

Especificação Técnica

Esse é um documento fictício de especificações técnicas exclusivas do protótipo desenvolvido pelo candidato no momento do case de processo seletivo.

Protótipo: API REST utilizando o Framework .NET 5.0

Linguagem predominante: C#

IDE de desenvolvimento: Microsoft Visual Studio Code (User) 1.71.2

Componentes essenciais:





- .NET 5.0 [Download .NET 5.0 SDK \(v5.0.103\) - Windows x64 Installer \(microsoft.com\)](#)



- . NET Framework 4.7.1 Developer Pack
- MySQL Installer – Community 1.6.3.0
- MySQL Router 8.0.30
- MySQL Server 5.7.39
- MySQL Shell 8.0.30
- MySQL Connector C++ 8.0.30
- MySQL Connector J 8.0.30
- MySQL Connector Net 8.0.30
- MySQL Connector/ODBC 8.0.30
- MySQL Workbench 8.0 CE

❖ **NOTA:** Os **PackageReferences** podem ser consultados nos arquivos .csproj de cada camada.

Papéis de Usuário (perfis):

-  ADM – Administrador
-  SAA – Secretaria Acadêmica
-  TUT – Tutor
-  STD – Estudante/Aluno

Atribuições (assignments)

Foram associadas aos perfis para implementação da gestão de acessos aos recursos da API:

- Cadastrar Curso
- Consultar Lista de Cursos
- Cadastrar Disciplina
- Gerir Grade Curricular
- Consultar Grade Curricular por Curso
- Cadastrar Aluno
- Consultar Lista de Alunos Por Curso
- Lançar Notas
- Consultar Boletim

Entidades:

Nome	Alias	Classe modeladora	Métodos disponíveis via API
CURSOS (courses)	CUR	CourseEntity	GetWhitId, GetAll
ALUNOS (students)	STD	StudentEntity	GetWhitId, Post, Put, Delete
TUTORES (tutors)	TUT	TutorEntity	GetWhitId
BOLETINS (bulletins)	BLT	BillEntity	GetWhitId{student}, Put
DISCIPLINAS (subjects)	SBJ	SubjectEntity	GetAll
USUÁRIOS (users)	USER	UserEntity	None
PERFIS (profiles)	PFL	ProfileEntity	None

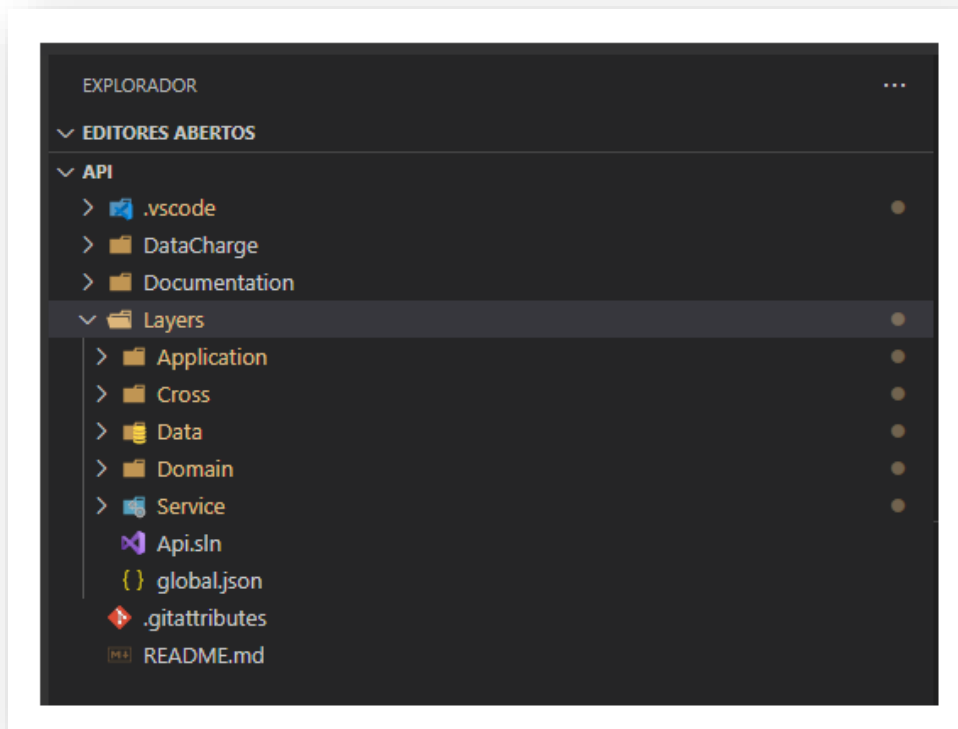
Arquitetura:

No modelo desenvolvido, cada camada é um projeto. Dessa forma esses projetos podem ser restaurados individualmente (**dotnet restore**), já pensando na manutenção dos fontes.

