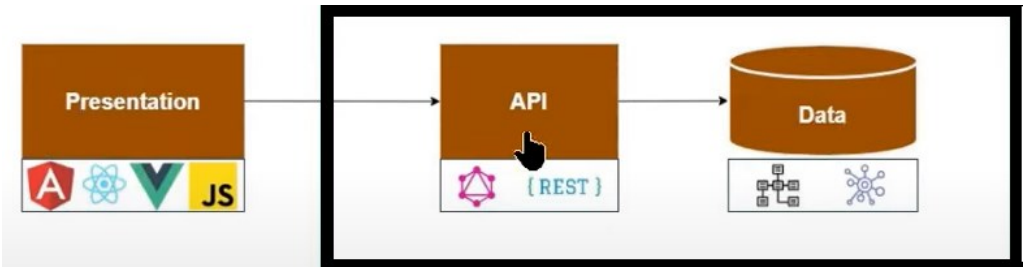
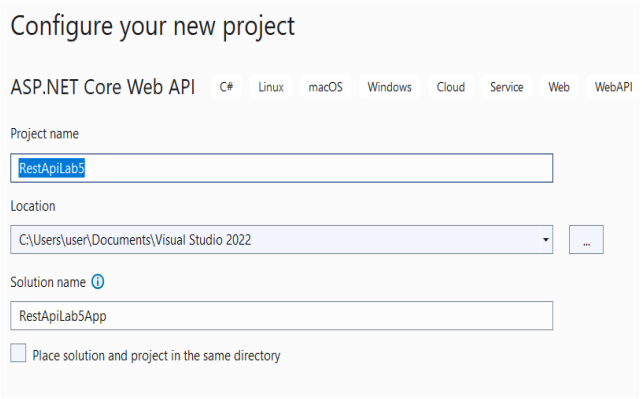
Υπηρεσίες και Συναλλαγές Ιστού – Χ. Γεωργιάδης

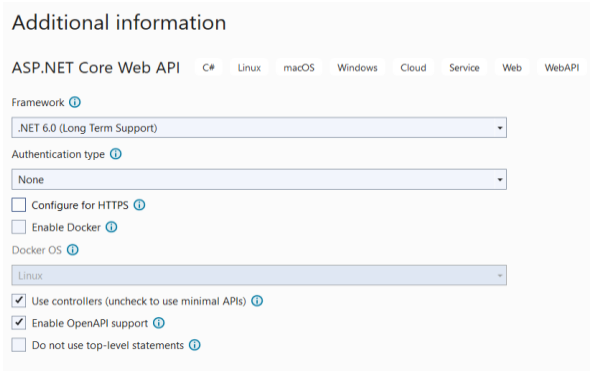
Lab5, VS_2022



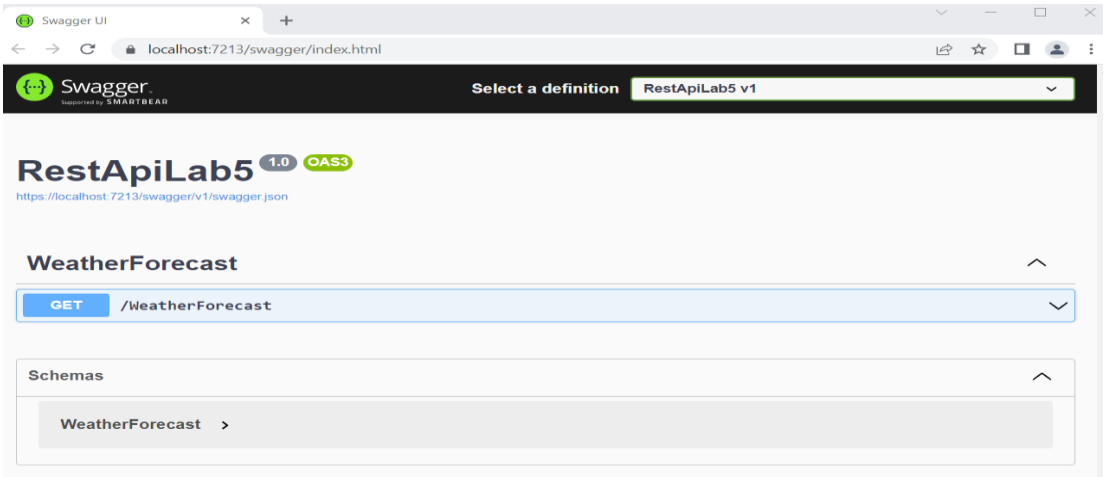
Εισαγωγή στις REST υπηρεσίες

1) File -> New Project
2) **ΠΡΟΣΟΧΗ:** Ως project template επιλέγουμε **ASP .NET Core Web API**, ονομάζουμε το project **RestApiLab5** και ονομάζουμε το solution **RestApiLab5App**





3) Αφήνουμε ως έχουν τις προκαθορισμένες ρυθμίσεις (.NET 6.0, Authentication: None, Use Controllers, Enable OpenAPI), ΕΚΤΟΣ της επιλογής HTPPS, στην οποία **αποεπιλέγουμε** το **Configure for HTTPS**,και ζητάμε **Create**
4) Κλείνουμε τη καρτέλα Overview, και ζητάμε την εκτέλεση του project μας για να δούμε τι αρχικά λειτουργεί (και τι έχει δημιουργηθεί) χωρίς οποιαδήποτε δική μας επέμβαση)

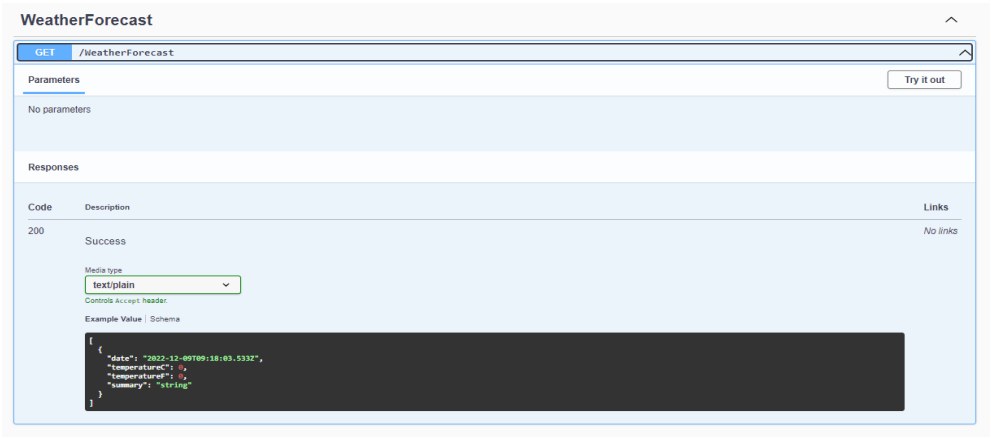


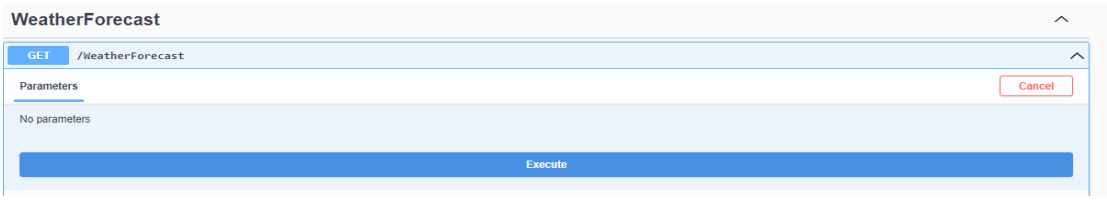
Αυτό που βλέπουμε είναι το αποτέλεσμα της **Swagger** (ή πλέον **OpenAPI** [*]), της διασύνδεσης χρήστη, και όχι μόνο, που χρησιμοποιεί το VS 2022 ως interface για τις REST YI.

