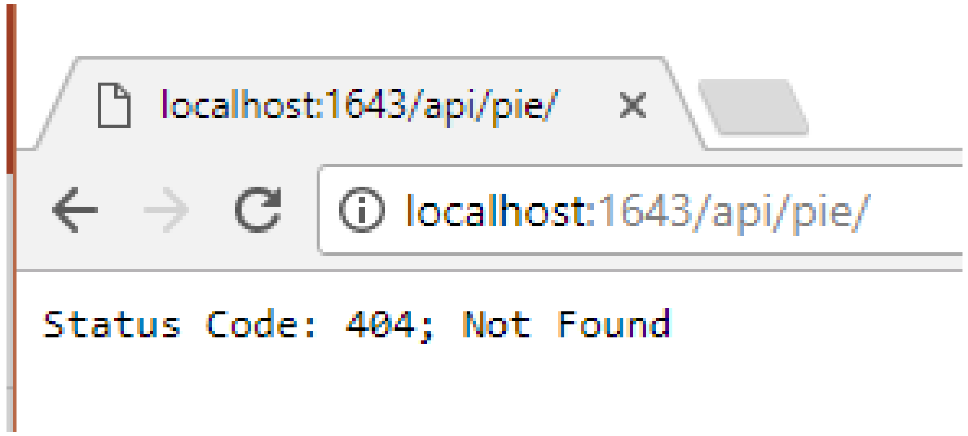
Controllers

- Similar to controllers in plain ASP.NET Core MVC
- Base ApiController is no more
  - All is now in one class: Controller
  - You can (and should) now have mixed classes

A very simple Action method

```
public class PieController : Controller
{
  public JsonResult Get()
  {
    var pies = new List<Pie>()
    {
      ...
    };
    return new JsonResult(pies);
  }
}
```

And the result is…



```
Status Code: 404; Not Found
```

SNOWBALL

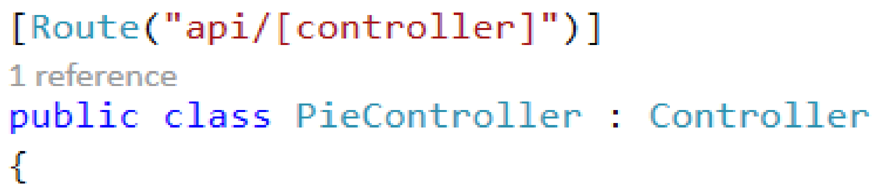
We’ll need some routing…

Routing

- Routing will allow us to match the request to the action method on the Controller
- 2 options exist
  - Convention-based routing
    - Useful for MVC applications
    - Similar to “old” model → the one we’ve already looked at!
  - Attribute-based routing
    - More natural fit for APIs

Working with the Route

- Convention is starting the route with *api*, followed by name of the controller
  - Can be reached using /api/pie



```
[Route("api/[controller]")]
1 reference
public class PieController : Controller
{
```

HTTP methods

- Routes are combined with HTTP methods
  - Used to indicate which action we want to do
  - Each will have a result, a payload
  - Response code is used to report on the outcome of the operation

