Udemy course: Ultimate ASP.NET pt. 4

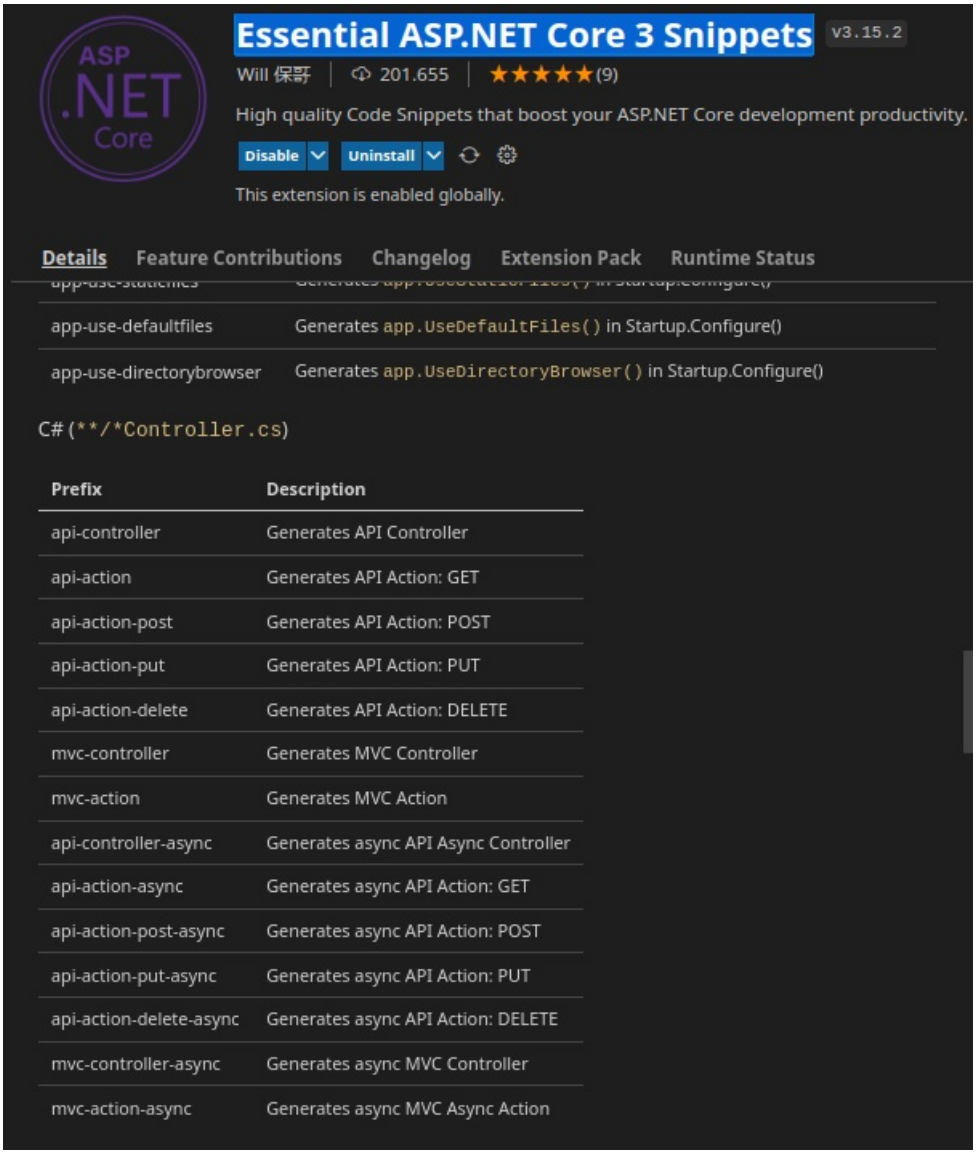
Scaffolding API Functionality

- scaffolding allows us to create easily controllers
- in particular allow easy interaction with our tables
- code which is boilerplatte, out of the box