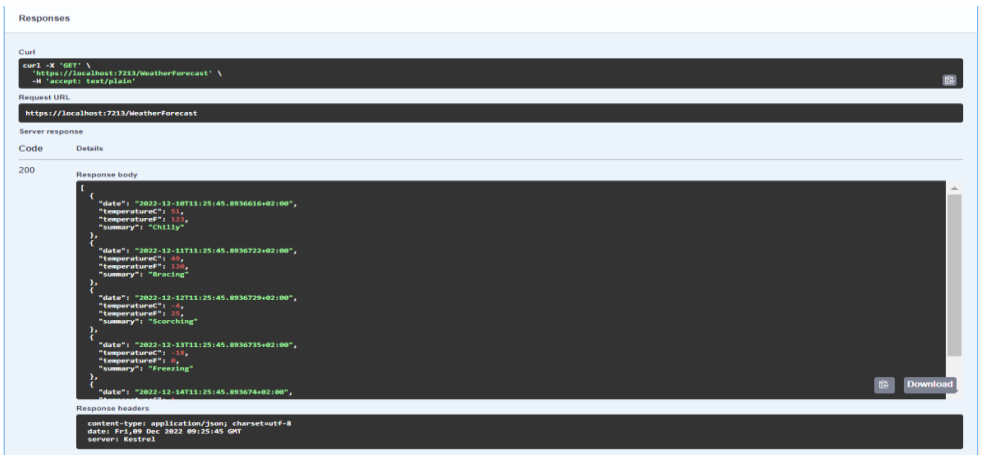
(*)The OpenAPI Specification, previously known as the Swagger Specification, is a specification for a machine-readable interface definition language for describing, producing, consuming and visualizing RESTful web services. It is an **open-source** collaboration project of the Linux Foundation. *(από wiki)*

Οι προβλέψεις καιρού παρέχονται σύμφωνα με το εμφανιζόμενο **Schema.**
Σημειώστε ότι οι REST υπηρεσίες μεταφέρουν δεδομένα χρησιμοποιώντας JSON μορφοποίηση

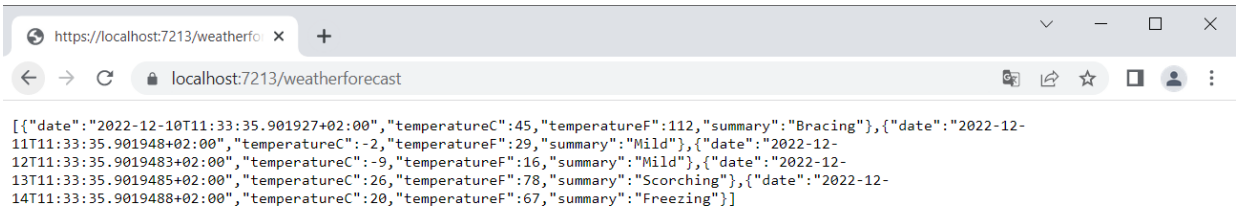
5) Πατάμε το **GET** στο WeatherForecast, μετά το **Try it out** και μετά το **Execute.** Βλέπουμε στα Responses τις προβλέψεις καιρού



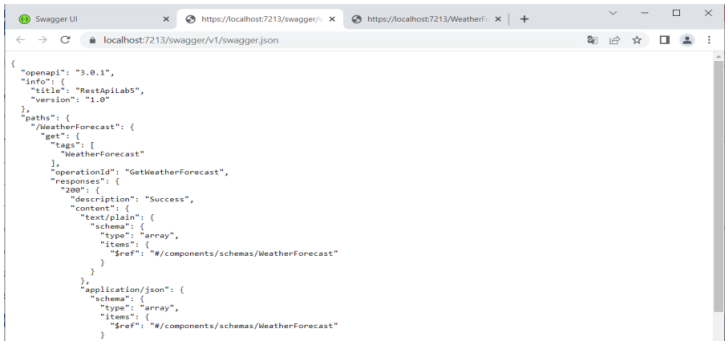




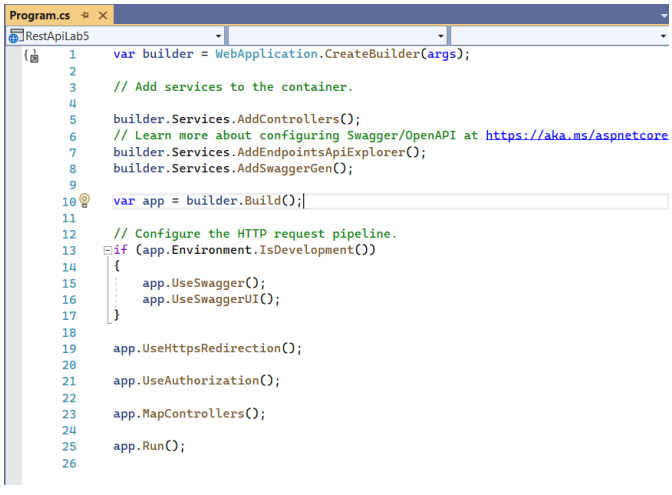
6) Ας δούμε τι παράγει η Rest υπηρεσία στο endpoint χωρίς την επίδραση του OpenAPI user interface: σε μια νέα καρτέλα του browser αντιγράψτε το URL της εφαρμογής που εκτελείται, κρατήστε μόνο το τμήμα του μέχρι το port και συμπληρώστε με το όνομα της υπηρεσίας REST. Δηλαδή για το παράδειγμά μας βάλτε στο url "**https://localhost:7213/WeatherForecast**"



7) Επίσης, αν πατήσουμε στον σύνδεσμο swagger που εμφανίζεται κάτω από το όνομα της εφαρμογής μας, μπορούμε να δούμε την τεκμηρίωση της υπηρεσίας σε JSON μορφή, σε μορφή δηλαδή που άλλες εφαρμογές μπορούν να κατανοήσουν



8) Ας δούμε όμως τον κώδικα που έχει παραχθεί μέχρι στιγμής αυτόματα, ο οποίος είναι υπεύθυνος για όλα αυτά. Σταματήστε την εκτέλεση της εφαρμογής και ανοίξτε πρώτα το αρχείο **Program.cs**. Είναι ένα τυπικό ASP.NET Core αρχείο (CreateBuilder, AddControllers, AddEndPoints, AddSwagger, UseHttpsRedirection, UseAuthorization, MapControllers, Run):

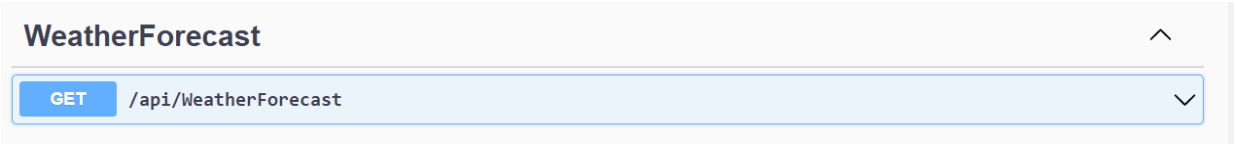


9) Μπορούμε να δούμε εάν το project μας έχει όντως ως Environment το **Development** (και όχι το Production) από το **lunchSettings** αρχείο *(Solution Explorer -> Properties -> launchSettings.json)*