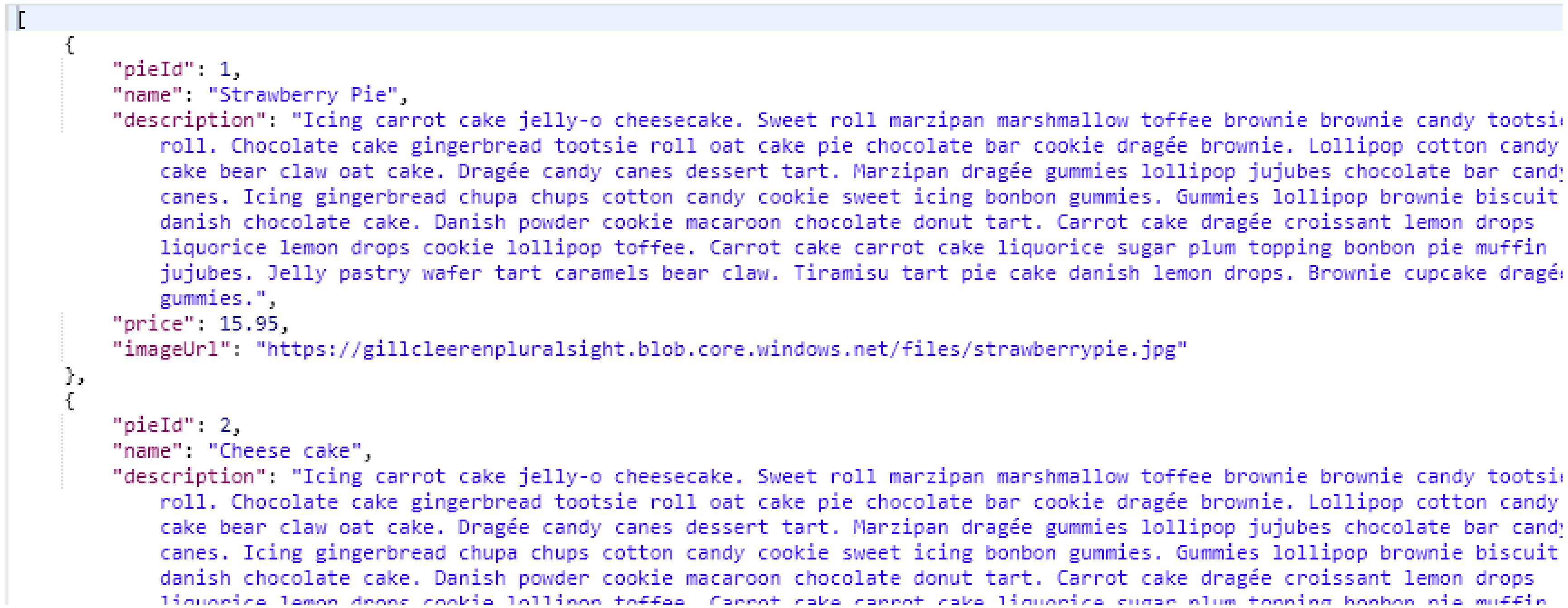
HTTP methods

Verb	Method	Result
GET	/api/pie	Gets collection of all pies
GET	/api/pie/1	Gets specific pie
POST	/api/pie	Contains data to create a new pie instance, will return saved data
PUT	/api/pie	Contains changed data to update an existing pie instance
DELETE	/api/pie/1	Deletes certain pie

An example

- API URL can be
  /api/pie/1
- **api**: indication that we’re going to work with the data part
- **pie**: indicate that we’re going to work with the collection of pies
- **1**: the actual object within the collection that we’re going to work with

The result: JSON



```
[
    {
        "pieId": 1,
        "name": "Strawberry Pie",
        "description": "Icing carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy tootsi
            roll. Chocolate cake gingerbread tootsie roll oat cake pie chocolate bar cookie dragée brownie. Lollipop cotton candy
            cake bear claw oat cake. Dragée candy canes dessert tart. Marzipan dragée gummies lollipop jujubes chocolate bar cand
            canes. Icing gingerbread chupa chups cotton candy cookie sweet icing bonbon gummies. Gummies lollipop brownie biscuit
            danish chocolate cake. Danish powder cookie macaroon chocolate donut tart. Carrot cake dragée croissant lemon drops
            liquorice lemon drops cookie lollipop toffee. Carrot cake carrot cake liquorice sugar plum topping bonbon pie muffin
            jujubes. Jelly pastry wafer tart caramels bear claw. Tiramisu tart pie cake danish lemon drops. Brownie cupcake dragé
            gummies.",
        "price": 15.95,
        "imageUrl": "https://gillcleerenpluralsight.blob.core.windows.net/files/strawberrypie.jpg"
    },
    {
        "pieId": 2,
        "name": "Cheese cake",
        "description": "Icing carrot cake jelly-o cheesecake. Sweet roll marzipan marshmallow toffee brownie brownie candy tootsi
            roll. Chocolate cake gingerbread tootsie roll oat cake pie chocolate bar cookie dragée brownie. Lollipop cotton candy
            cake bear claw oat cake. Dragée candy canes dessert tart. Marzipan dragée gummies lollipop jujubes chocolate bar cand
            canes. Icing gingerbread chupa chups cotton candy cookie sweet icing bonbon gummies. Gummies lollipop brownie biscuit
            danish chocolate cake. Danish powder cookie macaroon chocolate donut tart. Carrot cake dragée croissant lemon drops
            liquorice lemon drops cookie lollipop toffee. Carrot cake carrot cake liquorice sugar plum topping bonbon pie muffin
```

