REST API

HO
GENT

Content

- Demo case introduction
- What is REST?
- Building the Recipe API
- Adding token based authorization
- Publish on Azure

- Software
  - Visual Studio 2019 with latest updates
  - .NET Core 3.x

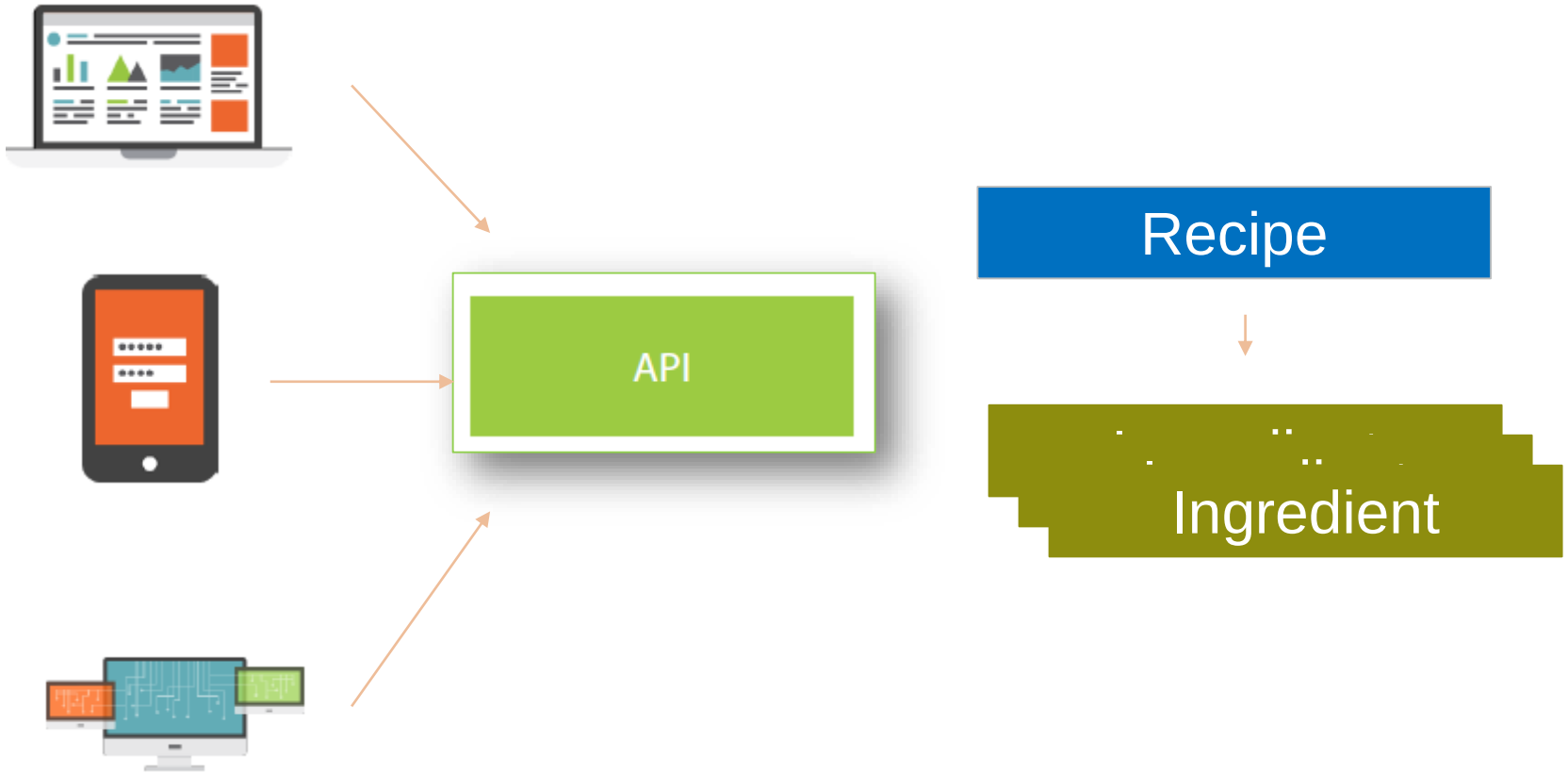
HO
GENT

Demo case introduction

Recipe Api

HO
GENT

Say Hello to “The Recipe API”



HO
GENT

Requirements

- API needs to be consumable from different types of clients (standards!)
- API needs to be friendly to consume (uniform interface)
- API needs to support CRUD operations
- …

- We'll learn how to build a REST-ful API that fits these for multiple (cross-platform) clients

HO
GENT

What is REST?

HO
GENT

What is REST?

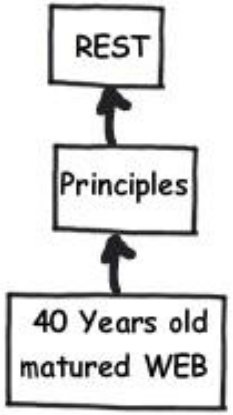
- REST (Representational State Transfer) was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.

> Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.
>
> — Roy Fielding

http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

What is REST?

- REST is an architecture style for designing networked applications.
  - REST is nothing but using the current features of the “Web” in a simple and effective way
    - 40 years old matured and widely accepted HTTP protocol.
    - Standard and Unified methods like POST, GET, PUT and DELETE.
    - Stateless nature of HTTP protocol.
    - Easy to use URI (Uniform resource identifier) to format to locate any web resource.

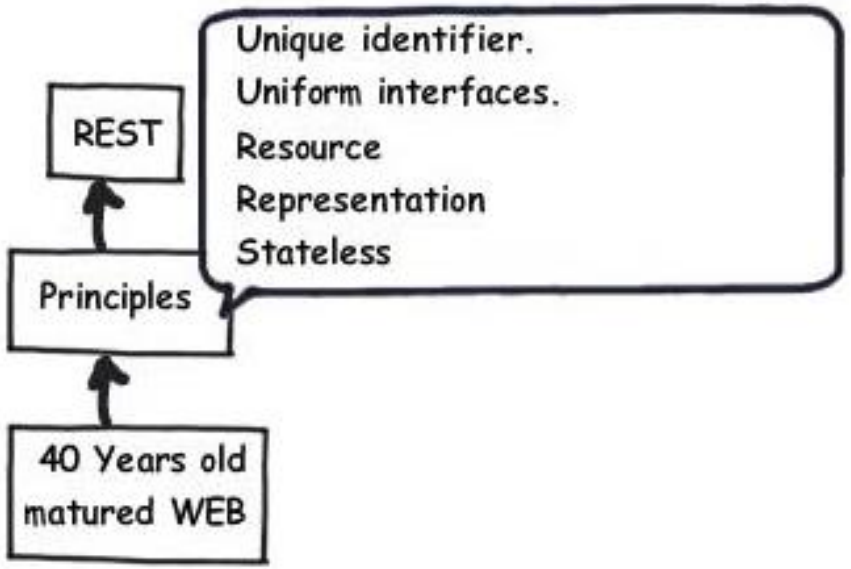


- REST leverages these amazing features of the web with some constraints

HO GENT

What is REST?

- REST is an architecture style for designing networked applications.
  - 7 principles of REST

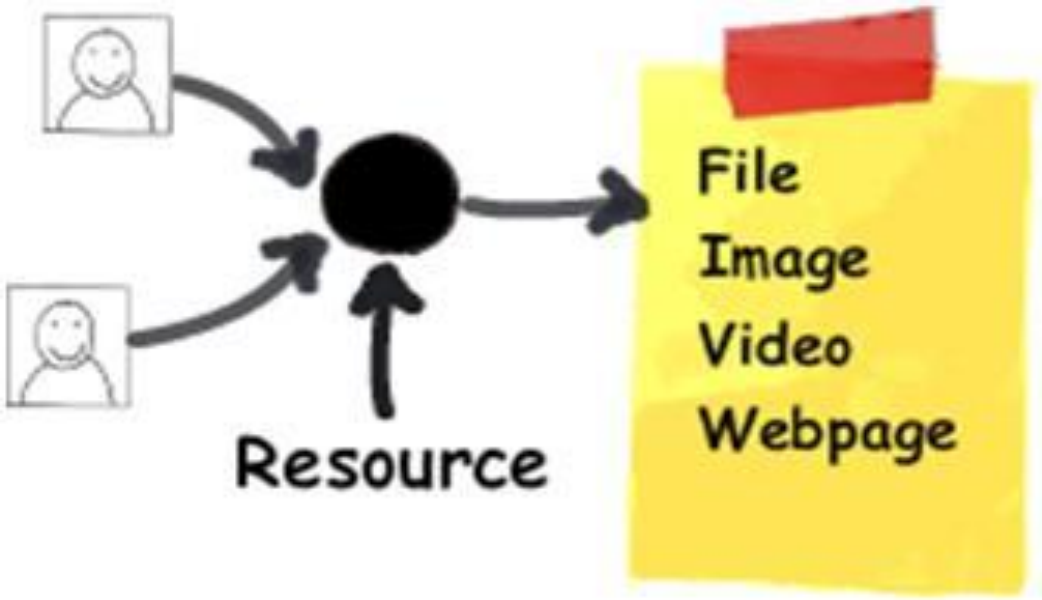


Shivprasad koirala. (n.d.). *Implementing 5 important principles of REST using WCF Services*. Retrieved from http://www.codeproject.com/Articles/283550/Implementing-important-principles-of-REST-using

HO GENT

What is REST?

- Principle 1: Everything is a **Resource**



www.hogent.be/images/logo.gif (Image resource)
www.hogent.be/students/1001 (Dynamically pulled resource)
www.hogent.be/videos/v001 (Video resource)
www.hogent.be/home.html (Static resource)

HO GENT

What is REST?

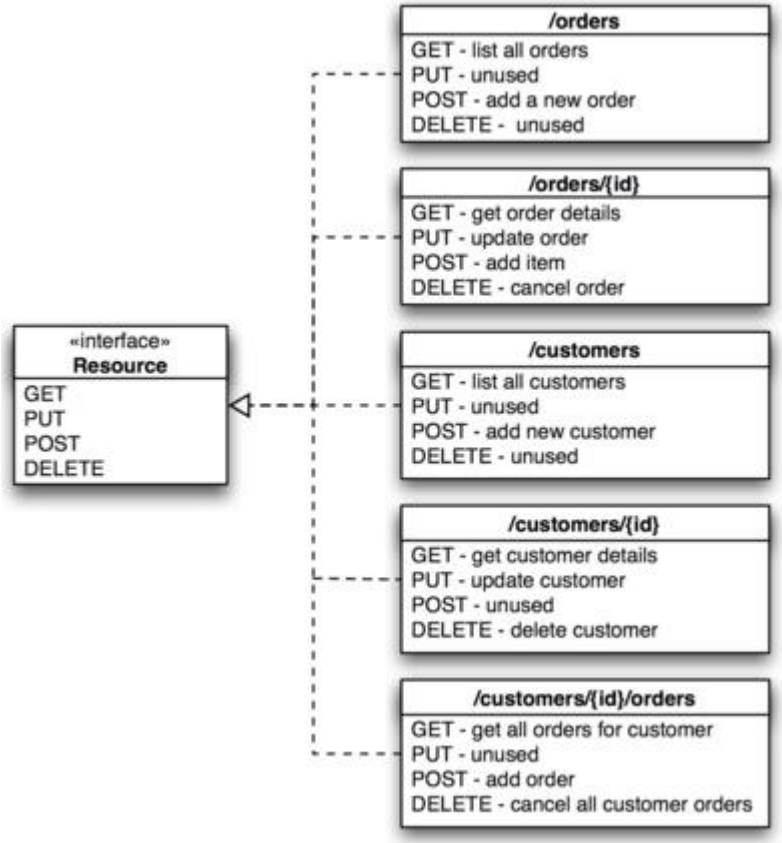
- Principle 2: Every Resource is Identified by a **Unique Identifier**
  - Resources are fundamental to the concept of REST
  - A resource has data, relationships to other resources, and methods that operate against it to allow for accessing and manipulating the associated information.
  - A Uniform Resource Locator (URL) identifies the online location of a resource. (directory structure-like URIs.)

Customer data	URI
Get Customer details with name “Shiv”	http://www.hogent.be/Customers/Shiv
Get Customer details with name “Raju”	http://www.hogent.be/Customers/Raju
Get orders from customer "Shiv"	http://www.hogent.be/Customers/Shiv/Orders
Get orders from customer “Raju”	http://www.hogent.be/Customers/Raju/Orders

HO GENT

What is REST?

