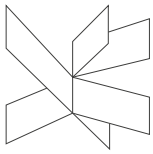


E- Recipe Platform

Project Report

Student Name	*Student No*
Dumitru Rares Bunea	*266983*
Vladimir Rotaru	*266914*

17626 characters

ICT Engineering

December 2, 2019

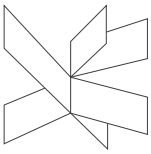
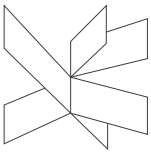
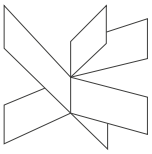


Table of Contents



List of figures and tables

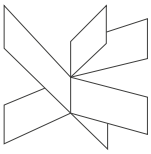


Abstract

Food is what all living creatures need in order to survive. It comes in different forms and aspects, for each individual and plant. Food might be sun energy, a plant, water or meat, it is always different, but its intent is to furnish energy and nutrition.

For this semester project a new electronic yellow book of recipes was proposed to be implemented. The name of this system followed to be “ERecipe”. The aim of the project was to create a platform where people all around the world could post recipes and also inspire themselves by researching other recipes posted.

The main technical challenges were to build a three-tier architecture system, using Java and C#. The system should consume and expose a web service and have a GUI for each client. The most important aspects of system development such as: analysis, design, implementation and testing; are described in this report



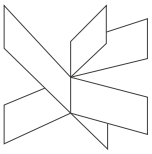
1. Introduction

Since ancient times, people as well as animals were hunting for food. It was one of the basic instincts of all living creatures. Throughout years, people have adapted themselves to their environments and hunting has been replaced with agriculture, pastoralism and farming.

Food has always served as a carrier of culture in human society, people were inventing different dishes that were meant to mirror their affiliation to a specific culture, country or region. They were keeping the instructions and ingredients needed to make the dish, also known as recipe, usually in written form, for future generations. It was their treasures and no other individuals could know their secrets of making a particular dish, except their family members.

As the times have changed, people started to share their recipes with the world as a way of promoting their country and culture. Also, traveling to other countries have enhanced the spirit of trying some new dishes. As an example, is famous pizza and croissants have spread the world and now there is no person who doesn't know these amazing dishes.

From all said above, has come the idea of making an E-Recipe platform available for everyone who wants to share a recipe and inspire themselves with other recipes. The system will make it possible for the user to log in, then, the user can create a new recipe specifying a name, a description, ingredients, steps, author and country from where the recipe has come from.



2. Requirements

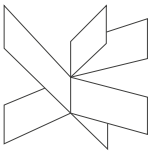
During the inception phase, some core requirements has been determined in order to create the project. The set of functional and non-functional requirements have settled the system functionalities and features. Both types of requirements will be covered in the upcoming sections.

2.1 Functional Requirements

Functional requirements were made in a table containing the name of the feature and the user story corresponding to the name.

Table 1: Functional requirements

NAME	User story
USER REGISTRATION	As a user I want to be able to create an account so I can start using the system
USER LOG-IN	As a user I want to log in order to have my information personalized
USER INFO UPDATE	As a user I want to update my personal information
RECIPES OVERVIEW	As a user I want to see recipes posted by other people
RECIPE CREATION	As a user I want to be able to create a new recipe
RECIPE UPDATE	As a user I want to be able to update my recipe
RECIPE DELETION	As a user I want to delete my recipe
RECIPE POSTING	As a user I want to post my recipe
RECIPE REVIEW	As a user I want to make review to the recipes
RECIPE SEARCH	As a user I want to search for recipes
RECIPE CATEGORIZING	As a user I want to have categories of recipes



2.2 Non-Functional Requirements

The non-functional requirements are defined below in a table, having an ID for each requirement as well as the description of the requirement.

Table 2: Non-functional requirements

ID	NON-FUNACTIONAL REQUIREMENT
1	The system must be heterogeneous
2	The system must be distributed
3	The system must be a three-tier architecture
4	The system must consume and expose a web service
5	The system must use a GUI for each client
6	The system should implement security features
7	The system should be maintainable
8	The system should have different layers of access

3 Analysis

3.1 Business Model

During the phase of analysis, based on the user stories outlined in section 2.1 – Functional Requirements, one actor has been identified – the **USER**. The actions that the user need to be able to perform are described below in a Use Case Diagram.

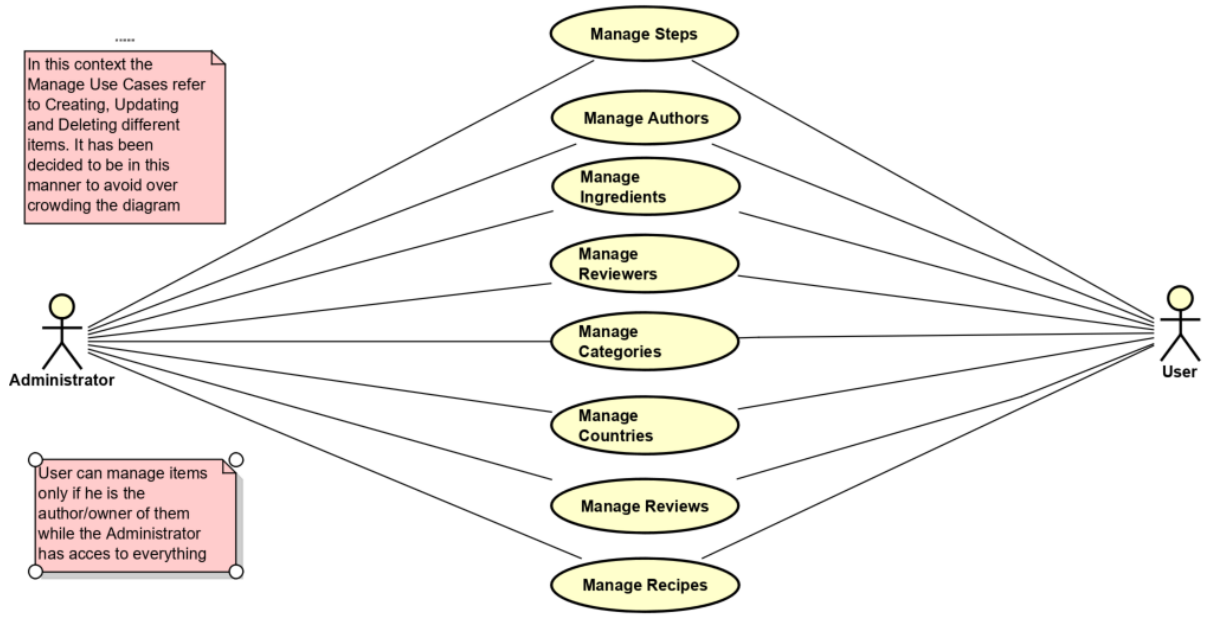
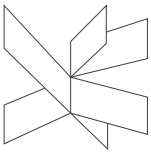