#19, Scaffolding Controlers and Actions

- when doing development
  - either start with the tables
  - or the functionality around it, which has the lowest number of foreign keys and then work your way up
- so we will start with countries
  - since it is the least tethered - least related one
- in Visual Studio:
  - in the solution explorer
    - Add Controller...
  - API -> API Controller with actions, using Entity Framework
- in Visual Studio Code:
  - new C#
    - Controller API
  - or better - more like the Visual Studio API Controller above:
    - have extension of C# code snippets installed
    - in the c sharp file of the controller enter
    - asp-api-controller and enter
  - but Note with that:
    - even then: it is not exactly the same as in Visual Studio
    - like EntityFrameworkCore is not used
    - the get request is not async, no await
    - and no injection with the database context!
  - another extension with snippets:
  - Essential ASP.NET Core 3 Snippets
  - Snippet Prefixes

Prefix	Description
nc-	General .NET Core Snippets
anc3-	ASP.NET Core 3 Snippets
api-	ASP.NET Core Web API Snippets
mvc-	ASP.NET Core MVC Snippets
services-	ASP.NET Core Snippets in `Startup.cs`
app-	ASP.NET Core Snippets in `Startup.cs`
middleware-	ASP.NET Core Middleware Snippets
signalr-	ASP.NET Core SignalR Snippets
grpc-	ASP.NET Core gRPC Snippets
ef-	Entity Framework Core Snippets

- 

  - with this use api-controller-async for example
  - and then add context (dependency injection) by hand
  - so add/change constructor:
    - ```
      private readonly HotelListingDbContext _context;

      public CountriesController(HotelListingDbContext context)

      {

      _context = context;

      }
      ```
- differences api/[controller] vs just [controller]
  - we will see when we are testing
- the attribute ApiController is afixed, but we inherit from ControllerBase
- code sample of our controller:
      ```

```
namespace HotelListing.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class CountriesController : ControllerBase
    {
        private readonly HotelListingDbContext _context;

        0 references
        public CountriesController(HotelListingDbContext context)
        {
            _context = context;
        }

        // GET: api/Countries
        [HttpGet]
        0 references
        public async Task<ActionResult<IEnumerable<Country>>> GetCountries()
        {
            return await _context.Countries.ToListAsync();
        }
```

- the selected parts: we call we inject our db context into our controller
- for that to work we had to:
  - register the DbContext in Program.cs!
  - with this we can inject it almost anywhere in our program
  - so if you want to interact with the database in a controller or you want to use another class to interact with the database, you can easily do that
- with this injection way: we don't have to declare a new instance of db instance, whenever we have a new class
- one of the SOLID principles!
- we don't have to instanciate a new DB context every single time!
- when it finishes with the database operations it will just destroy the database instance in the background - we are not in charge of it! It is also far more efficient and saves memory time.