- Principle 3: Describe resource functionality with HTTP methods

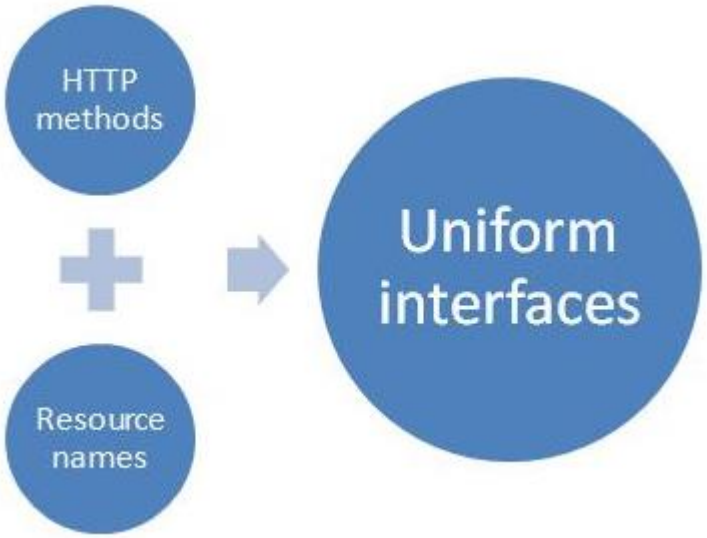


Method	Description
GET	to retrieve a representation of a resource.
POST	to create new resources and sub-resources
PUT	to update existing resources
PATCH	to partial update existing resources
DELETE	to delete existing resources

HO GENT

What is REST?

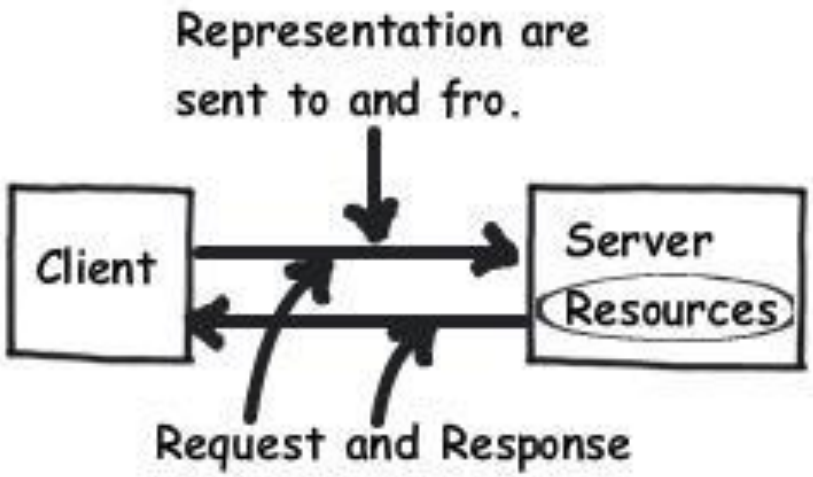
- Principle 3: Describe resource functionality with HTTP methods



HO GENT

What is REST?

- Principle 4: Communication is Done by **Representation**
  - You use Media Types for Representation : often XML and JSON
  - The Accept and Content-Type HTTP Headers describe the content being send or requested




```
{
Name:‘Jan',
Address:Gent'
}
```

HO GENT

What is REST?

- Principle 5: Responses : Give feedback to help developers succeed
  - Providing good feedback to developers on how well they are using your product
  - Good feedback involves positive validation on correct implementation, and an informative error on incorrect implementation that can help users debug and correct the way they use the product.
  - Response code
    - The client application behaved erroneously (client error - 4xx response code)
    - The API behaved erroneously (server error - 5xx response code)
    - The client and API worked (success - 2xx response code)
    - More on https://www.restapitutorial.com/httpstatuscodes.html

HO GENT

What is REST?

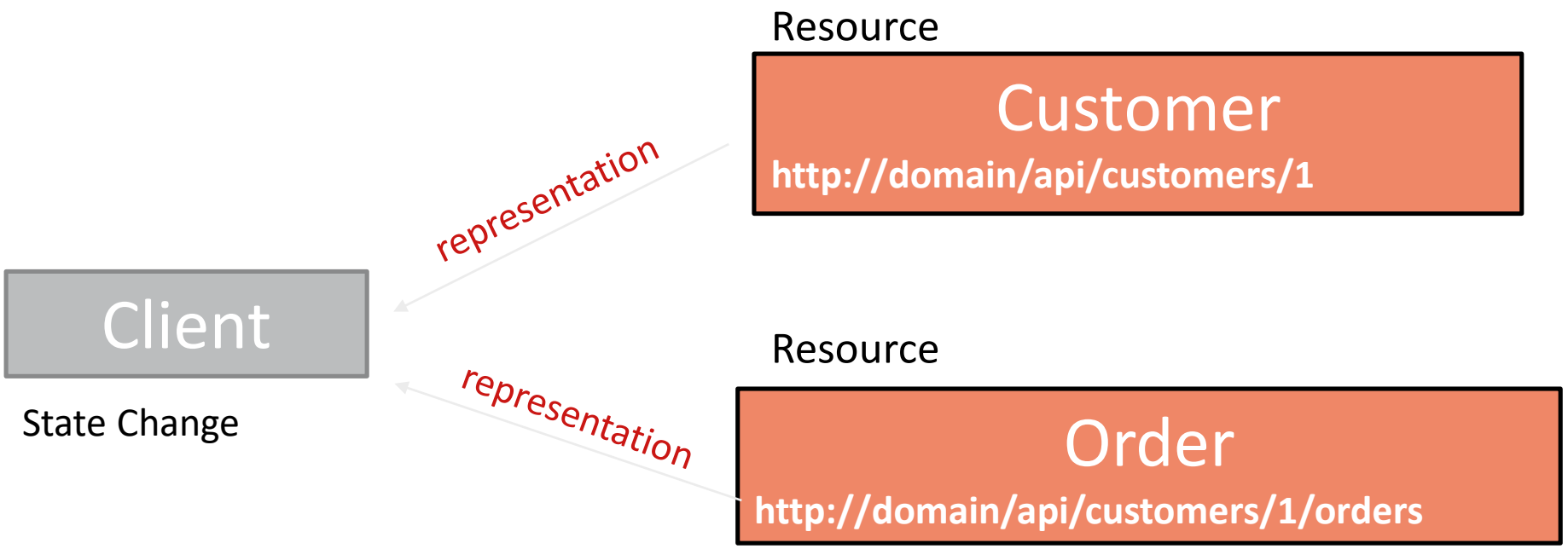
- Principle 5: Responses : Give feedback to help developers succeed
  - Example : a successful get
    - 200 response code
    - JSON with the requested data


```
{
“data”:[
{
“Username”:“example_user1”,
“created_time":“2013-12-23T05:51:14+0000 ”
},
{
“username”:“example_user2”,
“created_time":“2015-3-19T17:51:15+0000 ”
}
....
]
}
```


HO GENT

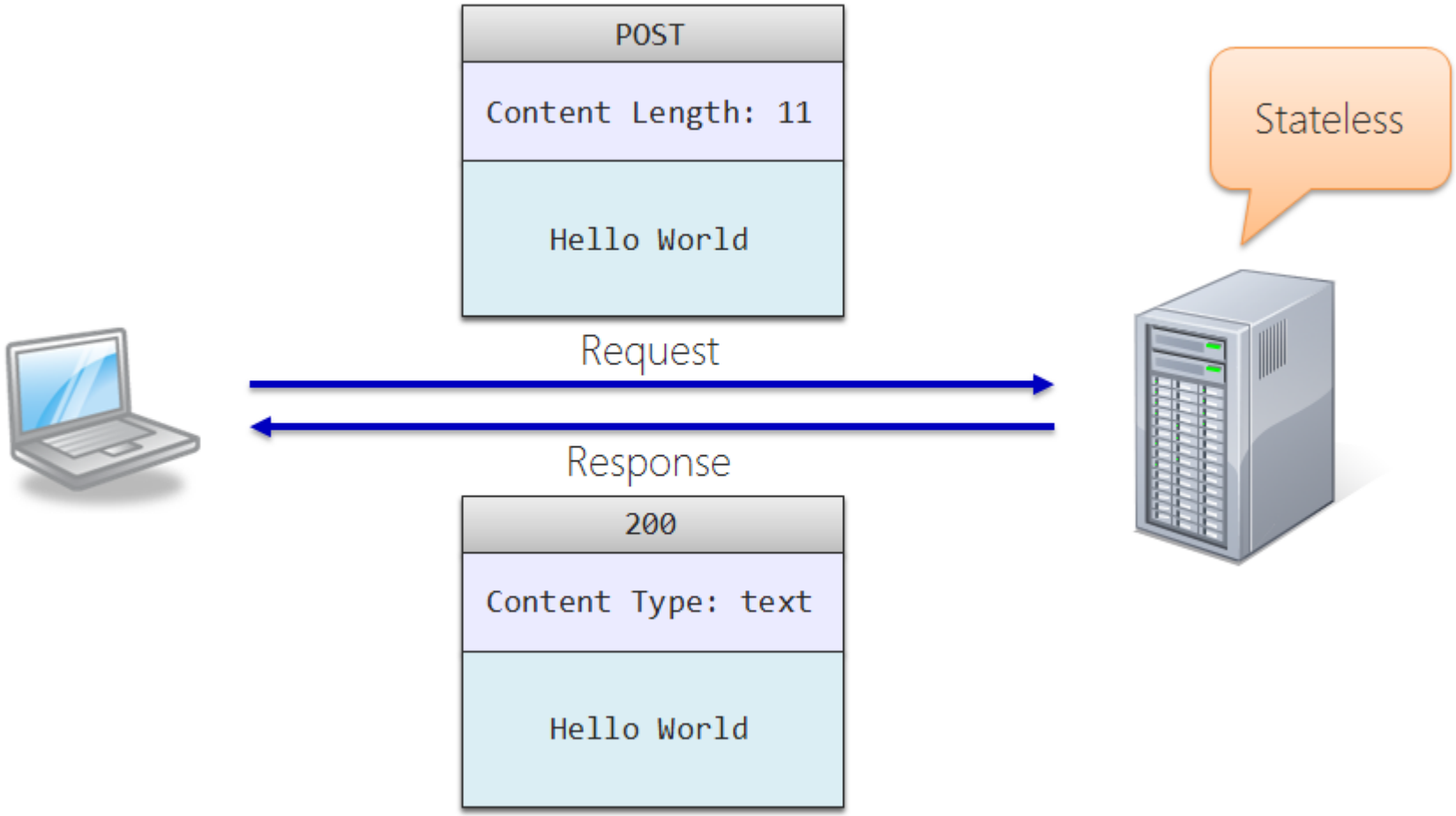
What is REST?

- Principle 6: Be Stateless
  - every request is independent and the server does not need to remember your previous request and states



HO GENT

What is REST?



HO
GENT

What is REST?

- Principle 7: Document your API

What is REST?

- Principle 7: Document your API
  - OpenAPI Specification
    - An **OpenAPI Specification (OAS)** (formerly known as Swagger) is a text (in YAML or JSON format) that provides a standard, programming language-independent description of a REST API.
    - OAS only specifies which functionality the API offers, not which implementation or dataset is hidden behind that API.
    - With OAS 3.0, both people and machines can view, understand and interpret the functionality of a REST API, without requiring access to the source code, additional documentation or analysis of network traffic.
    - From the documentation the client code can be generated

HO
GENT

What is REST?

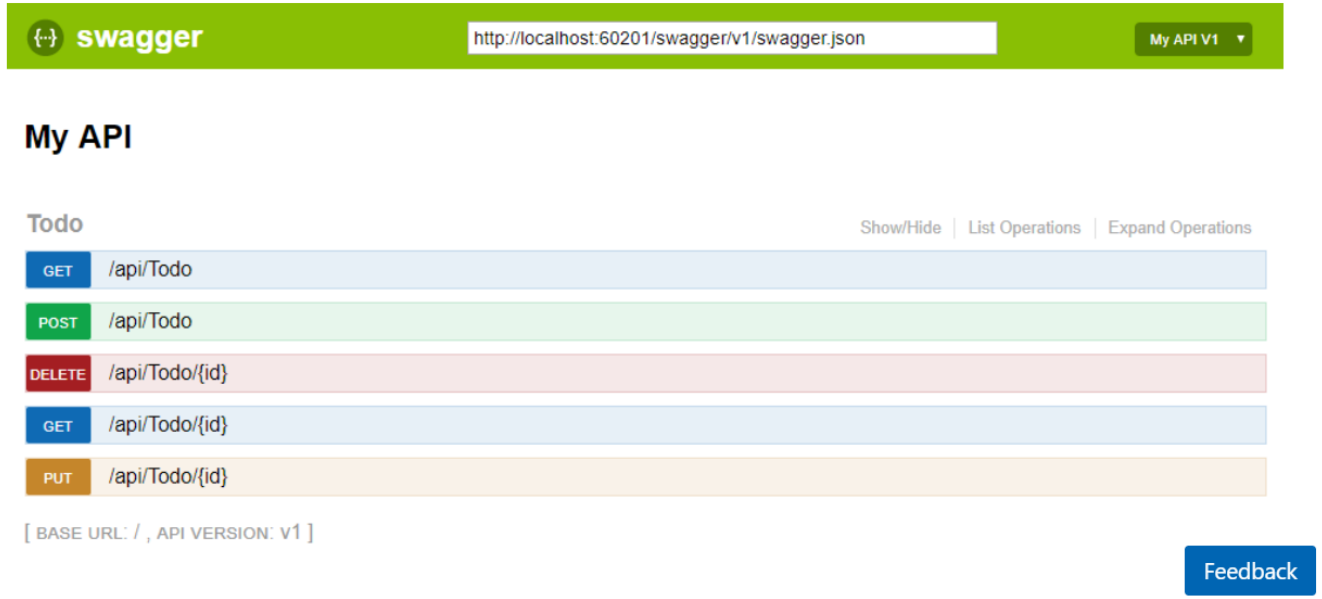
- Principle 7: Document your API
  - Swagger.json

```
'/api/Recipes/{id}": {
  "get": {
    "tags": [
      "Recipes"
    ],
    "summary": "Get the recipe with given id",
    "operationId": "Recipes_GetRecipe",
    "parameters": [
      {
        "name": "id",
        "in": "path",
        "required": true,
        "description": "the id of the recipe",
        "schema": {
          "type": "integer",
          "format": "int32"
        },
        "x-position": 1
      }
    ],
    "responses": {
      "200": {
        "description": "",
        "content": {
          "application/json": {
            "schema": {
              "nullable": true,
              "oneOf": [
                {
                  "$ref": "#/components/schemas/Recipe"
                }
              ]
            }
          }
        }
      },
```

HO GENT

What is REST?