Figure 1 - Use Case Diagram

As displayed in the above diagram, two type of users were designed to be implemented in out system. “User” impersonates a day-to-day person, with passion toward learning more about the culinary art, or just wanting to share his different already known recipes. On the other hand, the “Administrator” can be seen as a watching eye on the well-fare of the community.

The main difference between the two is that while the “User” can manage, which basically means creating, updating and deleting, only item that he has personally created/posted, the administrator can manage everything that would possibly exist in the UI.



The core classes of the system and the way they cooperate between one another are the actions the user can perform. The interaction is shown below in a Domain Model, containing the main objects and classes described earlier.

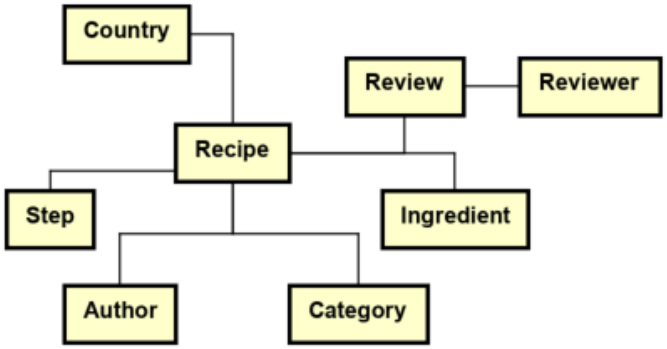


Figure 2 - Domain Model

As of the figure showed above, in the domain model, the main entities to be used in our system got a name and relations between them were made.

This Domain Model diagram was evolved, taking shape of a Conceptual Diagram to better explain the relationships between our entities.

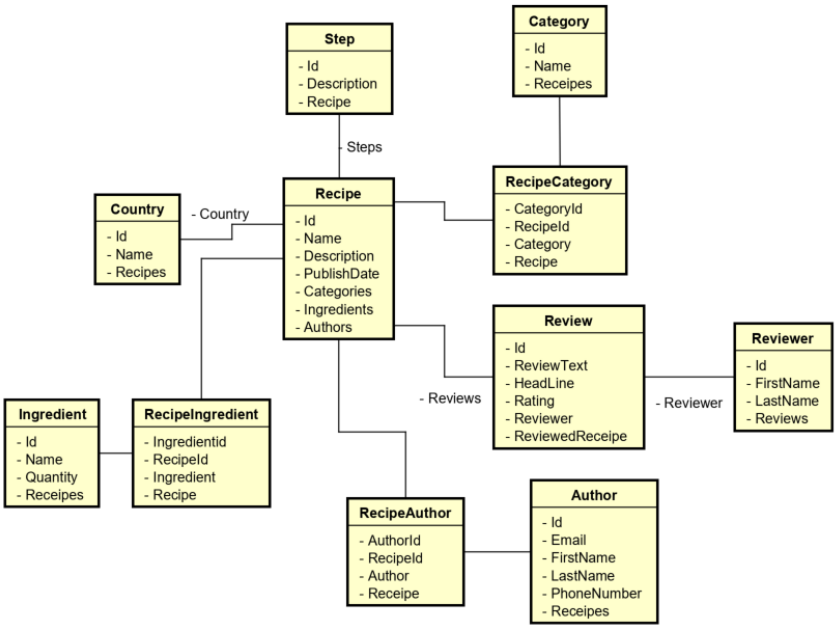


Figure 3 - Conceptual Diagram

With the help of the two previous presented diagrams, it is acknowledged that the entity that is the focus is the “Recipe”. In this business context, a recipe will contain as personal information a name, description and a publish date. With the assistance of the other differentiated entities, the recipe will also contain a country and lists of steps, ingredients and reviews. Due to multiplicity, three “different” entities were created to better form the many-to-many relations in our case: RecipeCategory, RecipeIngredient and RecipeAuthor. A simple explanation for this is that, for example, a recipe can have many authors, but at the same time an author can write more than one recipe. So, to better symbolize this relation, these joined classes were created to improve the navigation between our objects.

Also, to describe the basic flow between the user and the system, a small and simple sequence diagram was made, as shown in the figure below – System Sequence Diagram.

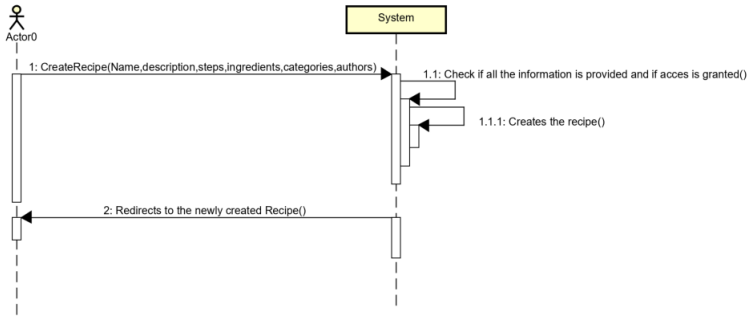


Figure 4 - System Sequence Diagram

The idea would be that every action that would be requested has to be validated. This means that the data has to be checked and users have to be verified if they are allowed to proceed with their actions.

