REST API Micro service (Preview)

Designing a micro service / REST API

In this article let us explore what it takes to build a truly enterprise grade micro service / REST API. Most of the concepts will be covered with examples in C# .NET, but they would hold good for any other matured language(s) / framework(s).

Any enterprise grade API these days follows REST pattern so that they can be invoked by clients such as desktop, mobile, browser based apps or even other backend services. The RESTful service itself can interact with databases, other services, message brokers while executing business logic to return some result back.

The idea to make use of standard patterns in the implementation is for following reasons

- Easy to on board new developer in the team
- Production support is easy
- Easier Cross team member migration
- Easy to maintain and extend the current features

Any framework that builds the REST API should be simple, easy to use, decoupled and follow standard patterns.

High level design elements

- **Abstraction**
  - Always code against abstractions using interface
  - Using interface makes it easy to change the underlying implementation with ease
  - Use IOC / Dependency injection to resolve and create objects
  - Lazy loading
  - SOLID principles while designing classes
- **Domain Driven Design**
  - Micro service to serve a specific business need
  - Mediator pattern for decoupled implementation
  - Well defined controller with end points and versioning
  - Middleware(s) for logging, exception handling, distributed id, authentication
- **Data Access**
  - CQRS for data - for performance, scalability
  - Unit of work
  - Repository Pattern
  - Pagination
- **Monitoring**
  - Distributed tracing
  - Structured logging
  - Health Checks
  - Detailed Exceptions
  - Configurable control over on demand log details
  - Metrices / KPIs etc.
  - Auditing
- **Event Driven**
  - Publisher / Consumer pattern
  - Supporting message broker for async / background jobs
- **Testing frameworks**
  - Unit testing with mocks
  - Benchmarking
  - BDD with frameworks like Cucumber
- **Authentication and authorization**
  - Authenticated and Authorized through standard OpenId tokens
  - Swagger UI support
- **Caching**
  - Distributed and in memory caching strategies

Build and Hosting options

• PAAS with auto scaling using CPU / Memory usage
• Access through API Gateway for throttling, Cache policy etc.
• CI / CD, Gated Check-in with code analysis support
• Docker, K8s with AKS, EKS, Fargate support

Feature Matrix

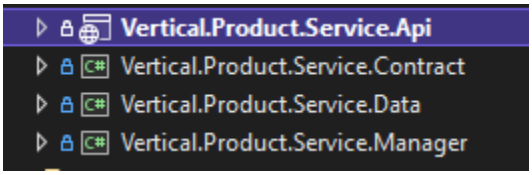
The first version of the reference architecture project would aspire to have following features built in. The matrix shows implementations details in .NET, we need to build similar for other languages like Node JS, Java or say Go.

Area	.NET
Framework	.NET 6
IDE	Visual Studio 2022 Windows & MacOS - Community Edition. MacOS version in Preview as of now
REST Framework	MVC
API Documentation	Swagger Integration
Async Framework	Azure Functions
Authenticating framework	JWT / Azure AD
Header standards	(correlation id)
Error handling standards	TBD
Logging framework (generic, pluggable)	Use Microsoft Ilogger and build custom implementation for AppInsights etc. Or use serilog Sink libs
IOC	Built in with .NET Core and above
Benchmarking framework (capture timer?)	https://benchmarkdotnet.org/?msclkid=1fec4770bcd411ecac8bef5794499ab7
Unit testing framework	MS Test / Rhino Mock / Automate using Plex
Patterns - Retry, Circuit breaker etc.	The Polly Project
Data Access Layer (Data Context, Repos, Unit of work patterns)	Entity Framework 6
Health and Diagnostics	Yes, Custom
Router / Http Calls	Home · jbogard/MediatR Wiki · GitHub Will have the http client injected so that we can use connection pool and control to change time out etc.
Caching	Azure Cache for Redis | Microsoft Azure IDistributedCache Interface (Microsoft.Extensions.Caching.Distributed) | Microsoft Docs Cache in-memory in ASP.NET Core | Microsoft Docs
Docker Support	Yes
Hosting	Out of the box support for Linux or Windows based PAAS, and Docker. On Premise is also an option
Azure Support	Cosmos DB, BLOB, Azure Key Vault, Azure App Configuration, SQL DB, Data Lake v2, Event Grid and Service bus publish - using Nuget Packages
AWS Support	TBD
AKS	TBD

AWS ECS/Fargate	TBD
AWS EKS	TBD

Reference Architecture Components

As shown below in the diagram we would have following 4 CS projects in VS 2K22. The project also refers several custom built nuget packages which we will discuss in a later article.