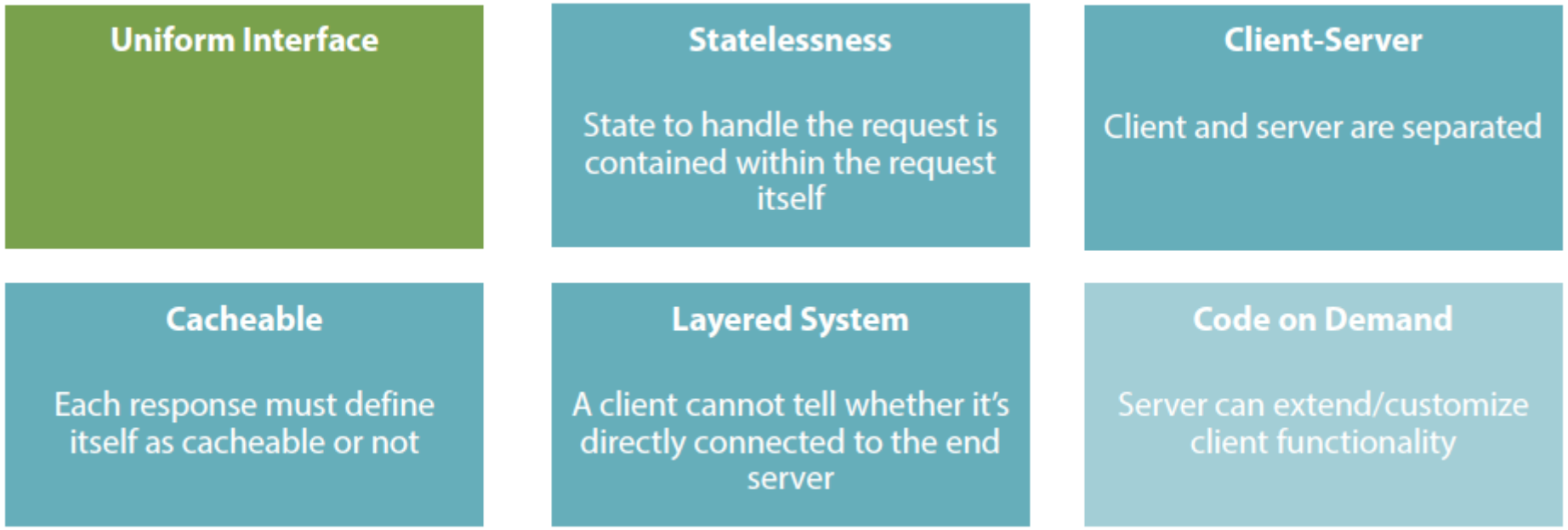
- Principle 7: Document your API
  - Swagger UI
    - to visualize and interact with the API's resources without having any of the implementation logic in place.
    - It's automatically generated from your OAS, with the visual documentation making it easy for back end implementation and client side consumption.



HO GENT

REST : Architectural constraints

- It is not a standard but a set of constraints
- To be called REST-ful, the API/system should adhere to 6 constraints



Dockx, K. (n.d.). *Building and Securing a RESTful API for Multiple Clients in ASP.NET*. Retrieved from https://app.pluralsight.com/library/courses/building-securing-restful-api-aspdotnet

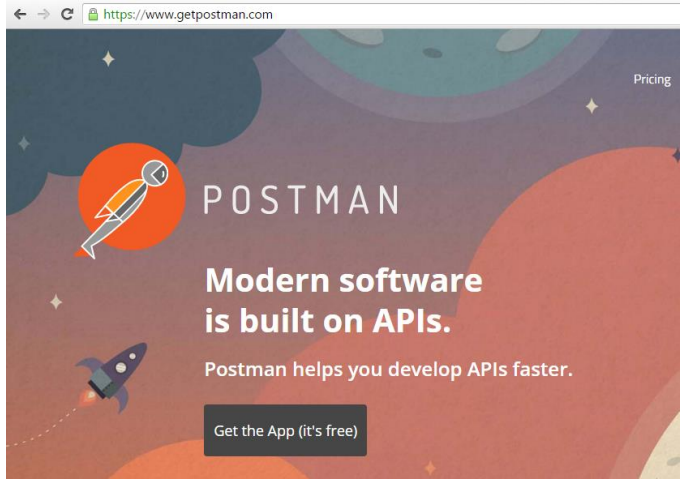
Positioning ASP.NET Core WebAPI

- **ASP.NET Core WebAPI** is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices
- For generating the documentation, you can use
  - **Swashbuckle.AspNetCore** is an open source project for generating Swagger documents for ASP.NET Core Web APIs.
  - **Or NSwag** is open source project for generating Swagger documents and integrating Swagger UI or ReDoc into ASP.NET Core web APIs.

HO
GENT

Consumer

- The consumer can be an MVC application
- The consumer can be an javascript/angular client
- The consumer can be the swagger UI, to document and test the api
- A Postman App, to test the api : https://www.getpostman.com/, also used for automated API testing https://www.postman.com/use-cases/api-testing-automation



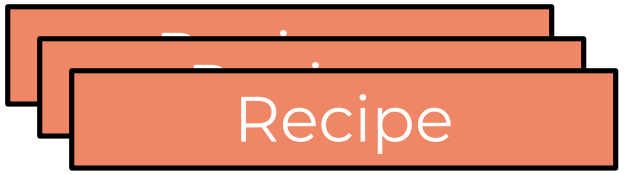
HO
GENT

Building the Recipe API

The API should be friendly to consume & should be consumable from different client types

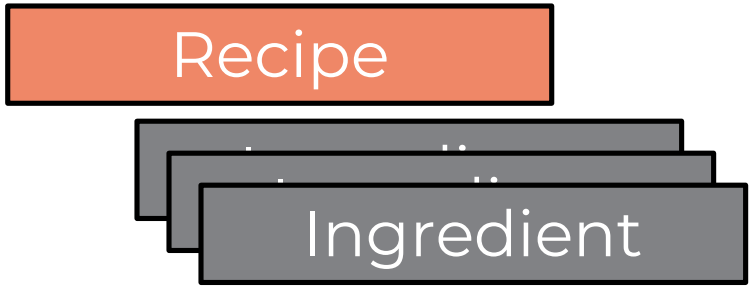
HO
GENT

Designing Resource & Resource Uri's



api/recipes

api/recipes/1



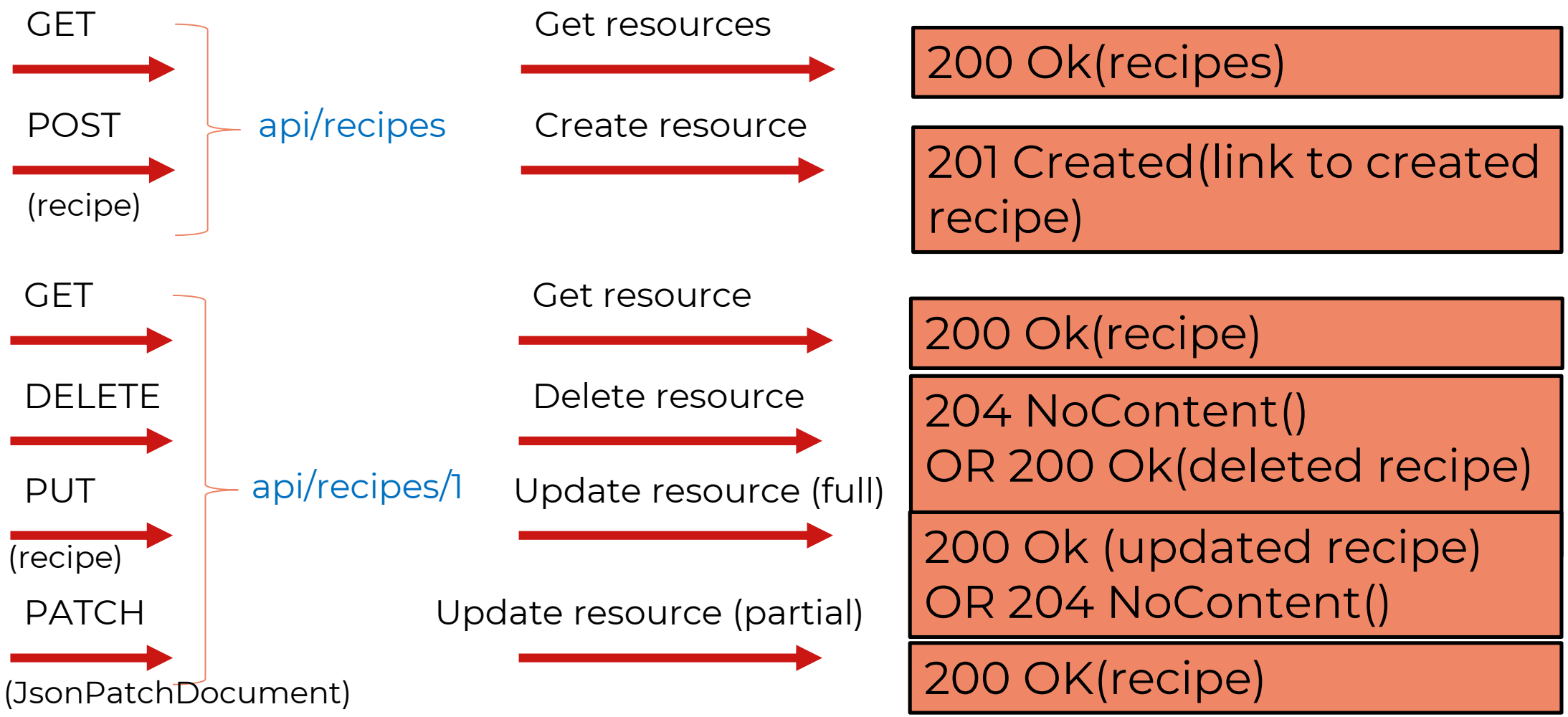
api/recipes/1/ingredients

api/recipes/1/ingredients/1

Representation:JSON

HO
GENT

Interacting with resources



HO GENT

The REST API endpoints

- All about recipes
  - **GET:** /recipes
  - **GET:** /recipes/{recipeid}
  - **POST:** /recipes
  - **PUT:** /recipes/{recipeid}
  - **DELETE:** /recipes/{recipeid}
  - Decide if the recipe will also include the ingredient details!!!!

REST Resource Naming Best Practices: more on https://restfulapi.net/resource-naming/
En https://github.com/microsoft/api-guidelines

- All about ingredients of a recipe (a **resource may contain sub-collection resources**)
  - **GET:** /recipes/{recipeid}/ingredients
  - **GET:** /recipes/{recipeid}/ingredients/{ingredientid}
  - **POST:**/recipes/{recipeid}/ingredients
  - **DELETE:** /recipes/{recipeid}/ingredients/{ingredientid}

Use sub-collection resources for associations!

HO GENT

The REST API endpoints

- Paging and searching
  - **GET:** /recipes? offset=0&limit=20
  - **GET:** /recipes?chef=Piet

Use querystring for paging and searching!

HO GENT

HTTP Status Codes

- GET: 200 (ok), 404 (not found), 400(bad request), 500 (internal server error)
- POST: 201 (created), 400, 500
- DELETE: 204 (no content) or 200(deleted recipe) 404, 400, 500
- PUT: 200 (changed recipe) or 204(no content), 404, 400, 500
- PATCH: 200, 404, 400, 500
- General: 401 (unauthorized), 403 (forbidden), 405 (method not allowed)

HO
GENT

Content Negotation

- Content Negotiation is a Best Practice
  - Use Accept header to determine how to format
    - GET /api/recipes/2
    - Accept: application/json
    - Host: localhost:8863
  - Not necessary to support all and have a sane default
    - Usually JSON for my tastes

HO
GENT

Exercise 1

- Explore the Vimeo api at https://developer.vimeo.com/api/reference

× Get a specific category

Request

```
GET https://api.vimeo.com/categories/{category}
```

Path parameters

Parameter	Data type	Description
category *	String	The name of the category.