4 Design

4.1 Data Flow

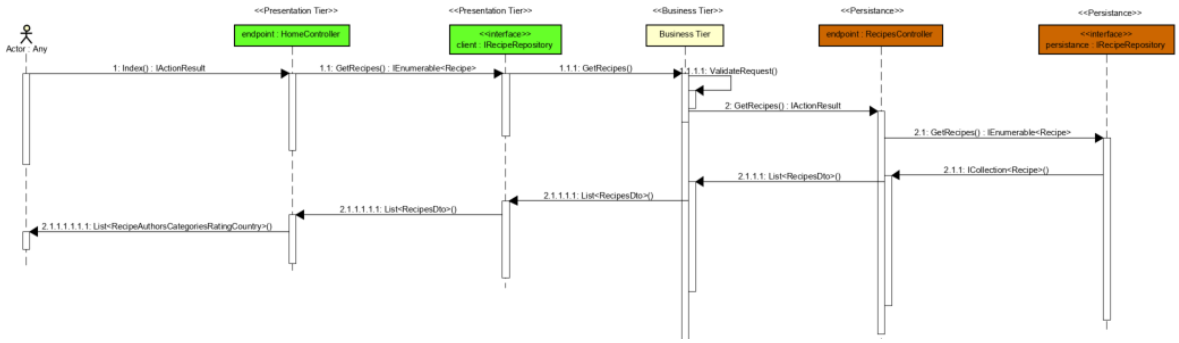


Figure 5 - System Sequence Diagram

This system sequence diagram is designed with the purpose of better understanding how data will be manipulated in our system. When a user makes a request form the Presentation Tier, the request is sent to the Business Logic tier which checks the validity of the data and if the user that made the request is authorized to procced. Then the request is retrieved trough the Persistence Tier of the system with different queries and it is sent back to the user. On the way back from the Persistence Tier to the user, data is altered in such a way that only the relevant information is displayed. Not relevant data, such as navigational proprieties, would not have any value in being presented or event being sent, since it would only consume resources without any true purpose.

4.2 Design Patterns

As the requirements suggest, the projects needs to implement a three-tier architecture. This type of architecture follows "Layered Pattern", being one of the most used patterns in software architecture. The idea of it is to split the system into layers, where each of layers is responsible for particular features of the system and provides sevices to the other layers through endpoints.

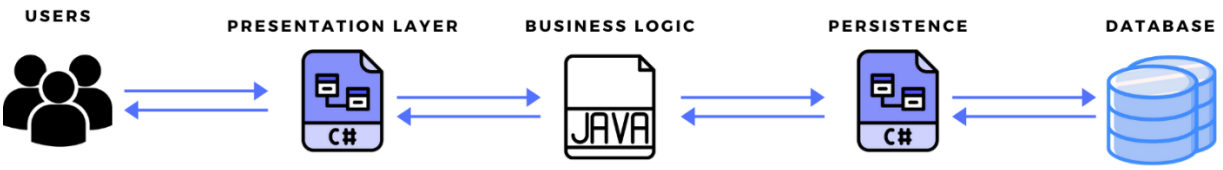
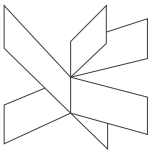


Figure 6 – Architecture Pattern

As it is shown in the diagram above, each layer has a particular purpose. The **Presentation Layer** is responsible for graphical design and user experience with the application. Also, presentation layer is responsible for interaction between users and Business Logic. In this case, in our project, the layer is written in ASP.NET Core, with a use of modified MVC pattern. The MVC pattern is not a usual one as it has Data Transfer Objects, Controllers, View Models and Views folders, but does not have the actual Models folder. Except MVC pattern, the Presentation Layer uses Razor Pages because it gives more control over the actions happening on the web page. In order to communicate with business logic, JSON serialized objects would be used.

The **Business Logic Layer** should contain the primary logic of the system. It was decided that this layer will be written in Java. The business logic tier should receive appropriate data sent from presentation layer, or the user interface, and should pass it to persistence layer as JSON. Then the persistence layer should send appropriate data back to the business logic, also in a JSON format. The business logic should send data back to the presentation layer.

The **Persistence Layer** is the layer that manipulates the **Database** through a connection string, that represents an individual code, some properties. It is generated by ASP.NET Core when creating the database.

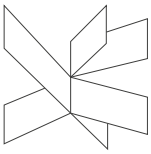


4.3 Class Diagram

Due to the big size, class diagram will be appended

A class diagram was designed for each tier. They were created to make future implementation easier, but also for a better understanding of the business context and the logic of the system that it is to be implemented.

The implementation of the model classes was “tested” using Doxygen, a software that turns C# code into xml, that Astah can understand and reproduce in diagrams. The implementations matched the initial design.



5 Implementation

5.1 Persistence Tier

The web server is implemented using C#, more exactly ASP.Net Core technologies.

It was decided that the way to address connecting, creating and populating a database would be using Entity's Framework code first approach.

This meant that the first step was to create different model classes and specify certain constraints for them. This was done in order to be sure that the final result was equivalent to the initial expectations. An example for such a model could be the "Author".

