Creating the List and Detail Pages

Gill Cleeren
CTO Xebia Microsoft Services Belgium

@gillcleeren

Overview

Creating a repository

Adding the category overview

Creating the category detail page

Working with a more complex model

Creating a Repository

We “can” use the DbContext from our controllers...

... but it’s often a better idea to separate logic into different classes.

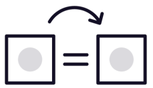
Repositories can be used to contain the EF Core logic.

Introducing a Repository

Mediates between domain and data access

Hides logic to retrieve or store data in one place

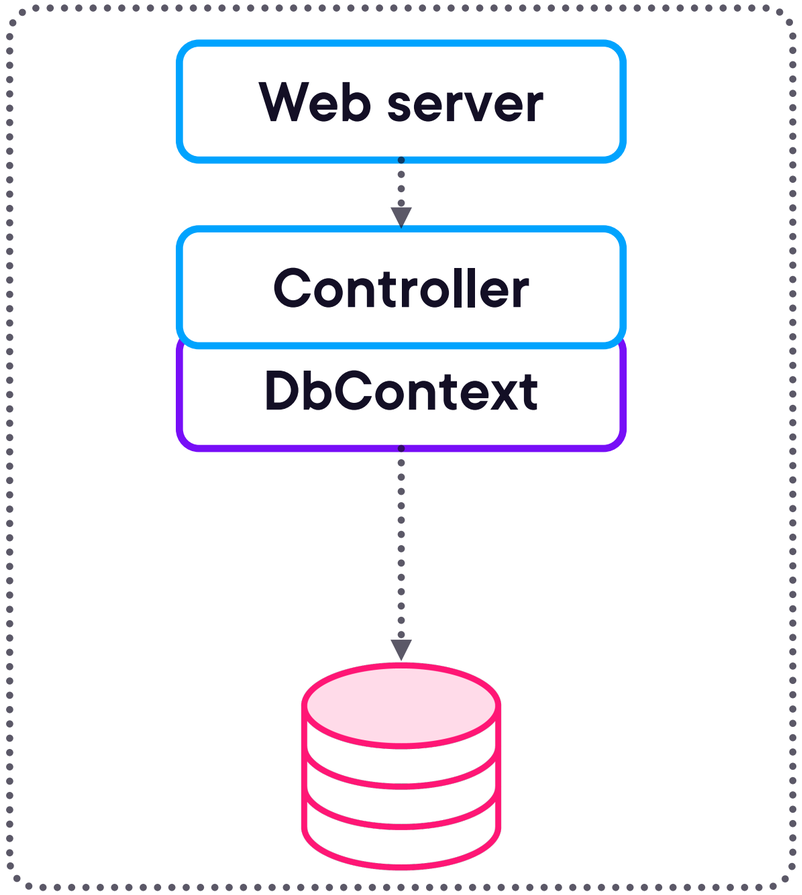
Improves the separation of concerns

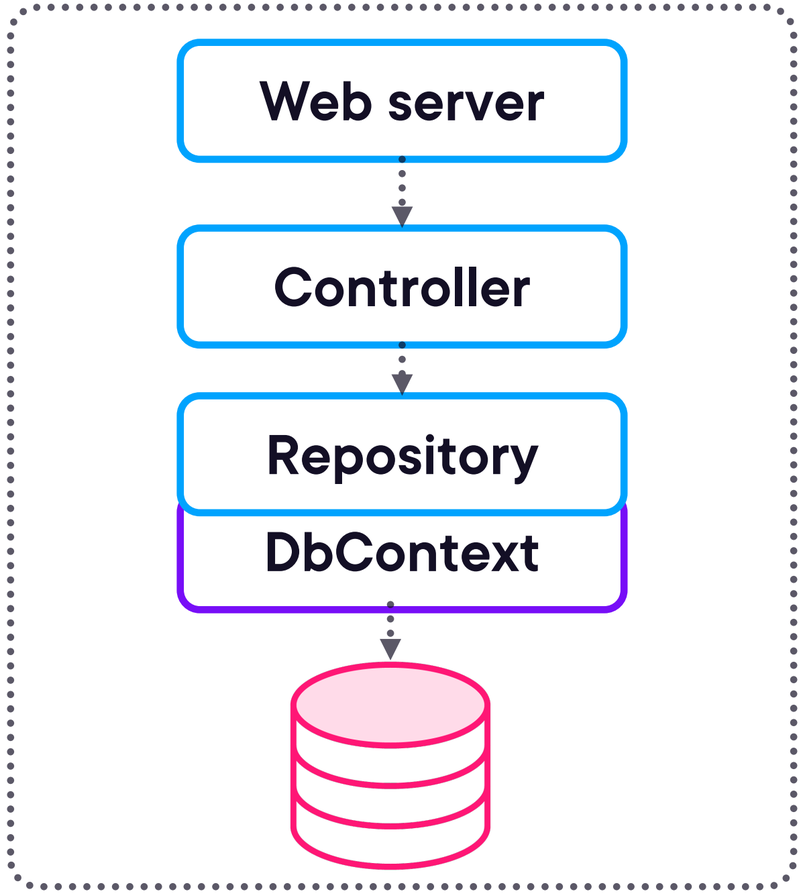


Reduces code duplication

Uses (and wraps) the database context

Adding a Repository





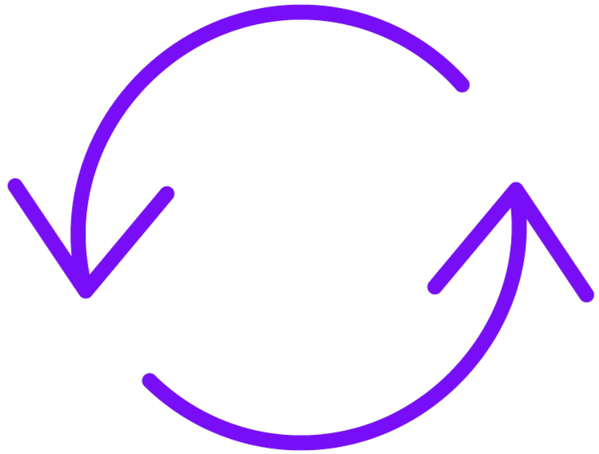
Adding the CategoryRepository

```
public class CategoryRepository : ICategoryRepository
{
    private readonly BethanysPieShopDbContext _bethanysPieShopDbContext;

    public CategoryRepository(BethanysPieShopDbContext bethanysPieShopDbContext)
    {
        _bethanysPieShopDbContext = bethanysPieShopDbContext;
    }

    public async Task<IEnumerable<Category>> GetAllCategoriesAsync()
    {
        ...
    }

}
```



Using asynchronous code

- Synchronous calls will block while waiting for completion
- Available threads on server are limited
- Asynchronous calls won’t block and increases number of concurrent calls

```
public async Task<IEnumerable<Category>> GetAllCategoriesAsync()
{
    return
        await _bethanysPieShopDbContext.Categories.OrderBy(c => c.CategoryId)
        .ToListAsync();
}
```

Using an Async Call on the Database Context

Only when we call ToListAsync() will a database call get executed: deferred execution

```
.Where(p => p.Id == categoryId)

.Single()

.Any()

.OrderBy()
```

- Perform a filter
- Get single instance
- Check if there is a return value
- Order items

