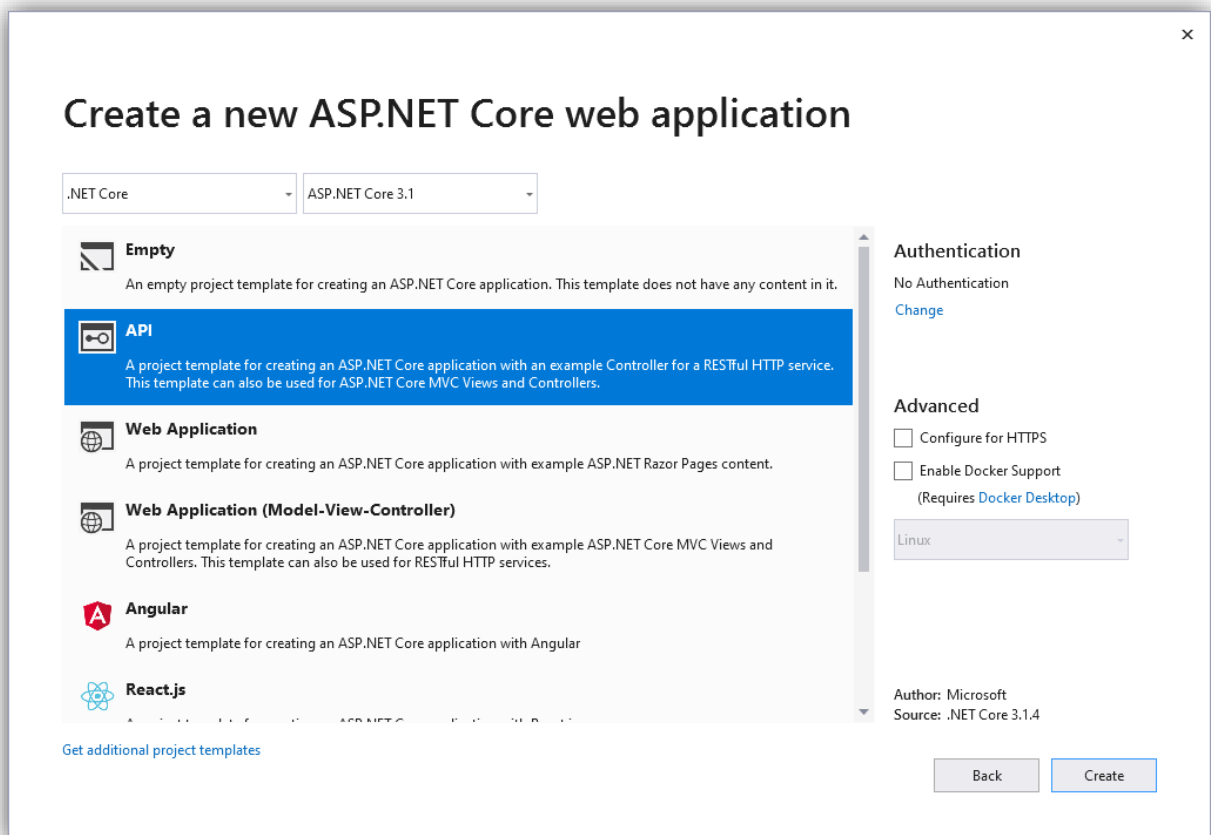
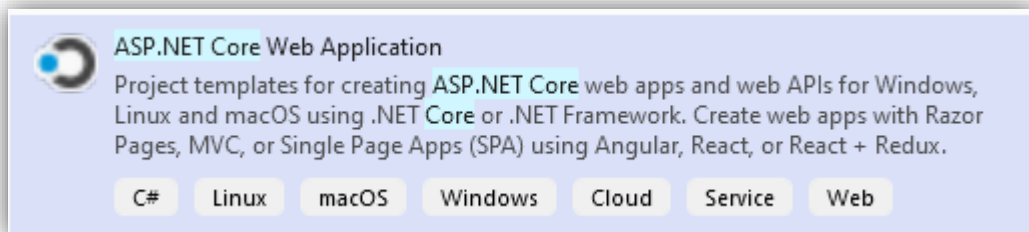


GET started

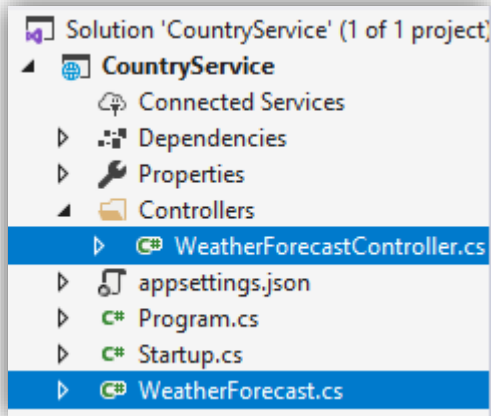
In dit hoofdstuk tonen we hoe een service kan worden opgezet en implementeren we enkel de GET methode. We bespreken hier nog niet de verschillende details en settings, de bedoeling is hier om een eenvoudig GET request te bekijken met de verschillende mogelijke antwoorden (responses).

Opzetten Visual Studio project

We starten een nieuw project en kiezen voor een ASP.NET Core Web Application. Aangezien we enkel een REST-service willen bouwen selecteren we daarna de optie API.