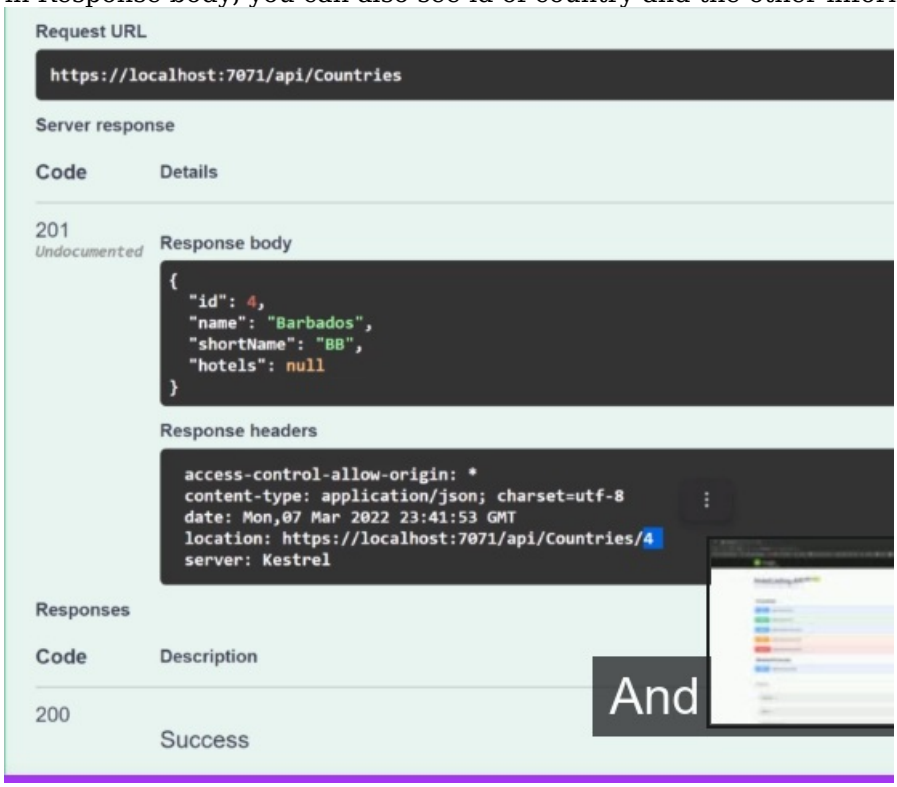
#20. Test and Understanding POST Endpoint

- in this section we will test our POST Endpoint
  - both with Swagger and with Postman
- our POST example - created with Scaffolding in Visual Studio:

```
// POST: api/Countries
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPost]
0 references
public async Task<ActionResult<Country>> PostCountry(Country country)
{
    _context.Countries.Add(country);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetCountry", new { id = country.Id }, country);
}
```

- why this HttpPost attribute?
  - it "Identifies an action that supports the HTTP POST method"
  - so it is a declaration to the controller; that whenever an POST request comes in
- Note about the terminology:
  - the method PostCountry is called an Action in MVC terms
  - api/Countries is the address; POST the request type
  - you return an ActionResult and an object of type Country
- on a POST request
  - go to the Countries table, and then add a country
  - Entity framework is one of the flagship object relation mapper for .NET
  - Countries add queues it up, and the save basically execute it
  - the return result includs the URL to get to that object ("GetCountry" here)
- test using swagger

    - execute the server
    - Swagger gives an example what is expected when you creating a country
    - you can fill in a country with any number of hotels
    - to give more than one, just comma separate - like it is done in JSON
    - and execute it to add that country
        - if you get a 201 code as response, all was well and the Country should be added
        - if you get a response status 500, there was a server error; possibly no connection to the database
        - you also see as request URL, the following URL: https://localhost:7213/api/Countries
        - Note: the route was defined in the beginning of the controller class!
        - [redacted]
        - in Response body, you can also see id of country and the other information
        - 

- what is wrong/ or subomptimal in testing with Swagger
    - we are not supposed of creating the ID when creating a request
    - when the id would already exist, you would also get a error code of 500, with "error while saving the entity changes" in response body
    - however with 0 as id it is fine; it will autoincrement
    - out of the box, testing works pretty well in Swagger
- testing with Postman
    - we can copy and paste the request url, and the request raw text from Swagger
    - the request text we can put below Body->raw and set JSON!
    - Note: logging is running so we can also use that
        - either via the file logging, terminal output or via Seq
        - following error appears:
        - DbUpdateException: An error occurred while saving the entity changes. See the inner exception for details. ---> Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity column in table 'Countries' when IDENTITY_INSERT is set to OFF.
- advantage of using Postman
    - the request will actually be saved
    - so we can do multiple tests more easily with the same requests
    - for longer troubleshooting sessions this is more convenient
        - in Swagger whenever you reload it, the request is lost
    - Note: when you get error code 415: Unsupported Media Type: check whether you selected JSON!
-

