Implementing Asynchronous Generic Repository in ASP.NET Core

We are going to convert synchronous code to asynchronous inside ASP.NET Core. First, we are going to learn a bit about asynchronous programming and why should we write async code. Then we are going to use our .NET Core project and rewrite it in an async manner.

We created a Generic Repository Pattern and a controller with actions that consumes that repository.

We are going to modify the code, step by step, to show how to convert the synchronous code to asynchronous one. Hopefully, this will help you understand how asynchronous code works and how to write it from scratch in your applications.

What is Asynchronous Programming

Async programming is a parallel programming technique that allows the working process to run separately from the main application thread. As soon as the work completes, it informs the main thread about the result, whether it was successful or not.

By using async programming, we can avoid performance bottlenecks and enhance the responsiveness of our application.

How so?

Because we are not sending requests to the main thread and blocking it while waiting for the responses anymore. Now, when we send a request to the main thread, it delegates a job to a background thread, thus freeing itself for another request. Eventually, a background thread finishes its job and returns it to the main thread. Then the main thread returns the result to the requester.

It is very important to understand that if we send a request to an endpoint, we probably won't be able to execute this request any faster in async mode. It is going to take the same amount of time as the sync request.

The only advantage is that in the async mode the main thread won't be blocked, and thus it will be able to process other requests.

Async, Await Keywords and Return Types

The `async` and `await` keywords play the crucial part in asynchronous programming. By using those keywords, we can write asynchronous methods without too much effort.

For example, if we want to create a method in an asynchronous manner, we need to add the `async` keyword next to the method's return type:

```
async Task<IEnumerable<Owner>> GetAllOwnersAsync()
```

By using the `async` keyword, we are enabling the `await` keyword and modifying the way method results are handled from synchronous to asynchronous:

```
await FindAllAsync();
```

In asynchronous programming we have three return types:

- `Task<TResult>`, for an async method that returns a value
- `Task`, to use for an async method that does not return a value
- `void`, which we can use for an event handler

What does this mean?

For example, if our sync method returns `int` then in the async mode it should return `Task<int>`, or if sync method returns `IEnumerable<string>` then the async method should return `Task<IEnumerable<string>>`.

But if our sync method returns no value (has `void` for the return type), then our async method should usually return `Task`. This means that we can use the `await` keyword inside that method but without the `return` keyword.

You may wonder why not return `Task` all the time? Well, we should use `void` only for asynchronous event handlers that require a `void` return type. Other than that, we should always return a `Task`.

Now let's do some refactoring in our completely synchronous code.

Modifying the `IRepositoryBase` Interface and the `RepositoryBase` Class

Our repository is interface based, so we need to start with interface changes. In the `Contracts` project open the `IRepositoryBase.cs` file.

Let's modify `Find` and `Save` method signatures:

```
public interface IRepositoryBase<T>
{
    IQueryable<T> FindAll();
    IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression);
    void Create(T entity);
    void Update(T entity);
    void Delete(T entity);
    Task SaveAsync();
}
```

In the code above we have changed the return types of `FindAll` and `FindByCondition` method signatures. We did that in order to be able to assign the async methods in the concrete repository classes.

We haven't changed the `Create`, `Update`, and `Delete` method signatures, because they don't modify any data, they just track changes to an entity and wait for EF Core's `SaveChanges` method to execute. But we do modify the `Save` method signature because it makes changes to our database.

After the interface modification, let’s modify the `RepositoryBase` class:

```
public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class
{
    protected RepositoryContext RepositoryContext { get; set; }

    public RepositoryBase(RepositoryContext repositoryContext)
    {
        this.RepositoryContext = repositoryContext;
    }

    public IQueryable<T> FindAll()
    {
        return this.RepositoryContext.Set<T>();
    }

    public IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression)
    {
        return this.RepositoryContext.Set<T>()
            .Where(expression);
    }

    public void Create(T entity)
    {
        this.RepositoryContext.Set<T>().Add(entity);
    }

    public void Update(T entity)
    {
        this.RepositoryContext.Set<T>().Update(entity);
    }

    public void Delete(T entity)
    {
        this.RepositoryContext.Set<T>().Remove(entity);
    }

    public async Task SaveAsync()
    {
        await this.RepositoryContext.SaveChangesAsync();
    }
}
```

We need to modify these methods in order to implement our interface modifications (names and return types). But now, the `Save` method has additional keywords `async` and `await` to make it asynchronous. Again, the `Create`, `Update`, and `Delete` methods are the same as before. They don’t modify any data but wait for the `SaveChangesAsync` method to execute.

Modifying the `IOwnerRepository` Interface and the `OwnerRepository` Class

We have modified our async `RepositoryBase` class, so let's continue on the other parts of our repository. In the `Contracts` project, there is also the `IOwnerRepository` interface with all the synchronous method signatures which we should change too.

So let's do that:

```
public interface IOwnerRepository
{
    Task<IEnumerable<Owner>> GetAllOwnersAsync();
    Task<Owner> GetOwnerByIdAsync(Guid ownerId);
    Task<OwnerExtended> GetOwnerWithDetailsAsync(Guid ownerId);
    Task CreateOwnerAsync(Owner owner);
    Task UpdateOwnerAsync(Owner dbOwner, Owner owner);
    Task DeleteOwnerAsync(Owner owner);
}
```

This interface modification is a bit different from the `IRepositoryBase` interface modification. The `Create`, `Update` and `Delete` method signatures are now asynchronous. That's because in these methods we are going to execute the `SaveAsync()` method. Because the `SaveAsync()` method is awaitable, we can use the `await` keyword, thus our methods need to have the `async` keyword and `Task` as a return type.