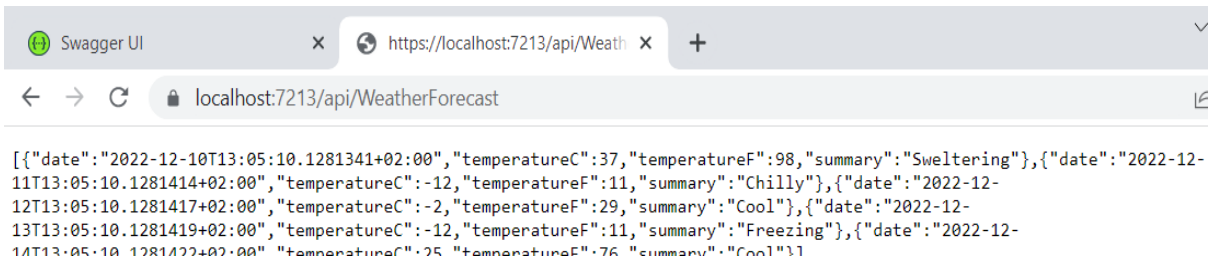
10) Ας δούμε το αρχείο του controller *(Solution Explorer -> Controllers -> WeatherForecastController.cs)*. Επειδή συνηθίζεται να υπάρχει στο path, στο URL, ένα τμήμα του με χαρακτήρες "api/", όταν μια δικτυακή εφαρμογή είναι API και όχι ιστότοπος με user interface (web forms), αλλάξτε το Route σε "api/[controller]"

```
[ApiController]
    [Route("api/[controller]")]
```

Το [controller] αντικαθίσταται όταν εκτελείται ο κώδικας από τον συγκεκριμένο WeatherForecastController, οπότε πλέον η εκτέλεση εμφανίζει:



Και η κλήση, χωρίς UI, μέσω browser, μπορεί να γίνει ως:



11) Μελετήστε τον κώδικα για την **HttpGet** στις γραμμές 21-31:

```
[HttpGet(Name = "GetWeatherForecast")]
  public IEnumerable<WeatherForecast> Get()
  {
      return Enumerable.Range(1, 5).Select(index => new WeatherForecast
      {
          Date = DateTime.Now.AddDays(index),
          TemperatureC = Random.Shared.Next(-20, 55),
          Summary = Summaries[Random.Shared.Next(Summaries.Length)]
      })
      .ToArray();
  }
```

Είναι η Get εντολή του http που καλείται, όταν σε έναν browser βάλουμε …/api/WeatherForecast

12) Στις γραμμές 9-12, ορίζεται ο πίνακας string Summaries που δίνει την δυνατότητα για τυχαίες προβλέψεις χαρακτηρισμού του καιρού (**Summaries[Random.Shared.Next(Summaries.Length)]**)

```
        private static readonly string[] Summaries = new[]
        {
          "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot", "Sweltering", "Scorching"
         };
```

13) Τυχαίες θερμοκρασίες Κελσίου παράγονται από την εντολή **TemperatureC = Random.Shared.Next(-20, 55),** ενώ η αντιστοίχιση σε θερμοκρασίες Φαρενάιτ γίνονται μέσω της κλάσης WeatherForecast, έτσι όπως φαίνεται στο αρχείο **WeatherForecast.cs,** το οποίο καθορίζει το σχήμα (**schema**) των δεδομένων πρόβλεψης καιρού. Σε αυτό το αρχείο βασίζεται ο browser για να εμφανίσει το Schema που είδαμε πριν.

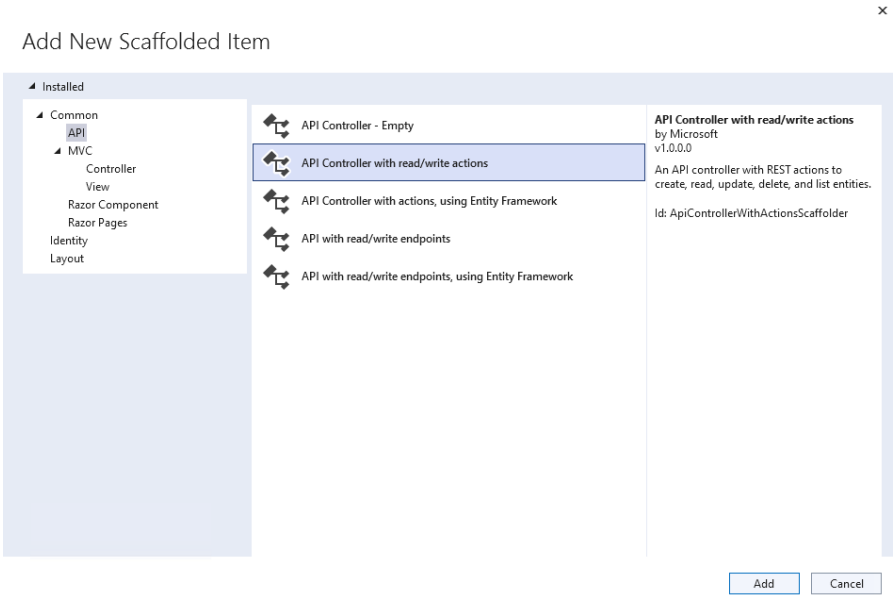
```
public class WeatherForecast
     {
         public DateTime Date { get; set; }

        public int TemperatureC { get; set; }

        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

        public string? Summary { get; set; }
  }
```

Στο Solution explorer, το sample code αρχείο WeatherForecast.cs και το sample controller αρχείο WeatherForecastController.cs μπορούμε να τα σβήσουμε, όταν θελήσουμε να προχωρήσουμε με δικό μας κώδικα.