O projeto foi desenvolvido utilizando a Biblioteca (Class library C#) net5.0 visando compatibilidade com outros sistemas operacionais além do Windows.

O projeto foi desenvolvido considerando o padrão MVC.

Posicionar no diretório Layers e a partir dele implementar as camadas.



Ao executar **dotnet restore** ou **dotnet build** nesse nível, os projetos de todas as camadas serão afetados. Opcionalmente pode-se posicionar na camada na qual deseja realizar esses procedimentos de forma a afetar somente a camada na qual se estiver posicionado no terminal.

Application

```
dotnet new webapi -n application -o Application --no-https
```

Application é a aplicação em si, classe que vai receber as requisições web. É nesse nível que deve ser executado o comando de inicialização `dotnet run`

Projeto **webapi** criado inicialmente sem implantação de protocolo seguro (HTTPS), devido a limitações do ambiente de desenvolvimento local. Recomenda-se fortemente que esse protocolo seja implementado no pacote a ser aplicado nos ambientes de stage e produção.

Implementadas classes **controllers** para cada entidade, responsáveis pela entrada dos endpoints. Endpoints corretamente roteados encontrarão esses controllers automaticamente através do .NET.

Todos os controllers foram programados para retornar o código do status das requisições. (<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>)

Referências entre camadas:

- `dotnet add .\Application\ reference .\Domain\`

- dotnet add .\Application\ reference .\Service\
- dotnet add .\Application\ reference .\Cross\

Domain dotnet new classlib -n Domain -o Domain -f net5.0

A camada de domínio é responsável pela implementação de classes, modelos as quais serão mapeadas para o banco de dados, além de obter as declarações de interfaces, constantes, Dtos, entidades, Models e Enums. Tudo em um mesmo local possibilitará às demais camadas da aplicação fazerem as injeções de dependências corretamente.

Por ser desenvolvido no modelo DDD, todas as entidades devem permanecer a camada Domain, sendo necessário criar uma referência a esta camada em todas as demais entidades, conforme demonstrado nesse documento em cada camada.

Interface de Repository (IRepository) - Tratada como interface padrão e que pode receber qualquer tipo de entidade na implementação do CRUD (Insert, Update, Delete, Select). Visando reaproveitamento e não redundância de código, conforme as boas práticas de Clean Code, permitindo inclusive que a API criada possa receber métodos síncronos e/ou assíncronos.

Structure | Cross dotnet new classlib -n Cross -f net5.0 -o Cross

Camada transversal que permite inclusão de chamadas a API internas e externas, extensões da estrutura atual e implementação de recursos como AutoMapper, com desempenho satisfatório. Essa camada foi implementada visando escalabilidade para o projeto, uma vez que configurando somente aqui ao invés de diretamente na camada de service permitirá reaproveitamento dos recursos implementados.

Referências entre camadas:

- dotnet add .\Cross\ reference .\Domain\
- dotnet add .\Cross\ reference .\Service\
- dotnet add .\Cross\ reference .\Data\

Validações de requisição - A validação das requisições foram desenvolvidas utilizando **Data Annotations**. Para este recurso utilizamos o namespace **System.ComponentModel.DataAnnotations**, este que provê atributos de classes e métodos capazes de alterar as convenções padrão e definir um comportamento personalizado que pode ser usado em vários cenários.

Dependency Injection- Nessa camada foram criados métodos para configurar a coleção de serviços utilizando o recurso **Transient**, de forma a criar uma nova instância a cada implementação da classe. A implementação dessa forma visa a escalabilidade e facilidade de manutenção do sistema em qualquer tempo, visto que se necessário mudar o serviço basta criar a nova classe que a partir desse momento toda a aplicação estará automaticamente enxergando o novo serviço.

O fato da camada de aplicação comunicar diretamente com a camada de dados pode ser considerado uma brecha de segurança. A implementação da camada Cross diminui consideravelmente essa brecha, pois quem acessa a camada Data é a Cross e não a Application.

DTO (Data Transfer Object) - Foi implementado como modelo na entidade UserLogin para não expor dados sensíveis do usuário no endpoint de requisição de token.

Security - O login da API é tokenizado. Para isso foi implementado **RSACryptoServiceProvider**

Structure | Data `dotnet new classlib -n Data -f net5.0 -o Data`

Camada que irá executar a gravação e leitura no banco de dados. Aqui estará instalado o EntityFramework, recurso responsável por gerenciar as transações do banco de dados.