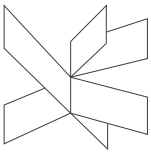
```
public class Author
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    10 references
    public int Id { get; set; }
    [Required]
    [MaxLength(100, ErrorMessage = "First Name cannot be longer than 100 characters")]
    10 references
    public string FirstName { get; set; }
    [Required]
    [MaxLength(200, ErrorMessage = "Last Name cannot be longer than 200 characters")]
    9 references
    public string LastName { get; set; }
    5 references
    public string Email { get; set; }
    5 references
    public string PhoneNumber { get; set; }
    1 reference
    public virtual ICollection<RecipeAuthor> RecipeAuthors { get; set; }
}
```

Figure 7 - Model

The RecipeDbContextClass was created. This class contain all the DbSets that will be exchanged with the database.

```
public RecipeDbContext(DbContextOptions<RecipeDbContext> options)
    : base(options)
{
}

7 references
public virtual DbSet<Recipe> Recipes { get; set; }
5 references
public virtual DbSet<Author> Authors { get; set; }
6 references
public virtual DbSet<Category> Categories { get; set; }
5 references
public virtual DbSet<Ingredient> Ingredients { get; set; }
8 references
public virtual DbSet<Review> Reviews { get; set; }
4 references
public virtual DbSet<Reviewer> Reviewers { get; set; }
4 references
public virtual DbSet<RecipeAuthor> RecipeAuthors { get; set; }
3 references
public virtual DbSet<RecipeCategory> RecipeCategories { get; set; }
4 references
public virtual DbSet<RecipeIngredient> RecipeIngredients { get; set; }
9 references
public virtual DbSet<Step> Steps { get; set; }
4 references
public virtual DbSet<Country> Countries { get; set; }
```

Figure 8 -Db Sets



After creating all of the required models the next step was to ensure that, once with the navigational proprieties, the proper constraints would be also set in the database, so a custom "OnModelCreating" method was created to ensure that the Model Builder that guides our migrations would create the proper connections. In this case connections mainly refer to primary and foreign keys in the database.This method resides in the RecipeDbContext class.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<RecipeCategory>()
        .HasKey(rc => new { rc.RecipeId, rc.CategoryId });
    modelBuilder.Entity<RecipeCategory>()
        .HasOne(r => r.Recipe)
        .WithMany(rc => rc.RecipeCategories)
        .HasForeignKey(r => r.RecipeId);
    modelBuilder.Entity<RecipeCategory>()
        .HasOne(c => c.Category)
        .WithMany(rc => rc.RecipeCategories)
        .HasForeignKey(c => c.CategoryId);
    modelBuilder.Entity<RecipeAuthor>()
       .HasKey(ra => new { ra.RecipeId, ra.AuthorID });
    modelBuilder.Entity<RecipeAuthor>()
        .HasOne(r => r.Recipe)
        .WithMany(ra => ra.RecipeAuthors)
        .HasForeignKey(r => r.RecipeId);
    modelBuilder.Entity<RecipeAuthor>()
        .HasOne(a => a.Author)
        .WithMany(ra => ra.RecipeAuthors)
        .HasForeignKey(a => a.AuthorID);
    modelBuilder.Entity<RecipeIngredient>()
       .HasKey(ri => new { ri.RecipeId, ri.IngredientId });
    modelBuilder.Entity<RecipeIngredient>()
        .HasOne(r => r.Recipe)
        .WithMany(ri => ri.RecipeIngredients)
        .HasForeignKey(r => r.RecipeId);
    modelBuilder.Entity<RecipeIngredient>()
        .HasOne(i => i.Ingredient)
        .WithMany(ri => ri.RecipeIngredients)
        .HasForeignKey(i => i.IngredientId);
}
```

Figure 9 - Model Builder

With this step being done, the database was tested by checking every key that was created, but also by autogenerating the ER diagram of the database. This was done by changing the connection string from a MSSQL server to a local Db, enabling auto migrations and then opening the database in Microsoft SQL Server Management Studio 17 were the diagram could be finally checked.

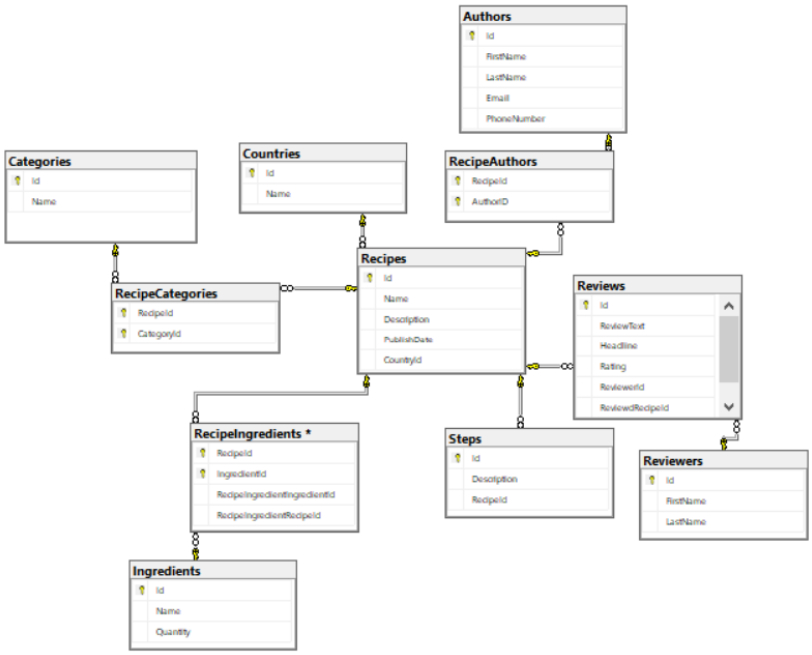
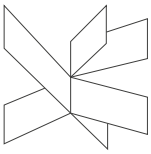


Figure 10 - Database Diagram



The result fell perfectly in line with already set expectations. But to be entirely sure that nothing could be forgotten the database got data added to it by seeding it through a class that has this sole purpose. The class is then called once in the startup and then commented out, to avoid adding the same data in the database every time the program would run.