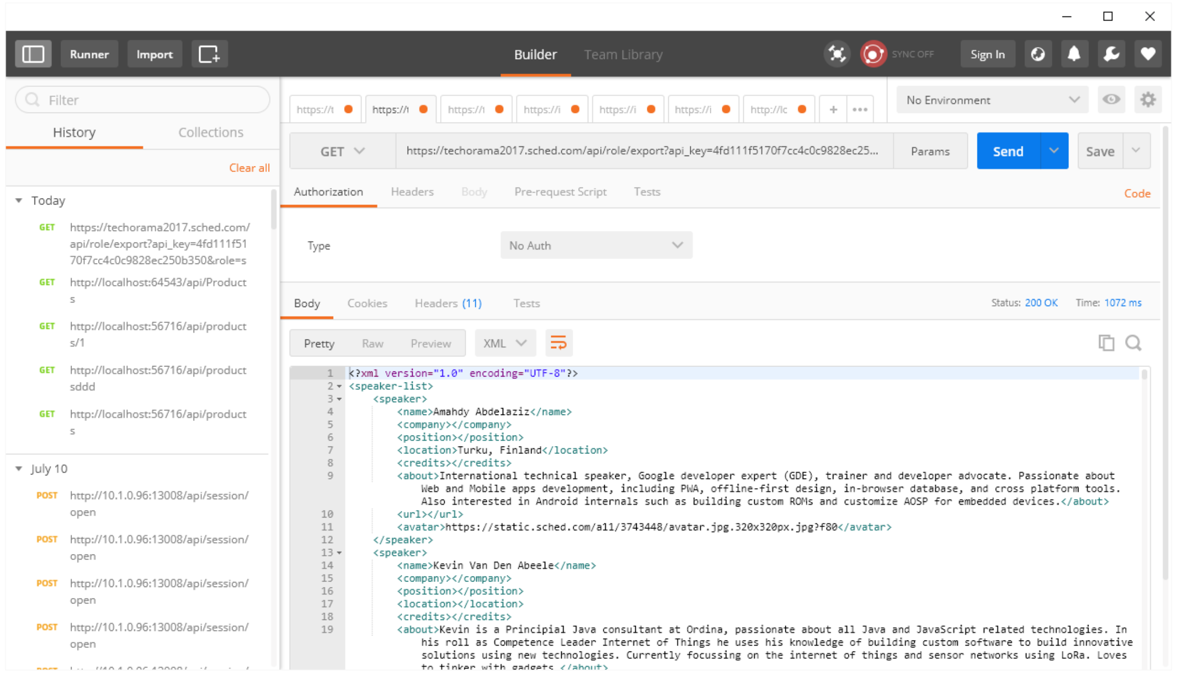
DEMO

Creating a simple API

Testing and using your controllers

Testing your controllers

- Different options
  - Fiddler
  - PostMan
  - Swashbuckle
    - NuGet package that will add API description page and allow for testing



Today

GET https://techorama2017.sched.com/api/role/export?api_key=4fd111f5170f7cc4c0c9828ec250b350&role=s

GET http://localhost:64543/api/Products

GET http://localhost:56716/api/products/1

GET http://localhost:56716/api/productsddd

GET http://localhost:56716/api/products

July 10

POST http://10.1.0.96:13008/api/session/open

POST http://10.1.0.96:13008/api/session/open

POST http://10.1.0.96:13008/api/session/open

POST http://10.1.0.96:13008/api/session/open

GET https://techorama2017.sched.com/api/role/export?api_key=4fd111f5170f7cc4c0c9828ec25...