* Required

Response

Details | Example | Reference

HTTP Status	Explanation
200 OK	The category was returned.
404 Not Found	No such category exists.

⊕ Get all categories

Try it out

Channels

→ Test your app

API Playground

```
GET /categories/{category}
```

Path parameters

Parameter	Value
category *	comedy

* Required

Make call

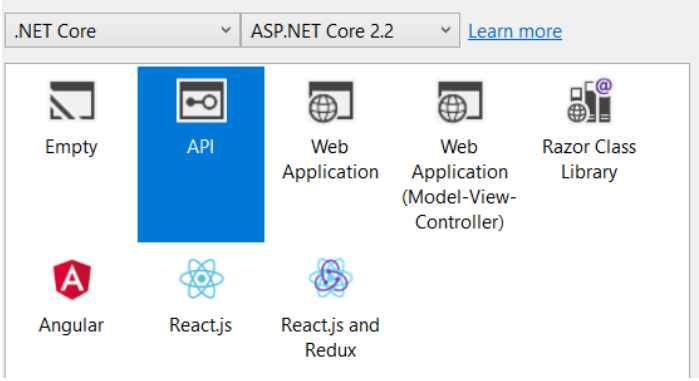
```
HTTP/1.1 403
Content-Type: text/html; charset=UTF-8

"<!doctype html>\n<html lang=\"en\">\n
<head>\n        <title>Vimeo Developer
Site</title>\n        <meta
name=\"description\" content=\"\">\n
<meta charset=\"utf-8\">\n        <meta http-
equiv=\"X-UA-Compatible\"
content=\"IE=edge\">\n        <meta
name=\"viewport\" content=\"width=device-width,
initial-scale=1\">\n        <meta name=\"csrf-
token\"
content=\"Ae7tTPphvFi4AG166h2UjxA6cHzFrVfIYZugm
Fuk\">\n        <link rel=\"canonical\"
href=\"\">\n        \n        <script>\n
window.dataLayer = [{'site_version': '2.0'}];\n
(function(w,d,s,l,i){w[l]=w[l]||
[];w[l].push({'gtm.start':\n
```

HO GENT

The Recipe Api

- Clone the project: https://github.com/ksa607/RecipeApi
  - It is an ASP.NET Core 3.1 Web application, choose the API template, No authentication. (The initial project contains a ValuesController)
  - Authentication will be added in a separate branch “authentication” for later use



- Examine the Model and Data classes.
  - Remarks : we work with disconnected entities (RecipeRepository). More on https://docs.microsoft.com/en-us/ef/core/saving/disconnected-entities
  - Shadow props en seeding (RecipeContext) : https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties and https://docs.microsoft.com/en-us/ef/core/modeling/data-seeding

HO GENT

The Recipe Api

- Step 1: install NSwag
- Step 2: Create the resource
- Step 3: Create REST API Controller
- Step 4: GET recipes
- Step 5: GET recipe by id
- Step 6: POST
- Step 7: PUT
- Step 8: DELETE
- Step 9: Documentation
- Step 10: CORS

Every step is a commit, so you can create branches to try things out

4fbe92bd	31/01/2020 14:23:25	Add search
ae96c9e5	31/01/2020 14:20:59	Suppress XML warning
59781ae2	31/01/2020 14:15:17	Enable CORS
3ed74d48	31/01/2020 14:12:55	Refactor Create Recipe method : working with DTO
37226d9d	31/01/2020 14:10:04	Add documentation
50a6e0e5	31/01/2020 14:03:38	Add POST, PUT, DELETE Recipe
37105b26	31/01/2020 13:48:44	Add GET Recipe by id
940a6ede	31/01/2020 13:46:41	Add GET recipes
59dd90c0	31/01/2020 13:37:58	Add RecipesController
506025cb	31/01/2020 13:35:16	Add NSwag
e71d46bc	31/01/2020 12:25:37	Add Models and Data classes
351a3e88	31/01/2020 12:04:52	Add project files.
331ebd96	31/01/2020 12:04:50	Add .gitignore and .gitattributes.

HO GENT

The Recipe Api

- Step 1 : Add Swagger
  - **Swashbuckle.AspNetCore** is an open source project for generating Swagger documents for ASP.NET Core Web APIs.
  - **NSwag** is open source project for generating Swagger documents and integrating Swagger UI or ReDoc into ASP.NET Core web APIs.

  - Nswag already uses OpenAPI 3.0, SwashBuckle still in beta

HO GENT

The Recipe Api

- Step 1 : Add Nswag (https://www.youtube.com/watch?v=lF9ZZ8p2Ciw)
  - Install the nuget package
    - Right-click the project in Solution Explorer > Manage NuGet Packages
    - Set the Package source to "nuget.org"
    - Enter "NSwag.AspNetCore" in the search box
    - Select the "NSwag.AspNetCore" package from the Browse tab and click Install
  - Configure Swagger Middleware in startup.cs
    - ConfigureServices

```
services.AddSwaggerDocument();
```

    - Configure (before useRouting : Register the Swagger generator and the Swagger UI)

```
app.UseOpenApi(); //Serves the registered OpenAPI/Swagger documents by default on `/swagger/{documentName}/swagger.json`
app.UseSwaggerUi3(); //Serves the Swagger UI 3 web ui to view the OpenAPI/Swagger documents by default on `/swagger`
```

HO
GE

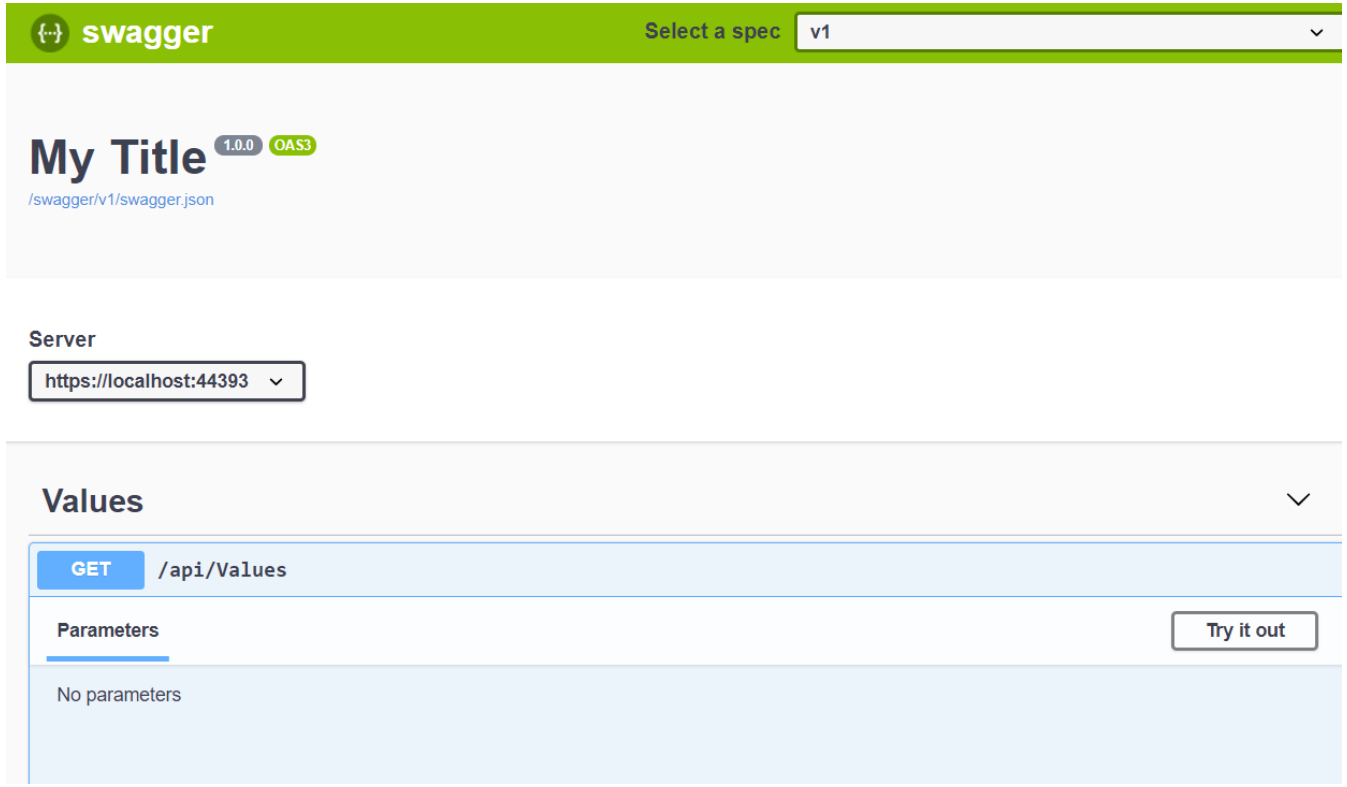
The Recipe Api

- Step 1 : Add Nswag
  - Launch the app.
  - http://localhost:<port>/swagger/v1/swagger.json to view the Swagger specification.
    - The core to the Swagger flow is the Swagger specification—by default, a document named *swagger.json*.
      - It's generated by the Swagger tool chain (or third-party implementations of it) based on your service. It describes the capabilities of your API and how to access it with HTTP. It drives the Swagger UI and is used by the tool chain to enable discovery and client code generation.
    - For more customisation see https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-nswag?view=aspnetcore-2.2&tabs=visual-studio%2Cvisual-studio-xml

HO GENT

The Recipe Api

- Step 1 : Add Nswag
  - Navigate to http://localhost:<port>/swagger to view the Swagger UI.



HO GENT

The Recipe Api

- Step 2 : Add Resources
  - The very first step in designing a REST API based application is – identifying the objects which will be presented as **resources**.

    - Recipe and Ingredient in the Models folder are the resources for our REST API
    - Add Data Annotations for validation

Remark : you can also create a viewmodel if the resource doesn't contain all the properties of the domain object (In Rest API we call this DTO's (Data Transfer Objects) or Entities)

HO
GENT

The Recipe Api

- Step 3 : Create the REST API (Controller)
  - Rightclick Controllers folder > Add > Controller > API Controller – Empty > name Recipe**s**Controller
    - Inherit from ControllerBase class : controller serves as a web API
    - [ApiController] attribute: indicates that the controller responds to web API requests.
    - [Route] : the url path. [controller] is replaced with name controller (class name minus the "Controller" suffix). It's important to use the plural name "recipes"!

HO GENT

The Recipe Api

- Step 3 : Create the REST API (Controller)
  - Inject the RecipeRepository (also see startup for DI config)

```
namespace recipeapi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class RecipesController : ControllerBase
     {
        private readonly IRecipeRepository _recipeRepository;
        public RecipesController(IRecipeRepository context)
        {
            _recipeRepository = context;
        }
    }
}
```

HO GENT

The Recipe Api

- Step 4 : GET Recipes
  - [HttpGet]: a method that responds to an HTTP GET request
  - ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message.
  - The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

```
// GET: api/Recipes
[HttpGet]
public IEnumerable<Recipe> GetRecipes()
{
    return _recipeRepository.GetAll().OrderBy(r => r.Name);
}
```

HO GENT

The Recipe Api

- Step 4 : GET Recipes
  - Test the api by calling the endpoint from a browser: https://localhost:<port>/api/recipes
    - The url path
      - Start with the template string in the controller's Route attribute. [controller] is replaced with the name of the controller
      - If the [HttpGet] attribute has a route template (for example, [HttpGet("/products")], this is appended to the path
    - A JSON-formatted response will be returned unless another format was requested and the server can return the requested format. Content negotiation occurs when the client specifies an Accept header. The default format used by ASP.NET Core MVC is JSON
  - Or use the swagger UI, go to http://localhost:<port>/swagger
    - To open swagger when you start the website : just open Properties of the application, go to Debug tab, and write Swagger in the "Launch browser" text box,

HO GENT

The Recipe Api

- Step 4 : GET Recipes
  - Remark: Make sure you include the related entities in the repository!!
  - Remark: extra encoding into a key/value pair
    - If the collection size is large, apply paging and/or filtering.
      - HTTP GET /recipes?offset=0&limit=20
      - HTTP GET /recipes?chef=Piet
    - Sorting :
      - HTTP GET /recipes?sort_by=name&order_by=asc
    - =>The keys become extra parameters of the method in the controller
  - To go immediately to swagger UI when running, update LaunchSettings.json in Properties folder

```
"RecipeApi": {
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "swagger",
  "applicationUrl": "https://localhost:5001;http://localhost:5000",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
```