```
public static class DbSeedingClass
{
    0 references
    public static void SeedDataContext(this RecipeDbContext context)
    {
        var recipeAuthors = new List<RecipeAuthor>()
        {
            new RecipeAuthor()...,
              new RecipeAuthor()...
        };

        context.RecipeAuthors.AddRange(recipeAuthors);
        context.SaveChanges();
    }
}
```

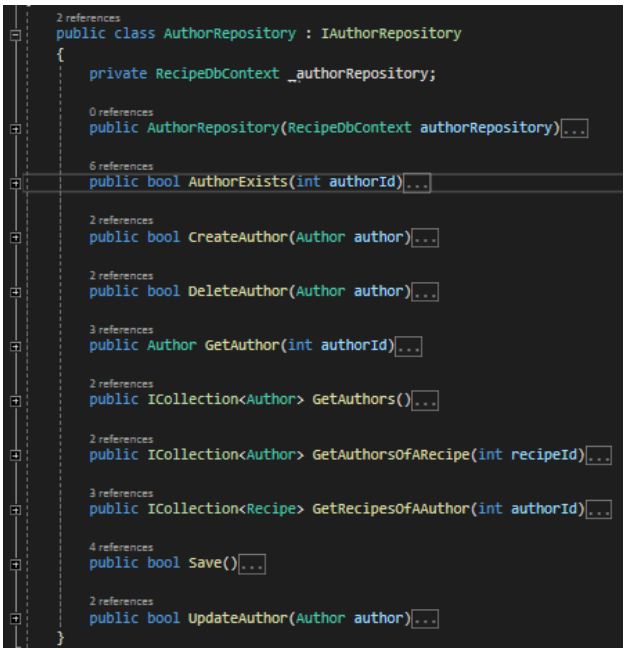
Figure 11 -Seeding

On of the greatest thing of .net Core is it's moduled state, that different components can be brought to a system as services. This is where Interfaces do their part. Connecting a concrete class through it's interface makes a system more maintainable and scalable.

For each of the model classes a IRepository and a concreate Repository class were created to exchange data with our database.

```
public interface IAuthorRepository
{
    2 references
    ICollection<Author> GetAuthors();
    3 references
    Author GetAuthor(int authorId);
    2 references
    ICollection<Author> GetAuthorsOfARecipe(int recipeId);
    3 references
    ICollection<Recipe> GetRecipesOfAAuthor(int authorId);
    6 references
    bool AuthorExists(int authorId);
    2 references
    bool CreateAuthor(Author author);
    2 references
    bool UpdateAuthor(Author author);
    2 references
    bool DeleteAuthor(Author author);
    4 references
    bool Save();
}
```

Figure 12 - IRepository



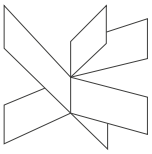
```
2 references
public class AuthorRepository : IAuthorRepository
{
    private RecipeDbContext _authorRepository;

    0 references
    public AuthorRepository(RecipeDbContext authorRepository)...

    6 references
    public bool AuthorExists(int authorId)...

    2 references
    public bool CreateAuthor(Author author)...

    2 references
    public bool DeleteAuthor(Author author)...

    3 references
    public Author GetAuthor(int authorId)...

    2 references
    public ICollection<Author> GetAuthors()...

    2 references
    public ICollection<Author> GetAuthorsOfARecipe(int recipeId)...

    3 references
    public ICollection<Recipe> GetRecipesOfAAuthor(int authorId)...

    4 references
    public bool Save()...

    2 references
    public bool UpdateAuthor(Author author)...
}
```

Figure 13 - Repository

A repository class implements a interface class. The reference to the repositories is made in the startup class.

```
services.AddScoped<ICountryRepository, CountriesRepository>();
services.AddScoped<ICategoryRepository, CategoryRepository>();
services.AddScoped<IAuthorRepository, AuthorRepository>();
services.AddScoped<IIngredientRepository, IngredientRepository>();
services.AddScoped<IReviewerRepository, ReviewerRepository>();
services.AddScoped<IReviewRepository, ReviewRepository>();
services.AddScoped<IStepRepository, StepRepository>();
services.AddScoped<IRecipeRepository, RecipeRepository>();
```

Figure 14 - Referencing In Startup



On top of the Repositories, Controller were built so that different endpoint would be exposed. With these endpoints, communication with other tiers of the system would be possible. But data cannot be sent in the form of the existing models to other system tiers, because these models contain navigational proprieties that only make sense to have in the Persistence tier.

And so, Data Transfer Objects had to be created, one for each model class we have.

```
public class AuthorDto
{
    3 references
    public int Id { get; set; }
    3 references
    public string Email { get; set; }
    3 references
    public string FirstName { get; set; }
    3 references
    public string LastName { get; set; }
    3 references
    public string PhoneNumber { get; set; }
}
```

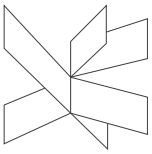
Figure 15 - DTO

These objects only contain the information that makes sense to have and use in the other tiers of our system.

The last step was to create the Controllers that would expose the needed endpoints, one controller for each model. The controllers work using HTTP requests and they were designed keeping in mind REST principles.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class AuthorsController : Controller
{
    private IAuthorRepository _authorRepository;
    private IRecipeRepository _recipeRepository;

    0 references
    public AuthorsController(IAuthorRepository authorsRepository, IRecipeRepository recipeRepository)
    {
        _authorRepository = authorsRepository;
        _recipeRepository = recipeRepository;
    }

    //api/authors
    [HttpGet]
    [ProducesResponseType(200, Type = typeof(IEnumerable<AuthorDto>))]
    [ProducesResponseType(400)]
    0 references
    public IActionResult GetAuthors()...

    //api/authors/authorId
    [HttpGet("{authorId}" , Name = "GetAuthor")]
    [ProducesResponseType(200, Type = typeof(AuthorDto))]
    [ProducesResponseType(400)]
    [ProducesResponseType(404)]
```

Figure 16 - Controller



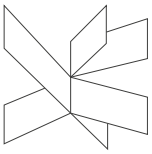
5.2 Business Logic Tier

Business Logic is the most important part of the system. Being situated between Presentation tier and Persistence tier, it plays a key role in three tier architecture. Its purpose is to handle the requests which are coming from the Presentation tier which must be filtered depending on the fact if the request sent is valid or not. In case of a valid request, then it communicates with the Persistence making the necessary data exchange. Also, it is responsible for information handling and validation.

