UNIVERSIDADE LUTERANA DO BRASIL

CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

MODELAGEM DE SOFTWARE

Artigo Acadêmico para Avaliação Parcial 2

Ariana Quadros de Almeida

Lucas Rodrigues Schwartzhaupt Fogaça

Torres, 2025

INTRODUÇÃO

Este artigo tem como objetivo apresentar o desenvolvimento de uma aplicação Web API voltada para o gerenciamento de clínicas veterinárias, conforme proposto para a Segunda Avaliação Parcial do primeiro semestre de 2025. O projeto utiliza a plataforma .NET, com persistência de dados via Entity Framework Core e banco de dados SQLite, e executado no Visual Studio Code.

O cenário proposto para o trabalho simula as necessidades de uma clínica veterinária, com a necessidade de funcionalidades como o cadastro, atualização, exclusão e consulta de tutores e animais de estimação. O desenvolvimento seguiu boas práticas de programação, como a aplicação do padrão Repository, uso de injeção de dependências e validações de dados essenciais, seguindo o que foi ensinado durante a cadeira de Modelagem de Software.

DESENVOLVIMENTO

Requisitos funcionais

A funcionalidade de cadastro, atualização, exclusão e consulta de tutores foi implementada utilizando a arquitetura em camadas, com a aplicação do padrão de repositório e o princípio de separação de responsabilidades. Foram utilizadas as classes *TutorController*, *TutorService*, *TutorRepository* e o repositório genérico *Repository<T>* para estruturar o fluxo de dados entre a API e o banco de dados.

O cadastro de tutores foi implementado através do método *AddTutorAsync*, localizado na classe *TutorService*.

```
        public async Task AddTutorAsync(Tutor tutor)
        {
            tutor.Pets = null;
            await _tutorRepository.AddAsync(tutor);
        }
```

E também do método *AddTutor*, no *TutorController*.

```
        public async Task<ActionResult<Tutor>> AddTutor([FromBody] Tutor tutor)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            await _tutorService.AddTutorAsync(tutor);
              return CreatedAtAction(nameof(GetTutorById), new { id = tutor.id }, tutor);
        }
```

A operação é realizada a partir de uma requisição HTTP POST, na qual os dados do tutor são recebidos pelo controlador. Após validação, o serviço invoca o repositório para persistir o novo registro no banco de dados SQLite.

A funcionalidade de atualização de dados foi desenvolvida por meio do método *UpdateTutorAsync*, no *TutorService*.

```
        public async Task UpdateTutorAsync(Tutor tutor)
        {
            await _tutorRepository.UpdateAsync(tutor);
        }
```

E também do método UpdateTutor, no TutorController.

```
        [HttpPut("{id}")]
             public  async  Task<IActionResult>  UpdateTutor(int  id,
[FromBody] Tutor tutor)
        {
            if (id != tutor.id)
            {
                return BadRequest();
            }

            await _tutorService.UpdateTutorAsync(tutor);
            return NoContent();
        }
```

O controlador recebe os dados atualizados através de uma requisição HTTP PUT, verifica se o identificador corresponde ao registro desejado e, em seguida, o serviço atualiza as informações do tutor utilizando o repositório.

A exclusão de tutores é realizada através do método *DeleteTutorAsync*, no *TutorService*.

```
        public async Task DeleteTutorAsync(int id)
        {
            await _tutorRepository.DeleteAsync(id);
        }
```

E também do método *DeleteTutor*, no *TutorController*.

```
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteTutor(int id)
        {
            await _tutorService.DeleteTutorAsync(id);
            return NoContent();
        }
```

A operação é acionada por meio de uma requisição HTTP DELETE, contendo o ID do tutor a ser removido. O serviço, então, executa a exclusão definitiva do registro no banco de dados via repositório.