HO GENT

The Recipe Api

- Step 5 : GET Recipe by id
  - [HttpGet]: method responds to an HTTP GET request. {id} is a placeholder variable. The id in the url will be provided to the method's id parameter
  - Action parameter binding: A parameter name matching a name in the route template is automatically bound using the request route data
  - If recipe exist : a Recipe object is returned with a 200 status code.
  - If recipe doesn't exist : 404. The NotFound helper method is invoked as a shortcut to return new NotFoundResult() (when executed will produce a 404 response.

```
[HttpGet("{id}")]
 public ActionResult<Recipe> GetRecipe(int id)
 {
     Recipe recipe = _recipeRepository.GetBy(id);
     if (recipe==null)  return NotFound();
     return recipe;  //of Ok(recipe)
 }
```

HO
GENT

The Recipe Api

- Step 5 : GET Recipes by ID
  - ActionResult<T> return type for Web API controller actions
    - It enables you to return a type deriving from ActionResult or return a specific type T. (Swagger uses this information to know what type will be returned)
    - It implements 2 implicit operators which do the work
      - `public static implicit operator ActionResult<TValue>(TValue value) { return new ActionResult<TValue>(value); }`
      - `public static implicit operator ActionResult<TValue>(ActionResult result) { return new ActionResult<TValue>(result); }`
      - More on https://joonasw.net/view/aspnet-core-2-1-actionresult-of-t
    - GetRecipe can return two different status values:
      - If no item matches the requested ID, the method returns a 404 NotFound error code.
      - Otherwise, the method returns 200 with a JSON response body.
  - Test the app by calling the endpoint from a browser: https://localhost:<port>/api/recipes/1 or using the Swagger UI

HO GENT

The Recipe Api

- Step 6 : CREATE Recipe

```
[HttpPost]
 public ActionResult<Recipe> PostRecipe(Recipe recipe)
 {
     _recipeRepository.Add(recipe);
     _recipeRepository.SaveChanges();

     return CreatedAtAction(nameof(GetRecipe),
               new { id = recipe.Id }, recipe);
 }
```

HO GENT

The Recipe Api

- Step 6 : CREATE Recipe
  - [HttpPost]: a method that responds to an HTTP POST request
  - Action parameter binding: Complex type parameters are automatically bound using the request body
  - CreatedAtAction method:
    - Returns a **201** response (= the standard response for an HTTP POST).
    - Adds a Location header to the response pointing to the url for the newly created response, and the object itself in the body. The url should be the url at which a GET request would return the object

```
content-type: application/json; charset=utf-8
date: Fri, 15 Feb 2019 15:28:33 GMT
location: https://localhost:44393/api/Recipes/3
server: Microsoft-IIS/10.0
status: 201
```

HO GENT

The Recipe Api

- Step 6 : CREATE Recipe
  - Handling of Model State errors
    - In MVC Controller: Model validation occurs before the execution of a controller action. It's the action's responsibility to inspect ModelState.IsValid and return an error response.
    - In web API controllers using the [ApiController] attribute: when ModelState.IsValid evaluates to false, an automatic **HTTP 400 (Bad Request)** response containing issue details is returned.
      - Example : leave the name of the recipe empty

Error:

Response body


```
{
  "errors": {
    "Name": [
      "The Name field is required."
    ]
  },
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "80000036-0003-fb00-b63f-84710c7967bb"
}
```


Response headers

```
content-type: application/problem+json; charset=utf-8
date: Fri, 15 Feb 2019 15:31:19 GMT
server: Microsoft-IIS/10.0
status: 400
```

HO GENT

The Recipe Api

- Step 7 : Update Recipe

```
[HttpPut("{id}")]
public IActionResult PutRecipe(int id, Recipe recipe)
 {
     if (id != recipe.Id)
     {
         return BadRequest();
     }
     _recipeRepository.Update(recipe);
     _recipeRepository.SaveChanges();
     return NoContent();     }
```

- Oefening : pas aan zodat 404 gereturned wordt als recipe met opgegeven id niet bestaat

HO GENT

The Recipe Api

- Step 7 : Update Recipe
  - [HttpPut]: a method that responds to an HTTP PUT request
  - similar to PostRecipe, except it uses HTTP PUT.
  - The response is **204 (No Content)** or **400 (Bad Request)** when ModelState validation fails or id's don't match
  - According to the HTTP specification: a PUT request requires the client to send the **entire updated entity**, not just the changes. For partial updates, use HTTP PATCH. The common use case is to **return 204** as a result of a **PUT** request, updating a resource, without changing the current content of the page displayed to the user. If the page should be changed to the newly updated page, the 200 should be used instead

HO GENT

The Recipe Api

- Step 8 : DELETE Recipe
  - [HttpDelete]: a method that responds to an HTTP DELETE request
  - The response is 204(No Content)

```
[HttpDelete("{id}")]
 public IActionResult DeleteRecipe( int id)
{
    Recipe recipe = _recipeRepository.GetBy(id);
    if (recipe == null)
    {
        return NotFound();
    }
    _recipeRepository.Delete(recipe);
    _recipeRepository.SaveChanges();
    return NoContent();
}
```

HO GENT

The Recipe Api

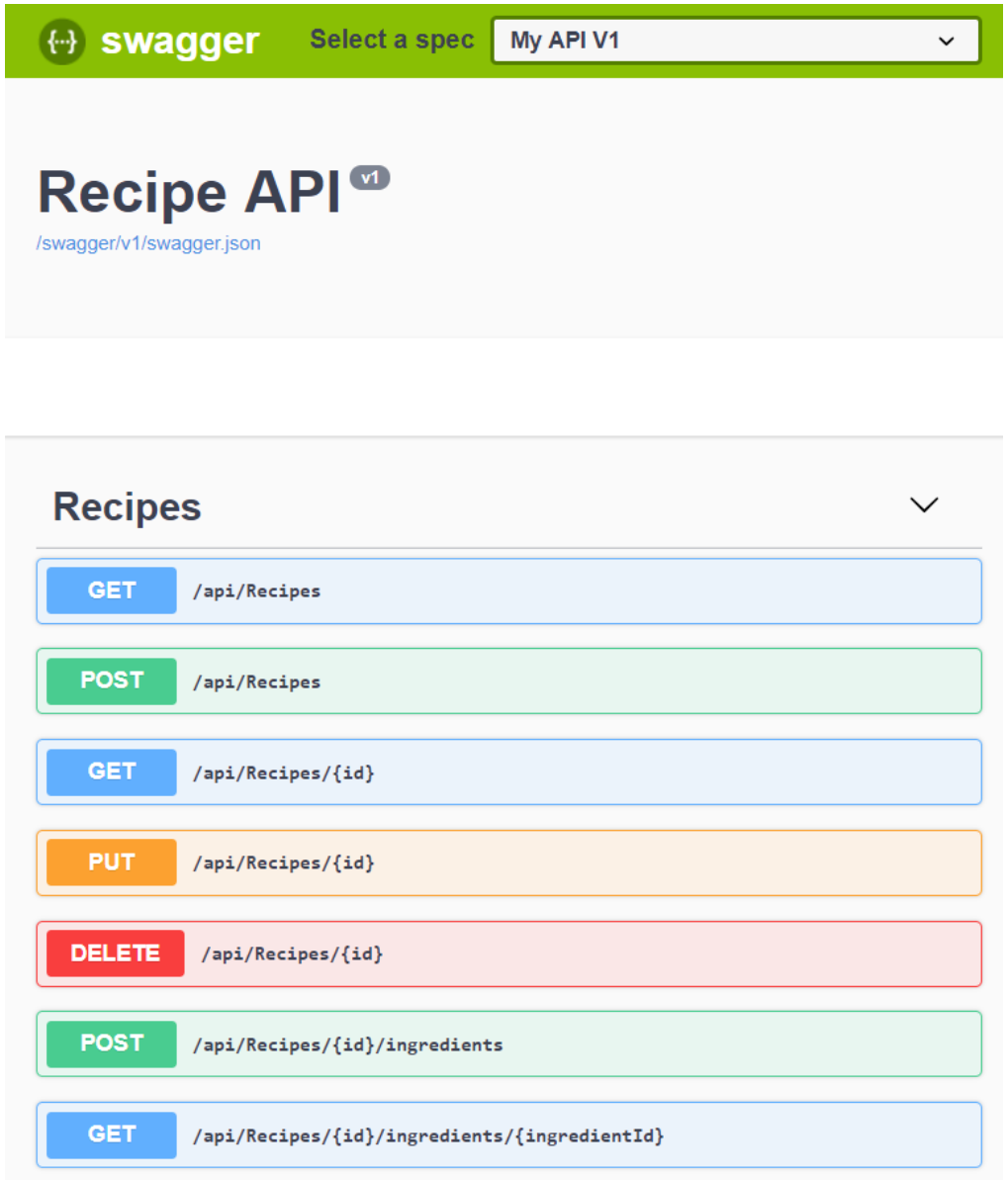
- In case of errors?
  - Return status code
  - Communicate the error info so you can help users to recover from error (be carefull with security issues like wrong password)

```
try
{
                    ….
}
catch (Exception e)
{
        return StatusCode(StatusCodes.Status500InternalServerError, e.Message );
}
```

HO GENT

Generating the documentation

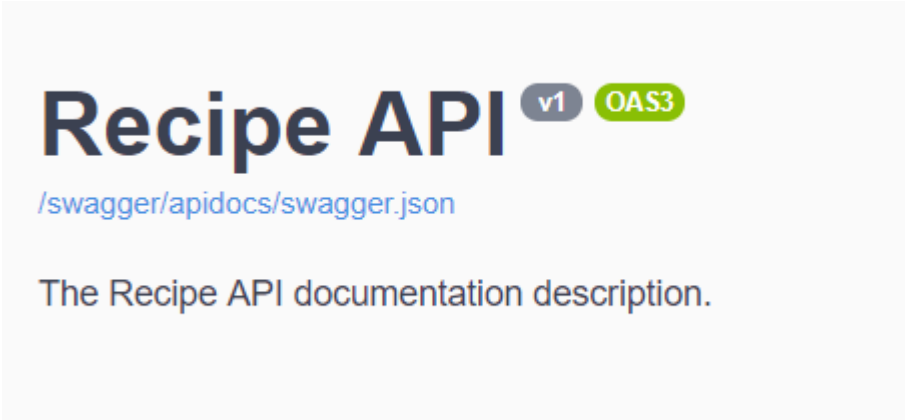
- Swagger UI
  - navigate to
  http://localhost:<port>/swagger
  - Renders the documentation
  - Renders a web based test UI
  - Generate client proxies
    - Like WCF service references
    - Many different generation tools
      - Main problem: Missing features in the generated problem (e.g. no support for discriminators, etc.)



HO GENT

Generating the documentation

- Customize and extend the documentation
  - API and info description




```
services.AddOpenApiDocument(c =>
{
c.DocumentName = "apidocs";
c.Title = "Recipe API";
c.Version = "v1";
c.Description = "The Recipe API documentation description.";
});
```

HO GENT

Generating the documentation

- Customize and extend the documentation
  - Adding /// comments to generate documentation
    - First enable xml comments
      - Rightclick in solution explorer > Edit recipeapp.csproj. Add the following

```
<PropertyGroup>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

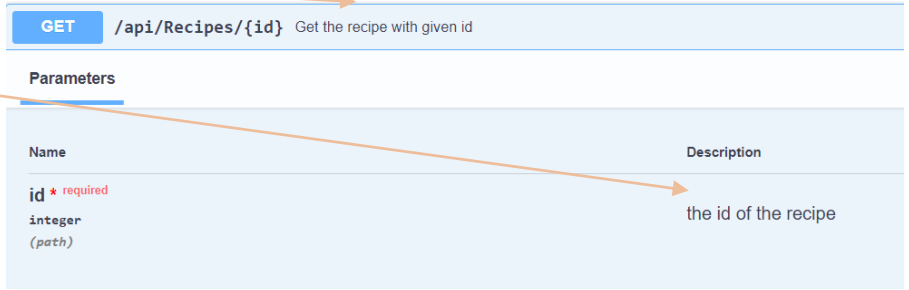
HO GENT

Generating the documentation

- Customize and extend the documentation
  - Adding /// comments to generate documentation

```
/// <summary>
/// Get the recipe with given id
/// </summary>
/// <param name="id">the id of the recipe</param>
/// <returns>The recipe</returns>
[HttpGet("{id}")]

public ActionResult<Recipe> GetRecipe(int id)
```



- VS shows a lot of 1591 warnings : In Solution explorer > rechtsklik Project > properties > Build, Errors and warning : add 1591

HO GENT

Generating the documentation

- Customize and extend the documentation
  - Data Annotations found in the System.ComponentModel.DataAnnotations namespace alter underlying json schema

```
public class Recipe
{
        #region Properties
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public DateTime Created { get; set; }
        public string Chef { get; set; }
        public ICollection<Ingredient> Ingredients { get; set; }
        #endregion
}
```

HO GENT

Generating the documentation

- Customize and extend the documentation
  - Add the [Produces("application/json")] attribute to the API controller.
    - The Response Content Type drop-down selects this content type as the default for the controller's GET actions:

HO
GENT

Generating the documentation

- Customize and extend the documentation
  - Describe response types
    - NSwag uses Reflection to infer the types
    - Use data annotations[ProducesResponseType] to tell clients which HTTP status codes this action is known to return.

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Recipe> PostRecipe(Recipe recipe)
```