#21 Test and Understand GET endpoints

- in our countries controller API we got two GET requests
    - one without id

    - one with
- the more simple GET method without id:
    - ```
      // GET: api/Countries
      [HttpGet]
      0 references
      public async Task<ActionResult<IEnumerable<Country>>> GetCountries()
      {
          // Select * from Countries
          return await _context.Countries.ToListAsync();
      }
      ```
    - with the await we know, that we expect a list
    - another possiblity is to wrap it in a OK method - to have it a bit more explicit with the return type
    - OK is giving the Statuscode 200
        - ```
          return Ok(await _context.Countries.ToListAsync());
          OkObjectResult ControllerBase.Ok(object? value) (+ 1 overload)
          Creates an OkObjectResult object that produces an StatusCodes.Status200OK response.
          Returns:
          The created OkObjectResult for the response.
          GET: api/
          ```
        - or for improvement of readability:
          ```
          var countries = await _context.Countries.ToListAsync();
          return Ok(countries);
          ```
- the other GET method for querying an ID:
    - ```
      [HttpGet("{id}")]
      0 references
      public async Task<ActionResult<Country>> GetCountry(int id)
      {
          var country = await _context.Countries.FindAsync(id);

          if (country == null)
          {
              return NotFound();
          }

          return country;
      }
      ```
    - Note: in case no entry found, it will return error code 404
    - we can also warp the return code in an Ok():
      ```
          return Ok(country);
      }
      ```
- Note: if you change the attribute of the method from
    - ```
      [HttpGet("{id}")]
      ```
    - to just:
    - ```
      [HttpGet]
      ```
        - you would get following error in Swagger:
        - "Failed to load API Definition"
    - when you execute that method for example via the API you will get:
    - an ambiguous match exception
    - the API request URL would become the same by the attributetag change - but two equal URL don't work
- Note: you can modify the attribute tag to allow more complex query urls:
    - ```
      // GET: api/Countries/5
      [HttpGet("{id}/hotelId/{hotelId}")]
      ```
    - to query not only country but additionally for an hotel id
- about terminology:
    - template equally use to attribute tag
- so from that section you should understand
    - the importance of the templates and how these work
    - how the two get methods work
    - modify the template to have advanced queries

#22: Test and Understand PUT Endpoint

- PUT request is usually meant to do an update

- the PUT needs the id and also the object
- our example PUT action:
  -
```
// PUT: api/Countries/5
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPut("{id}")]
0 references
public async Task<IActionResult> PutCountry(int id, Country country)
{
    if (id != country.Id)
    {
        return BadRequest();
    }

    _context.Entry(country).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!CountryExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```
- PUT:
  - replaces existing data with new data
- in case the id does not belong to country id
  - a 400 status code is returned
- we can add messages to BadRequests also
  - just add a string as argument
  - e.g. "Invalid Record Id"
- every entity in Entity Framework has an Entity State
- example on that change:
  -
```
_context.Countries.Add(country);
```
  - in this line, the entity get the state Add
  - so when we save the country it knows, it should be added
  - the other states:
  -
```
EntityState.
Added
Deleted
Detached
Modified
Unchanged
```
  - in the PUT example: it is a Modified state
- Note about the exception handling in this action:
  - you of course should not just throw!
  - only if you want to kill the runtime
- testing the PUT action in Swagger
  - Swagger gives an example how the output is supposed to look like
  - so that is an advantage to Postman
- about CountryExists:
  - that is found at the bottom of the controller:
  -
```
private bool CountryExists(int id)
{
    return _context.Countries.Any(e => e.Id == id);
}
```

#23: Test and Understand DELETE Endpoint

- our DELETE action:

- ```
  // DELETE: api/Countries/5
  [HttpDelete("{id}")]
  0 references
  public async Task<IActionResult> DeleteCountry(int id)
  {
      var country = await _context.Countries.FindAsync(id);
      if (country == null)
      {
          return NotFound();
      }

      _context.Countries.Remove(country);
      await _context.SaveChangesAsync();

      return NoContent();
  }
  ```
- basically very simple
- just like our PUT and GET action
- the same URL with the ID at the end
- when it is not found - e.g. ID was already deleted before - status code is 404
  ```