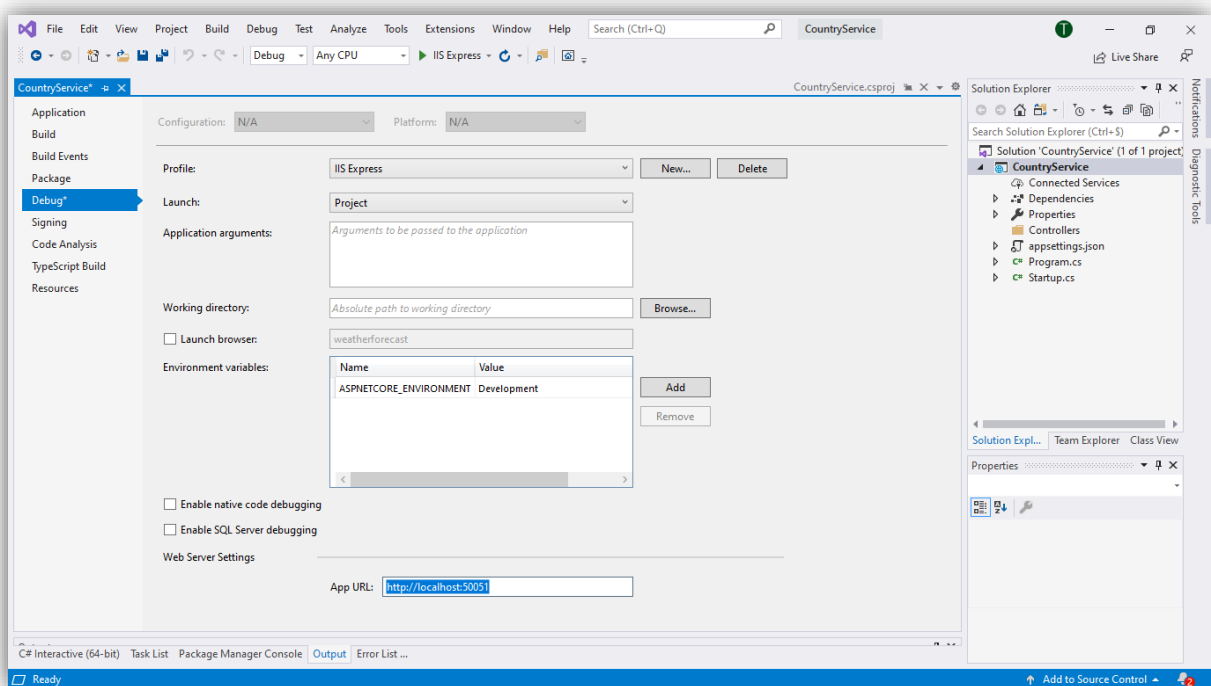


Het resultaat is een project met de volgende folders en bestanden :



We verwijderen daarbij de WeatherForecast.cs en WeatherForecastController.cs bestanden die standaard als voorbeeld worden aangemaakt, maar die we niet gaan gebruiken.

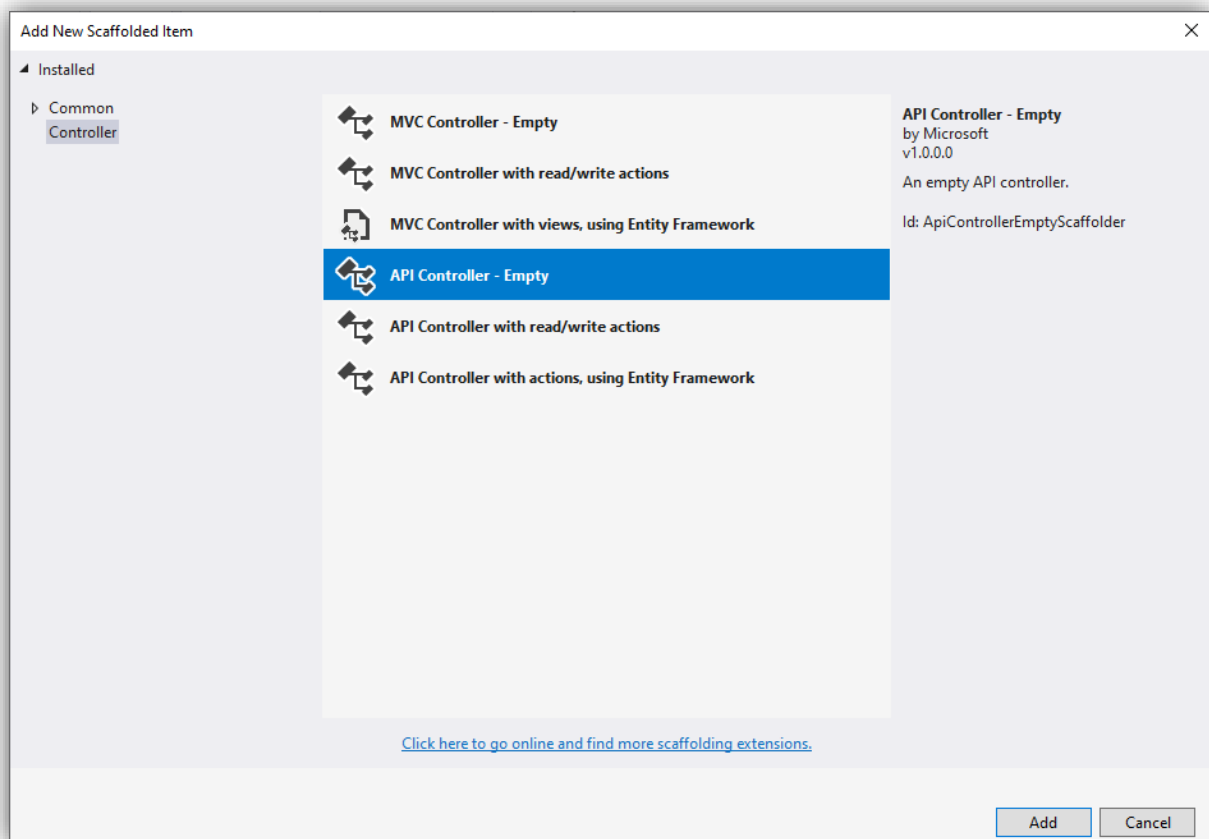
Verder maken we nog een aantal aanpassingen in de configuratie vooraleer te starten. Bij 'Launch' kiezen we voor project zodat we alles in een console zien verschijnen, 'launch browser' zetten we uit en bij App URL vullen we in "http://localhost:50051".