    - In ASP.NET Core 2.2 or later, conventions can be used as an alternative to explicitly decorating individual actions with [ProducesResponseType], put it on the controller

```
[ApiConventionType(typeof(DefaultApiConventions))]
```

HO GENT

Exercise : create the methods for ingredients of a recipe

- Get 1 ingredient for a given recipe
- Add an ingredient for a recipe

HO
GENT

Further refine the methods

- You can further refine the methods by using DTO's (Data Transfer objects : comparable with ViewModels)
  - When adding a recipe, the user can fill in an id, but will get an error message if different from 0
  - Solution : Define a DTO :

```
using System.ComponentModel.DataAnnotations;
namespace RecipeApi.DTOs
{
    public class IngredientDTO    {
        [Required]
        public string Name { get; set; }
        public double? Amount { get; set; }
        public string Unit { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;
namespace RecipeApi.DTOs
{
    public class RecipeDTO
    {
        [Required]
        public string Name { get; set; }
        public string Chef { get; set; }
        public IList<IngredientDTO>
Ingredients { get; set; }
    }}
```

Further refine the methods

– Controller

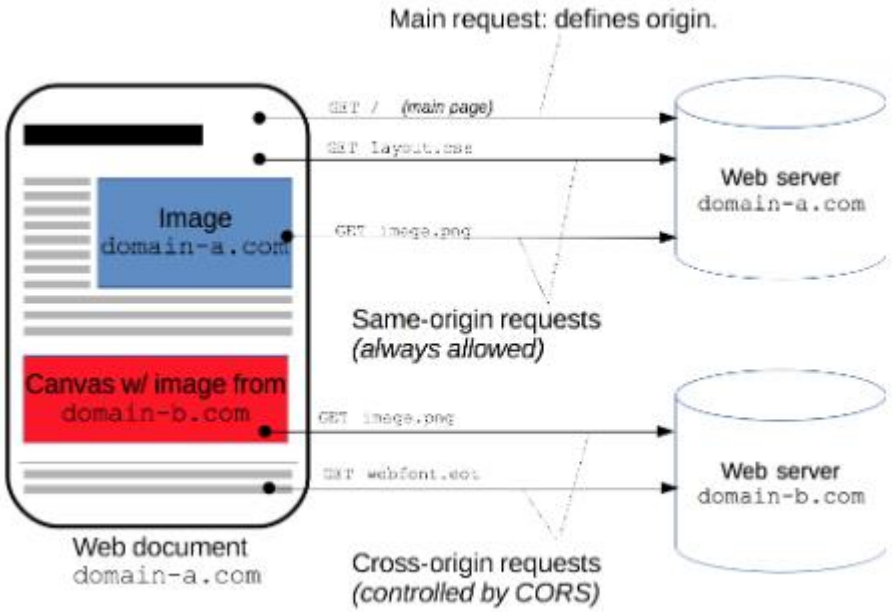
```
[HttpPost]
        public ActionResult<Recipe> PostRecipe(RecipeDTO recipe)
        {
            Recipe recipeToCreate = new Recipe() { Name = recipe.Name, Chef = recipe.Chef };
            foreach (var i in recipe.Ingredients)
                recipeToCreate.AddIngredient(new Ingredient(i.Name, i.Amount, i.Unit));
            _recipeRepository.Add(recipeToCreate);
            _recipeRepository.SaveChanges();

            return CreatedAtAction(nameof(GetRecipe), new { id = recipeToCreate.Id },
recipeToCreate);
        }
```

HO GENT

Enabling CORS

- Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.



Cross-Origin Resource Sharing (CORS). (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

HO GENT

Enabling CORS

- Register CORS services in Startup.cs, ConfigureServices

```
 services.AddCors(options =>
       options.AddPolicy("AllowAllOrigins", builder =>
builder.AllowAnyOrigin()));
```

- Apply CORS policies globally to the app via middleware in Startup.cs, Configure

```
 app.UseCors("AllowAllOrigins");
```

- More on https://docs.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-2.2

HO GENT

Using Postman app to test

- Download from https://www.getpostman.com/
- Switching off SSL Certificate Verification, File > Settings > General
- Open the Postman Console to see more error logging (View > Show Postman Console)
- To send information to server
  - Body tab > select raw and in the dropdown json(application/json)

HO GENT

Authentication

JWT token based

HO
GENT

Authentication

- Introduction
- Storing passwords
- Token based authentication
- Implementing token based authentication in the rest api
  - Add Identity
  - Enable token based authentication in rest api
  - Enable token based authentication in swagger
  - Create AccountController
  - Add Authorize attribute

HO
GENT

Authentication

- many sites want to provide personalized content, and hence need a system to register and identify users
- user data is considered sensitive, and many (sometimes strict) laws apply to how you handle user data (GDPR)
- users (rightfully) expect you to store their data securely and with care; storing 6 letter passwords in a cleartext file won't cut it anymore

Authentication

- implementing all steps correctly is far from trivial, as few things involving encryption tend to be
- luckily, we're also far from the first ones to attempt this, by now it's pretty well known how to tackle these problems
- still, caution and diligence are advised; data breaches happen, users re-use their same passwords, exposing user info is always bad, even on seemingly 'unimportant' sites

Authentication steps

- create a database table (collection), PROPERLY store username and password
- provide routes to login / register new users in your backend
- choose a secure way to communicate who's logged in between front and backend (e.g. jwt)
- protect all backend routes which shouldn't be accessible by everyone
- properly store and send your token (or cookie) in the frontend so you do get access where you should

HO
GENT

Authentication don’t’s

- NEVER trust the frontend, sanitize everything in your backend
- DO NOT rely on 'secrets' or urls only you know to secure something
- DO NOT invent your own encryption mechanism, that is really, really (really) hard
- NEVER trust the frontend
- NEVER TRUST THE FRONTEND
- BITGRAIL CRYPTO 'HACK', EARLY 2018, $170 MILLION STOLEN
  - "There was a bug, on the withdraw page. But this check was only on java-script client side, you find the js which is sending the request, then you inspect element - console, and run the java-script manually, to send a request for withdrawal of a higher amount than in your balance. Bitgrail delivered this withdrawal. How many people did this? Who knows. This bug was later closed."

HO
GENT

Storing passwords

- I hope you already know it's not a good idea to store passwords as plain text

user	password
rudy	ydur1
shaniah	slagroomtaart
...	...

  – while it might be convenient to check passwords, and retrieve lost ones, if your database is leaked, everything is out in the open

HO
GENT

Storing passwords

- encrypting/decrypting the password is slightly better, people need to get hold of both the database AND your encryption key

user	password
rudy	encrypt('ydur1', 'mysecret')
shaniah	encrypt('slagroomtaart', 'mysecret')
...	...

  – still, in the end this is pretty bad, once both do get out, every single user his info is out in the open again, and if you can't trust your database to be safe, can you really say database+secret is much safer?

HO
GENT

Storing passwords

- it's much better to store a one way hash of the password, if the user tries to login, you hash again and can check if he is who he claims he is

user	password
rudy	onewayhash('ydur1')
shaniah	onewayhash('slagroomtaart')
...	...

  - but if your database is leaked, you can't retrieve the password based on the hash ('one way'), retrieving lost passwords becomes a bit more cumbersome though (you can only assign a new one, not retrieve an existing)
  - still, while you can't retrieve a password from a hash, you can easily calculate hashes from passwords (that's the whole point), and many people tend to use very common passwords ('password', 'password1', 'pass123', ...)

HO
GENT

Storing passwords

- it's much better to store a one way hash of the password, if the user tries to login, you hash again and can check if he is who he claims he is

user	password
rudy	onewayhash('ydur1')
shaniah	onewayhash('slagroomtaart')
...	...

  – so it's possible to pre-calculate all the common passwords (= rainbow table) and then simply check if you find users with any of those hashes; you can't find a login for a specific user this way, but you can probably find many logins for many users

HO
GENT

Storing passwords

user	hash	salt
rudy	onewayhash('ydur15FA482B')	5FA482B
shaniah	onewayhash('slagroomtaartAAC62F9')	AAC62F9
...	...	...

- this is prevented by adding a 'salt', for every user a random string is generated and stored, and the hash is now a hash of password+salt
- while you can still easily generate a hash for a user (simply add the salt to whatever password you're trying), you can't precompute all common passwords anymore, as each user has a different salt

HO
GENT

PASSWORDS & HASHES

- you store a good password as a (secure) hash, this works because computing the hash is (relatively) slow, so brute forcing becomes hard
- what makes a good password? unfortunately, the 'world of hashes' changed the last years, and changed a lot
- GPU's happen to be very good at hashing and people started to use GPU power for general computing
- how long do you think it takes to brute force a truly random (lowercase, uppercase, number) 8 character password?
- think again https://www.grc.com/haystack.htm

HO
GENT

PASSWORD RULES

- password rules are bullshit, the only thing you achieve is annoy your regular users (and show you don't know anything about password safety to your knowledgeable users)

- size is the only thing that matters that's what she said
- your password is probably to short, use passphrases
- better yet, use a decent password manager, many of you will have (root) access to large and important computers during your careers, start good habits now

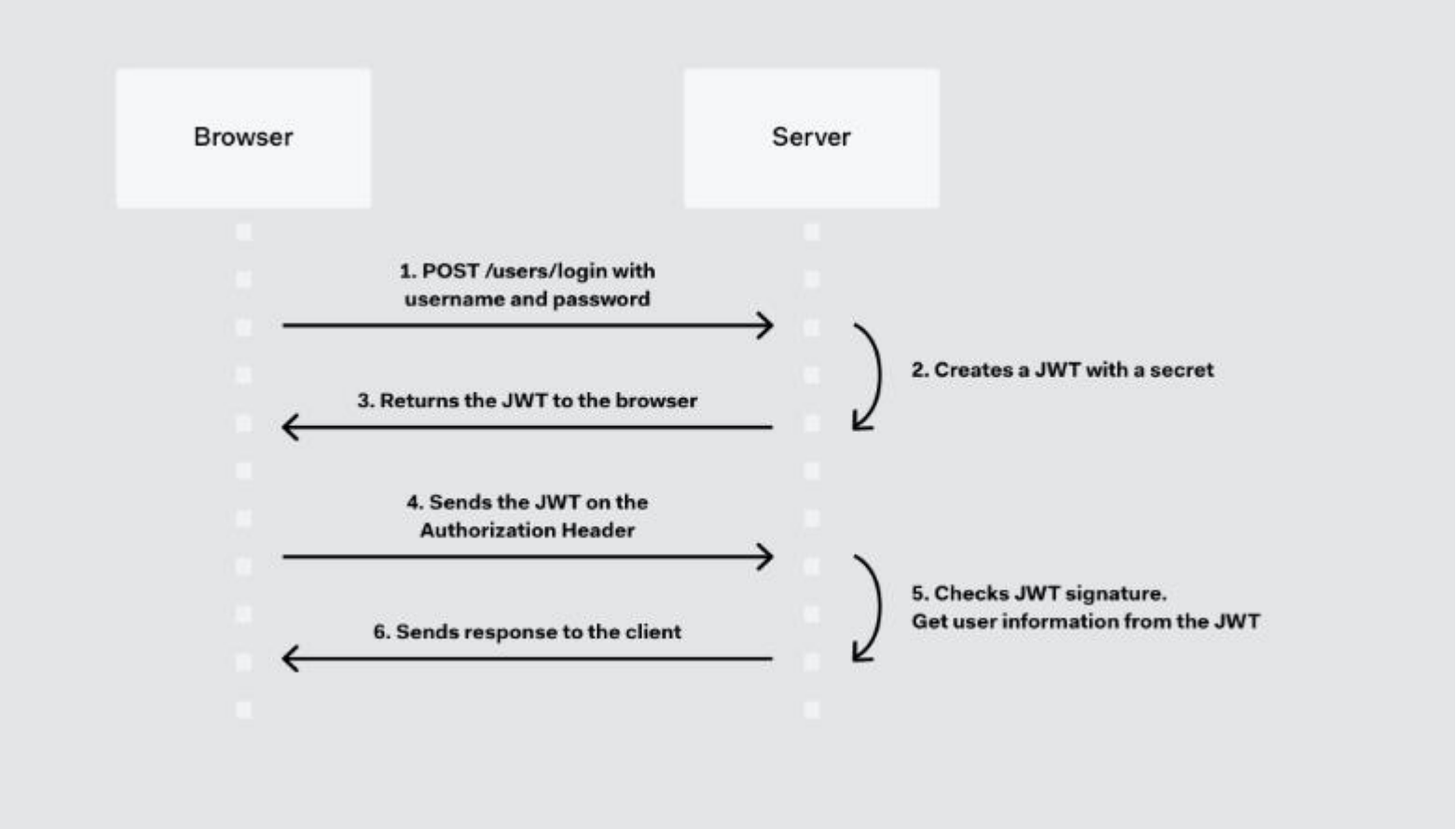
HO
GENT

Token based authentication

- we need a way to store login status on the frontend, so users don't have to re-login on every single requests
- we'll use json web tokens (JWT) for this and store them in the browser's localstorage