Type: No Auth

Status: 200 OK Time: 1072 ms

```xml
<?xml version="1.0" encoding="UTF-8"?>
<speaker-list>
    <speaker>
        <name>Amahdy Abdelaziz</name>
        <company></company>
        <position></position>
        <location>Turku, Finland</location>
        <credits></credits>
        <about>International technical speaker, Google developer expert (GDE), trainer and developer advocate. Passionate about
            Web and Mobile apps development, including PWA, offline-first design, in-browser database, and cross platform tools.
            Also interested in Android internals such as building custom ROMs and customize AOSP for embedded devices.</about>
        <url></url>
        <avatar>https://static.sched.com/a11/3743448/avatar.jpg.320x320px.jpg?f80</avatar>
    </speaker>
    <speaker>
        <name>Kevin Van Den Abeele</name>
        <company></company>
        <position></position>
        <location></location>
        <credits></credits>
        <about>Kevin is a Principial Java consultant at Ordina, passionate about all Java and JavaScript related technologies. In
            his roll as Competence Leader Internet of Things he uses his knowledge of building custom software to build innovative
            solutions using new technologies. Currently focussing on the internet of things and sensor networks using LoRa. Loves
            to tinker with gadgets.</about>
```

SNOWBALL

DEMO

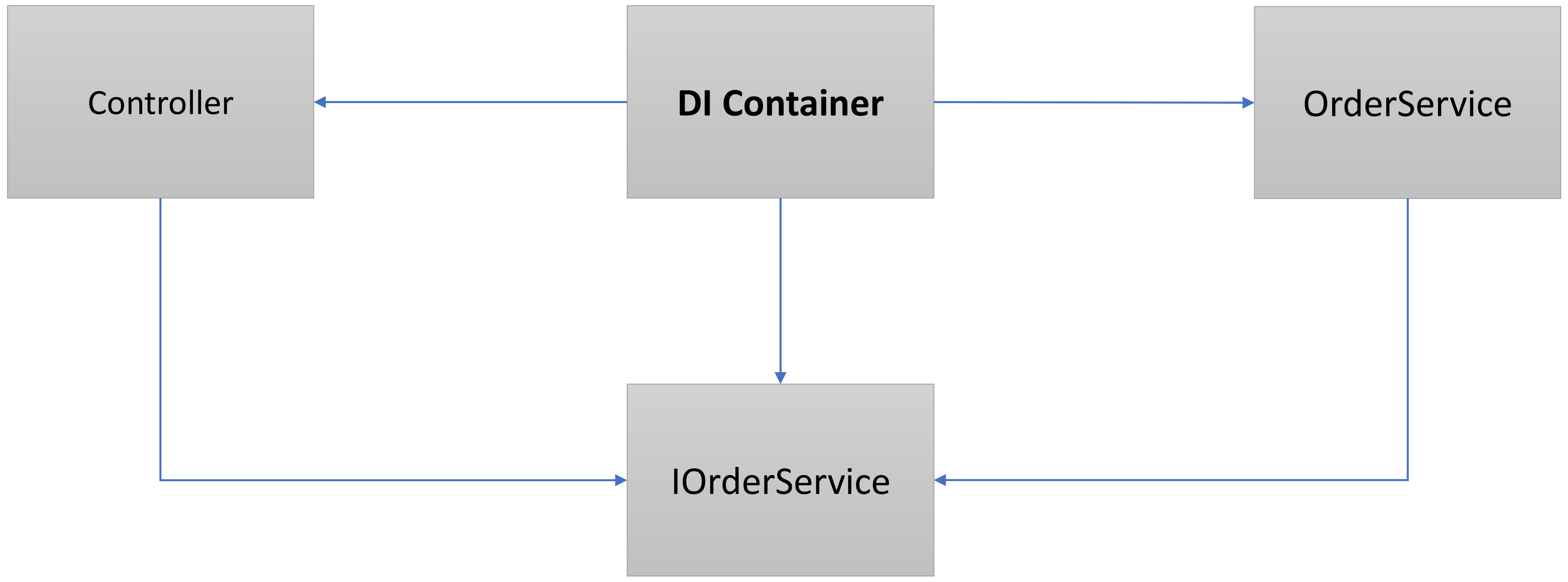
Testing the API using Postman

Using Dependency Injection and repositories

Dependency Injection

- Type of inversion of control (IoC)
- Another class is responsible for obtaining the required dependency
- Results in more loose coupling
  - Container handles instantiation as well as lifetime of objects
- Built-in into ASP.NET Core
  - Can be replaced with other container if needed

Dependency Injection



Registering dependencies

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSingleton<ISomethingService, SomethingService>();