EntityFramework: `dotnet tool install --global dotnet-ef --version 3.1.29`

BaseEntity - Todas as entidades possuirão um GUID como identificador único e chave primária, visto que não há necessidade deste projeto case contemplar definições de máscaras de identificação para cada entidade.

O que é um GUID? GUID (também conhecido como UUID) é um acrônimo para 'Globally Unique Identifier' (ou 'Universal Unique Identifier'). É um número inteiro de 128 bits usado para identificar recursos. O termo GUID geralmente é usado por desenvolvedores que trabalham com tecnologias da Microsoft, enquanto o UUID é usado em todos os outros lugares.

Quão único é um GUID? 128 bits é grande o suficiente e o algoritmo de geração é único o suficiente para que, se 1.000.000.000 GUIDs por segundo fossem gerados por 1 ano, a probabilidade de uma duplicata seria de apenas 50%. Ou se cada humano na Terra gerasse 600.000.000 GUIDs, haveria apenas 50% de probabilidade de uma duplicata.

Como os GUIDs são usados? Os GUIDs são usados no desenvolvimento de software empresarial em C#, Java e C++ como chaves de banco de dados, identificadores de componentes ou em qualquer outro lugar onde seja necessário um identificador verdadeiramente exclusivo. GUIDs também são usados para identificar todas as interfaces e objetos na programação COM.

Criei uma classe chamada **ContextFactory** definindo nela as especificações dos campos da base de dados. Dessa forma a aplicação se torna multibanco, podendo ser utilizada tanto por SQLServer quanto por MySQL, por exemplo, sem precisar modificar o código-fonte, mudando apenas o driver de conexão.

Especifiquei o relacionamento das tabelas diretamente nas **Entities**, inclusive aplicando os recursos do componente **DataAnnotations**. Entendo que isso é uma melhor prática em relação a definir nos **Mappings**. Dessa forma o próprio framework se encarrega de criar as **Foreign Keys** nas entidades, criando automaticamente o relacionamento, que pode e deve ser conferido e validado sempre após as execuções de **Migrations**.

Criei alguns **Seeds** para pequenos volumes de dados, como por exemplo o cadastro de Cursos e o cadastro de Perfis.

Procedimento para criação/manutenção de tabelas e atributos da base via migrations:

Sempre que precisei criar uma nova entidade, desenvolvi na sequência:

Entities (Domain) > Interfaces (Domain) > Mapping Entities (Data) > MyContext > Gerar as migrations (dotnet ef migrations add <YourMigration>) > Atualizar o Banco de Dados (dotnet ef database update)

Seguir esse procedimento padrão garante a integridade da estrutura modelada.

Service dotnet new classlib -n Service -f net5.0 -o Service

Camada para validação de conteúdos e execução de regras de negócio diversas, fazendo intermédio entre a Application e a Data.

Referências entre camadas:

- dotnet add .\Service\ reference .\Domain\
- dotnet add .\Service\ reference .\Data\
- dotnet add .\Service\ reference .\Cross\

Autenticação da API com JWT (Jason Web Token) - Os **JSON Web Tokens** são um método RFC 7519 padrão opensource para representar declarações de forma segura entre duas partes. JWT.IO permite decodificar, verificar e gerar JWT. Na autenticação, quando o usuário fizer login com sucesso usando suas credenciais, um JSON Web Token será retornado. Sempre que o usuário desejar acessar uma rota ou recurso protegido, o agente do usuário deve enviar o JWT no cabeçalho Authorization usando o esquema Bearer.

Desenvolvido Endpoint exclusivo para obtenção do token a ser solicitado em requisições das demais entidades. Exemplo de retorno de uma requisição de token bem-sucedida:

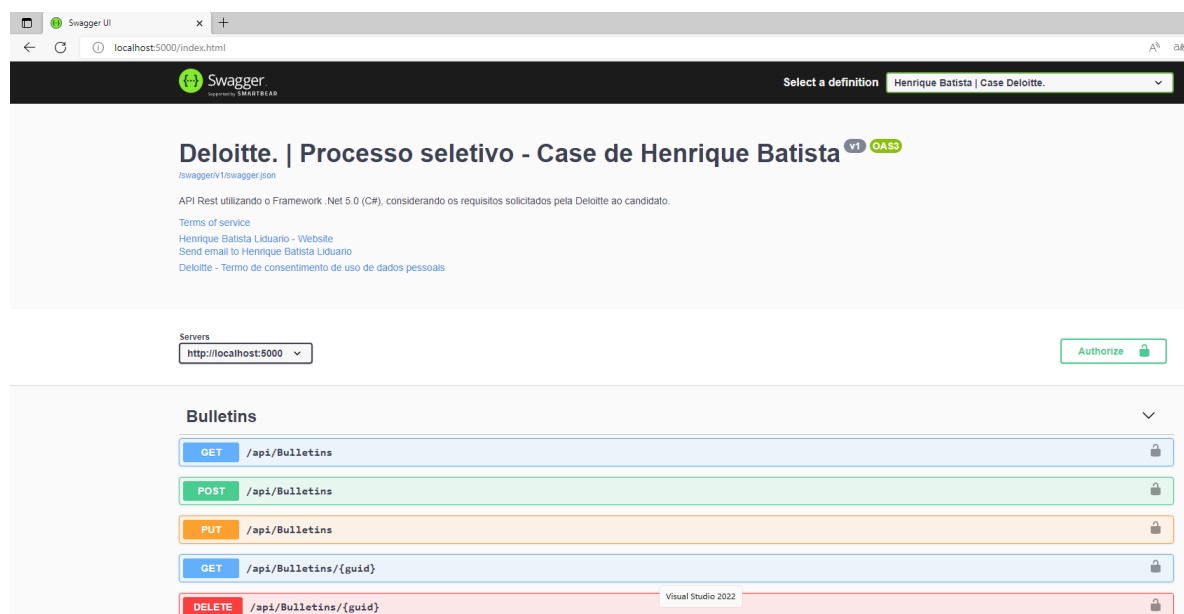
Por que escolhi usar JWT? Devido a alguns benefícios dos JSON Web Tokens (JWT) em relação aos Simple Web Tokens (SWT) e Security Assertion Markup Language Tokens (SAML), como por exemplo: 1. Pelo fato do JSON ser menos detalhado que o XML, seu tamanho é menor, tornando o JWT mais compacto e leve que o SAML, sendo então uma melhor opção para ser passado em ambientes HTML e HTTP.

2. Em termos de segurança, o SWT só pode ser assinado simetricamente por um segredo compartilhado usando o algoritmo HMAC. No entanto, os tokens JWT e SAML podem usar um par de chaves pública/privada na forma de um certificado X.509 para assinatura. Assinar XML com assinatura digital XML sem introduzir brechas de segurança obscuras é muito difícil quando comparado à simplicidade de assinar JSON. 3. JSON é comum na maioria das linguagens de programação, visto que mapeiam diretamente para objetos. Já o XML não possui um mapeamento natural de documento para objeto. Por isso o trabalho com JWT é mais prático e facilitado do que com asserções SAML. 4. JWT é usado na escala da Internet, e isso facilita o processamento do lado cliente em várias plataformas, especialmente em dispositivos móveis.

Após a implementação do JWT, todos os endpoints no Controller foram protegidos com o modelo de autenticação Bearer, usando recursos do **Microsoft.AspNetCore.Authorization**.

Conclusão:

O protótipo da API pode ser apreciado e homologado fazendo uso da ferramenta **Swagger** que foi implementada no Startup do projeto. Após startar a aplicação (dotnet run), acesse o caminho padrão dos endpoints. No meu caso é `http://localhost:5000`



O código-fonte do protótipo, bem como mais alguns itens da documentação estão disponíveis no meu git:

<https://github.com/HenriqueBatistaLiduario/CaseApi>

Implementações que eu gostaria de ter feito porém não foi possível devido ao prazo:

- ☺ Implementar Criptografia de senha para armazenar no banco.
- ☺ Criar uma semicamada de DTOs e implementar tratamentos para todas as entidades (fiz apenas na entidade Login), para evitar a exposição de dados desnecessários no FrontEnd.
- ☺ Implementar Seeds de Migrations para todas as entidades. (Criei apenas para as entidades com pouco volume de registros. Para as demais escrevi scripts em SQL para popular manualmente o banco de dados.
- ☺ Implementar Models para todas as entidades, conforme as boas práticas do MVC.

- ☺ Consistir Assignments do perfil no momento da requisição. Os perfis foram populados com seus respectivos Assignments, mas está consistindo somente a atribuição do perfil ao usuário.
- ☺ Escreveria uma semicamada de testes dentro da camada Data, para agilizar a realização de testes unitários das implementações, utilizando Microsoft.Net.Test.sdk e xUnit.
- ☺ Gostaria de validar a conversão para a versão 6.1.

“Agradeço a oportunidade de mostrar meu trabalho.”

Henrique Batista Liduario