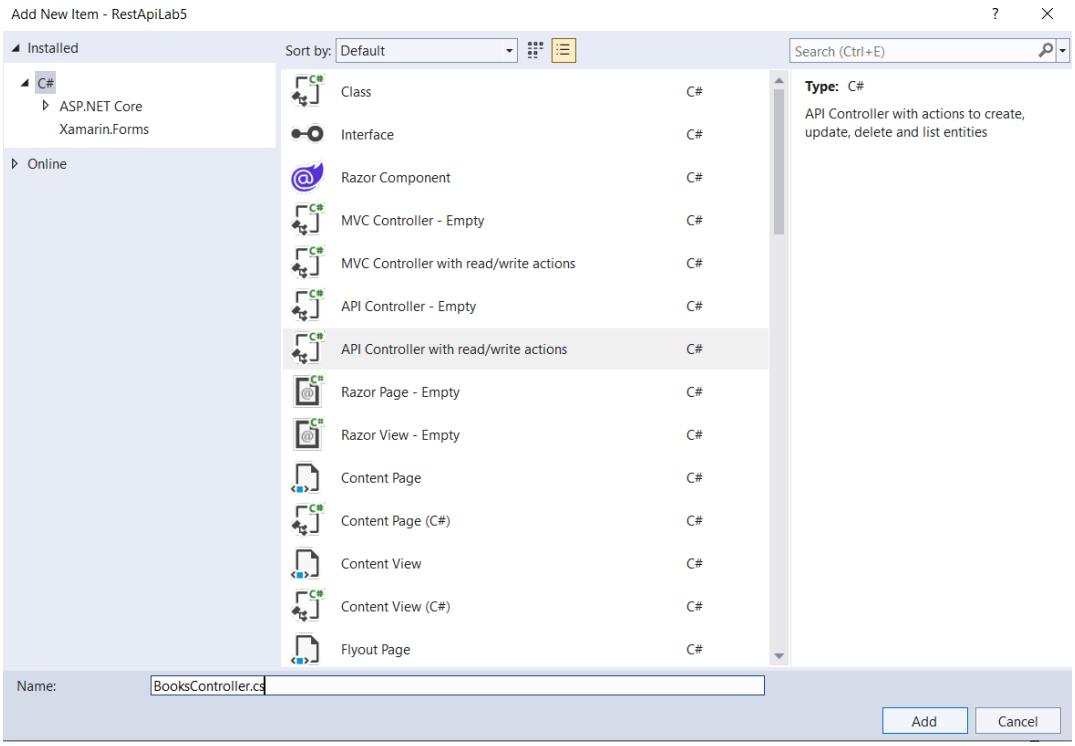
****

Ας πάμε όμως πέρα από τα όσα αυτόματα έχουν δημιουργηθεί. Ας πάμε να γράψουμε δική μας λειτουργικότητα – δικό μας κώδικα. Ας δημιουργήσουμε έναν πλήρη API Controller

14) Θα προσθέσουμε έναν νέο Controller. Με δεξί κλικ στο φάκελο Controllers, ζητάμε *Add -> Controller -> API -> API Controller with read/write actions*



Ως όνομα, έστω ότι βάζουμε **BooksController.cs**



Έχει δημιουργηθεί (**BooksController.cs**) ο ακόλουθος κώδικας, ένα καλό template, πλαίσια κώδικα, που πρέπει στη συνέχεια να συμπληρώσουμε με επιπλέον κώδικα:

```
public class BooksController : ControllerBase
    {
        // GET: api/<BooksController>
        [HttpGet]
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/<BooksController>/5
        [HttpGet("{id}")]
        public string Get(int id)
        {
            return "value";
        }

        // POST api/<BooksController>
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }

        // PUT api/<BooksController>/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/<BooksController>/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
```

Βλέπουμε ότι έχουν δημιουργηθεί τα πλαίσια για την γνωστή στις REST YI σειρά μεθόδων (**Get, Get id, Post, Put id, Delete id**). Οι τρεις από αυτές βλέπουμε ότι είναι με παράμετρο id.

15) Για διευκόλυνση, μπορούμε στα αντίστοιχα σημεία των σχολίων, να βάλουμε τον ακριβή τρόπο κλήσης των μεθόδων

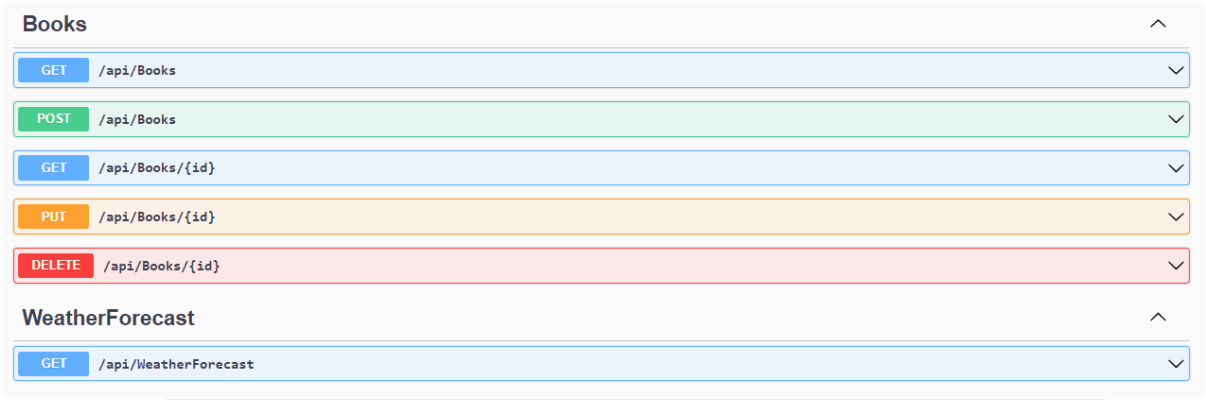
```
// GET: api/Books
// GET: api/Books/5
// POST: api/Books
// PUT: api/Books/5
// DELETE: api/Books/5
```

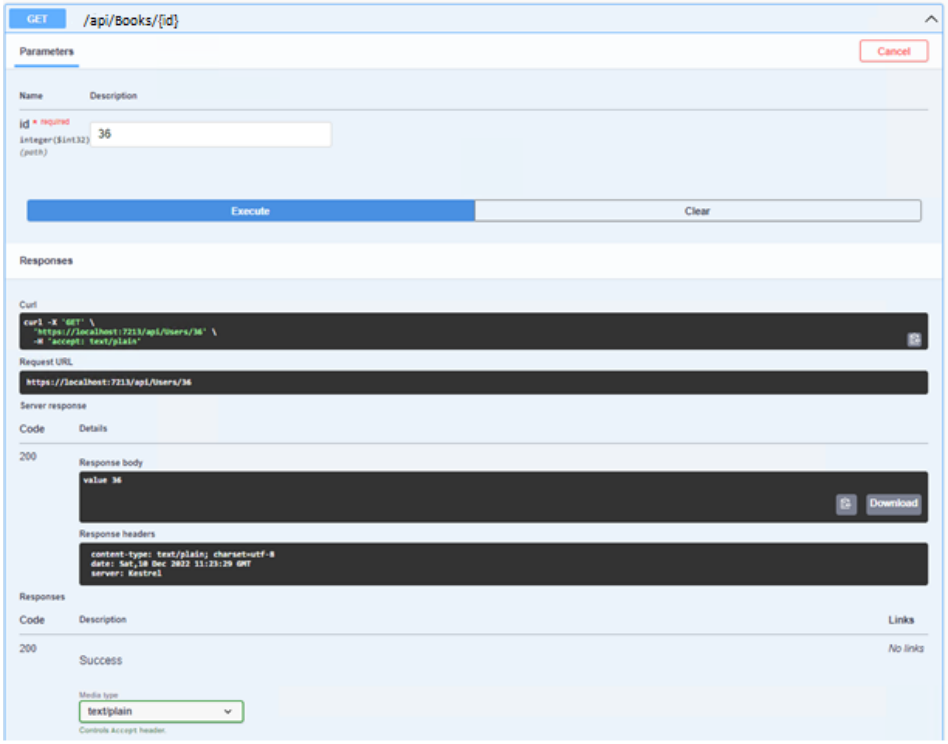
16) Ας «πειράξουμε» το πλαίσιο κώδικα για την μέθοδο GET με παράμετρο id, βάζοντας την ακόλουθη εντολή return:

```
// GET api/Books/5
      [HttpGet("{id}")]
      public string Get(int id)
     {
         return $"value {id}"; //string interpolation
```