  // Add framework services.
  services.AddMvc();
}
```

Registering dependencies

- AddSingleton
- AddTransient
- AddScoped

Dependency injection in controller

- Constructor injection: adds dependencies into constructor parameters

```
public class PieController : Controller
{
  private readonly ISomethingService _something;

  public PieController(ISomethingService something)
  {
    _something = something;
  }
}
```

Use a **repository** to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer

The repository

```
public class PieRepository : IPieRepository
{
  private List<Pie> _pies;

  public PieRepository()
  {
    _pies = new List<Pie>()
      {
        ...

      };
  }

  public IEnumerable<Pie> Pies => _pies;
}
```

Registering the repository

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSingleton<IPieRepository, PieRepository>();
}
```

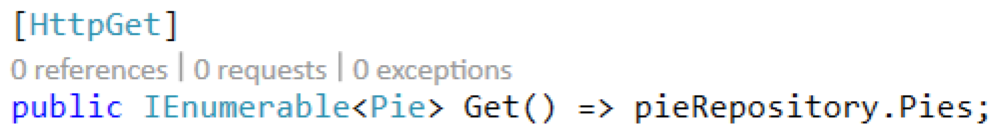
DEMO

Adding a Repository and Dependency Injection

Adding more functionality on the controller

Action methods

- Action methods need to be declared with attribute that specifies the HTTP method

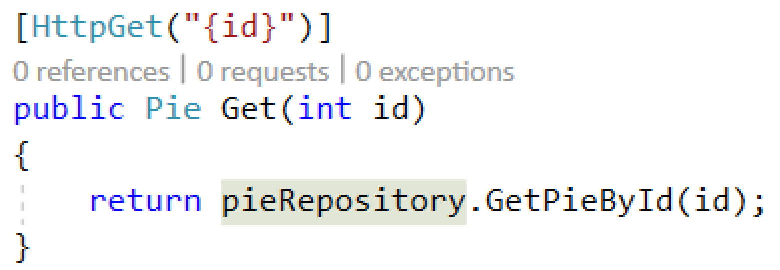


```
[HttpGet]
0 references | 0 requests | 0 exceptions
public IEnumerable<Pie> Get() => pieRepository.Pies;
```

- Available attributes
  - HttpGet
  - HttpPost
  - HttpPut
  - HttpDelete
  - HttpPatch
  - HttpHead
  - AcceptVerbs: for multiple verbs on one method

Action methods

- We can pass values (routing fragment)

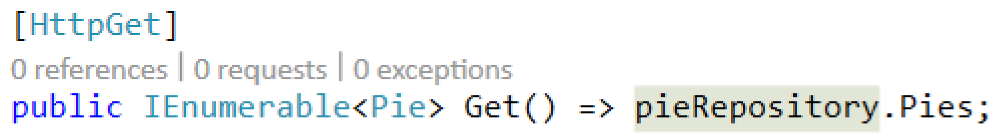


```
[HttpGet("{id}")]
0 references | 0 requests | 0 exceptions
public Pie Get(int id)
{
    return pieRepository.GetPieById(id);
}
```

  - Can be reached by combining the route with the parameter
    - /api/pie/{id}
- Routes in general don’t contain an action segment
  - Action name isn’t part of the URL to reach an action method
  - HTTP method is used to differentiate between them