Communication between tiers was made through webservices. Persistence would expose and Business Logic would consume the information. Then, Business Logic would expose the received information for the Presentation tier.

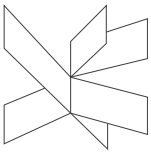
```
public static String GetAuthors() throws IOException
{

    String readLine = null;
    URL urlForGetRequest = new URL(baseRoute+"authors");
    HttpURLConnection conection = (HttpURLConnection) urlForGetRequest.openConnection();
    conection.setRequestMethod("GET");

    int responseCode = conection.getResponseCode();
    if (responseCode == HttpURLConnection.HTTP_OK) {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(conection.getInputStream()));
        StringBuffer response = new StringBuffer();
        while ((readLine = in .readLine()) != null)
        {
            response.append(readLine);
        } in .close();
        System.out.println(response.toString());
    }

    return readLine;
}
```

In the figure above, data from Persistence tier is being retrieved to Business Logic. All of the authors are retrieved by using the “GET” method. Before storing the object, it is checked if the response code is “OK”. Retrieved data is stored in a Json object and then it is sent further, to presentation tier.



```
@RestController
public class CountryController {
    @Autowired
    private CountryDaoService service;
    @GetMapping("/countries")
    public String GetCountries() throws IOException
    {
        return service.GetCountries();
    }
    @GetMapping("/countries/{id}")
    public String GetCountry(@PathVariable int countryId) throws IOException
    {
        return service.GetCountry(countryId);
    }
    @GetMapping("/countries/recipes/{id}")
    public String GetCountryOfRecipe(@PathVariable int recipeId) throws IOException
    {
        return service.GetCountryOfRecipe(recipeId);
    }
    @GetMapping("/countries/{id}/recipes")
    public String GetRecipesFromCountry(@PathVariable int countryId) throws IOException
    {
        return service.GetRecipesFromCountry(countryId);
    }
}
```

In the picture above data is transferred to presentation tier by using the RESTful web services. First, class is marked as a controller by setting the “@RestController” mark. Auto wiring happens through the created property “service” of type “CountryDaoService”. Every method has a personal mapping where the information will be stored and ready for consumption. Also, a “@PathVariable” is used so that the information can be divided into separate objects.



5.3 Presentation Tier

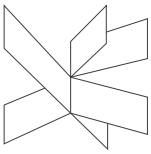
The presentation tier is created with the purpose of displaying data to the use and should be connected to the Business Logic Tier of the system. Since the Business Tier of the system was not read in due time, the Presentation Tier, for now, is connected straight to the Persistence Tier of the system. This is done by consuming the API expose by the Persistence Tier of the system and sending HTTP request to the endpoints.

Information is displayed with the help of ASP.NET core Razor pages. Minor styling has been added to the project. This was achieved by using Bootstrap.

All Recipes

Name	Authors	Country	Published	Rating	Categories	
Cheese Cake	JamesPotter View PabloPablito View	Mexic	Nov 2019	No rating yet	burger View fast View	View Recipe
File mignion	JamesPotter View PabloPablito View	Mexic	Nov 2019	No rating yet	burger View fast View	View Recipe
MexicanBurger	JamesPotter View	America	Nov 2019	3 Stars	burger View	View Recipe
Nacho cheeseburger	PabloPablito View	Mexic	Nov 2019	3 Stars	fast View burger View	View Recipe
Polenta	JamesPotter View PabloPablito View	Romania	Dec 2019	No rating yet	burger View fast View	View Recipe
Tiramis	JamesPotter View PabloPablito View	Mexic	Nov 2019	No rating yet	burger View fast View	View Recipe

Figure 17 - Home Page



For the data transfer to be consistent equivalent DTO objects had to be implemented in this tier.

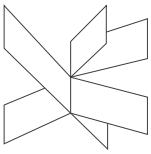
```
public class AuthorDto
{
    namespace ERecipePresentation.DTO
        public class AuthorDto
    1 reference
    public string Email { get; set; }
    5 references
    public string FirstName { get; set; }
    5 references
    public string LastName { get; set; }
    1 reference
    public string PhoneNumber { get; set; }
}
```

Figure 18 -DTO

The connection between these tiers is made again using IRepositories and Repositories classes. The difference is that in this case, the Repositories are aiming to our Persistence Tier and not to a database.

```
public class AuthorRepository : IAuthorRepository
{
    2 references
    public AuthorDto GetAuthor(int authorId)
    {
        AuthorDto author = new AuthorDto();

        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri("http://localhost:49951/api/");

            var response = client.GetAsync($"authors/{authorId}");
            response.Wait();

            var result = response.Result;

            if (result.IsSuccessStatusCode)
            {
                var readTask = result.Content.ReadAsAsync<AuthorDto>();
                readTask.Wait();

                author = readTask.Result;
            }
        }

        return author;
    }
```

Figure 19 - Repository



The IRepositories and Repositories have to be registered in the Startup class along with other components to make routing possible.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddScoped<ICountryRepository, CountryRepository>();
    services.AddScoped<ICategoryRepository, CategoryRepository>();
    services.AddScoped<IAuthorRepository, AuthorRepository>();
    services.AddScoped<IIngredientRepository, IngredientRepository>();
    services.AddScoped<IReviewerRepository, ReviewerRepository>();
    services.AddScoped<IReviewRepository, ReviewRepository>();
    services.AddScoped<IStepRepository, StepRepository>();
    services.AddScoped<IRecipeRepository, RecipeRepository>();
}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseStatusCodePages();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}");
    });
}
```

Figure 20 - Startup

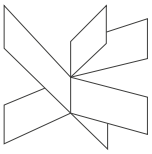
If the Repositories are responsible of retrieving data, Controllers are created to send if further up the system, in this case toward the Views. Each data transfer object has is represented by it's own controller, but in this case they are not enough for displaying required data for each of our web page. This because on each Razor Page you can only use one model, but for example on a recipe page, next to the recipe itself, you also need to display the authors, ingredients, steps, categories, reviews and linked reviewers. So to make this possible, View Models are created according to the needs of each page.

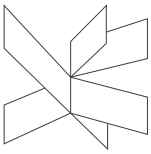
```
4 references
public class CategoriesRecipesViewModel
{
    4 references
    public CategoryDto Category { get; set; }
    2 references
    public IEnumerable<RecipeDto> Recipes { get; set; }
}
```

Figure 21 - ViewModel

This view model is created so that on a category page, created for the GetCategotyById, recipes that belong to that specific category can also be displayed.

In our context, for each action in a controller, a Razor Page is created.





6 Testing

6.1 Persistence Tier

With the project being forced to advance at a vey fast pace, for the presentation and persistence layers no unit tests have been done. This was a risk and a sacrifice that had to be made.

With the persistence layer being finished, a suite of tests for the controllers was created. The implementation itself was extremely helpful with testing, because every controller method specifies all the possible response types and the url format that needed to be accessed.

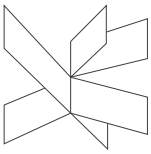
```
//api/countries/countryId
[HttpPut("{countryId}")]
[ProducesResponseType(204)]
[ProducesResponseType(400)]
[ProducesResponseType(422)]
[ProducesResponseType(500)]
[ProducesResponseType(404)]
0 references
public IActionResult UpdateCountry(int countryId, [FromBody]Country updatedCountryInfo)...
```

Figure 22 - Response Types

The test suite that was prepared for the controllers contains tests for all the possible scenarios that can be encountered in our current business context.

Controller Name	Method Name	Parameters	Scenario	Expected Result	Actual Result	Validation
Controller	GetAction (in out contex, always retrieves al ist)	No Parameters	No issue	A list of all the the desired object as Dto objects along with a 200 status code	A list of all the the desired object as Dto objects along with a 200 status code	
			InternalError	A 400 status code along with the model state	A 400 status code along with the model state	
	GetById	Object.Id	Using an existing Id as a parameter	The desired Object as a Dto object along with a 200 status code	The desired Object as a Dto object along with a 200 status code	
			Using a non-existing Id as a parameter	A 404 status code	A 404 status code	
			InternalError	A 400 status code along with the model state	A 400 status code along with the model state	
	GetProprietyOfObject	Propriety.Id	Using an existing Id as a parameter	200 status code along with the desire Propriety Dto	200 status code along with the desire Propriety Dto	
			Using a non-existing Id as a parameter	A 404 status code	A 404 status code	
			InternalError	A 400 status code along with the model state	A 400 status code along with the model state	
	GetObjectListFromPropriety	Object.Id	Using an existing Id as a parameter	Return a list of Object Dto objects that are linked to the given Propriety with a 200 status code	Return a list of Object Dto objects that are linked to the given Propriety with a 200 status code	
			Using a non-existing Id as a parameter	A 404 status code	A 404 status code	
			InternalError	A 400 status code along with the model state	A 400 status code along with the model state	
	Create	Object	valid object	201 Created at route(redirecting)	201 Created at route(redirecting)	
			null object	422 unprocessable entity	422 unprocessable entity	
			malformed object	400 bad request	400 bad request	