A naming standard for any application must be set and followed by the organization. Right now the proposal is to use “Vertical.Product.Service” as a prefix. For example “Origination.Underwriting.Decision” or “Servicing.Loan.PayOff”.

The above example has 4 CS projects

- **API** exposes the endpoints via controllers, should not have any code except for model validation, authenticate / authorize through AOP. There will be guidance on how to name controllers, methods and API version etc. in this series. The API would use a router or a middleware to invoke a handler or manager class method using IOC.
- **Contract** project will define POCO classes used by the APIs.
- **Data** project will refer the Entity framework Nuget package built by the architecture team that takes care of data context, abstract repositories and unit of work pattern. Developers will use the data project to write LINQ queries.
- **Manager** project is the heart of the service with all logic and business rule built by the developer.

Let us see code example

API Controller Method

```
namespace Vertical.Product.Service.Api.Controllers
{
    [ApiController]
    [Authorize]
    [Route("v1/[controller]")]
    public class PlaygroundController : Controller
    {

        private readonly Lazy<IMediator> _mediator;
        private readonly Lazy<ILogger<PlaygroundController>> _logger;

        public PlaygroundController(Lazy<IMediator> mediator,
Lazy<ILogger<PlaygroundController>> logger)
        {
            _mediator = mediator;
            _logger = logger;
        }

        [HttpPost]
        [ProducesResponseType(typeof(string), (int)HttpStatusCode.OK)]
        [Route("CreateManualLoan")]
        [ProducesResponseType((int)HttpStatusCode.BadRequest)]
        public async Task<ActionResult<string>> CreateManualLoan
(CreateManualLoanRequest request)
        {
            _logger.Value.LogInformation("CreateManualLoan Enter");
            var result = await _mediator.Value.Send(request);
            _logger.Value.LogDebug($"{request}", request);
            _logger.Value.LogInformation("CreateManualLoan Exit");
            return Ok(result);
        }
    }
}
```

In the above code snippet, we see a method called `CreateManualLoan` as an end point in the controller. You will notice, `IMediator`and `ILogger`being injected as lazy objects. Lazy initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements.

`CreateManualLoan`method is using logger to log messages but since it is an abstracted interface, behind the scene we can swap out the actual implementation using IOC without impacting developers. We will see more about this later in the series.

You will also notice that the controller method does not have any business code written in it, instead it invokes a class and it's method in the "Manager" CS project. Let's see how we do this.

We are using an open source C# library to implement this rather than custom code, here is a link to the library if you need

Home · jbogard/MediatR Wiki · GitHub

```
var result = await _mediator.Value.Send(request);
```

the above line invokes a method called "Handle" based on the IOC registration. This is called Mediator pattern so that the code is decoupled. Let's see this in detail -

When the method `Send` is invoked, mediator class looks up at the IOC to find out the class registered for the "request" object class `CreateManualLoanRequest`

Data Contract with Data Attributes based validations

```
namespace Vertical.Product.Service.Contract.Playground
{
    public class CreateManualLoanRequest : IRequest<string>
    {
        [Required]
        [StringLength(100)]
        public string? BorrowerFirstName { get; set; }
        public string? BorrowerLastName { get; set; }
        public DateTime BorrowerDOB { get; set; }
        public string? PropertyAddress { get; set; }
        public int LoanType { get; set; }

        [Range(0, 999.99)]
        public long BaseLoanAmount { get; set; }
    }
}
```

Handler Implementation

```
 namespace Vertical.Product.Service.Manager.Loan.Command
{
    public class CreateManualLoanHandler :
IRequestHandler<CreateManualLoanRequest, string>
    {
        public CreateManualLoanHandler()
        {
        }

        public async Task<string> Handle(CreateManualLoanRequest
request, CancellationToken cancellationToken)
        {
            return await Task.FromResult(Guid.NewGuid().ToString());
        }
    }
}
```

Mediator Registration

This is part of the service framework and called at Startup

```
namespace Vertical.Product.Service.Manager
{
    public static class ManagerDependency
    {
        public static void Register(IServiceCollection sericeCollection)
        {
            sericeCollection.AddMediatR(typeof(ManagerDependency));
        }
    }
}
```

All in all, the core framework is handling following, while developer focuses on their business code

- Authentication
- Health check for Service
- Standard pattern like mediator so the code is easily readable and standard
- IOC out of the box
- Logging abstractions
- Application settings in the configuration JSON
- IOptionsMonitor which can be used with Azure Application Configuration for auto reload of the settings on change without restarting the application / service
- *More features to be added*

A sample code with working implementation has been checked in to the repository

fairwayindependentmc / architecture-template-csharp — Bitbucket