Action results

- API action methods don’t rely on ViewResult
  - Instead, return data



```
[HttpGet]
0 references | 0 requests | 0 exceptions
public IEnumerable<Pie> Get() => pieRepository.Pies;
```

  - MVC will make sure they get returned in a correct format
    - By default JSON
    - It’s possible to change this, even from the client
      - Media Type formatting is applied here

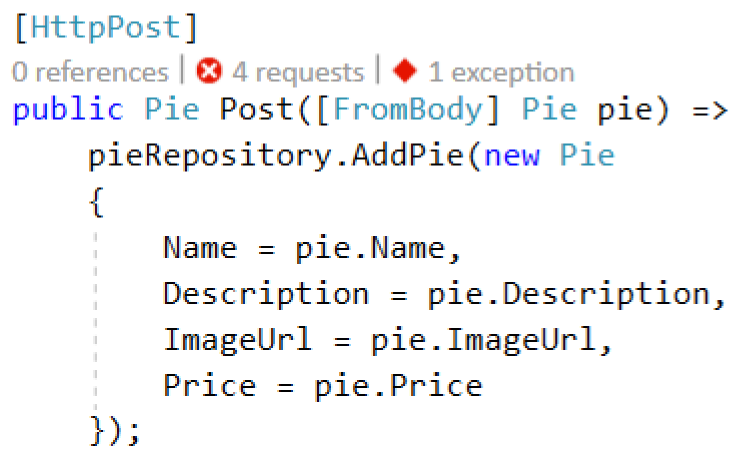
SNOWBALL

Status codes

- Status code are part of the response from the API
- Used for information:
  - Whether or not the request succeeded
  - Why did it fail

Status codes

- MVC is good at knowing what to return
  - If we return null, it’ll return 204 – No Content
- We can override this by returning a different IActionResult
  - NotFound leads to 404
  - Ok just sends the object to the client and returns 200



```
[HttpPost]
0 references | 4 requests | 1 exception
public Pie Post([FromBody] Pie pie) =>
    pieRepository.AddPie(new Pie
    {
        Name = pie.Name,
        Description = pie.Description,
        ImageUrl = pie.ImageUrl,
        Price = pie.Price
    });
```

Most important Status codes

- 20X: Success
  - 200: OK
  - 201: Created
  - 204: No content
- 40X: Client error
  - 400: Bad request
  - 403: Forbidden
  - 404: Not found
- 50X: Server error
  - 500: Internal server error

Using Ok()

```
[HttpGet]
public IActionResult Get()
{
  return Ok(pieRepository.Pies);
}
```

SNOWBALL

DEMO

Completing the Controller and repository

Adding Entity Framework Core

Entity Framework (EF) Core is a lightweight and extensible version of the popular Entity Framework data access technology

Entity Framework Core

- ORM
- Lightweight
- Cross-platform
- Open source
- Works with several databases
- Code-first

The DbContext

```
public class AppDbContext: DbContext
{
  public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
  {
  }

  public DbSet<Pie> Pies { get; set; }
}
```

DEMO

Adding Entity Framework Core

Content formatting

Content formatting

- MVC has to work out which data format it should return
  - Basically, which encoding to use
- Can be influenced by the client request
- Content format depends on
  - Format(s) accepted by client
  - Format(s) that can be produced
  - Content policy specified by the action
  - Type returned by the action
- Can be hard to figure out
  - In most cases, the default will do

Default content policy

- Easiest situation: client and action method define no restrictions on the format that can be used
  - If method returns string, string is returned to client, content-type header is set to text/plain
  - All other data types are returned as JSON, content-type is set to application/json
- Strings aren’t returned as JSON because of possible issues
  - Double quoted strings: “”Hello world””
  - An int is simply returned as “2”

DEMO

Default content policy

Content negotiation

- Clients will typically include an Accept header
  - Specifies possible accepted formats (MIME types)
- Browser will send
  - Accept: text/html,application/xhtml +xml,application/xml;q=0.9,image/webp,*/*;q=0.8
  - Prefers HTML and XHTML, can accept XML and Webp
  - Q indicated preference (xml is less preferred)
  - */* indicated that all will work
    - But only for 0.8

