Realisation document: FoodBridge

By: Simeon Markov

Institution Name: Fontys UAS

Course/Class: ICT/EN08

Date: 2025-12-04

Introduction

This document highlights the project development phase, detailing the technical structure, implementation strategy and development workflow. It includes explanation of the core logic, explaining major parts of the architecture like controllers, components, writing and dealing with migrations (database changes, modifications).

Project structure

This section presents the system architecture and main components.

System architecture:

Architecture: React & Asp.NET (Client-Server communication)

- **Backend:** Asp.NET handles routing, controllers, models, validation
- **Frontend:** React handles frontend (React components).

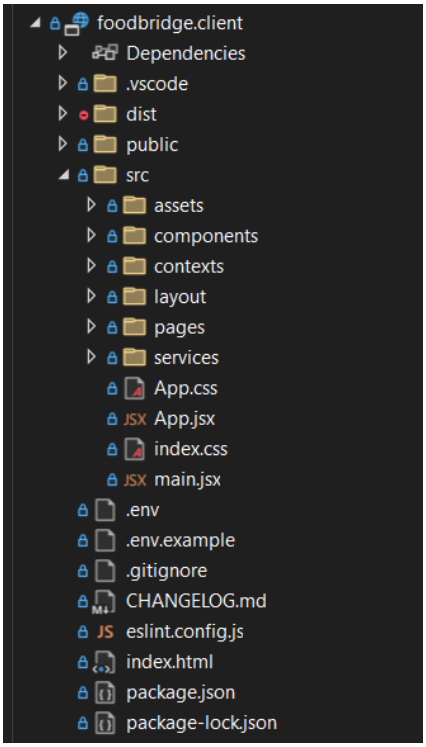

Figure 2: Client-side structure

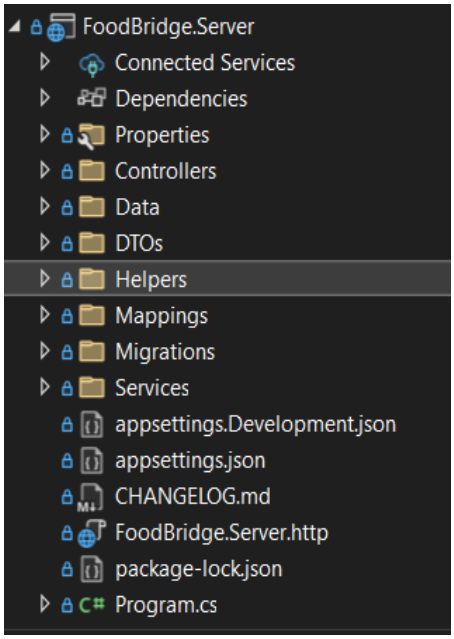

Figure 1: Server-side structure

Design vs Realisation phase

UI Additions

- *Manage Resource:* The manage resource page was modified to include a search bar and filtration, so that the privileged users could easily find and record to either delete or edit it.

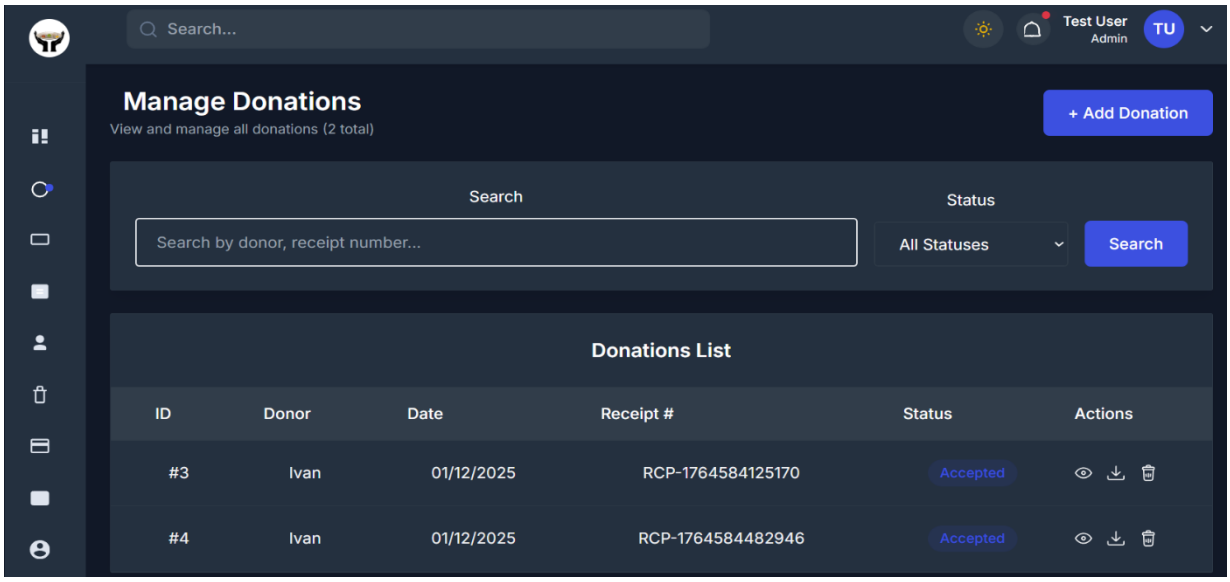

Figure 3: Manage resource page

The page contains internal links to create, edit, and delete a resource.

Code documentation

Main logic

This section explains the main logic behind the project (DotVVM three-tier architecture).

Example flow: First a model with a migration file is created, where the migration file is used to define the model objects, so that the ORM (Entity Framework in this case) could pass it as data object (code-first approach) and the model prepares the data that the application is going to work with (business layer). Then a react component reads from the entity, handles user's requests, passes data to the defined controller, which then returns a response to the page.

```
namespace FoodBridge.Server.Data.Models
{
    [Table("Donations")]
    public class Donation
    {
        [Key]
        public int DonationId { get; set; }

        [Required]
        public int DonorId { get; set; }

        [Required]
        public DateTime DonationDate { get; set; }

        [Required]
        [StringLength(50)]
        public string ReceiptNumber { get; set; }

        [Required]
        [StringLength(50)]
        public string Status { get; set; } // Pending, Inspection, Approved, Rejected, Archived
```

Figure 4: Donation entity defined

Explanation: Each field that will correspond to a column in the table is defined class of the object type (Donation -> Donations). Each annotation like **Key**, **Required** is a constraint for that column in the database.

Controllers: Exposing ASP.NET backend API for the server-side rendering.

```
[HttpGet]
[ProducesResponseType(typeof(ApiResponse<PagedResultDto<ProductDto>>), StatusCodes.Status200OK)]
public async Task<IActionResult> GetAll([FromQuery] ProductFilterDto filter)
{
    try
    {
        var result = await _productService.GetAllAsync(filter);
        return Ok(ApiResponse<PagedResultDto<ProductDto>>.SuccessResponse(result));
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error retrieving products");
        return StatusCode(500, ApiResponse.ErrorResponse("An error occurred while retrieving products")
    }
}
```

Figure 5: Product controller

Explanation: This is the Product controller, where in fig. 5 is shown the fetch method for all available products from the database. It is a Get method for accessing the particular resource. If a successful connection is established, display all the available products, otherwise log an error and return status 500 (server error).

```
[HttpPost]
[Authorize(Roles = "Admin")]
[ProducesResponseType(typeof(ApiResponse<ProductDto>), StatusCodes.Status201Created)]
[ProducesResponseType(typeof(ApiResponse), StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Create([FromBody] CreateProductDto dto)
{
    try
    {
        if (!ModelState.IsValid)
            return BadRequest(ApiResponse.ErrorResponse("Invalid product data", ModelState.Values
            .SelectMany(v => v.Errors)
          .Select(e => e.ErrorMessage)
    .ToList()));

        var product = await _productService.CreateAsync(dto);
        return CreatedAtAction(nameof(GetById), new { id = product.ProductId },
              ApiResponse<ProductDto>.SuccessResponse(product, "Product created successfully"));
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error creating product");
        return StatusCode(500, ApiResponse.ErrorResponse("An error occurred while creating the product"
    }
}
```

Figure 6: POST Method in Product controller

Explanation: This is a Post method for doing updates on the product resource (submitting data to the server). It checks if it is accessed by the admin, then establishes a successful connection otherwise it returns an unauthorized access error.

Services: The backbone of the application. The logic performed on the data is done through them.

```
public async Task<bool> DeleteAsync(int id)
{
    try
    {
        var product = await _context.Products
        .FirstOrDefaultAsync(p => p.ProductId == id);

        if (product == null)
            return false;

        // Soft delete
        product.IsActive = false;

        await _context.SaveChangesAsync();

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error deleting product {ProductId}", id);
        throw;
    }
}
```

Figure 7: Part of Product Service (Delete method)

Explanation: The logic flow is as follows: First the ID of the resource (Product) is taken as a parameter from the URL (Controller is responsible for sending the data, routing), then an instance of the Product class is created, looking for the ID that matches with the product's one and ensuring that there is an actual product. Afterwards the flag field is set to false, indicating soft delete and at last the changes are saved to the database. If an error occurs the 'catch' from the 'try-catch' construction catches the type of the error and the logger logs it into the console.

Components: The views (client side) of the application, responsible for showing frontend to the end-user.

```
const handleSave = async () => {
    if (!validateForm()) return;

    try {
        let response;
        if (modalMode === 'create') {
            response = await productsAPI.create(formData);
        } else {
            response = await productsAPI.update(selectedProduct.productId, formData);
        }

        if (response.success) {
            setSuccess(`Product ${modalMode === 'create' ? 'created' : 'updated'} successfully`);
            handleCloseModal();
            fetchProducts();
        }
    } catch (err) {
        setError(err.response?.data?.message || `Failed to ${modalMode} product`);
    }
};
```

Figure 8: Snippet from the Product component page (save functionality)

Explanation: Before submitting the data to the server, the data is being handled, which means that a validation of each field is done (e.g. allowed nulls, required fields, incorrect data type). Then the state is checked of the product (is the modal dialog open or not? It is in the state or create or edit?) and based on that the data is submitted and an response message to the user is returned otherwise an error message is displayed.

NOTE: For extensive reading of the code documentation please visit:
https://github.com/Simeon31/FoodBridge

Development methodology and techniques

Architectural Patterns & Principles

Pattern/Principle	**Description**
DRY / Single Responsibility	Reusable services and components. Each method, service does one particular job.
Separation of concerns	Dependency Injections, DTOs.
Clean architecture	Monolith architecture, client-server communication.

Table 2: Architectural Patterns & Principles

Conclusion

In conclusion, the project successfully implements a scalable, maintainable food bank system with modern technologies and architectural best practices, delivering a responsive user experience with real-time features and extensibility through modular design.