	Update	Object and Object.Id	valid object	204 no content	204 no content	
			null object	422 unprocessable entity	422 unprocessable entity	
			malformed object	400 bad request	400 bad request	
			inexisting object id	404 not found	404 not found	
	Delete	Object.Id	valid id	204 no content	204 no content	
			another object depends on the target	409 conflict	409 conflict	
			inexisting object id	404 not found	404 not found	

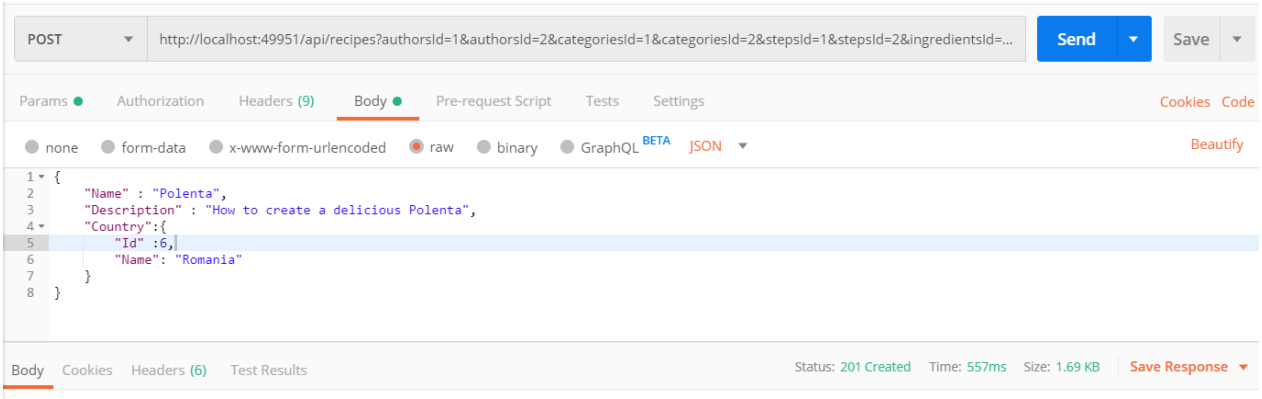
Figure 23 - Test Suite

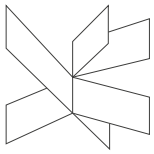


The test suite was a success and it proved that it tested every piece of functionality that the Persistence Tier had to offer.

It is true that one minor flaw was discovered during tests. That is that while trying to update a "Recipe", the response would consist in a 400 Bad Request status code, although the item in discussion would still be updated.

The suite was tested using Postman.



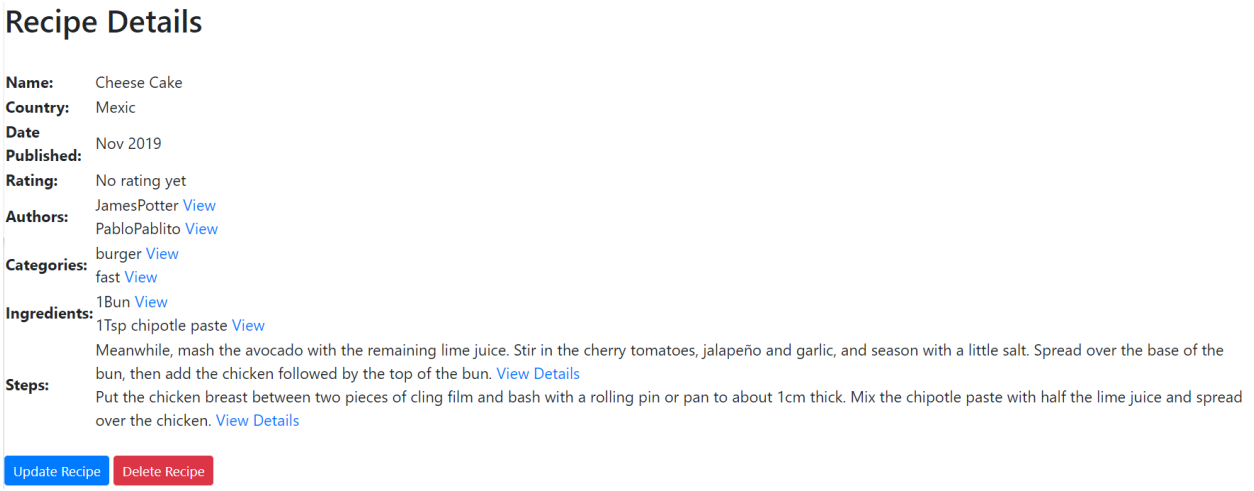


6.2 Business Logic Tier

Testing in Business Logic was done using black-box testing. Information is retrieved successfully from the persistence tier. However, only GET methods were implemented and no testing has been executed on POST, PUT or DELETE.

6.3 Presentation Tier

Since time was still pressuring down on the project, no unit tests were written for the presentation tier, but since the result is a simple website that covers most of the functionality implemented in the first two tires, white box testing was considered sufficient at the time.



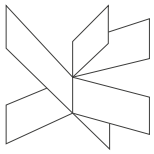


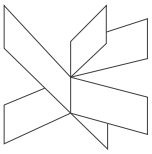
Figure 24 - Layout

7 Results and Discussions

The purpose of this project was to create an electronic book of recipes, called E-Recipe, where user could inspire themselves and try new recipes, as well as post their recipe. The group succeeded in creating the system, accomplished the requirements of having a layered architecture, consume and expose a web service as well as having a GUI for each client. However, there were some issues with creating the Business Logic Tier, written in Java programming language.

In the end, this is a functional project of two-tiers with a friendly user interface, written in C# programming language. The graphical design of the user interface is as user friendly as possible, however, there could have been implemented some more graphical features.

Overall there is room for future improvements and building more features.



8 Conclusion

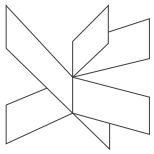
This semester project was challenging from the beginning till the end. The final idea has come a long way till it was developed in a functional and usable application.

A rocky overview of the system is first made in the analysis part, described in the section above. It illustrates system's behavior as well as how it will communicate with the user. Furthermore, the analysis part has taken the team to the opinion of not implementing any security features yet, as it requires lots of time the team didn't have.

In the design section, there can be seen all the diagrams created and a brief explanation of every piece. The diagrams help the team to have a better understanding of how the system will exchange information between the tiers. Also, diagrams sharped the need of using design patterns in order to have a more readable and maintainable project.

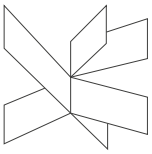
The project implementation section took most of the time as it was a huge system to put into action. However, it was successful implemented along with some White Box testing described in the section above.

As a result, the group members have concluded that a good part of the requirements were accomplished. The system still needs some future improvements, but overall it is great project in the existing timespan.



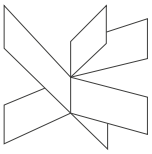
9 Project Future

For future updates of the system, there are a few important functions to be implemented. First, for security features of the system, to have a login functionality in order to personalize users and distinguish between one another. Also, there could be implemented two actors, this being the user and the administrator – who will be responsible for managing users and will have access to more features than the user. Second, implementation of the middle tire, or the business logic tier, written in Java programming language. Third, there could be implemented a more eye-catching user interface, so the user will have a better experience with the application.



Sources of information

Britannica, T. E. o. E., 2019. *Encyclopaedia Britannica.* [Online]
Available at: https://www.britannica.com/topic/food
[Accessed 2 October 2019].



Appendices

Appendix A: Project Description

Appendix B: Use Cases

Appendix C: Test Cases

Appendix D: ERecipe Diagrams

Appendix E: GitHub