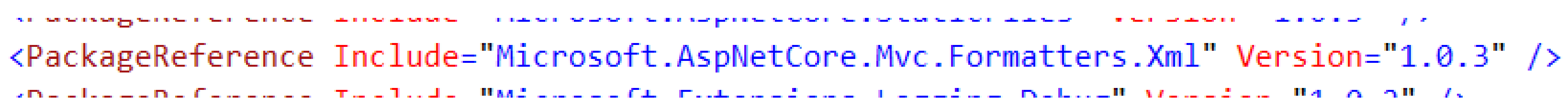
Changing the ACCEPT header

- If we ask the API for XML…
  - Headers Accept="application/xml"
- We’ll get back json… (Thank you MVC)

- MVC doesn’t know about XML at this point

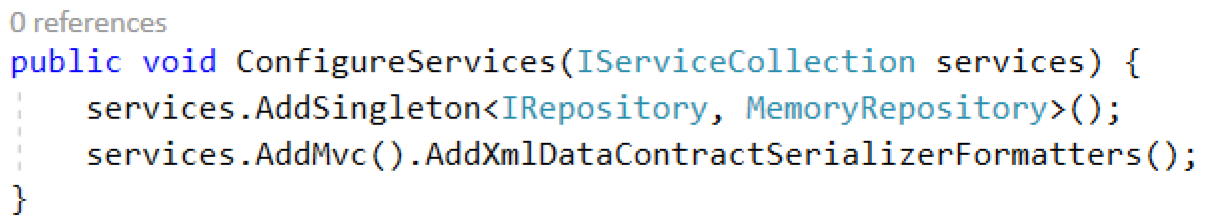
XML please

- Requires extra package to be included



```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Formatters.Xml" Version="1.0.3" />
```

- And as always, some configuration change in the Startup class



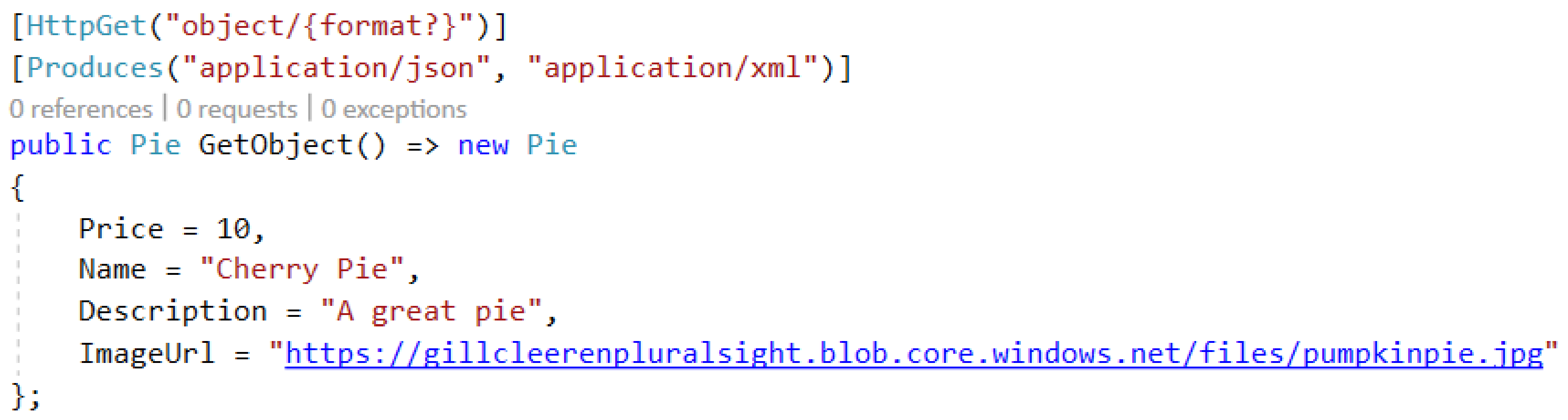
```
0 references
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IRepository, MemoryRepository>();
    services.AddMvc().AddXmlDataContractSerializerFormatters();
}
```