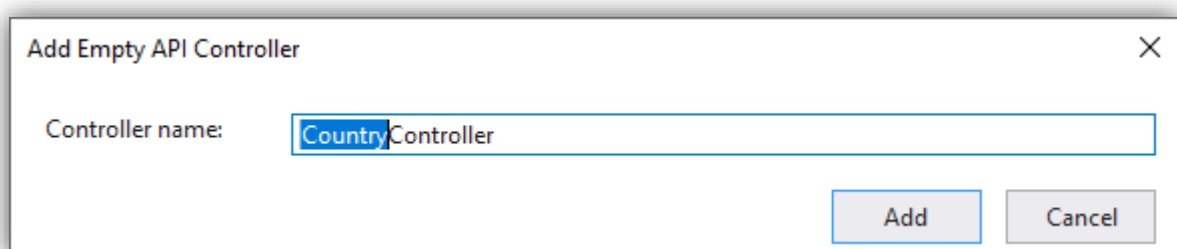
Controller

De volgende stap is het toevoegen van een 'controller', deze zorgt ervoor dat de bronnen of resources via de API ter beschikking worden gesteld aan de gebruikers. We kunnen de controller zien als een klasse die C# vertaalt naar HTTP.

Om een controller toe te voegen selecteren we de folder controllers en met de rechtermuisknop kunnen we een menu oproepen waarin we via 'Add' en 'Controller' het volgende scherm kunnen oproepen.



In dit scherm kiezen we voor een lege API Controller (omdat we nu toch enkel de GET requests gaan implementeren) en geven deze de naam CountryController.

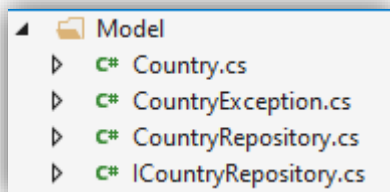


Onze controller klasse erft van de klasse ControllerBase en bevat ook de annotaties [ApiController] en [Route] die aangeeft bij welke URL de controller zal worden opgeroepen (ook hierover later meer).

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class CountryController : ControllerBase
{
```

Model

Het is natuurlijk de bedoeling om via onze service bronnen/resources ter beschikking te stellen en dus is het ook noodzakelijk om een bron te definiëren en een klasse te voorzien die deze kan bewerken. We voegen daarom de volgende 4 klassen toe in de folder 'Model' :



Een eenvoudige klasse Country die slechts een beperkt aantal attributen bevat.

```
public class Country
{
    0 references
    public Country(string name, string capital, string continent)
    {
        Name = name;
        Capital = capital;
        Continent = continent;
    }
    5 references
    public Country(int id, string name, string capital, string continent)
    {
        Id = id;
        Name = name;
        Capital = capital;
        Continent = continent;
    }
    7 references
    public int Id { get; set; }
    2 references
    public string Name { get; set; }
    2 references
    public string Capital { get; set; }
    2 references
    public string Continent { get; set; }
}
```

Een Exception klasse voor als er iets fout gaat :

```

public class CountryException : Exception
{
    4 references
    public CountryException(string message) : base(message)
    {
    }
}

```

De bewerkingen op de collectie van onze objecten implementeren we door middel van de CountryRepository en om deze klasse los te kunnen koppelen definiëren we ook de interface ICountryRepository.

```

public interface ICountryRepository
{
    0 references
    void AddCountry(Country country);
    0 references
    Country GetCountry(int id);
    0 references
    IEnumerable<Country> GetAll();
    0 references
    void RemoveCountry(Country country);
    0 references
    void UpdateCountry(Country country);
}

```

We houden de implementatie van de repository bewust heel eenvoudig zodat we ons kunnen focussen op de werking van de service. Er is daarom gekozen om dictionary te gebruiken die slechts 5 objecten bevat en deze in de constructor te initialiseren.

```

public class CountryRepository : ICountryRepository
{
    private Dictionary<int, Country> data = new Dictionary<int, Country>();

    0 references
    public CountryRepository()
    {
        data.Add(1, new Country(1, "België", "Brussel", "Europa"));
        data.Add(2, new Country(2, "Peru", "Lima", "Zuid-Amerika"));
        data.Add(3, new Country(3, "Duitsland", "Berlijn", "Europa"));
        data.Add(4, new Country(4, "Zweden", "Stockholm", "Europa"));
        data.Add(5, new Country(5, "Noorwegen", "Oslo", "Europa"));
    }
}

```

```

public IEnumerable<Country> GetAll()
{
    return data.Values;
}

2 references
public Country GetCountry(int id)
{
    if (data.ContainsKey(id))
        return data[id];
    else
        throw new CountryException("country doesn't exist");
}

```

```

public void AddCountry(Country country)
{
    if (!data.ContainsKey(country.Id))
        data.Add(country.Id, country);
    else
        throw new CountryException("country already added");
}
1 reference
public void RemoveCountry(Country country)
{
    if (data.ContainsKey(country.Id))
        data.Remove(country.Id);
    else
        throw new CountryException("country doesn't exist");
}
1 reference
public void UpdateCountry(Country country)
{
    if (data.ContainsKey(country.Id))
        data[country.Id] = country;
    else
        throw new CountryException("country doesn't exist");
}

```

Om de repository te kunnen gebruiken in de controller voegen we een variabele van het type `ICountryRepository` toe in de klasse `CountryController` en initialiseren deze in de constructor.

```

public class CountryController : ControllerBase
{
    private ICountryRepository repo;

    0 references
    public CountryController(ICountryRepository repo)
    {
        this.repo = repo;
    }
}

```

Via dependency injection (hierover later meer) geven we de klasse `CountryRepository` mee in de constructor van de controller. Om dit te realiseren voegen we de volgende lijn code toe in de `ConfigureServices` methode van de `Startup` klasse :

```

Services.AddSingleton<ICountryRepository, CountryRepository>();

```

```

public class Startup
{
    0 references
    public Startup(IConfiguration configuration) {...}

    1 reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.AddSingleton<ICountryRepository, CountryRepository>();
    }
}

```

GET return types

Specifiek return type

Om een methode te linken aan een GET request volstaat het om de annotatie [HttpGet] toe te voegen voor de methode die moet worden uitgevoerd. (Routing zal in één van de volgende hoofdstukken worden besproken)

Specifiëren wat de methode precies teruggeeft kan door middel van een specifiek return type. In het volgende voorbeeld geven we een IEnumerable<Country> terug als antwoord.