Para a consulta de tutores, foram implementados os métodos *GetAllTutorsAsync* e *GetTutorByIdAsync*, disponíveis tanto no *TutorController* quanto no *TutorController*.

```
        [HttpGet]
                public   async   Task<ActionResult<IEnumerable<Tutor>>>
GetAllTutors()
        {
            var tutors = await _tutorService.GetAllTutorsAsync();
            return Ok(tutors);
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<Tutor>> GetTutorById(int id)
        {
            var tutor = await _tutorService.GetTutorByIdAsync(id);
            if (tutor == null)
            {
                return NotFound();
            }
            return Ok(tutor);
        }
```

```
        public async Task<IEnumerable<Tutor>> GetAllTutorsAsync()
        {
            return await _tutorRepository
                .Query()
                .Include(t => t.Pets)
                .ToListAsync();
        }

        public async Task<Tutor> GetTutorByIdAsync(int id)
        {
            return await _tutorRepository
                .Query()
                .Include(t => t.Pets)
                .FirstOrDefaultAsync(t => t.id == id);
        }
```

O método *GetAllTutorsAsync* retorna todos os tutores cadastrados, incluindo seus respectivos pets, utilizando a extensão *Include* do *Entity Framework Core* para realizar o carregamento dos relacionamentos. Já o método *GetTutorByIdAsync* busca um tutor específico a partir de seu identificador único (ID). Ambas as consultas são realizadas por meio de requisições HTTP GET.

A mesma coisa ocorre com os animais de estimação, no *PetController* e no *PetService*, cumprindo o requisito de criar, deletar, atualizar e buscar os animais de estimação, tanto por ID, ou buscando todos.

Para garantir que cada animal de estimação esteja associado a um tutor específico, foi implementado um relacionamento de muitos-para-um entre as entidades *Pet* e *Tutor* utilizando os recursos do *Entity Framework Core*. Esse relacionamento assegura a integridade referencial no banco de dados e permite o carregamento estruturado das informações.

A entidade Pet foi projetada com a propriedade *TutorId*, que funciona como chave estrangeira referenciando a entidade *Tutor*. Além disso, a propriedade de navegação *Tutor* foi definida para estabelecer a associação entre os objetos no domínio da aplicação.

```
        [Required]
        public int TutorId { get; set; }

        [ForeignKey("TutorId")]
        public Tutor? Tutor { get; set; }
    }
```

Por outro lado, a entidade *Tutor* contém uma coleção do tipo *ICollection<Pet>*, representando a relação de um-para-muitos, ou seja, um tutor pode ter vários pets registrados.

```
        public ICollection<Pet> Pets { get; set; } = new List<Pet>();
```

Durante as operações de consulta aos tutores, o carregamento dos pets associados é realizado através da extensão *Include* do *Entity Framework Core*. Nos métodos *GetAllTutorsAsync* e *GetTutorByIdAsync*, presentes na classe *TutorService*, a inclusão explícita da coleção *Pets* permite o carregamento antecipado *(eager loading)* dos dados relacionados, garantindo que as informações dos animais estejam acessíveis junto com os dados do tutor.

```
public async Task<IEnumerable<Tutor>> GetAllTutorsAsync()
{
    return await _tutorRepository
        .Query()
        .Include(t => t.Pets)
        .ToListAsync();
}
```

Essa abordagem permite que a aplicação mantenha a consistência entre os dados e possibilita consultas otimizadas e completas, cumprindo o requisito de que cada pet esteja obrigatoriamente vinculado a um tutor específico no momento do cadastro.

Para garantir que os dados inseridos no sistema sejam corretos e completos, foram implementadas validações obrigatórias nos campos essenciais, como nome e telefone, utilizando os recursos nativos do *Entity Framework Core* e do *ASP.NET Core.*

Nas classes *Tutor* e *Pet*, foram utilizados atributos de anotação como *[Required]* e *[MaxLength]* para definir regras básicas de validação.

O atributo *[Required]* garante que o campo não pode ser deixado em branco ou nulo, enquanto o *[MaxLength]* limita o número máximo de caracteres permitidos, evitando inserções excessivas ou fora do padrão.

```
[Required]
[MaxLength(100)]
public string Name { get; set; } = string.Empty;

[Required]
public string Email { get; set; } = string.Empty;

[Required]
[MaxLength(15)]
public string Phone { get; set; } = string.Empty;
```

Essas validações também estão presentes na entidade *Pet*, especialmente no campo *Name*, além de outros campos obrigatórios como *Especies*, *Breed* e *TutorId*.

Além da validação nas entidades, o controlador também faz uma verificação dos dados recebidos antes de executar as operações de cadastro e atualização.

Isso é feito por meio do *ModelState*, uma estrutura do *ASP.NET Core* que verifica automaticamente se os dados recebidos atendem as regras de validação definidas.

```
        [HttpPost]
          public async Task<ActionResult<Tutor>> AddTutor([FromBody]
Tutor tutor)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            await _tutorService.AddTutorAsync(tutor);
              return CreatedAtAction(nameof(GetTutorById), new { id =
tutor.id }, tutor);
        }
```

Essas práticas ajudam a garantir que somente dados válidos sejam enviados ao banco de dados, prevenindo erros futuros e mantendo a integridade das informações cadastradas.