DEMO

Adding support for XML

Overriding the Action Data Format

- We can specify the data format on the action method itself
  - Produces is filter
  - Changes the content type
  - Argument is the returned data format



```
[HttpGet("object/{format?}")]
[Produces("application/json", "application/xml")]
0 references | 0 requests | 0 exceptions
public Pie GetObject() => new Pie
{
    Price = 10,
    Name = "Cherry Pie",
    Description = "A great pie",
    ImageUrl = "https://gillcleerenpluralsight.blob.core.windows.net/files/pumpkinpie.jpg"
};
```

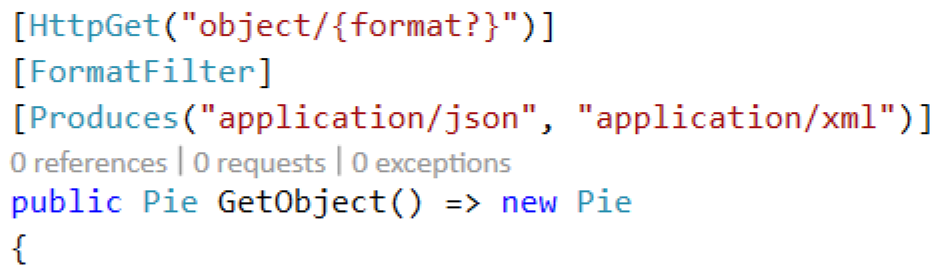
Passing the data format using the route or the query string

- ACCEPT header can't always be controlled from client
- Data format can therefore also be specified through route or query string to reach an action
- We can define a shorthand in the Startup class
    - xml can now be used to refer to application/xml

```
services.AddMvc()
    .AddXmlDataContractSerializerFormatters()
    .AddMvcOptions(opts => {
        opts.FormatterMappings.SetMediaTypeMappingForFormat("xml",
            new MediaTypeHeaderValue("application/xml"));
    });
```

Passing the data format using the route or the query string

- On the action, we now need to add the FormatFilter
- Route can now also include the format (optional)
  - If combined with Produces and other type is requested, 404 is returned
  - Produces is not required to be used here though
- Upon receiving the shorthand (xml), the mapped type from the Startup is retrieved



```
[HttpGet("object/{format?}")]
[FormatFilter]
[Produces("application/json", "application/xml")]
0 references | 0 requests | 0 exceptions
public Pie GetObject() => new Pie
{
```

DEMO

Content Negotiation

Versioning the API

Versioning an API

- Something that's often forgotten by developers…
- Versioning is required
  - Clients evolve, API needs to follow as well
  - Old versions often still required
- Many ways exist
  - URI versioning
  - Header versioning
  - Accept-headers versioning
  - …

Aspnet-api-versioning

- Open-source package
  - microsoft.aspnetcore.mvc.versioning
- Supports
  - ASP.NET Web API
  - ASP.NET Web API with oData
  - ASP.NET Core
- Added using
  - services.AddApiVersioning();

Versioning using Querystring parameter

```
namespace My.Services.V1
{
    [ApiVersion("1.0")]
    public class HelloWorldController : Controller
    {
        public string Get() => "Hello world v1.0!";
    }
}
namespace My.Services.V2
{
    [ApiVersion("2.0")]
    public class HelloWorldController : Controller
    {
        public string Get() => "Hello world v2.0!";
    }
}
```

Versioning using URL Path Segment

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
public class HelloWorldController : Controller {
    public string Get() => "Hello world!";
}

[ApiVersion("2.0")]
[ApiVersion("3.0")]
[Route("api/v{version:apiVersion}/helloworld")]
public class HelloWorld2Controller : Controller {
    [HttpGet]
    public string Get() => "Hello world v2!";

    [HttpGet, MapToApiVersion( "3.0" )]
    public string GetV3() => "Hello world v3!";
}
```

Versioning using Header

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddApiVersioning(o => o.ApiVersionReader = new HeaderApiVersionReader("api-version"));
}
```

DEMO

Versioning the API