```

// GET: api/Country
[HttpGet]
0 references
public IEnumerable<Country> Get()
{
    return repo.GetAll();
}

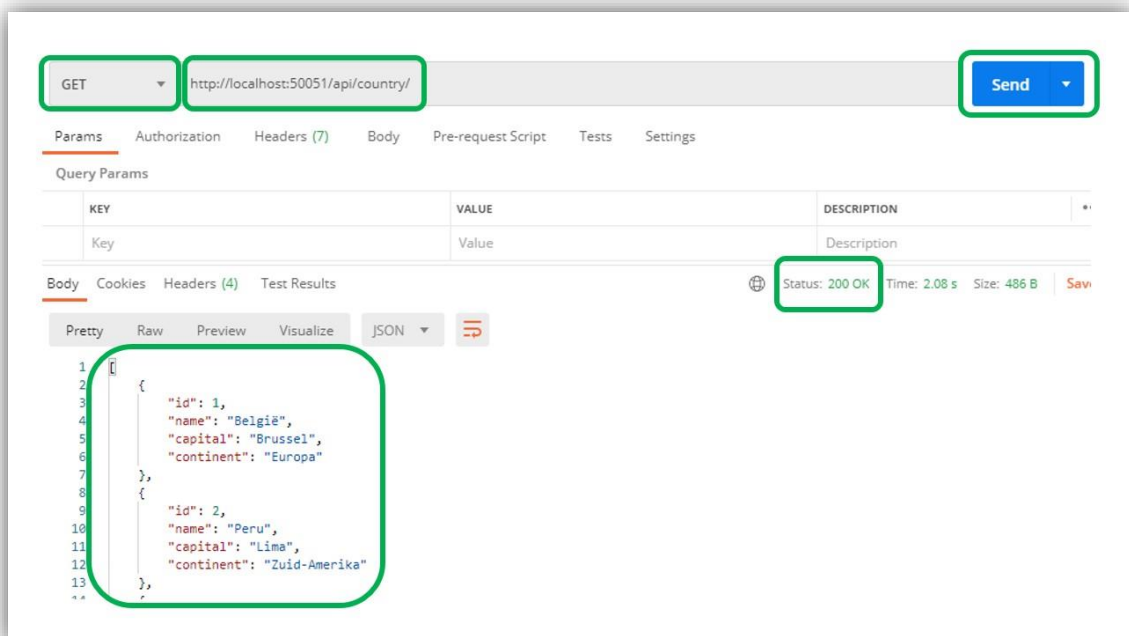
```

Voeren we onze code uit dan verschijnt het volgende console-venster :


```
C:\NET\VS19\ASP\CountryService\bin\Debug\netcoreapp3.1\CountryService.exe
Info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:50051
Info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
Info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
Info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\NET\VS19\ASP\CountryService
```

Om te interageren met de service kunnen we gebruik maken van de tool "Postman". De tool kan gedownload worden van de website <https://www.postman.com/> en is eenvoudig in gebruik.

Het sturen van een GET request kan door de url <http://localhost:50051/api/country/> in te geven, GET te selecteren en dan op de Send knop te klikken. Het antwoord dat we terugkrijgen is de lijst van Country-objekten (in JSON formaat) en de status-code 200 Ok.



Om een specifiek Country-object op te vragen op basis van zijn id voegen we een tweede methode toe in de controller klasse die er als volgt uit ziet :

```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public Country Get(int id)
{
    return repo.GetCountry(id);
}
```

De HttpGet notatie bevat nu ook de string "{id}" die aangeeft dat er een parameter id in de URL aanwezig zal zijn en die aan de methode zal worden doorgegeven. De parameter wordt achteraan de 'route' van de controller toegevoegd. De methode geeft nu slechts 1 object meer terug die behoort bij de meegegeven id.

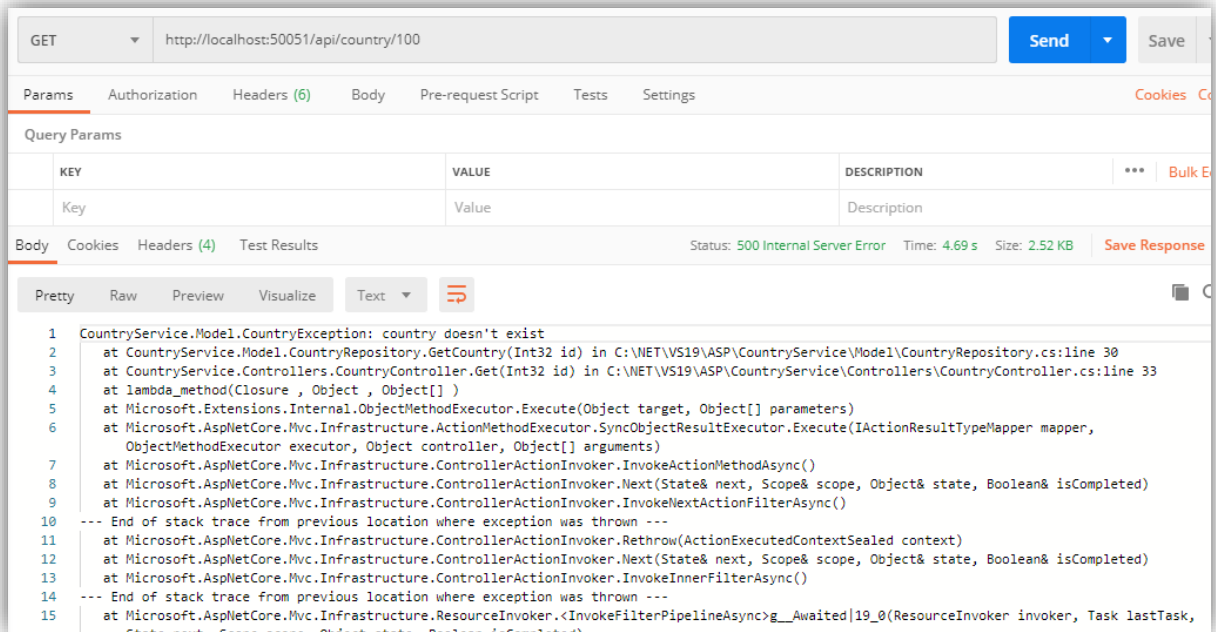
The screenshot shows a web browser's developer tools interface. The top bar indicates a GET request to `http://localhost:50051/api/country/1`. Below this, the 'Params' tab is active, showing a table with one query parameter: 'Key' with the value '1'. The 'Body' tab is also active, showing the response body in JSON format. The status is '200 OK' and the time taken is '866 ms'.

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body: Cookies Headers (4) Test Results Status: 200 OK Time: 866 ms