Requisitos não funcionais

A aplicação também atende aos requisitos não funcionais definidos na proposta do projeto, que envolvem tecnologias utilizadas, organização do código e padrões de desenvolvimento.

A persistência dos dados foi implementada utilizando o *Entity Framework Core*, um ORM *(Object-Relational Mapper)* moderno e amplamente utilizado na plataforma *.NET*. O banco de dados utilizado foi o *SQLite*, que é uma solução leve e eficaz para aplicações pequenas e médias.

A conexão com o banco de dados foi configurada no arquivo *Program.cs*, onde o contexto de dados é injetado no contêiner de dependência da aplicação.

```
            builder.Services.AddDbContext<PetshopContext>(options =>
                options.UseSqlite("Data Source=petshop.db"));
```

O *AppDbContext* representa o contexto principal da aplicação e expõe as tabelas Tutors e Pets por meio das propriedades *DbSet<Tutor>* e *DbSet<Pet>*, mapeando diretamente as entidades para o banco de dados.

O código-fonte da aplicação foi organizado e disponibilizado em um repositório público no *GitHub*, conforme solicitado. A estrutura do repositório segue boas práticas de organização, com separação clara entre as camadas da aplicação (*Controllers*, *Services*, *Repositories*, *Entities*, etc.), o que favorece a legibilidade.

A aplicação foi construída seguindo boas práticas de desenvolvimento de software. Um dos principais padrões adotados foi o Padrão *Repository*, que centraliza a lógica de acesso a dados, desacoplando essa responsabilidade dos serviços e controladores.

Foi criada uma interface genérica *IRepository<T>* e sua respectiva implementação *Repository<T>*, utilizadas por serviços como *TutorService* e *PetService*. Isso permite a reutilização de código e facilita a manutenção.

```
        private readonly IRepository<Tutor> _tutorRepository;

        public TutorService(IRepository<Tutor> tutorRepository)
        {
            _tutorRepository = tutorRepository;
        }
```

CONCLUSÃO

O desenvolvimento do sistema para gerenciamento de clínicas veterinárias permitiu aplicar na prática diversos conceitos abordados na disciplina de Modelagem de Software. Durante a implementação, foi possível compreender melhor o papel dos requisitos funcionais e não funcionais no planejamento e construção de uma aplicação robusta.

Um dos principais desafios enfrentados foi garantir a correta separação de responsabilidades entre as camadas da aplicação, especialmente ao implementar o padrão de repositório de forma genérica. Também exigiu atenção especial a questões como o carregamento de relacionamentos entre entidades (por exemplo, Tutor e seus Pets), que em alguns testes no Postman, não retornou corretamente o esperado.

De forma geral, o projeto serviu como uma experiência completa de aplicação dos conhecimentos adquiridos em sala de aula, e demonstrou como o uso de boas práticas e ferramentas adequadas contribui significativamente para o desenvolvimento de sistemas mais estruturados, confiáveis e fáceis de manter.

REFERÊNCIAS BIBLIOGRÁFICAS

MICROSOFT LEARN. *Entity Framework Core*. Disponível em: https://learn.microsoft.com/pt-br/ef/core/ . Acesso em: 19 mai. 2025.

AULA ULBRA. *Aula quatorze de Modelagem de Software: Diagrama de Classes Web API, Interfaces, Services e Injeção de Dependências*. Disponível em: https://ulbra.instructure.com/courses/16192/pages/diagrama-de-classes-web-api-interfaces-services-e-injecao-de-dependencias?module_item_id=391890 . Acesso em: 19 mai. 2025.

AULA ULBRA. *Aula quinze de Modelagem de Software: Implementando Persistência com Entity Framework Core, SQLite e Padrão Repository*. Disponível em:https://ulbra.instructure.com/courses/16192/pages/aula-implementando-persistencia-com-entity-framework-sqlite-e-padrao-repository?module_item_id=398555. Acesso em: 19 mai. 2025.

SQLITE. Disponível em: https://www.sqlite.org/index.html . Acesso em: 19 mai. 2025.

MICROSOFT. *Dotnet*. Disponível em: https://dotnet.microsoft.com/pt-br/ . Acesso em: 19 mai. 2025.