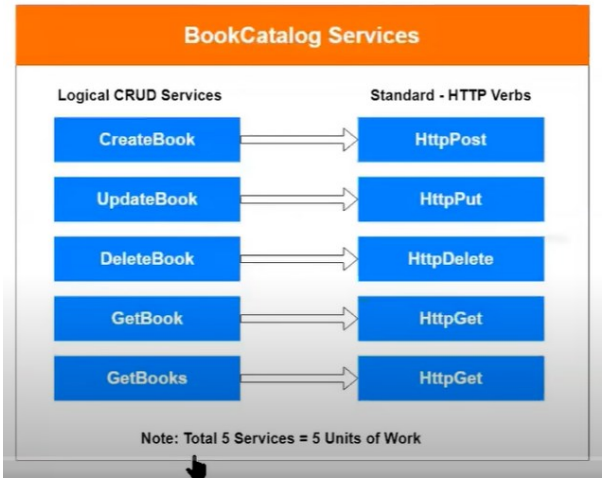
}

17) Ας ζητήσουμε εκτέλεση, και ας δοκιμάσουμε την γενική GET, την GET με παράμετρο, αλλά και τις υπόλοιπες:





Θα υποστηρίξουμε υπηρεσίες καταλογογράφησης βιβλίων, σύμφωνα με την ακόλουθη αντιστοίχιση:



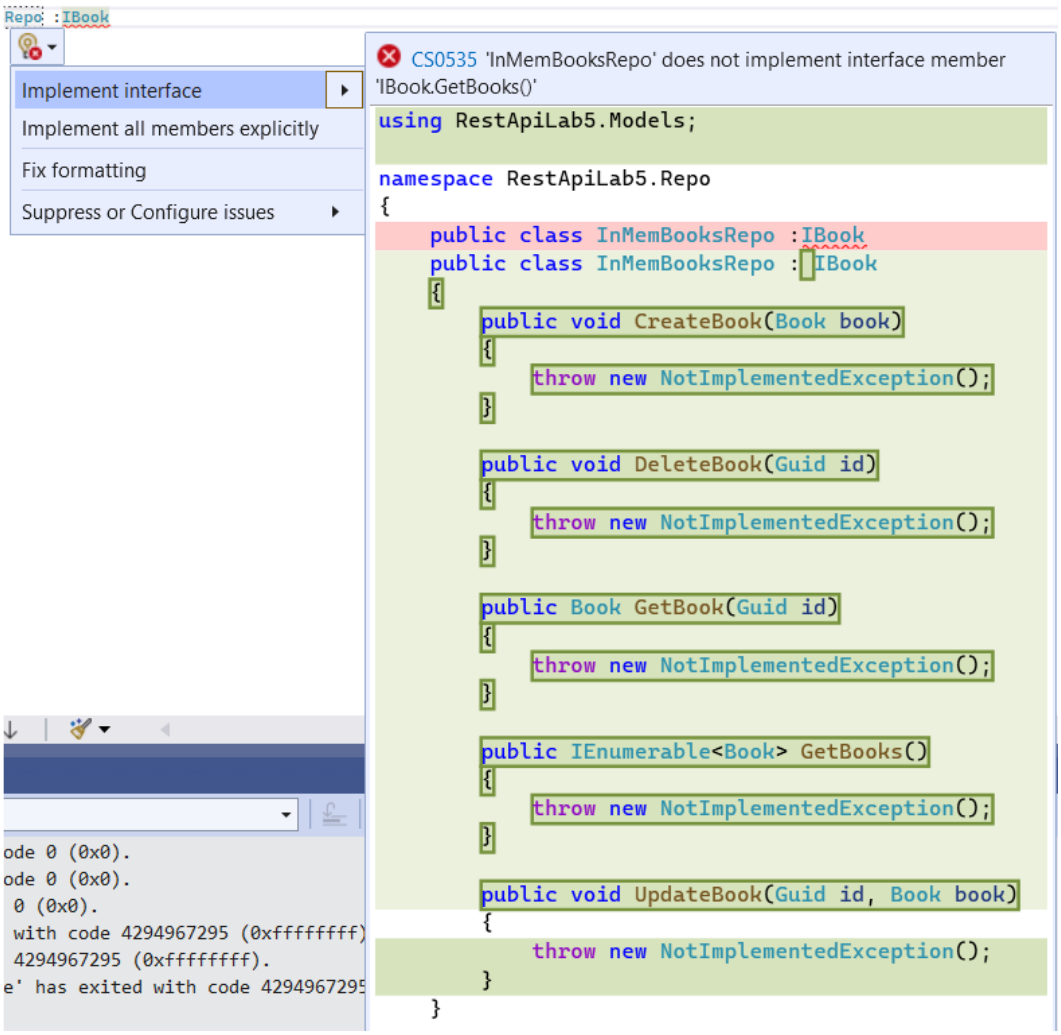
18) Δημιουργούμε στο Solution Explorer, έναν φάκελο **Models**, και μέσα σε αυτόν δημιουργούμε μια κλάση (class), ένα αρχείο **Book.cs**. Αυτό θα είναι το βασικό σχήμα (**schema**) για τα δεδομένα βιβλίου. Προσέξτε πώς ο editor του VS μας βοηθά να βάλουμε τα πεδία που θέλουμε.

```
public class Book
    {
        [Required]
        public Guid Id { get; set; }

        [Required]
        public string? Title { get; set; }

        [Required]
        [Range(0, 100)]
        public decimal Price { get; set; }
    }
```

19) Θέλουμε να ασχοληθούμε με το πού θα αποθηκεύονται τα δεδομένα των βιβλίων, και σε αυτό το θέμα είναι σημαντική η έννοια του αποθετηρίου (**repository**). Δημιουργούμε λοιπόν έναν ακόμη φάκελο στο Solution Explorer, με το όνομα **Repo**. Επειδή θέλουμε η εφαρμογή μας να είναι database agnostic (αγνωστική β.δ.), δηλαδή να είναι μια εφαρμογή που χρησιμοποιεί μια βάση δεδομένων αλλά δεν γνωρίζει και δεν έχει τρόπο (ή δεν ενδιαφέρεται να μάθει) ποια μηχανή βάσης δεδομένων τελικά θα είναι, θα δημιουργήσουμε μέσα στον φάκελο αυτό αρχικά μια διασύνδεση (**interface**), ένα αρχείο **IBook.cs**. Και πάλι προσέξτε πώς ο editor του VS μας βοηθά να βάλουμε τις πέντε λειτουργίες-μεθόδους που θέλουμε.

```
public interface IBook
    {
        public IEnumerable<Book> GetBooks();
        public Book GetBook(Guid id);
        public void CreateBook(Book book);
        public void UpdateBook(Guid id, Book book);
        public void DeleteBook(Guid id);
    }
```

20) Πάμε τώρα να δημιουργήσουμε ένα αποθετήριο. Σε αυτήν την άσκηση θα χρησιμοποιήσουμε **in-memory database**. Οπότε μέσα στον φάκελο **Repo** θα δημιουργήσουμε μια κλάση, ένα αρχείο **InMemBookRepo.cs**. Σημειώστε ότι συμπληρώνουμε στη κεφαλίδα της κλάσης ": IBook", αφού θέλουμε η διασύνδεση του αποθετηρίου μας να είναι η IBook που δημιουργήσαμε προηγουμένως. Η κόκκινη υπογράμμιση που εμφανίζεται, μας ενημερώνει ότι δεν έχουν υλοποιηθεί οι λειτουργίες που η διασύνδεση προβλέπει. Για αυτό και επιλέγουμε στο μικρό εικονίδιο επιδιόρθωσης, την επιλογή "**Implement interface**".