```
1 {
2   "id": 1,
3   "name": "België",
4   "capital": "Brussel",
5   "continent": "Europa"
6 }
```

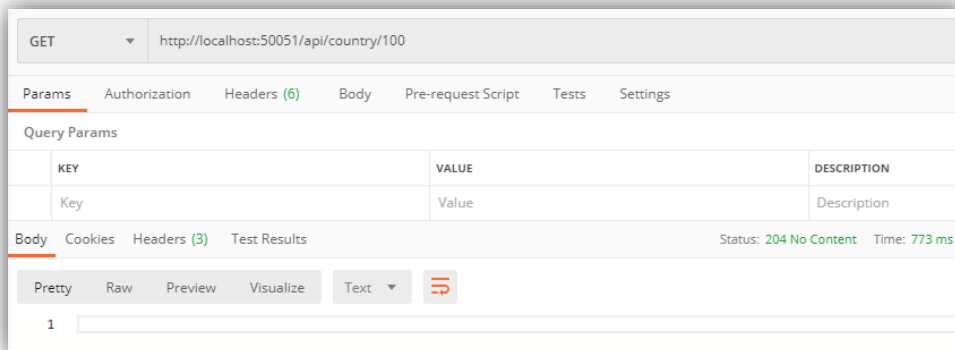
Wat gebeurt er nu als de betreffende id niet kan gevonden worden ? We proberen het uit.



Het ingeven van een niet bestaande id geeft als resultaat een statuscode 500 (internal server error) en in de body van de message vinden we de exception. Dit is echter niet de bedoeling. In de eerste plaats is dit geen fout van de server, maar een fout van de gebruiker (client) en ten tweede willen we zeker niet de exception weergeven in de message (dit geeft namelijk info over de interne werking en kan tot beveiligingsproblemen leiden).

Een mogelijke oplossing is om de fout op te vangen in de Get methode en een null-waarde terug te geven als er geen object kan worden gevonden voor de opgegeven id.

```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public Country Get(int id)
{
    try
    {
        return repo.GetCountry(id);
    }
    catch (CountryException ex)
    {
        return null;
    }
}
```



GET http://localhost:50051/api/country/100

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (3) Test Results Status: 204 No Content Time: 773 ms

Pretty Raw Preview Visualize Text

1

In plaats van een internal server error krijgen we nu de statuscode 204 No Content. Ook dit is niet de oplossing die we zoeken. In plaats daarvan wensen we een status code 400 die aangeeft dat het om een client probleem gaat. Om dit te bereiken kan de code worden aangepast door expliciet de statuscode in te stellen bij de response, zoals in het volgende voorbeeld.

```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public Country Get(int id)
{
    try
    {
        return repo.GetCountry(id);
    }
    catch (CountryException ex)
    {
        Response.StatusCode = 400;
        return null;
    }
}
```

GET http://localhost:50051/api/country/100

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 903 ms

Pretty Raw Preview Visualize Text

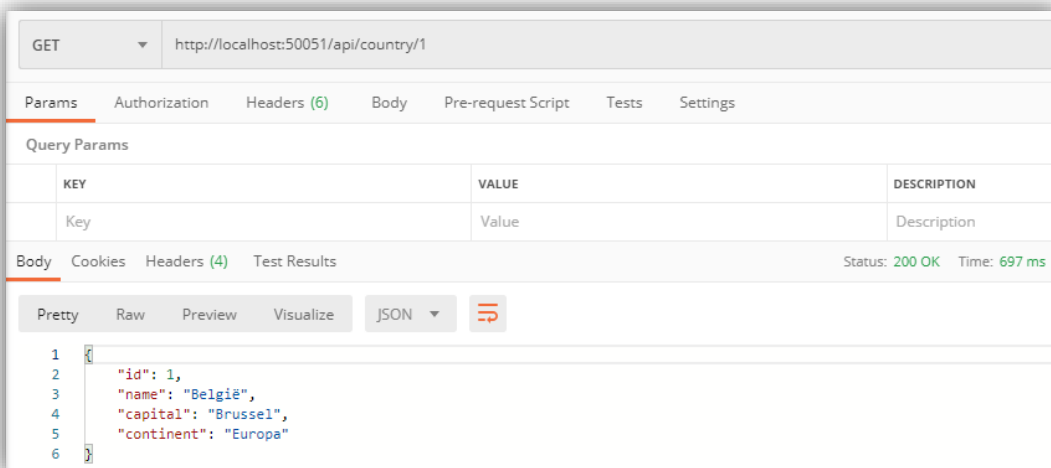
1

IActionResult en ActionResult<T>

In plaats van een specifiek type terug te geven kunnen we ook een `IActionResult` teruggeven die een statuscode combineert met het resultaat (data). Deze interface wordt door een aantal klassen geïmplementeerd zoals `Ok()`, `NotFound()`, `NoContent()` en `BadRequest()`. De methode kan dan herschreven worden zoals in het volgende voorbeeld :