Demo

Creating the repository class

Adding an interface

Registering in the Services collection

Adding the Category Overview

Demo

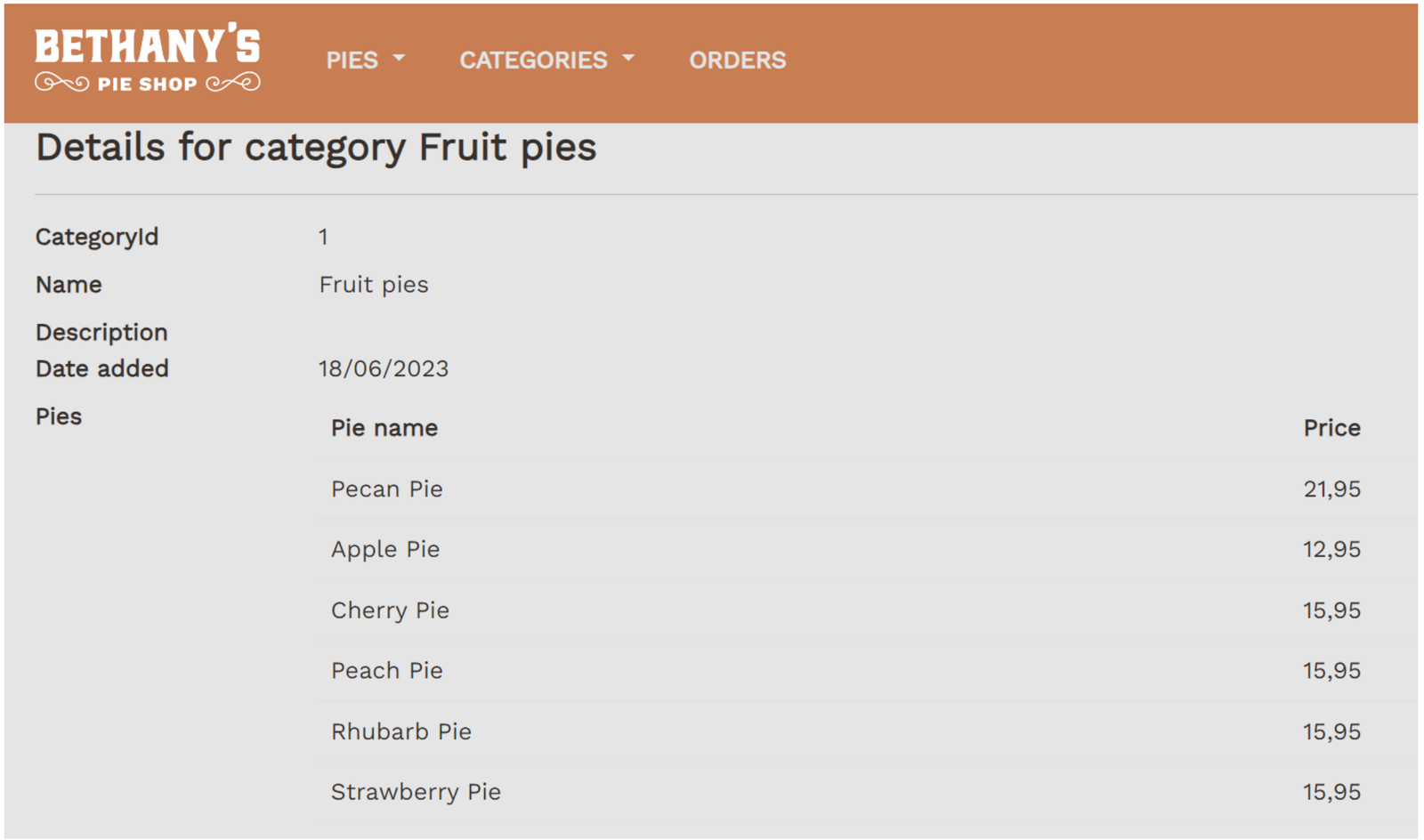
Adding the CategoryController

Introducing a view model class

Creating the view

Creating the Category Detail Page

The Detail Page



Loading Related Data

Eager loading

Explicit loading

Lazy loading

```
public async Task<Category?> GetCategoryByIdAsync(int id)
{
    return await
        _bethanysPieShopDbContext.Categories.Include(p =>p.Pies)
            .FirstOrDefaultAsync(c => c.CategoryId == id);
}
```

Using Eager Loading

Include()
Can be used multiple times
Can cause a lot of data to be loaded!

```
public async Task<Category?> GetCategoryByIdAsync(int id)
{
     return await
          _bethanysPieShopDbContext.Categories.Include(p => p.Pies)
          .ThenInclude(p =>p.Ingredients)
          .FirstOrDefaultAsync(c => c.CategoryId == id);
}
```

Using ThenInclude()

```
public async Task<Category?>? GetCategoryByIdAsync(int categoryId)
{
    var category = _bethanysPieShopDbContext.Categories
            .Single(b => b.CategoryId == categoryId);

            _bethanysPieShopDbContext.Entry(category)
                .Collection(b => b.Pies)
                .Load();
}
```

Using Explicit Loading

Using Raw SQL Queries

```
await  _bethanysPieShopDbContext.Categories
       .FromSqlInterpolated($"Select * from dbo.Categories")
       .Where(c => c.Name.StartsWith("F"))
       .ToListAsync();
```

Demo

Adding the Category Detail page

Demo

Exploring the Pie Overview and Detail pages

Working with a More Complex Model

The Model Classes We'll Use

Demo

Adding the order overview

Creating the order detail page

Summary

Repositories add better separation of concerns

EF Core relies heavily on LINQ

Loading related data can be done in several ways

Up Next:

Working with New Items