21) Βάζουμε αρχικά τον κώδικα για να υποστηρίξουμε τις λειτουργίες GetBooks (GET) και GetBook (Get με id):

```
public class InMemBookRepo : IBook
    {
        // δημιουργία αποθετηρίου ως λίστα
        private List<Book> _Books;

        // ο κατασκευαστής (constructor) του αντικειμένου InMemBookRepo
        public InMemBookRepo()
        {
            // καταχώριση ενός πρώτου βιβλίου
            _Books = new() { new Book
            { Id=Guid.NewGuid(), Title="Book 0", Price=10}
           };
        }

       public IEnumerable<Book> GetBooks()
        {
            return _Books;
        }

        public Book GetBook(Guid id)
        {
```

```
            var book = _Books.Where(x => x.Id == id).SingleOrDefault();
            return book;
        }

        public void CreateBook(Book book)
        {
            throw new NotImplementedException();
        }

        public void DeleteBook(Guid id)
        {
            throw new NotImplementedException();
        }

        public void UpdateBook(Guid id, Book book)
        {
            throw new NotImplementedException();
        }
    }
```

22) Πηγαίνουμε στον Controller, στο αρχείο **BooksController.cs**, και συμπληρώνουμε με τον ακόλουθο κώδικα, που αφορά τις δύο GET λειτουργίες:

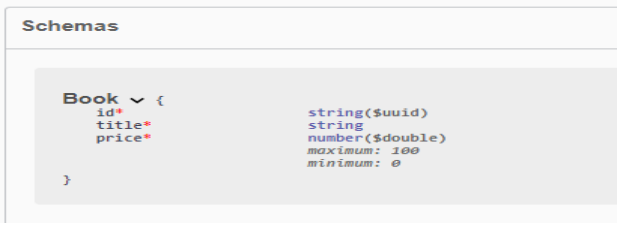
```
public class BooksController : ControllerBase
    {
        private IBook _BookRepo;

        // o constructor
        public BooksController(IBook bookRepo)
        {
            _BookRepo = bookRepo;
            // _BookRepo = new InMemBookRepo();
        }

        // GET: api/Books
        [HttpGet]
        public ActionResult<IEnumerable<Book>> GetBooks()
        {
            return _BookRepo.GetBooks().ToList();
        }

        // GET api/Books/5
        [HttpGet("{id}")]
        public ActionResult<Book> GetBook(Guid id)
        {
            var book = _BookRepo.GetBook(id);
            if (book==null)
                return NotFound();
            return book;
        }

        // POST api/Books
              [HttpPost]
              public void Post([FromBody] string value)
        {
                }
```

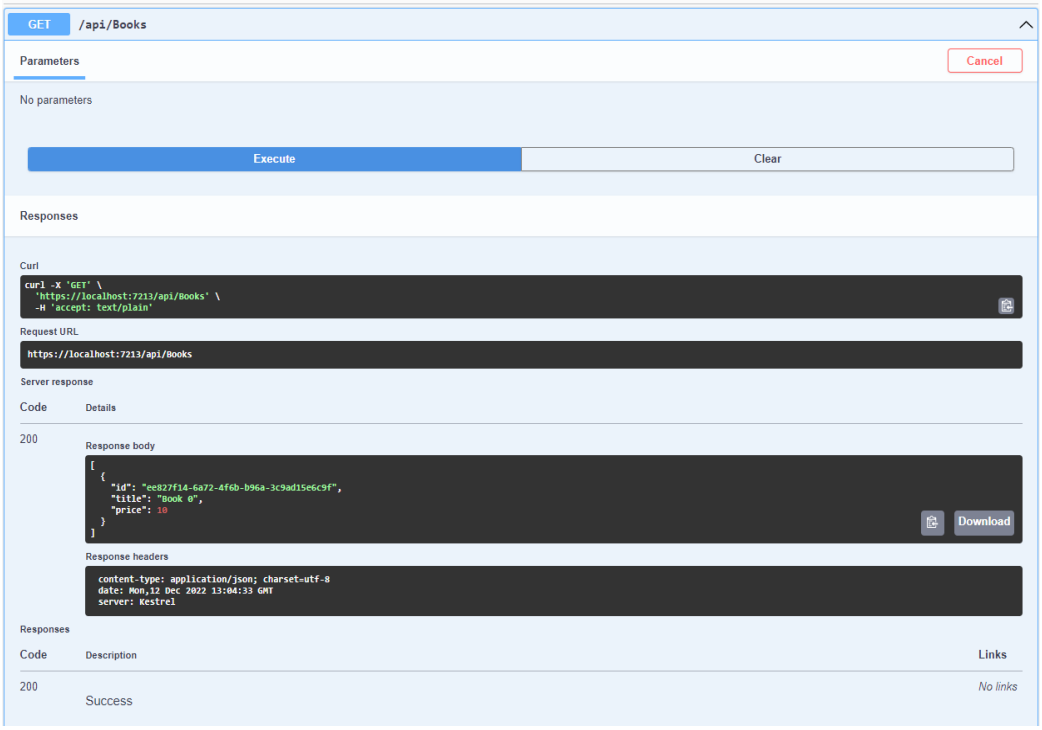
```
        // PUT api/Books/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/Books/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
```

23) Για να αρχικοποιεί η asp.net μόνο μια φορά το πρώτο βιβλίο χρειάζεται η ακόλουθη ρύθμιση στο αρχείο **Program.cs**, πριν την **AddControllers();**

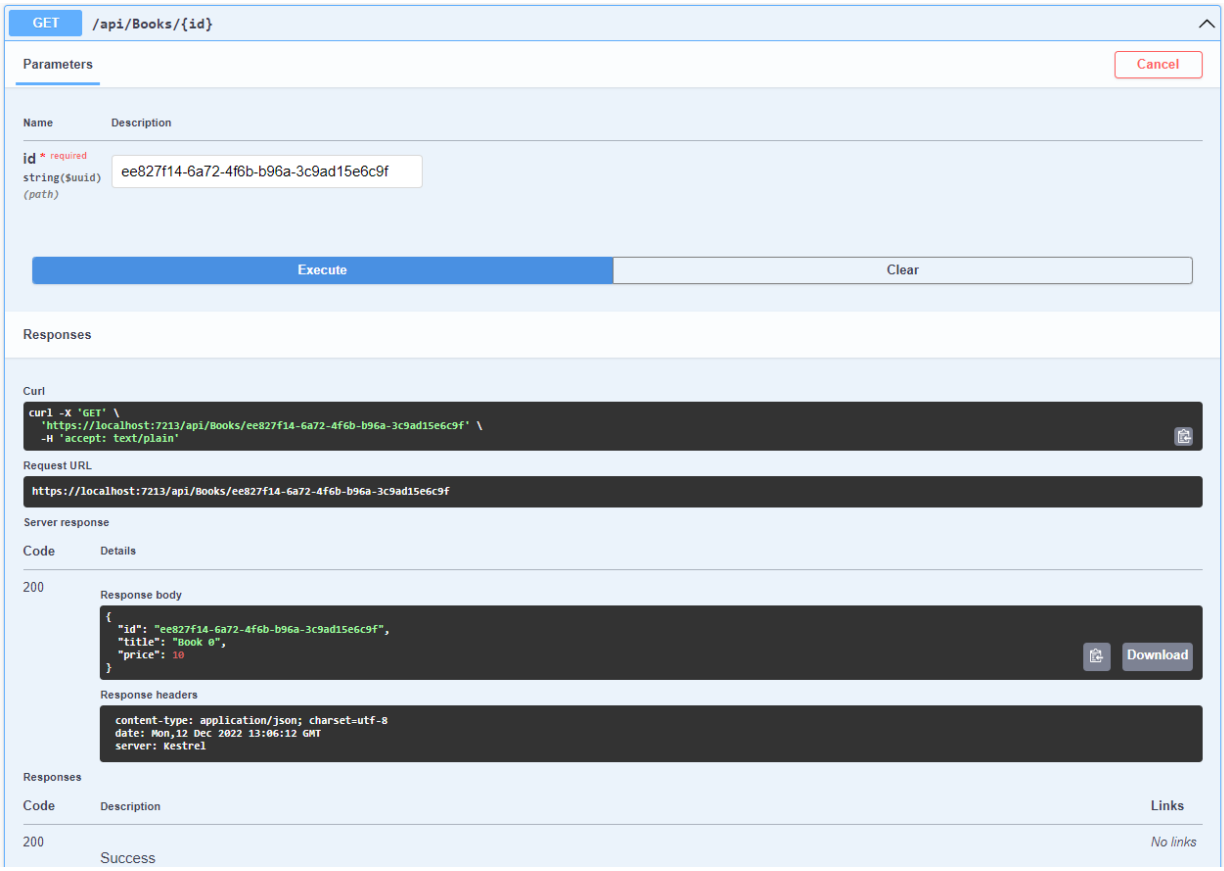
```
builder.Services.AddSingleton<IBook, InMemBookRepo>();
```