```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public IActionResult Get(int id)
{
    try
    {
        return Ok(repo.GetCountry(id));
    }
    catch (CountryException ex)
    {
        return NotFound();
    }
}
```

En de resultaten zien er dan als volgt uit :



GET ▼ http://localhost:50051/api/country/1

Params Authorization Headers (6) Body Pre-request Script Tests Settings

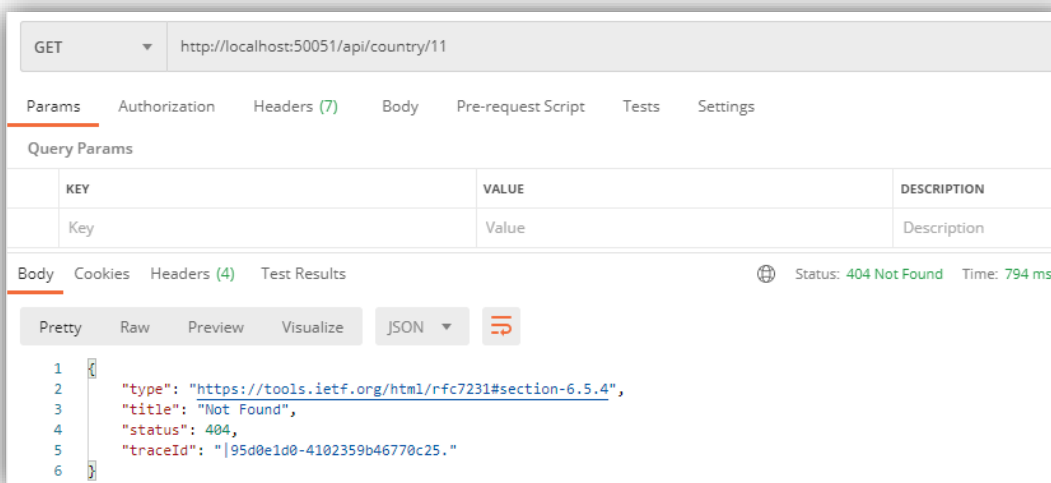
Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (4) Test Results Status: 200 OK Time: 697 ms

Pretty Raw Preview Visualize JSON ≡

```
1 {
2   "id": 1,
3   "name": "België",
4   "capital": "Brussel",
5   "continent": "Europa"
6 }
```



GET ▼ http://localhost:50051/api/country/11

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (4) Test Results ⊕ Status: 404 Not Found Time: 794 ms

Pretty Raw Preview Visualize JSON ≡

```
1 {
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
3   "title": "Not Found",
4   "status": 404,
5   "traceId": "|95d0e1d0-4102359b46770c25."
6 }
```

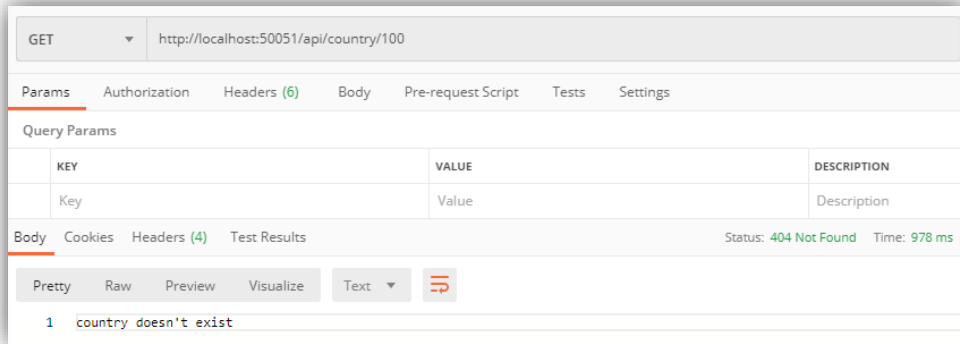
Door gebruik te maken van `ActionResult<T>` hoeven we het resultaat niet meer via de `Ok()` methode in te pakken en terug te geven, we kunnen nu rechtstreeks een object van het type `<T>` teruggeven. Ook aan client zijde is het nu duidelijk welk object er wordt teruggegeven en is er geen casting meer nodig.