In accordance with the interface changes, let's modify our `OwnerRepository.cs` class, which we may find in the `Repository` project:

```
public class OwnerRepository : RepositoryBase<Owner>, IOwnerRepository
{
    public OwnerRepository(RepositoryContext repositoryContext)
        : base(repositoryContext)
    {
    }

    public async Task<IEnumerable<Owner>> GetAllOwnersAsync()
    {
         return await FindAll()
            .OrderBy(x => x.Name)
            .ToListAsync();
    }

    public async Task<Owner> GetOwnerByIdAsync(Guid ownerId)
    {
         return await FindByCondition(o => o.Id.Equals(ownerId))
             .DefaultIfEmpty(new Owner())
             .SingleAsync();
    }

    public async Task<OwnerExtended> GetOwnerWithDetailsAsync(Guid ownerId)
```

```
        {
             return await FindByCondition(o => o.Id.Equals(ownerId))
                 .Select(owner => new OwnerExtended(owner)
                 {
                     Accounts = RepositoryContext.Accounts
                     .Where(a => a.OwnerId.Equals(owner.Id))
                     .ToList()
                 })
                 .SingleOrDefaultAsync();
        }

        public async Task CreateOwnerAsync(Owner owner)
        {
            owner.Id = Guid.NewGuid();
            Create(owner);
            await SaveAsync();
        }

        public async Task UpdateOwnerAsync(Owner dbOwner, Owner owner)
        {
            dbOwner.Map(owner);
            Update(dbOwner);
            await SaveAsync();
        }

        public async Task DeleteOwnerAsync(Owner owner)
        {
            Delete(owner);
            await SaveAsync();
        }
    }
```

Controller Modification

Finally, we need to modify all of our actions in the `OwnerController` to work asynchronously.

So, let's first start with the `GetAllOwners` method:

```
    [HttpGet]
    public async Task<IActionResult> GetAllOwners()
    {
        try
        {
            var owners = await _repository.Owner.GetAllOwnersAsync();
            return Ok(owners);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Some error in the GetAllOwners method: {ex}");
            return StatusCode(500, "Internal server error");
    }
}
```

We haven't changed much in this action. We've just changed the return type and added the `async` keyword to the method signature. In the method body, we can now `await` the `GetAllOwnersAsync()` method. And that is pretty much what we should do in all the actions in our controller.

So let's modify all the other actions.

`GetOwnerById`

```
[HttpGet("{id}", Name = "OwnerById")]
public async Task<IActionResult> GetOwnerById(Guid id)
{
    try
    {
        var owner = await _repository.Owner.GetOwnerByIdAsync(id);

        if (owner.IsEmptyObject())
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }
        else
        {
            _logger.LogInfo($"Returned owner with id: {id}");
            return Ok(owner);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside GetOwnerById action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

`GetOwnerWithDetails`

```
[HttpGet("{id}/account")]
public async Task<IActionResult> GetOwnerWithDetails(Guid id)
{
    try
    {
        var owner = await _repository.Owner.GetOwnerWithDetailsAsync(id);

        if (owner.IsEmptyObject())
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
```

```
                return NotFound();
            }
            else
            {
                _logger.LogInfo($"Returned owner with details for id: {id}");
                return Ok(owner);
            }
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong inside GetOwnerWithDetails action:
{ex.Message}");
            return StatusCode(500, "Internal server error");
        }
 }
```

CreateOwner

```
[HttpPost]
public async Task<IActionResult> CreateOwner([FromBody]Owner owner)
{
    try
    {
        if (owner.IsObjectNull())
        {
            _logger.LogError("Owner object sent from client is null.");
            return BadRequest("Owner object is null");
        }

        if (!ModelState.IsValid)
        {
            _logger.LogError("Invalid owner object sent from client.");
            return BadRequest("Invalid model object");
        }

        await _repository.Owner.CreateOwnerAsync(owner);

        return CreatedAtRoute("OwnerById", new { id = owner.Id }, owner);
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside CreateOwner action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

UpdateOwner

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateOwner(Guid id, [FromBody]Owner owner)
{
    try
    {
        if (owner.IsObjectNull())
        {
            _logger.LogError("Owner object sent from client is null.");
            return BadRequest("Owner object is null");
        }

        if (!ModelState.IsValid)
        {
            _logger.LogError("Invalid owner object sent from client.");
            return BadRequest("Invalid model object");
        }

        var dbOwner = await _repository.Owner.GetOwnerByIdAsync(id);
        if (dbOwner.IsEmptyObject())
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }

        await _repository.Owner.UpdateOwnerAsync(dbOwner, owner);

        return NoContent();
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside UpdateOwner action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

`DeleteOwner`

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteOwner(Guid id)
{
    try
    {
        var owner = await _repository.Owner.GetOwnerByIdAsync(id);
        if (owner.IsEmptyObject())
        {
            _logger.LogError($"Owner with id: {id}, hasn't been found in db.");
            return NotFound();
        }
        await _repository.Owner.DeleteOwnerAsync(owner);

        return NoContent();
    }
    catch (Exception ex)
    {
        _logger.LogError($"Something went wrong inside DeleteOwner action:
{ex.Message}");
        return StatusCode(500, "Internal server error");
    }
}
```

Conclusion

We have converted the synchronous repository to asynchronous. With a couple of changes, we created a more responsive application.