24) Ας ζητήσουμε εκτέλεση, και ας δούμε τα όσα έως τώρα έχουμε υλοποιήσει. Μπορούμε να δούμε τη περιγραφή (schema) του βιβλίου, και να δοκιμάσουμε την γενική GetBooks και την GetBook με παράμετρο:





25) Αναζητείστε με βάση το id του ενός βιβλίου που έχει καταχωρηθεί, και εντοπίστε το:



26) Για την CreateBook - θέλουμε όποιος καταχωρεί βιβλίο να μην μπορεί να βάλει δικό του id, αλλά το id να δημιουργείται από τον server. Θα χρειαστούμε λοιπόν ένα δεύτερο σχήμα (schema) που δεν θα περιέχει το πεδίο id. Δημιουργούμε στο φάκελο **Models**, μια κλάση, ένα αρχείο με το όνομα **CreateOrUpdateBookSchema.cs**, αλλάζουμε τη λέξη class με τη λέξη record και βάζουμε όλα τα πεδία που είχε το αρχικό σχήμα, εκτός του id:

```
public record CreateOrUpdateBookSchema
    {
        [Required]
        public string? Title { get; set; }
        [Required]
        [Range(0, 100)]
        public decimal Price { get; set; }
    }
```

27) Πηγαίνουμε στον **BooksController.cs** και εμπλουτίζουμε τον κώδικα για την υποστήριξη της μεθόδου **POST**:

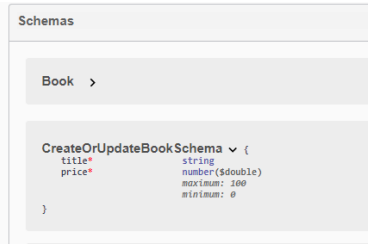
```
// POST api/Books
[HttpPost]
public ActionResult CreateBook(CreateOrUpdateBookSchema book)
        {
            var mybook = new Book();
            mybook.Id=Guid.NewGuid();
            mybook.Title=book.Title;
            mybook.Price=book.Price;
```

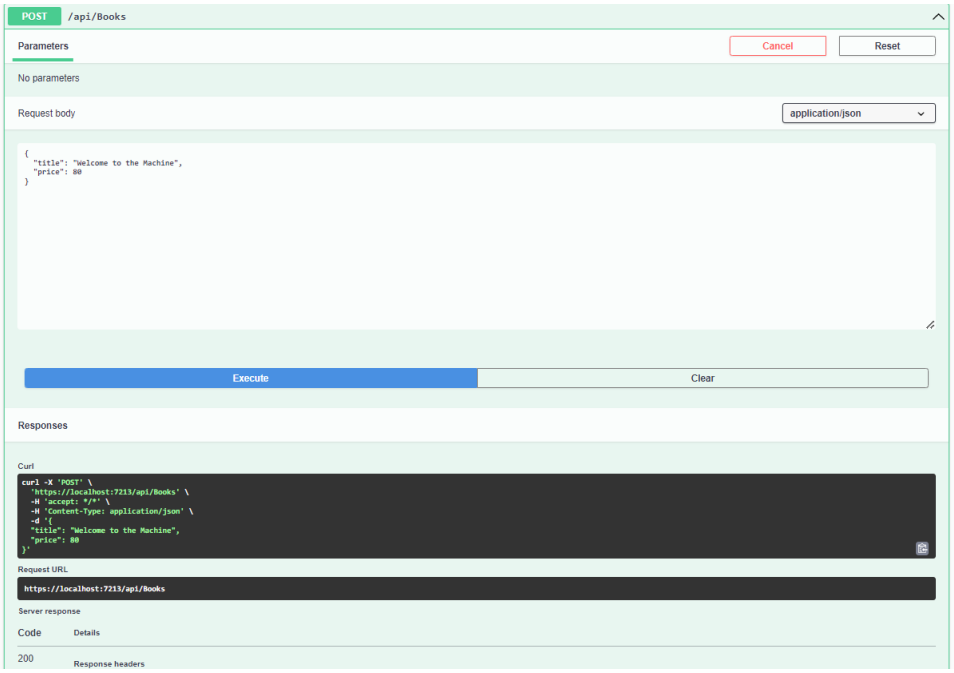
```
            _BookRepo.CreateBook(mybook);
            return Ok();
        }
```

28) Πηγαίνουμε στο αρχείο **InMemBookRepo.cs** και συμπληρώνουμε τον κώδικα για την μέθοδο **CreateBook**:

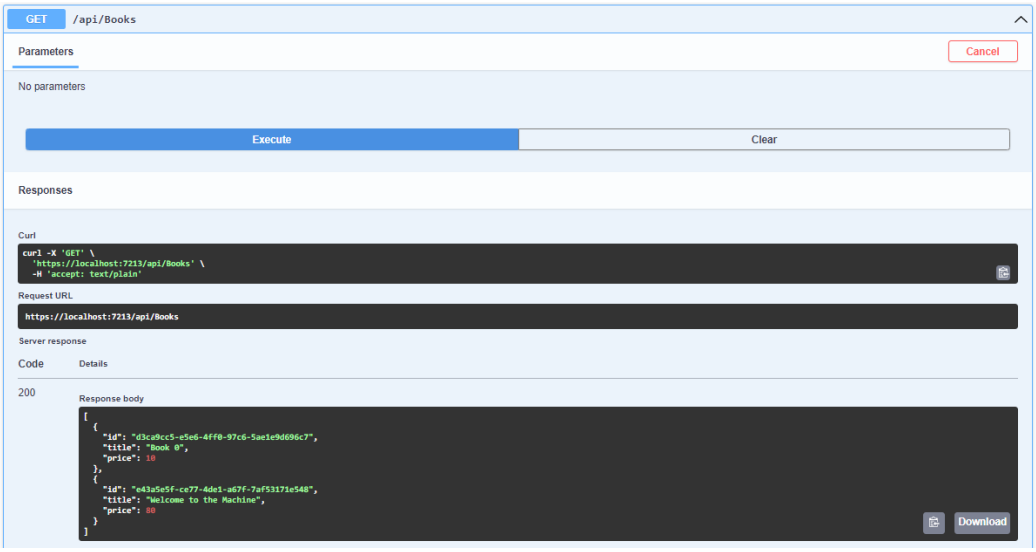
```
        public void CreateBook(Book book)
            {
                _Books.Add(book);
            }
```