```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public ActionResult<Country> Get(int id)
{
    try
    {
        return repo.GetCountry(id);
    }
    catch (CountryException ex)
    {
        return NotFound();
    }
}
```

Wensen we extra info mee te geven met de `NotFound` status code dan kunnen we dat op de volgende manier doen :

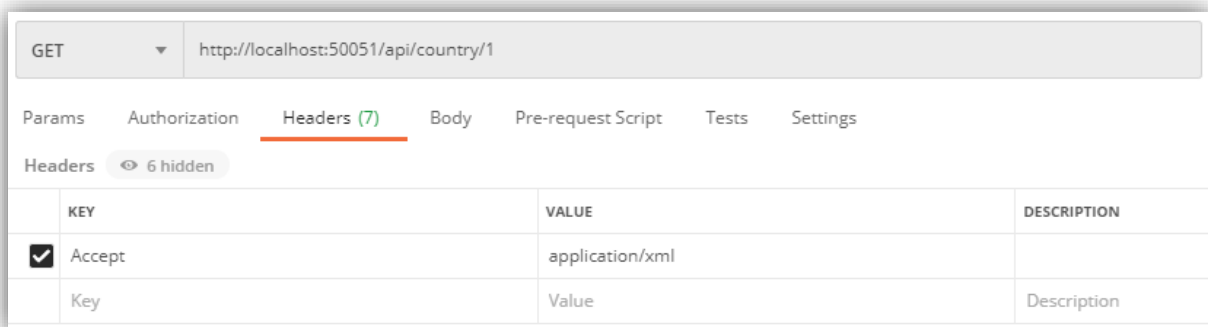
```
// GET: api/Country/5
[HttpGet("{id}", Name = "Get")]
0 references
public ActionResult<Country> Get(int id)
{
    try
    {
        return repo.GetCountry(id);
    }
    catch (CountryException ex)
    {
        return NotFound(ex.Message);
    }
}
```

Het resultaat is dan het volgende :



GET return format

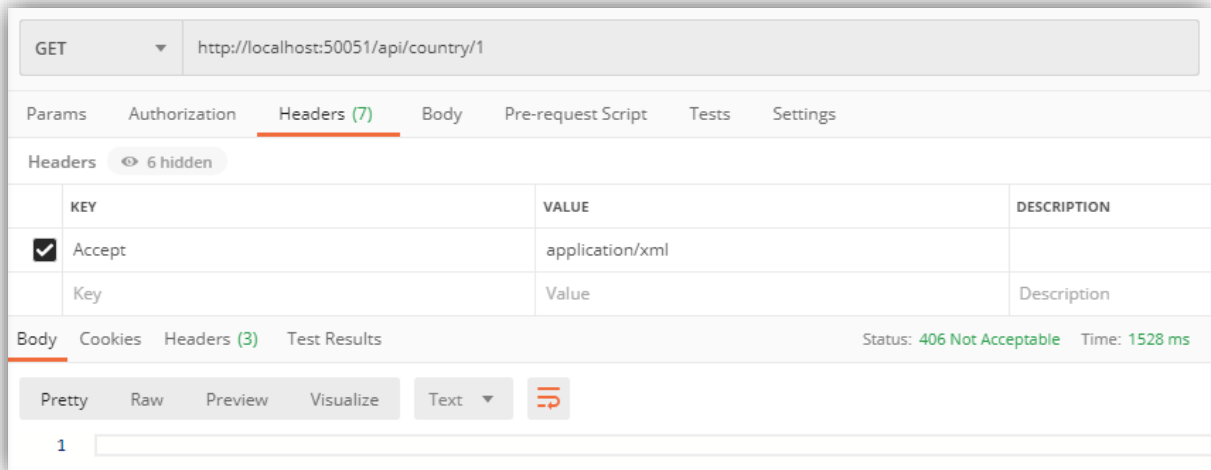
Tot nu toe hebben we de data in de response body steeds als JSON meegegeven. Maar er zijn nog andere mogelijkheden natuurlijk en één veelgebruikte daarvan is XML. Om het antwoord in XML terug te krijgen moeten we bij het sturen van het verzoek aangeven dat we een XML antwoord wensen. Dat kunnen we doen door in de header van het verzoek het key value paar `Accept : application/xml` mee te geven.



Daarnaast moeten we ook onze service configureren. Daarvoor zijn er twee settings nodig, in de eerste plaats willen we vermijden dat we een antwoord in het verkeerde formaat accepteren. Met andere woorden als we XML vragen en de service kan dit niet aanbieden dan wensen we geen JSON (default) te zien maar wel een status code 406 Not Acceptable. Om dit te bekomen passen we de code `services.AddControllers` aan in de `ConfigureServices` van de `Startup` klasse als volgt :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(setup=>setup.ReturnHttpNotAcceptable=true);
    services.AddSingleton<ICountryRepository, CountryRepository>();
}
```

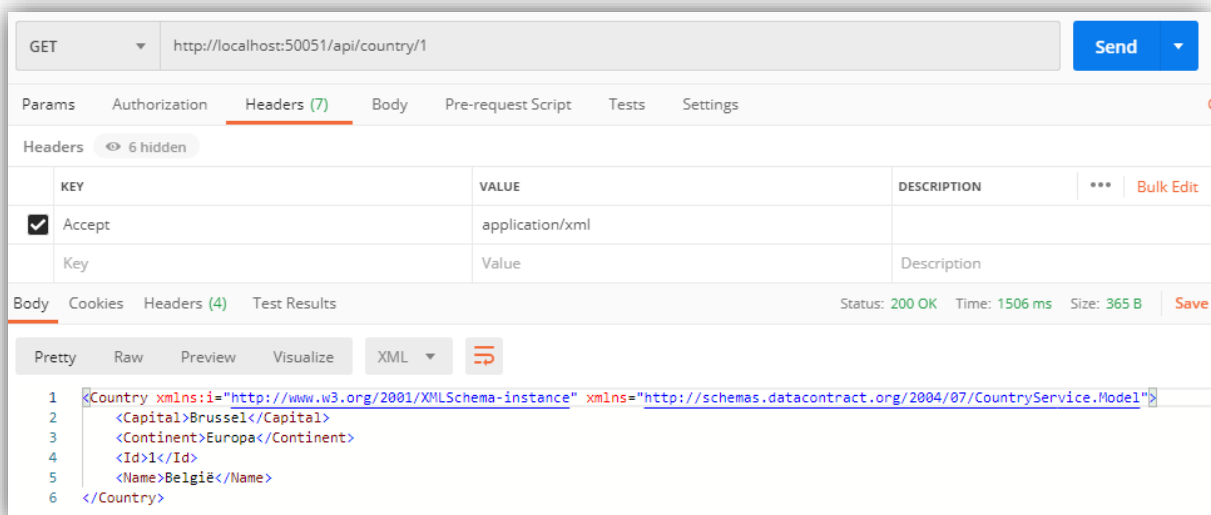
Het resultaat ziet er dan als volgt uit :



Om het nu ook mogelijk te maken om het antwoord in XML te geven moeten we ook nog de methode `AddXmlDataContractSerializerFormatters` oproepen in de `ConfigureServices` methode en dat ziet er als volgt uit :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(setup => {
        setup.ReturnHttpNotAcceptable = true;
    }).AddXmlDataContractSerializerFormatters();
    services.AddSingleton<ICountryRepository, CountryRepository>();
}
```

Als we nu expliciet vragen om het antwoord in XML te krijgen lukt dat ook.



Opmerking : voeg lege constructor toe in klasse country.

HEAD

Een HEAD request lijkt op een GET request maar geeft enkel de headers terug en niet de body. Deze request wordt voornamelijk gebruikt om te kijken of een bepaalde request een bestaand antwoord zal geven (zonder de eigenlijke data op te vragen). We kunnen hiermee bijvoorbeeld controleren of een bepaalde bron bestaat, wat interessant kan zijn als het bijvoorbeeld over een grote bron gaat aangezien het HEAD verzoek dan een stuk sneller zal zijn dan het GET verzoek. HEAD requests kunnen ook interessant zijn voor caching, indien het document is aangepast (vb LastModified / ContentLength header info is anders) kan een nieuwe versie worden afgehaald.

In code is het vrij eenvoudig om een HEAD request toe te voegen, we hoeven enkel een extra annotatie toe te voegen.

```
// GET: api/Country
[HttpGet]
[HttpHead]
0 references
public IEnumerable<Country> GetAll()
{
    return repo.GetAll();
}
```

```
[HttpGet("{id}")]
[HttpHead("{id}")]
0 references
public ActionResult<Country> Get(int id)
{
    try
    {
        return repo.GetCountry(id);
    }
    catch (CountryException ex)
    {
        return NotFound(ex.Message);
    }
}
```

HEAD	http://localhost:50051/api/country/1		
Params	Authorization	Headers (7)	Body
Query Params			
	KEY	VALUE	DESCRIPTION
	Key	Value	Description
Body	Cookies	Headers (3)	Test Results
	KEY	VALUE	
	Date	Wed, 19 Aug 2020 18:27:35 GMT	
	Content-Type	application/json; charset=utf-8	
	Server	Kestrel	

Filtering en Searching

Indien we niet alle landen wensen te selecteren, maar enkel een selectie daarvan kunnen we gebruik maken van filtering. De GetAll methode moet nu worden aangepast zodat er met een filter kan worden gewerkt.

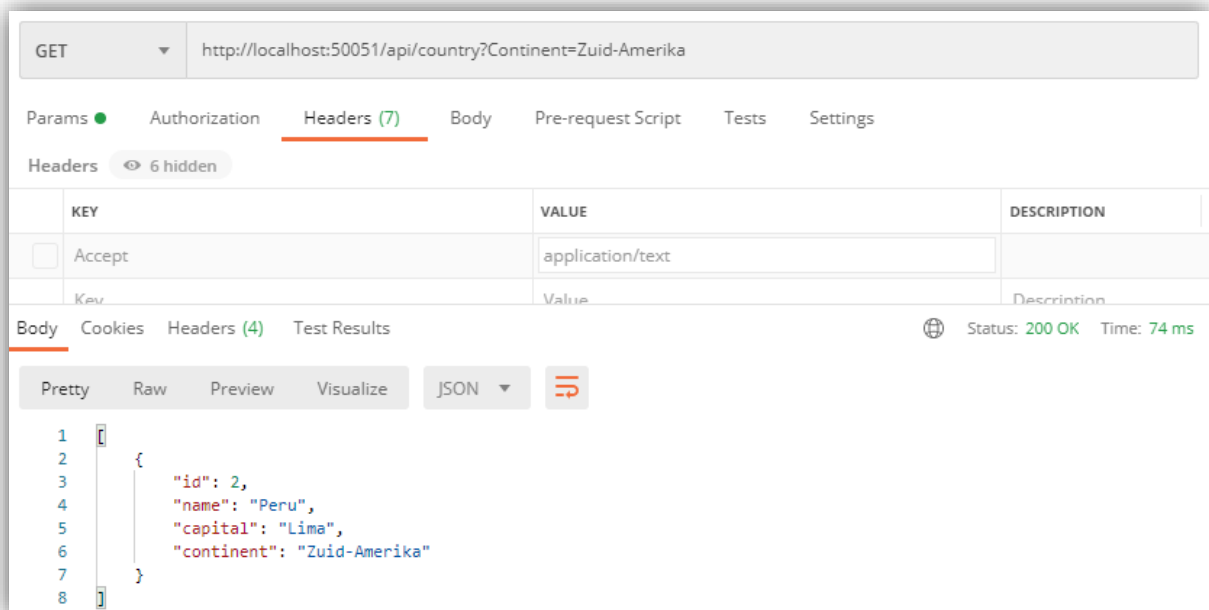
In ons voorbeeld implementeren we nu de mogelijkheid om te filteren op basis van de variabele continent. Naast de parameter en zijn type voegen we ook nog toe vanwaar deze parameter zal komen uit het GET request via een annotatie. Hier is geopteerd om de parameter uit de Query input te halen.

```
// GET: api/Country
[HttpGet]
[HttpHead]
0 references
public IEnumerable<Country> GetAll([FromQuery] string continent)
{
    if (!string.IsNullOrEmpty(continent))
        return repo.GetAll(continent);
    else
        return repo.GetAll();
}
```

Aangezien we wensen te filteren, moeten we dit ook implementeren in onze repository en voegen we volgende functie toe :

```
public IEnumerable<Country> GetAll(string continent)
{
    return data.Values.Where(x => x.Continent == continent);
}
```

Het GET request ziet er dan als volgt uit : <http://localhost:50051/api/country?Continent=Zuid-Amerika>. De query parameters worden aan de URL toegevoegd door middel van een vraagteken en een naam/waarde paar.



Wensen we meerdere parameters te gebruiken dan kan dat bijvoorbeeld als volgt :

```
// GET: api/Country
[HttpGet]
[HttpHead]
0 references
public IEnumerable<Country> GetAll([FromQuery] string continent, [FromQuery] string capital)
{
    if (!string.IsNullOrEmpty(continent))
    {
        if (!string.IsNullOrEmpty(capital.Trim()))
        {
            return repo.GetAll(continent, capital);
        }
        else
        {
            return repo.GetAll(continent);
        }
    }
    else
    {
        return repo.GetAll();
    }
}
```

```
public IEnumerable<Country> GetAll(string continent, string capital)
{
    return data.Values.Where(x => x.Continent == continent && x.Capital == capital);
}
```

GET

http://localhost:50051/api/country?Continent=Europa&Capital=Oslo

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Headers

6 hidden

	KEY	VALUE	DESCRIPTION
<input type="checkbox"/>	Accept	application/text	
	Key	Value	Description

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 755 ms

Pretty

Raw

Preview

Visualize

JSON

```
1  []
2  {
3      "id": 5,
4      "name": "Noorwegen",
5      "capital": "Oslo",
6      "continent": "Europa"
7  }
8  ]
```