HO
GENT

Token based authentication



HO
GENT

Token based authentication

- a jwt has (more on https://jwt.io/introduction/)
  - a header: a JSON object containing meta-information about the type of JWT and hash algorithm used to encrypt the data. this JSON is Base64Url encoded
  - a payload: a JSON object containing the actual data shared between source and target; these data are coded in *claims*, that is statements about an entity, typically the user. The payload is then Base64Url encoded
  - a signature: this section allows to verify the integrity of the data (to verify that the message wasn't changed along the way), since it represents a digital signature. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

  The three sections are combined together into a sequence of Base64 strings separated by dots so that the data can be easily sent around in HTTP-based environments. It can be stored in local storage.
- you encrypt the signature on the server using a secret only you know
- then when the token is sent back to you, you check the signature using that secret, and you know the user is who she claims to be

HO GENT

Token based authentication

- There are two types of JSON Web Token (JWT) claims:
  - Reserved: Claims defined by the JWT specification to ensure interoperability with third-party, or external, applications. The JWT specification defines 7 reserved claims that are not required
    - iss (issuer): Issuer of the JWT
    - sub (subject): Subject of the JWT (the user)
    - aud (audience): Recipient for which the JWT is intended
    - exp (expiration time): Time after which the JWT expires
    - nbf (not before time): Time before which the JWT must not be accepted for processing
    - iat (issued at time): Time at which the JWT was issued; can be used to determine age of the JWT
    - jti (JWT ID): Unique identifier; can be used to prevent the JWT from being replayed (allows a token to be used only once)
  - Custom: Claims that you define yourself. Name these claims carefully, such as through namespacing , to avoid collision with reserved claims or other custom claims

HO GENT

Token based authentication

https://github.com/ksa607/RecipeApi.git
Branch 'Authentication'

- Step 1 : Add Identity – see commit "Added Identity and Customer classes"
- Step 2 : enable token based authentication in rest api
- Step 3 : enable token based authentication in swagger
- Step 4 : Create AccountController
- Step 5 : Add Authorize attribute
- Step 6 : Test it out
- Stap 7 : Get the favourite recipes of the authenticated user

HO GENT

Authentication

- Step 1 : Add Identity – see commit “Added Identity and Customer classes
  - Added package Microsoft.AspNetCore.Identity.EntityFrameworkCore
  - Startup.cs: added Identity (not DefaultIdentity (so nu UI))
  - Domain : Customer- CustomerFavorites - ICustomerRepository
  - Data : RecipeContext (:IdentityDbContext, mapping, DbSet) and RecipeDataInitializer

HO
GENT

Authentication

- Step 2 : enable token based authentiation
  - Add package , packages Microsoft.AspNetCore.Authentication.JwtBearer
  - enable token authentication
    - Adding token authentication to your API: use JwtBearerAuthentication middleware
    - in startup.cs in ConfigureServices

```
services.AddAuthentication(x => {
  x.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>            {
  x.SaveToken = true;
  x.TokenValidationParameters = new TokenValidationParameters{
    ValidateIssuerSigningKey = true,
    IssuerSigningKey = new SymmetricSecurityKey(
        Encoding.UTF8.GetBytes(Configuration["Tokens:Key"])),
    ValidateIssuer = false,
    ValidateAudience = false,
    RequireExpirationTime = true
};
});
```

  - in startup.cs in Configure, to make the service available (after app.UseRouting)

```
app.UseAuthentication();
```

HO GENT

Authentication

- Step 2 : enable token based authentication
  - in startup.cs in ConfigureServices : configure the JWT-based authentication service
    - AddAuthentication : to register JWT authentication schema, JwtBearerDefaults.AuthenticationScheme (used by the AuthenticateAsync method)
    - Then configure the authentication schema with options (JwtBearerOptions) for JWT bearer
      - SaveToken is used to indicate whether the server must save token server side to validate them. So even when a user have a properly signed and encrypted it will not pass token validation if it is not generated by the server.
      - TokenValidationParameters : Contains a set of parameters that are used when validating a token
        - » IssuerSigningKey = the signing key of the tokens
        - » controls if validation of the signing key that signed the securityToken is called (ValidateIssuerSigningKey = true/false(default)).
        - » validate the server that created that token (ValidateIssuer = true (default) /false (a token can specify an issuer claim)
        - » ensure that the recipient of the token is authorized to receive it (ValidateAudience = true (default)/false (a token can specify an audience claim);
        - » Ensure token has an expiration value (RequireExpirationTime = true(default)/false) and that the token is not expired (ValidateLifeTime = true(default)/false)

HO GENT

Authentication

- Step 2 : enable token based authentication
  - Add the server secret
    - you need a secret only the server knows to encrypt/decrypt this signature, but where should you store this secret? a variable exposed in github is obviously a bad idea
    - Store the key in a safe place.
      - Appsettings.json is a bad idea
      - Store the key in your environment variables or Secret Manager (rightclick project > Manage User Secrets) !!! this way our code can use the secret, but it's not stored anywhere in our source files (and github) or database
      - if you work in a team, you obviously need to share this secret, but not through github!
      - most online hosts have a way of setting environment variables (we'll see how to do this in heroku/Azure later)

```
{ "Tokens": {
    "Key": " IfThisEndsUpInGithubYouFailTheClass"
}}
```

HO GENT

Authentication

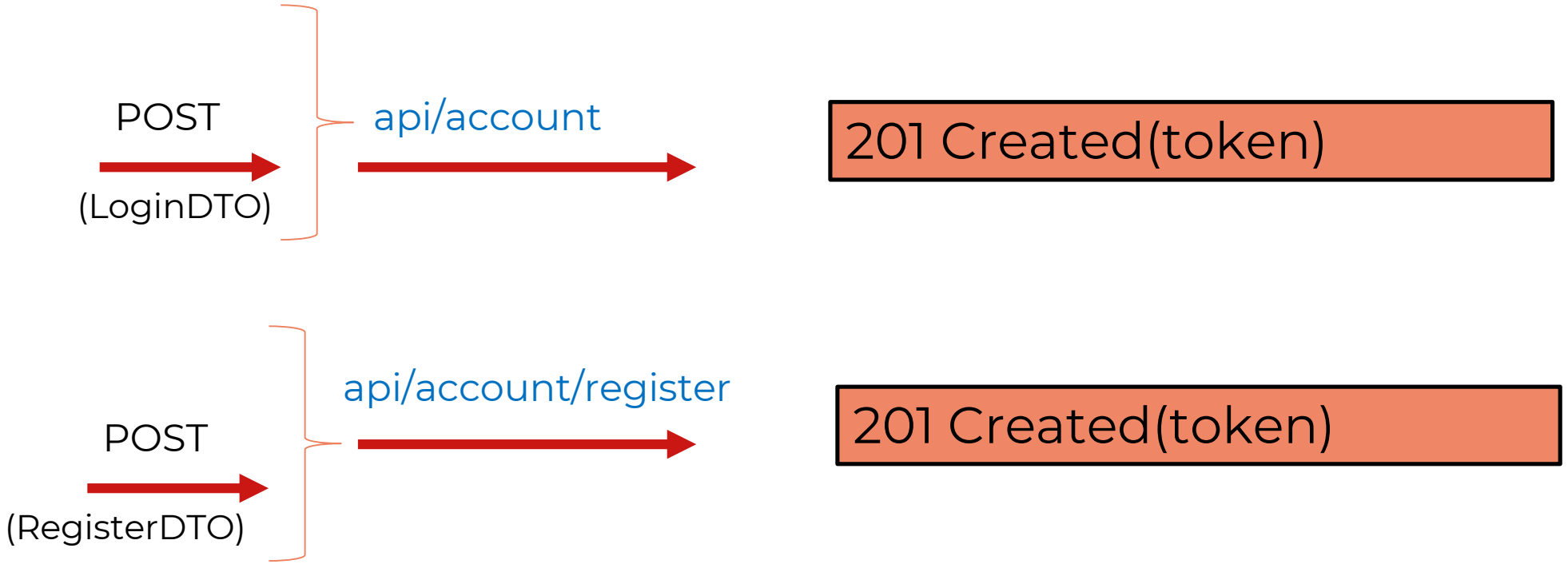
- Step 3 : enable authentication in swagger (https://github.com/RicoSuter/NSwag/wiki/AspNetCore-Middleware)

```
services.AddOpenApiDocument(c =>
        {
            c.DocumentName = "OpenAPI 3";
            c.Title = "Recipe API";
            c.Version = "v1";
            c.Description = "The Recipe API documentation description.";
            c.AddSecurity("JWT", new OpenApiSecurityScheme
            {
                Type = OpenApiSecuritySchemeType.ApiKey, //use API keys for authorization. An API key is a token that a client provides when making API calls.
                In = OpenApiSecurityApiKeyLocation.Header, //token is passed in the header
                Name = "Authorization", //name of header to be used
                Description = "Type into the textbox: Bearer {your JWT token}. " //description above textfield to enter bearer token
            });

            c.OperationProcessors.Add(
                new AspNetCoreOperationSecurityScopeProcessor("JWT")); //adds the token when a request is send
        });
```

Authentication

- Step 4 : AccountController



HO GENT

Authentication

- Step 4 : AccountController
  - First create a LoginDTO in the DTO's folder, to supply the credentials

```
 using System.ComponentModel.DataAnnotations;

namespace RecipeApi.DTOs
{
    public class LoginDTO
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        public string Password { get; set; }
    }
}
```

HO GENT

Authentication

- Step 4 : AccountController
  - Add an empty API controller and Inject SignInManager, UserManager, ICustomerRepository and IConfiguration

```
public class AccountController : ControllerBase    {
       private readonly UserManager<IdentityUser> _userManager;
       private readonly SignInManager<IdentityUser> _signInManager;
       private readonly ICustomerRepository _customerRepository;
       private readonly IConfiguration _config;
       public AccountController(
         SignInManager<IdentityUser> signInManager,
         UserManager<IdentityUser> userManager,
         ICustomerRepository customerRepository,
         IConfiguration config)       {
           _signInManager = signInManager;
           _userManager = userManager;
           _customerRepository = customerRepository;
           _config = config;
       }
}
```

Authentication

- Step 4 : AccountController
  - add an authentication API to your application so that user can authenticate to get new JWTs.

```
[AllowAnonymous]
[HttpPost]
public async Task<ActionResult<String>> CreateToken(LoginDTO model)
{
   var user = await _userManager.FindByNameAsync(model.Email);
   if (user != null){
       var result = await _signInManager.CheckPasswordSignInAsync(user, model.Password, false);
       if (result.Succeeded){
         string token = GetToken(user);
         return Created("", token); //returns only the token
       }
  }
  return BadRequest();
}
```

Authentication

- Step 4 : AccountController
  - Generate token

```
private String GetToken(IdentityUser user)
        {      // Create the token
            var claims = new[]       {
              new Claim(JwtRegisteredClaimNames.Sub, user.Email),
              new Claim(JwtRegisteredClaimNames.UniqueName, user.UserName)    };
            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Tokens:Key"]));
            var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
            var token = new JwtSecurityToken(
              null,null,claims,
              expires: DateTime.Now.AddMinutes(30),
              signingCredentials: creds);
            return new JwtSecurityTokenHandler().WriteToken(token);
        }
    }
```

Authentication

- Step 4 : AccountController
  - Generate token (https://tools.ietf.org/html/rfc7519)
    - When the user is succesfully logged in, create a jwt that identifies the user securely.
    - Store claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.
    - Securely sign the JWT claims, so when it gets back to the server we can confirm the identity, often roles are stored here too (admin or not)
    - you want tokens to expire, so add an expiration date and sign it too
    - If you want to play with JWT and put these concepts into practice, you can use jwt.io Debugger to decode, verify, and generate JWTs.

HO GENT

Authentication

- Step 5 : RecipeController
  - Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema.

```
Authorization: Bearer <token>
```

  - This can be stateless authorization mechanism. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case. If token is invalid: 401 unauthorized is send

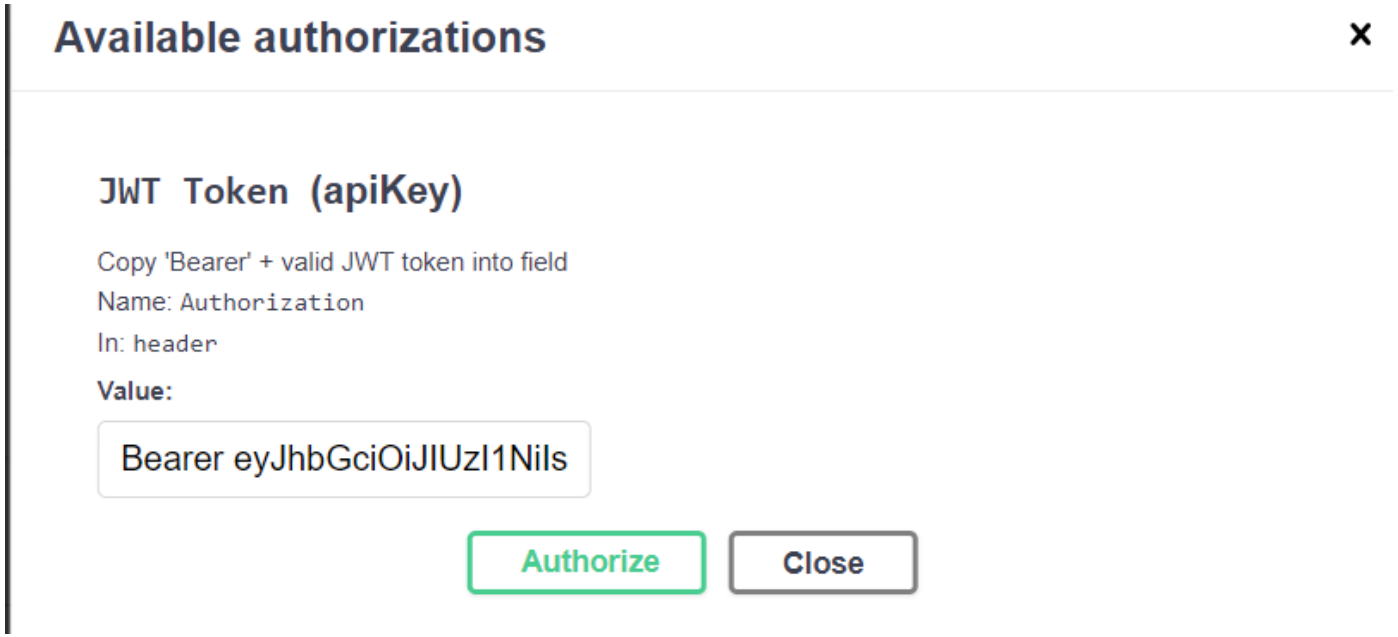
```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

  - Use [AllowAnonymous] for methods without authentication needs

HO GENT

Authentication

- Step 6 : Swagger UI, test it out
  - Login with email and password. The response contains the token value
  - Click the Authorize Button
  - Enter Bearer followed by a space and the token value



HO GENT

Authentication

- Step 4 : AccountController
  - Register method : Add RegisterDTO in DTOs folder

```
public class RegisterDTO : LoginDTO
    {
        [Required]
        [StringLength(200)]
        public String FirstName { get; set; }

        [Required]
        [StringLength(250)]
        public String LastName { get; set; }

        [Required]
        [Compare("Password")]
        [RegularExpression("^((?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])|(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[^a-zA-
Z0-9])|(?=.*?[A-Z])(?=.*?[0-9])(?=.*?[^a-zA-Z0-9])|(?=.*?[a-z])(?=.*?[0-9])(?=.*?[^a-zA-Z0-9])).{8,}$",
ErrorMessage = "Passwords must be at least 8 characters and contain at 3 of 4 of the following: upper case
(A-Z), lower case (a-z), number (0-9) and special character (e.g. !@#$%^&*)")]
        public String PasswordConfirmation { get; set; }
    }
```

Authentication

- Step 4 : AccountController
  - Register method

```
[AllowAnonymous]
[HttpPost("register")]
public async Task<ActionResult<String>> Register(RegisterDTO model)
{
  IdentityUser user = new IdentityUser { UserName = model.Email, Email = model.Email };
  Customer customer = new Customer { Email = model.Email, FirstName = model.FirstName,
LastName = model.LastName };
  var result = await _userManager.CreateAsync(user, model.Password);
  if (result.Succeeded) {
    _customerRepository.Add(customer);
    _customerRepository.SaveChanges();
    string token = GetToken(user);
    return Created("", token);
}
return BadRequest();
}
```

Authentication

- Step 7 : Get Favourite recipes of signed in user

```
[HttpGet("Favourites")]
    public IEnumerable<Recipe> GetFavourites()
    {
        Customer customer = _customerRepository.GetBy(User.Identity.Name);
        return customer.FavoriteRecipes;
    }
```

HO GENT

Extra

- Add RoleClaims
  - add the policies in the startup.cs configureServices (see Web 3)

```
services.AddAuthorization(options =>    {
    options.AddPolicy("AdminOnly", policy => policy.RequireClaim(ClaimTypes.Role, "admin"));});
```

  - add the claims for the user on registering (and in the initializer) (see Web 3)

```
await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Role, "admin"));
```

  - Add the Authorize Attribute (see Web 3)

```
[Authorize(Policy = "admin")]
```

  - Add the role claims to the JWT token

```
var roleClaims = await _userManager.GetClaimsAsync(user);
var claims = new List<Claim>() { …..}
claims.addRange(roleClaims);
```

HO GENT

Extra

- Secure your API using Auth0 : https://auth0.com/docs/quickstart/backend/aspnet-core-webapi/01-authorization en/of https://auth0.com/blog/how-to-build-and-secure-web-apis-with-aspnet-core-3/

- [JsonProperty], [JsonIgnore]

HO GENT

Testing the API

HO
GENT

Postman

- For Bearer authentication
  - Authorization tab > type: Bearer and enter the token (just the token value without the “”)

- For automated testing: https://www.postman.com/automated-testing
  - Create collections
  - For each scenario, create Request and write test (use the snippets)

- Postman tutorial: https://www.toolsqa.com/postman-tutorial/

HO GENT

Publish on Azure

HO
GENT

Publish on Azure

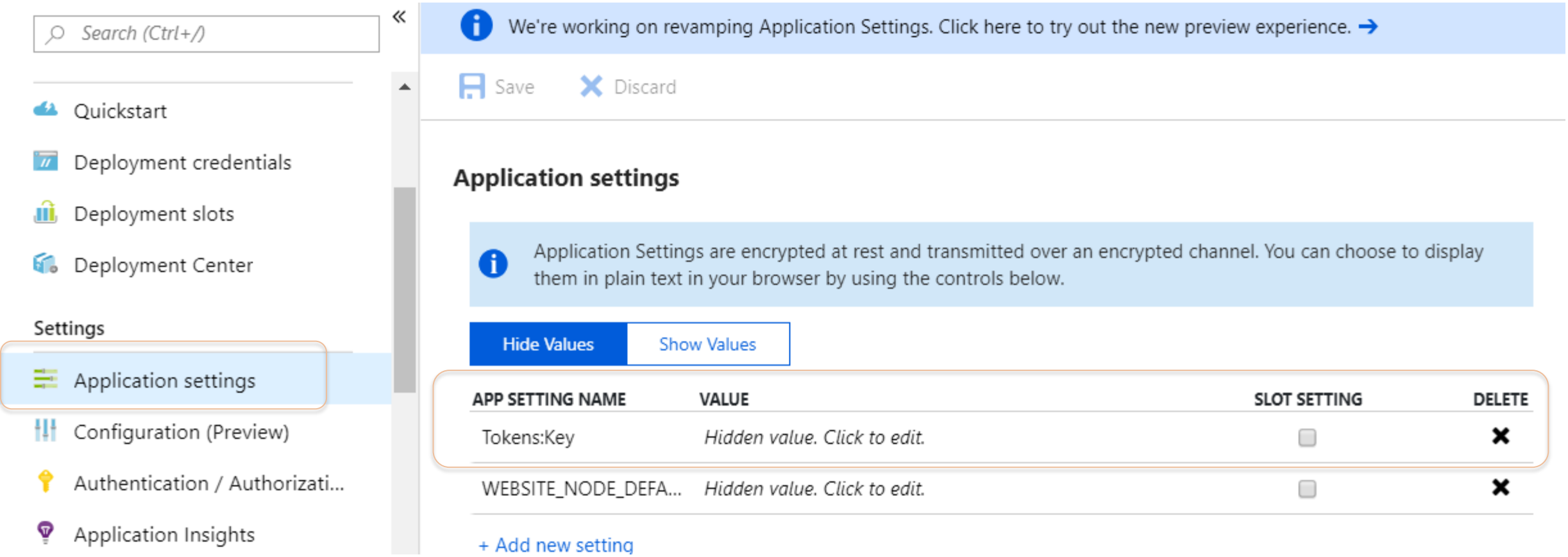
Met je Hogent account kan je aanmelden op https://azure.microsoft.com/nl-nl/free/students/ en heb je een tegoed van 100$

- Remove the code in startup.cs for running the initializer
- Installleer nuget package Microsoft.EntityFrameworkCore.Tools
- Create a migration for the database, open the Package Manager Console
  - Add-Migration InitialCreate
  - Update-database (verwijder eerst je locale database)
- Rightclick the project > Publish
  - Kies voor App Services, follow the tutorial
  - More on https://docs.microsoft.com/en-us/aspnet/core/tutorials/publish-to-azure-webapp-using-vs?view=aspnetcore-3.1
  - Connection string name = RecipeContext

HO GENT

Publish on Azure

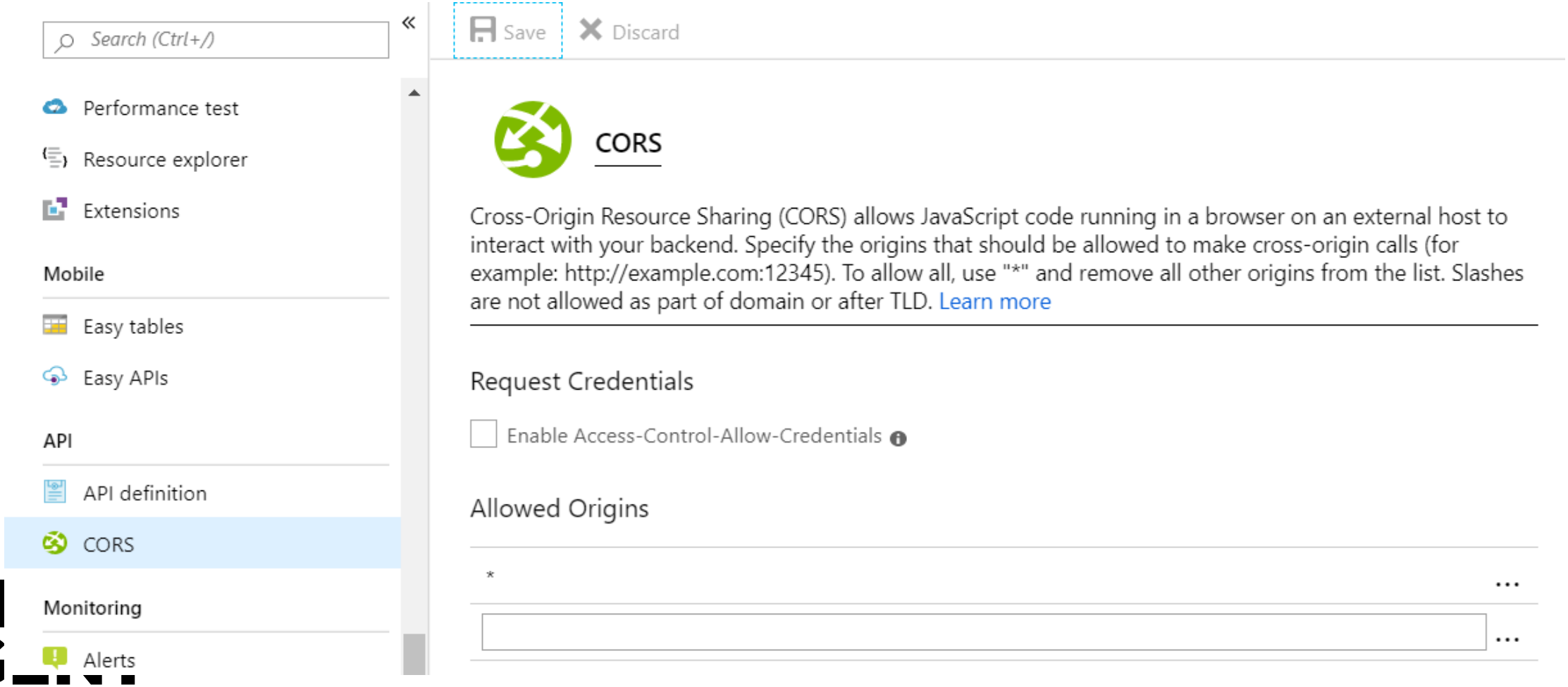
- Enter the key Tokens:Key in the application settings on azure



GENT

Publish on Azure

- Enable Cors



Publish on Azure

- The Angular app
  - Use package https://github.com/Azure/ng-deploy-azure to Deploy Angular apps to Azure using the Angular CLI
    - ng add @azure/ng-deploy to add the package to your app
      - you may be prompted you to sign in to Azure, providing a link to open in your browser and a code to paste in the login page.
    - ng run recipeapp:deploy to deploy your project to Azure.

HO GENT