29) Εκτελέστε και δοκιμάστε στον browser την μέθοδο **POST**. Παρατηρήστε και το νέο schema που εμφανίζει ο browser:





Πέρα από τον κωδικό επιτυχίας 200 που λαμβάνουμε, αν ζητήσουμε την GetBooks θα δούμε πλέον ότι το βιβλίο που καταχωρήσαμε με την POST, συμπεριλαμβάνεται στη λίστα των βιβλίων:



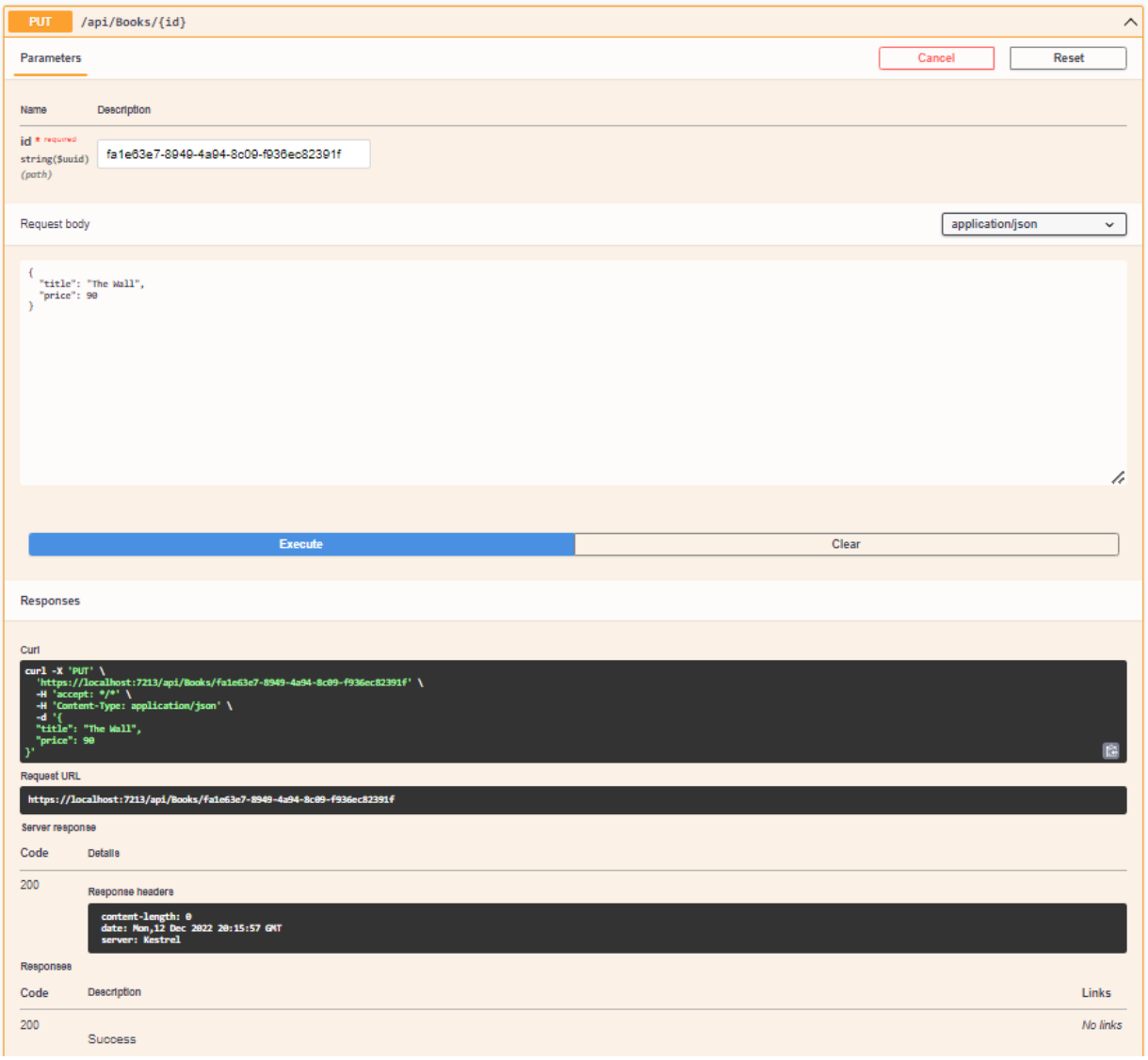
30) Για την UpdateBook – ο απαιτούμενος κώδικας είναι παρόμοιος με αυτόν της CreateBook. Πηγαίνουμε στον **BooksController.cs** και εμπλουτίζουμε τον κώδικα για την υποστήριξη της μεθόδου **PUT:**

```
// PUT api/Books/5
[HttpPut("{id}")]
public ActionResult UpdateBook(Guid id, CreateOrUpdateBookSchema book)
        {
            var mybook = _BookRepo.GetBook(id);
            if (mybook == null)
                return NotFound();
            mybook.Title = book.Title;
            mybook.Price = book.Price;

            _BookRepo.UpdateBook(id,mybook);
            return Ok();
        }
```

31) Πηγαίνουμε στο αρχείο **InMemBookRepo.cs** και συμπληρώνουμε τον κώδικα για την μέθοδο **UpdateBook:**

```
public void UpdateBook(Guid id, Book book)
        {
            var bookIndex = _Books.FindIndex(x => x.Id == id);
            if (bookIndex>-1)
                _Books[bookIndex]=book;
        }
```

32) Εκτελέστε και δοκιμάστε στον browser την μέθοδο **PUT:** πρώτα μέσω της GET αντιγράψτε το id από καταχωρημένο βιβλίο και μετά μέσω της PUT αλλάξτε για το βιβλίο με το συγκεκριμένο id τον τίτλο ή και την τιμή του:



33) Αν βάλετε ένα id που δεν υπάρχει, τότε θα λάβετε μήνυμα «Not found”:

34) Για την DeleteBook – ο απαιτούμενος κώδικας είναι παρόμοιος με αυτόν της UpdateBook. Πηγαίνουμε στον **BooksController.cs** και εμπλουτίζουμε τον κώδικα για την υποστήριξη της μεθόδου **DELETE**:

```
// DELETE api/Books/5
```

```
[HttpDelete("{id}")]
public ActionResult DeleteBook(Guid id)
{
    var mybook = _BookRepo.GetBook(id);
    if (mybook == null)
        return NotFound();

    _BookRepo.DeleteBook(id);
    return Ok();
}
```

35) Πηγαίνουμε στο αρχείο **InMemBookRepo.cs** και συμπληρώνουμε τον κώδικα για την μέθοδο **DeleteBook**:

```
public void DeleteBook(Guid id)
    {
        var bookIndex = _Books.FindIndex(x => x.Id == id);
        if (bookIndex > -1)
            _Books.RemoveAt(bookIndex);
    }
```

36) Εκτελέστε και δοκιμάστε στον browser την μέθοδο **DELETE**: πρώτα μέσω της PUT καταχωρήστε ένα βιβλίο, μετά μέσω της GET αντιγράψτε το id από ένα από τα καταχωρημένα βιβλίο και μετά μέσω της DELETE σβήστε το βιβλίο με το συγκεκριμένο id: