# Instituto Politécnico do Cávado e do Ave

## Relatório de Integração de Sistemas de Informação


Licenciatura em Engenharia de Sistemas Informáticos

Pedro Martins, nº 23527

Luís Anjo, nº 23528

Diogo Silva, nº 23893

Barcelos, Portugal
2023/2024

# Introdução

# Índice de Figuras

# 1 - Introdução

No âmbito das unidades curriculares de Projeto Aplicado, Programação de Dispositivos Móveis e Integração de Sistemas de Informação foi proposto o desenvolvimento de uma Aplicação Móvel no qual será possível disponibilizar alojamentos locais para reserva, assim como, efetuar a reserva dos mesmos.

Desenvolvemos o presente relatório com o objetivo de descrever o contexto e os detalhes do projeto, bem como as funcionalidades da aplicação desenvolvida e os requisitos funcionais e não funcionais da mesma. Estas funcionalidades serão representadas através de Diagramas de Casos de Uso, Diagrama de modelo de dados (ER), Diagramas de Atividades, Diagramas de Estados, Diagramas de Sequência de Ecrã e Mockups da aplicação.

# 2 - Descrição do projeto

A nossa aplicação móvel servirá para os utilizadores poderem efetuar alugueres de alojamentos e disponibilizar o seu alojamento para aluguer. Os **utilizadores** do nosso projeto serão pessoas comuns que pretendem efetuar uma reserva em um determinado alojamento seja para viver, férias ou até para um estudante que pretende ingressar na faculdade, tendo este último a oportunidade de alugar um quarto e a possibilidade de partilhar a casa com outros estudantes universitários para que a mesma fique mais barata.

Os utilizadores da aplicação vão contribuir para a rentabilidade do nosso projeto, uma vez que, ao efetuarem reservas nos alojamentos, iremos receber uma comissão.

# 3 - Objetivos de negócio / Mais Valias

- Este projeto irá ajudar as pessoas devido às **dificuldades de encontrar espaços de aluguer**, estas desconhecem ou não têm um alojamento para alugar, e vai ajudar nesse sentido, dando um passo à frente de outras apps como o "airbnb" ao ter a possibilidade de alugar um quarto compartilhado por parte dos universitários.
- Antes da publicação de um alojamento ou quarto universitário, este irá passar por uma análise do administrador, de forma a manter um estatuto de alojamentos a bom preço em relação à qualidade do produto. Esta análise dará competitividade às plataformas da atualidade.
- Em caso de alguma dúvida, a equipa de suporte estará ativa 24/7 de forma a evitar que os usuários migrem para outras plataformas concorrentes.

# 4 - API (Backend)

Para o desenvolvimento da API da nossa plataforma usamos a linguagem de programação C# e a framework .NET 6.0. Foram definidos os endpoints da API (rotas), que serão os caminhos para aceder aos recursos e executar operações específicas. Criamos middlewares para filtrar os acessos aos endpoints.

De forma a priorizar a segurança usamos as bibliotecas jwt e Bcrypt. A biblioteca jwt foi usada para gerir os tokens da plataforma, enquanto a biblioteca Bcrypt foi usada para encriptar a password do utilizador.

Para o armazenamento dos dados da plataforma usamos o SQL. Para o gerenciamento da base de dados usamos o Microsoft SQL Server Management Studio (MSSMS).

Por fim, para que os membros do grupo tenham acesso à base de dados em tempo real, colocamos a mesma online a partir do AzureDevOps.

Repositório Api: https://github.com/ISI2324/tp02-03

# 5 -Endereço das API's

EzBooking: https://bookapih.azurewebsites.net/

AuthenticationEzBooking:
https://authenticationezbooking20231222152833.azurewebsites.net/

# 6 - Desenvolvimento das API's

Para a implementação da nossa plataforma foram desenvolvidas 2 API's, EzBooking e AuthenticationEzBooking.

EzBooking será uma API dedicada à gestão e partilha de alojamentos, enquanto a AuthenticationEzBooking será uma outra API dedicada apenas à autenticação dos utilizadores.

# 7 -Testes Integração

Para a realização dos testes de integração usamos o Postman.

## 7.1 - Teste (Criar alojamento)

Para realizar o teste da criação do alojamento tivemos de realizar a sequência até que o utilizador consiga efetuar a criação. Para isso, o teste terá de criar um utilizador, entrar na aplicação fazendo um login e por fim, criar o alojamento.

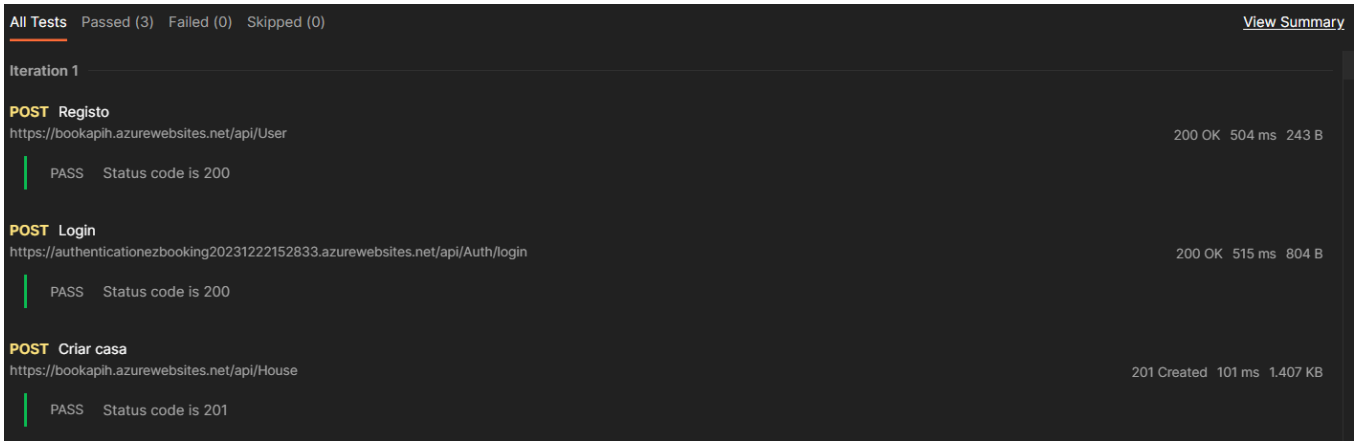



*Figura 1 - Teste (Criar alojamento)*

## 7.2 - Teste (Efetuar reserva)

Para o sucesso da criação da reserva o teste terá de criar um utilizador, entrar na aplicação fazendo um login e por fim, criar o alojamento.

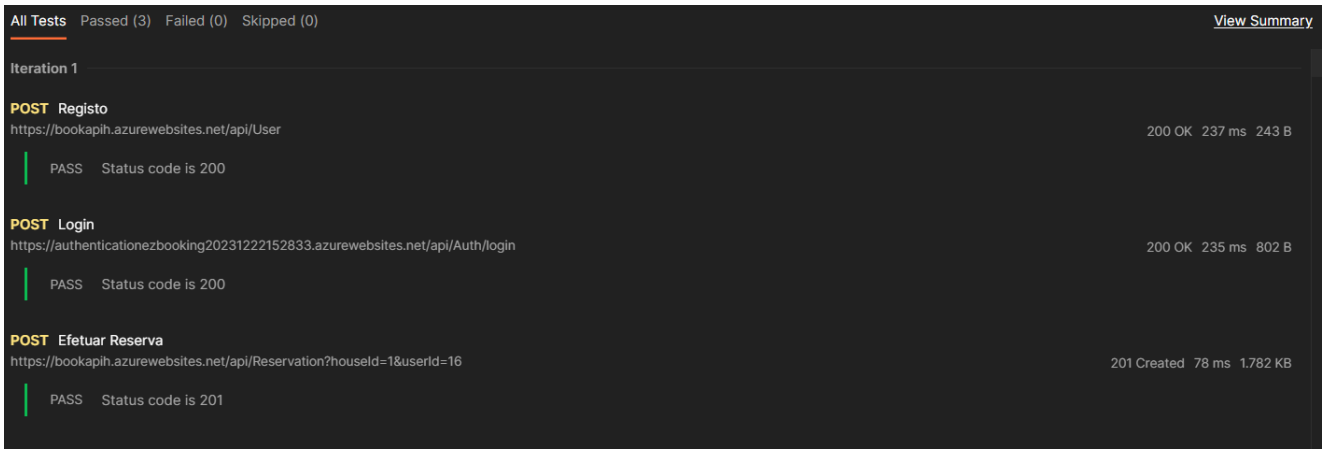
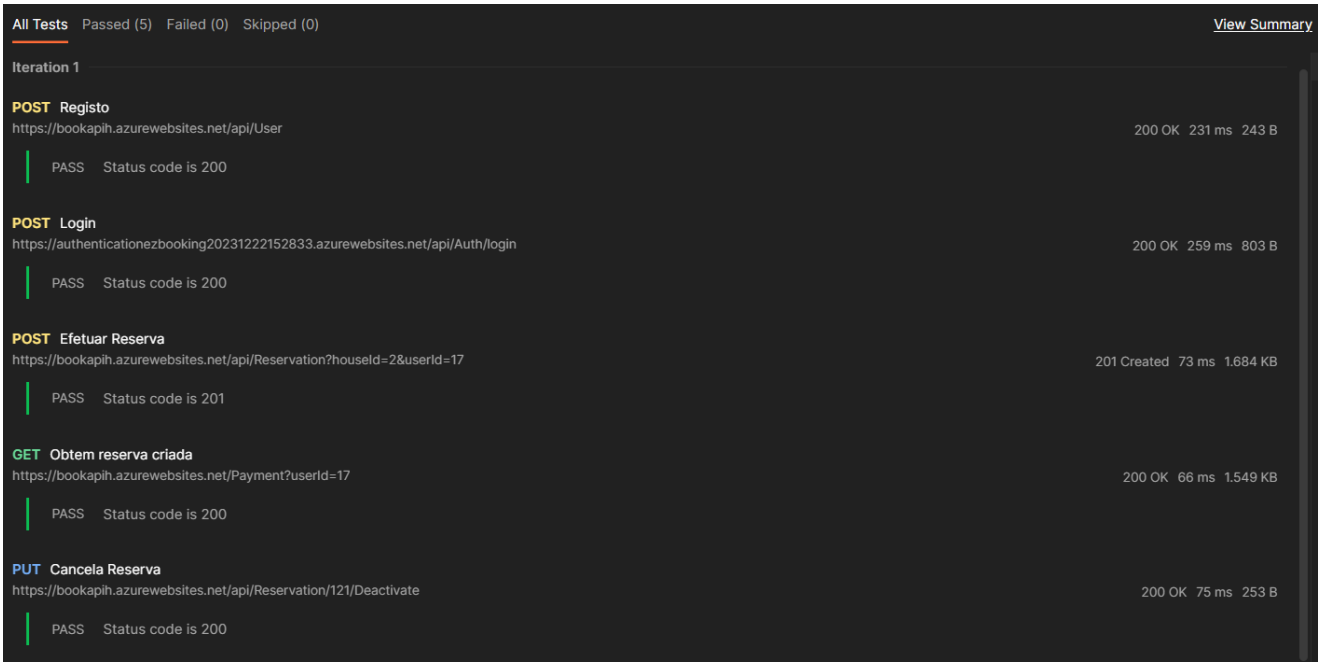



*Figura 2 - Teste (Efetuar reserva)*

## 7.3 - Teste (Cancelar Reserva)

Para que o cancelamento da reserva seja possível, o teste terá de criar um utilizador, entrar na aplicação fazendo um login, efetuar uma reserva, obter a última reserva criada e por fim, cancelar a mesma.



## 7.4 - Teste (Realizar feedback)

Para a sequência da criação do feedback, o teste terá de criar um utilizador, entrar na aplicação fazendo um login, efetuar uma reserva, alterar a reserva para o estado completo e por fim, realizar o feedback.

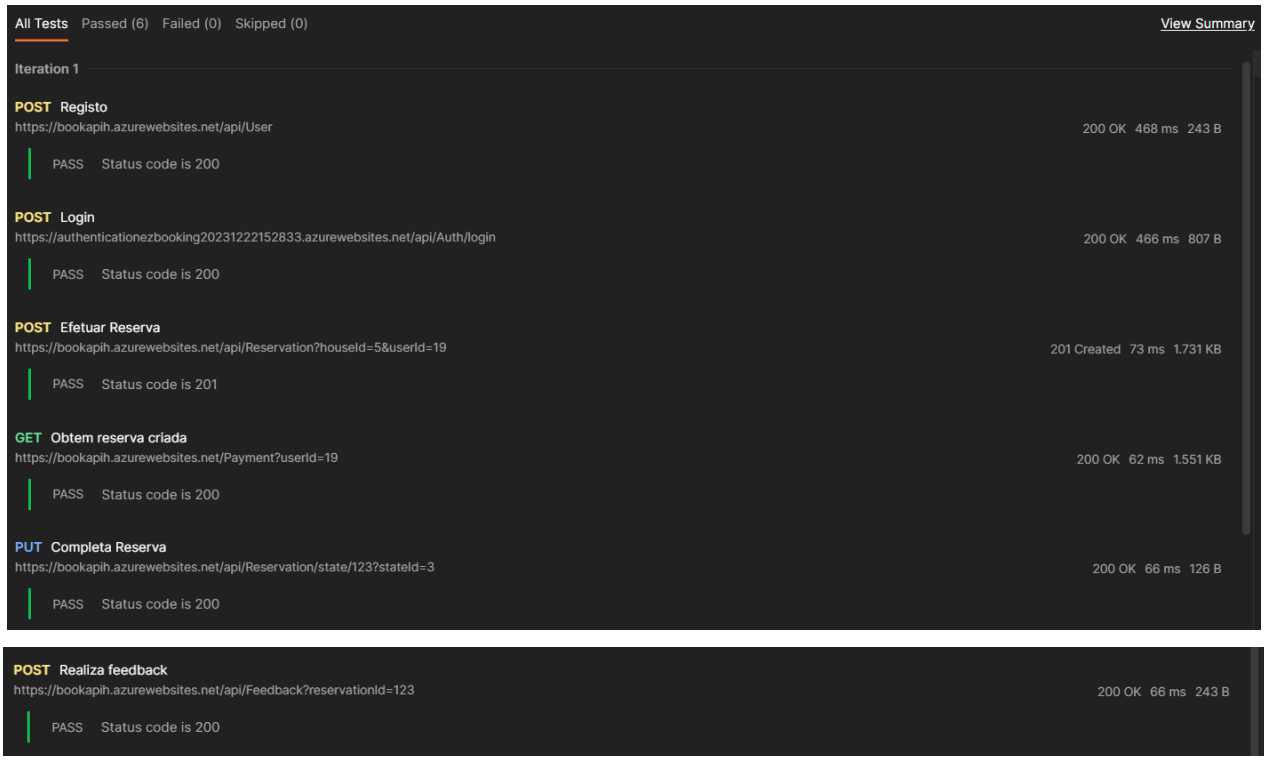






*Figura 4 - Teste (Realizar feedback)*

# 8 - DataContext

Usamos o arquivo DataContext para a criação das tabelas indicando também o nome das mesmas na base de dados.

```
13 references
public DbSet<House> Houses{ get; set; }
5 references
public DbSet<PostalCode> PostalCodes { get; set; }
5 references
public DbSet<StatusHouse> StatusHouses { get; set; }
6 references
public DbSet<User> Users { get; set; }
9 references
public DbSet<Reservation> Reservations { get; set; }
6 references
public DbSet<ReservationStates> ReservationStates { get; set; }

6 references
public DbSet<Feedback> Feedbacks { get; set; }
5 references
public DbSet<Payment> Payments { get; set; }
4 references
public DbSet<PaymentStates> PaymentStates { get; set; }

2 references
public DbSet<Images> Images { get; set; }
5 references
public DbSet<UserTypes> UserTypes { get; set; }
0 references
public DbSet<RevokedTokens> RevokedTokens { get; set; }
```

*Figura 5 - DataContext*

# 9 - Seed (Inserção de dados na base de dados)

Criamos um arquivo Seed. Este servirá como uma ferramenta essencial para iniciar o desenvolvimento, garantindo uma base de dados com vários dados de exemplo. Esta inserção de dados irá facilitar testes e demonstrações durante o desenvolvimento do sistema.

Exemplos da criação de dados no Seed:

```
var user1 = new User
{
    name = "Pedro",
    email = "pedro@alunos.ipca.pt",
    password = hashedPassword1,
    phone = 123456789,
    token = null,
    status = true,
    image = null,
    imageFormat = null,
    userType = userType1
};
```

*Figura 7 - Seed (Exemplo do Utilizador)*

```
var house1 = new House
{
    name = "Example House 1",
    doorNumber = 123,
    floorNumber = 2,
    price = 100.0,
    rooms = 4,
    guestsNumber = 4,
    road = "Example Road 1",
    propertyAssessment="dffddsf2",
    sharedRoom = false,
    PostalCode = postalCode1,
    StatusHouse = statusHouse1,
    User = user1
};
```

*Figura 6 - Seed (Exemplo do Alojamento)*

```
var reservation1 = new Reservation
{
    init_date = new DateTime(2023, 11, 18),
    end_date = new DateTime(2023, 11, 20),
    guestsNumber = 1,
    User = user1,
    House = house1,
    ReservationStates = statusReservation1
};
```

*Figura 8 - Seed (Exemplo da Reserva)*

Após a criação das variáveis com o respetivo conteúdo de cada modelo, chamamos a função "AddRange" pertencente a dataContext para inserir os dados:

```
dataContext.PostalCodes.AddRange(postalCode1, postalCode2);
dataContext.StatusHouses.AddRange(statusHouse1, statusHouse2, statusHouse3);
dataContext.Houses.AddRange(house1, house2);
dataContext.UserTypes.AddRange(userType1, userType2, userType3);
dataContext.Users.AddRange(user1, user2, user3);
dataContext.Feedbacks.AddRange(feedback1, feedback2, feedback3);
dataContext.PaymentStates.AddRange(paymentState1, paymentState2, paymentState3);
dataContext.Payments.AddRange(payment1, payment2, payment3);
dataContext.Images.AddRange(image1,image2,image3);

dataContext.ReservationStates.AddRange(statusReservation1, statusReservation2, statusReservation3, statusReservation4, statusReservation5);
dataContext.Reservations.AddRange(reservation1, reservation2, reservation3);

dataContext.SaveChanges();
```

*Figura 9 - Seed (Inserção dos dados)*

# 10 - Attributes

Para filtrar os acessos aos endpoints foi criado um arquivo com attributes. Os attributes serão chamados nos métodos dos controladores conforme este exemplo:

```
/// <summary>
/// Obtém todos os utilizadores.
/// </summary>
/// <returns>Uma lista de Utilizadores.</returns>
[HttpGet]
[AuthAuthorize]
[AdminAuthorize]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
0 references
public async Task<ActionResult<IEnumerable<User>>> GetUsers()
{
    var users = await _userRepo.GetUsers();

    if (users == null || users.Count == 0)
    {
        return NotFound("Nenhum utilizador encontrado."); //404
    }

    return Ok(users);
}
```

*Figura 10 - Exemplo Listagem de Utilizadores*

Este método do controlador está restringido. Apenas administradores logados terão acesso a este endpoint.

## 10.1 - AuthAttribute

O atributo começa por aceder ao token que está no header, caso este não encontre o token dará o erro 401 "Acesso não autorizado".

Após a receção do token acede ao repositório da autenticação e usa a função "TokenIsRevoked" para verificar se o token já foi anteriormente revocado.

Após a verificação do token, irá verificar se o token é valido.

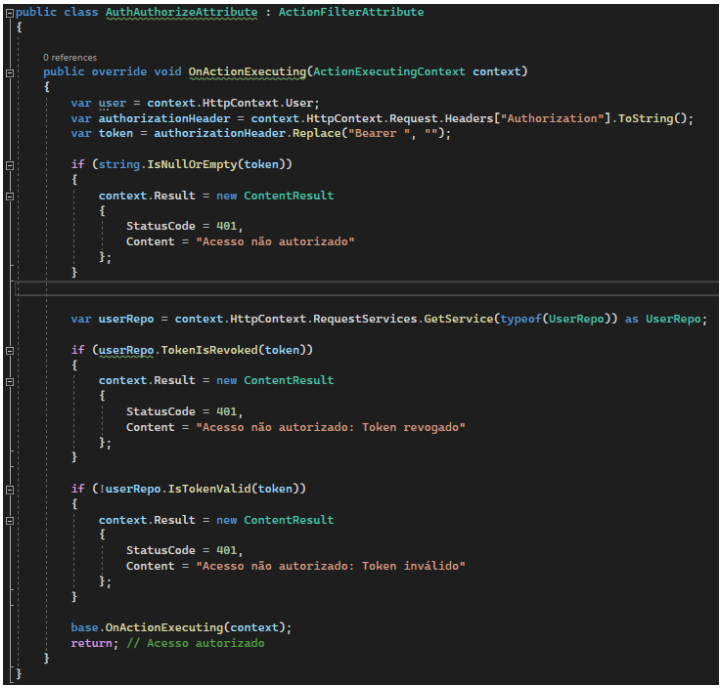



*Figura 11 - AuthAttribute*

## 10.2 - AdvertiserAttribute

Este atributo recebe o utilizador que está logado na plataforma e acede ao seu tipo de utilizador que está guardado no seu token. Caso o valor do tipo de utilizador seja maior ou igual a 2, ou seja, uma permissão maior ou iguais às do anunciante, este terá acesso ao endpoint.

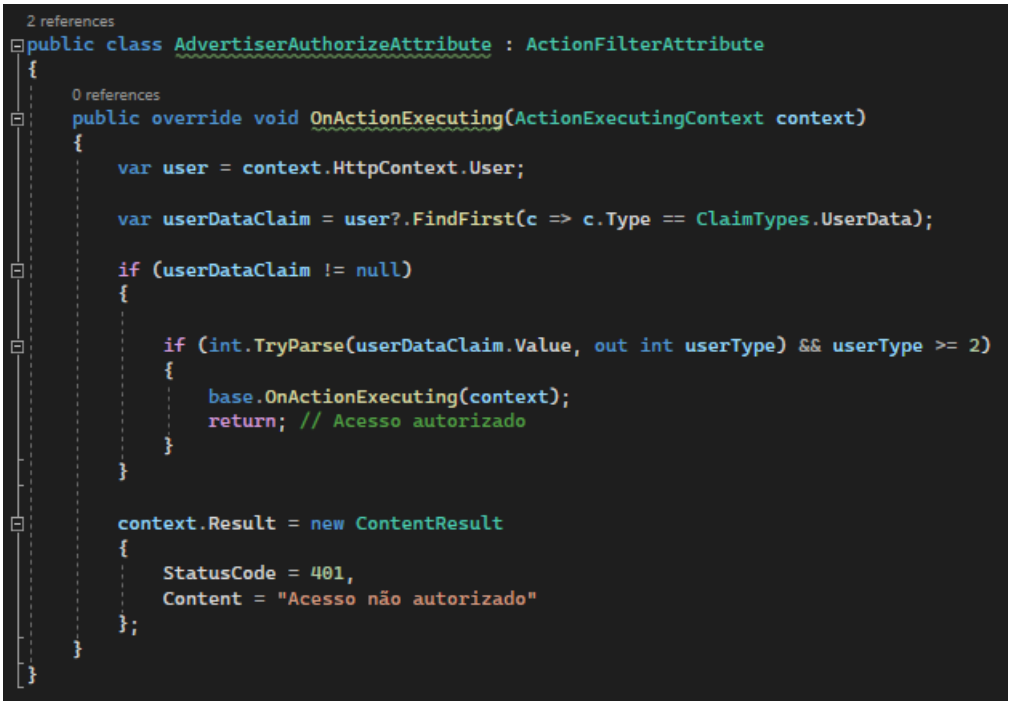



*Figura 12 - AdvertiserAttribute*

### 10.3 - AdminAttribute

O atributo e verifica se o tipo de utilizador é igual a 3, ou seja, se o utilizador é um administrador. Caso o utilizador seja um administrador, este terá acesso ao endpoint.

```
public class AdminAuthorizeAttribute : ActionFilterAttribute
{
    0 references
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var user = context.HttpContext.User;

        if (user != null && user.HasClaim(c => c.Type == ClaimTypes.UserData && c.Value == "3"))
        {
            base.OnActionExecuting(context);
            return; // Acesso autorizado
        }
        else
        {
            context.Result = new ContentResult
            {
                StatusCode = 401,
                Content = "Acesso não autorizado"
            };
        }


    }
}
```

*Figura 13 – AdminAttribute*

## 11 - Exemplos Swagger

Criamos um documento com exemplos de dados do swagger. Este documento irá alterar os dados predefinidos do swagger.

Exemplos do alojamento:

```
public const string NameExample = "Casa Azul";
public const int DoorNumberExample = 123;
public const int FloorNumberExample = 2;
public const double PriceExample = 100.0;
public const double PriceYearExample = 12000.0;
public const int GuestsNumberExample = 5;
public const string RoadExample = "Rua Principal";
public const string PropertyAssessmentExample = "A4gffX";
public const int CodDoorExample = 456;
public const bool SharedRoomExample = false;
```

*Figura 14 - Exemplos Swagger Alojamento*

Inserção dos exemplos no swagger:

```
if (context.Type == typeof(House))
{
    schema.Example = new OpenApiObject
    {
        ["name"] = new OpenApiString(Examples.NameExample),
        ["doorNumber"] = new OpenApiInteger(Examples.DoorNumberExample),
        ["floorNumber"] = new OpenApiInteger(Examples.FloorNumberExample),
        ["price"] = new OpenApiDouble(Examples.PriceExample),
        ["guestsNumber"] = new OpenApiInteger(Examples.GuestsNumberExample),
        ["rooms"] = new OpenApiInteger(Examples.GuestsNumberExample),
        ["road"] = new OpenApiString(Examples.RoadExample),
        ["propertyAssessment"] = new OpenApiString(Examples.PropertyAssessmentExample),
        ["codDoor"] = new OpenApiInteger(Examples.CodDoorExample),
        ["sharedRoom"] = new OpenApiBoolean(Examples.SharedRoomExample),
        ["postalCode"] = new OpenApiObject
        {
            ["postalcode"] = new OpenApiInteger(Examples.postalCode),
            ["concelho"] = new OpenApiString(Examples.concelho),
            ["district"] = new OpenApiString(Examples.district)
        }
    };
}
```

*Figura 15 - Inserção do exemplo*

Após a inserção do exemplo no swagger, será este o resultado:

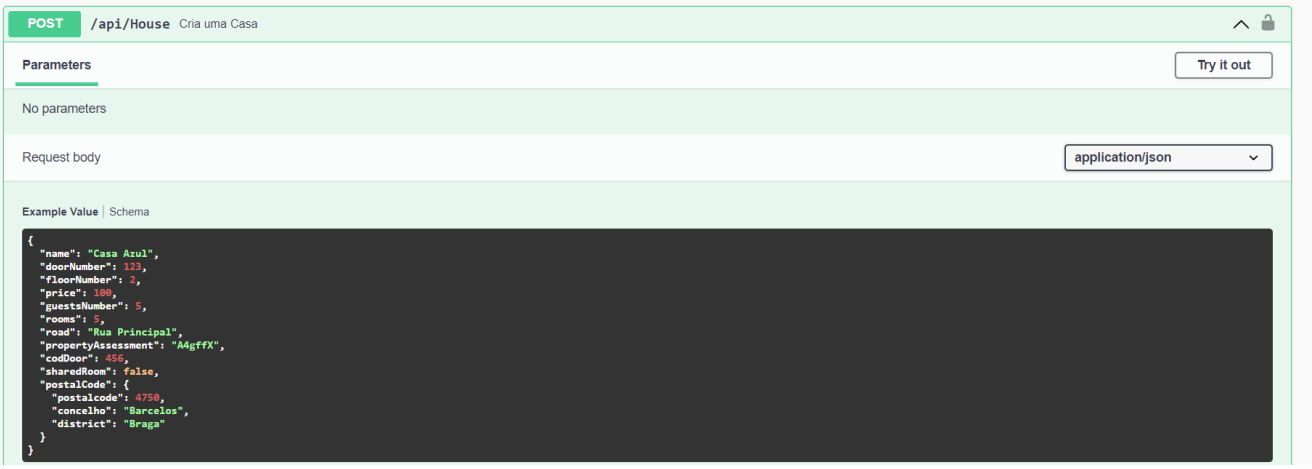
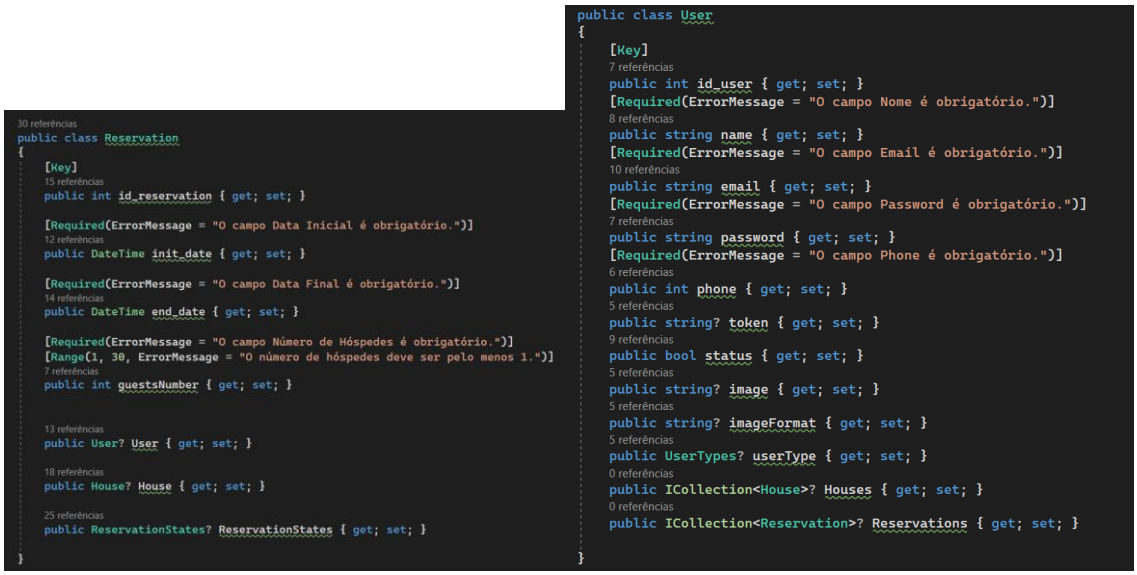



*Figura 16 - Swagger com exemplos*

# 12 - API - Modelos

Os Modelos foram usados para definir as estruturas das tabelas, bem como as relações entre elas. Para demonstrar os exemplos de modelos desenvolvidos no nosso código temos os exemplos:



```
30 referências
public class Reservation
{
    [Key]
    15 referências
    public int id_reservation { get; set; }

    [Required(ErrorMessage = "O campo Data Inicial é obrigatório.")]
    12 referências
    public DateTime init_date { get; set; }

    [Required(ErrorMessage = "O campo Data Final é obrigatório.")]
    14 referências
    public DateTime end_date { get; set; }

    [Required(ErrorMessage = "O campo Número de Hóspedes é obrigatório.")]
    [Range(1, 30, ErrorMessage = "O número de hóspedes deve ser pelo menos 1.")]
    7 referências
    public int guestsNumber { get; set; }


    13 referências
    public User? User { get; set; }

    18 referências
    public House? House { get; set; }

    25 referências
    public ReservationStates? ReservationStates { get; set; }

}
```

```
public class User
{
    [Key]
    7 referências
    public int id_user { get; set; }
    [Required(ErrorMessage = "O campo Nome é obrigatório.")]
    8 referências
    public string name { get; set; }
    [Required(ErrorMessage = "O campo Email é obrigatório.")]
    10 referências
    public string email { get; set; }
    [Required(ErrorMessage = "O campo Password é obrigatório.")]
    7 referências
    public string password { get; set; }
    [Required(ErrorMessage = "O campo Phone é obrigatório.")]
    6 referências
    public int phone { get; set; }
    5 referências
    public string? token { get; set; }
    9 referências
    public bool status { get; set; }
    5 referências
    public string? image { get; set; }
    5 referências
    public string? imageFormat { get; set; }
    5 referências
    public UserTypes? userType { get; set; }
    0 referências
    public ICollection<House>? Houses { get; set; }
    0 referências
    public ICollection<Reservation>? Reservations { get; set; }
}
```

*Figura 17 - Modelo da Reserva / User na API*

Como exemplo, no modelo da Reserva definimos os atributos e o tipo de cada atributo. No caso, temos um id, que será um inteiro, as datas inicial e final da reserva, que são dois atributos “DateTime”, e que são dois campos obrigatórios, e o número de hóspedes que será um inteiro.

Temos também em cada modelo as relações necessárias para o funcionamento da aplicação. Para tal, o modelo da Reserva tem os atributos das Classes Utilizador, Casa e Estados de Reserva. No caso da referência do Utilizador, temos uma relação de um para Muitos, ou seja, uma Reserva terá apenas um Utilizador, mas um Utilizador poderá ter várias Reservas.

# 13 - API - Repositórios

Os Repositórios da API foram utilizados para gerir o conteúdo relativo aos dados de cada modelo.

## 13.1 - API AuthenticationEzBooking

A gestão do conteúdo no repositório da API da autenticação foi feita diretamente na base de dados usando comandos em query.

### 13.1.1 - Criação do token

Para a implementação do login na plataforma usamos a biblioteca Bcrypt para encriptar a palavra-passe do utilizador e a biblioteca JWT para gerir os tokens dos utilizadores.

Este método irá gerar um token do tipo HmacSha512Signature. O token irá conter *claims* os claims servirão para ter acesso aos dados dos utilizadores a partir do seu token. Estes irão guardar o email, tipo de utilizador e o seu respetivo ID.

Por fim, definimos que o token será expirado após 1 dia.

```
1 reference
public string CreateToken(int userId, int userType)
{
    string userEmail = GetUserEmailById(userId);
    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.Email, userEmail),
        new Claim(ClaimTypes.UserData, userType.ToString()),
        new Claim(ClaimTypes.NameIdentifier, userId.ToString(), ClaimValueTypes.Integer32),
    };

    var key = new SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes(
        _configuration.GetSection("AppSettings:Token").Value));

    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

    var token = new JwtSecurityToken(
        claims: claims,
        expires: DateTime.Now.AddDays(1),
        signingCredentials: creds);

    var jwt = new JwtSecurityTokenHandler().WriteToken(token);

    return jwt;
}
```

*Figura 18 - Criação do token*

### 13.1.2 - Acesso a variáveis

Para acessar a base de dados a partir do repositório da API da autenticação usamos o SqlConnection para nos conectarmos com a base de dados para permitir posteriormente os

comandos query. Usamos o tipo de conexão mencionado em DefaultConntection nas propriedades do sistema:

```
"ConnectionStrings": {
 "DefaultConnection": "Data Source=bkbd.database.windows.net;Initial
Catalog=bookinghouses;User ID=rootadmin;Password=Root123!;Connect
Timeout=30;Encrypt=True;Trust Server Certificate=False;Application
Intent=ReadWrite;Multi Subnet Failover=False"},
```

Exemplo para saber o token a partir do ID do utlizador:

```
0 references
public string GetTokenById(int userId)
{
    using (SqlConnection conn = new SqlConnection(_configuration.GetConnectionString("DefaultConnection")))
    {
        conn.Open();

        string query = "SELECT token FROM Users WHERE id_user = @userId";

        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            cmd.Parameters.AddWithValue("@userId", userId);

            object result = cmd.ExecuteScalar();

            string token = (result != null) ? result.ToString() : string.Empty;

            return token;
        }
    }
}
```

*Figura 19 - Acesso ao token*

### 13.1.3 - Revocação do token

Para a revocação do token criamos 2 funções. A função RevokeToken servirá para inserir o token do utilizador na tabela RevokedTokens. Isto será usado a implementação do método logout:

```
2 references
public void RevokeToken(string token)
{

    using (SqlConnection conn = new SqlConnection(_configuration.GetConnectionString("DefaultConnection")))
    {
        conn.Open();

        string query = "INSERT INTO RevokedTokens (Token, RevocationDate) VALUES (@token, GETDATE())";

        using (SqlCommand cmd = new SqlCommand(query, conn))
        {
            cmd.Parameters.AddWithValue("@token", token);

            cmd.ExecuteNonQuery();
        }
    }
}
```

*Figura 20 - Revocar token*

A segunda função isTokenValid servirá para validar se o token do utilizador é valido, ou seja, se ainda não excedeu a data de expiração:

```
2 references
public bool IsTokenValid(string token)
{
    var tokenHandler = new JwtSecurityTokenHandler();

    try
    {
        var jsonToken = tokenHandler.ReadToken(token) as JwtSecurityToken;


        if (jsonToken?.ValidTo != null && jsonToken.ValidTo < DateTime.UtcNow)
        {
            return false;
        }


        return true;
    }
    catch (Exception)
    {
        return false;
    }
}
```

*Figura 21 - Validação do token*

## 13.2 - API EzBooking

```
1 referência
public async Task<ICollection<Reservation>> GetReservations()
{
    return await _context.Reservations
    .Include(r => r.House)
    .Include(r => r.User)
    .Include(r => r.ReservationStates)
    .OrderBy(r => r.id_reservation)
    .ToListAsync();
}

1 referência
public async Task<Reservation> GetReservationPayment(int userId)
{
    return await _context.Reservations
    .Where(r => r.User.id_user == userId && r.ReservationStates.id == 1)
    .Include(r => r.ReservationStates)
    .Include(r => r.House)
    .ThenInclude(r => r.PostalCode)
    .Include(r => r.House)
    .ThenInclude(h => h.User)
    .OrderByDescending(r => r.id_reservation)
    .FirstOrDefaultAsync();
}

8 referências
public async Task<Reservation> GetReservationById(int id)
{
    return await _context.Reservations
        .Include(r => r.House)
        .ThenInclude(h => h.PostalCode)
        .Include(r => r.User)
        .Include(r => r.ReservationStates)
        .FirstOrDefaultAsync(r => r.id_reservation == id);

}
```

*Figura 22 - Funções para Obter dados do Repositório das Reservas*

Em cada repositório, temos as respetivas funções “Get” utilizadas para obter os dados das tabelas de cada modelo. Neste caso, temos três funções:

- A função “GetReservations” que irá obter todas as reservas presentes na base de dados;

- A função “GetReservationPayment” que irá obter o pagamento associado a uma reserva;

- A função “GetReservationById” que irá obter uma reserva através do seu Id.

```
public async Task<ICollection<object>> AvailableHouses(string? location, int guestsNumber, bool? checkedV, DateTime startDate, DateTime endDate)
{
    var houses = await _context.Houses
        .Where(h => (location == null || h.PostalCode.district == location) &&
               (guestsNumber == null || h.guestsNumber >= guestsNumber) &&
               (h.sharedRoom == checkedV) &&
               (!h.Reservations.Any() ||
               h.Reservations.All(reservation =>
                    reservation.ReservationStates.id == 1 || reservation.ReservationStates.id == 3 ||
                    reservation.ReservationStates.id == 4 || reservation.ReservationStates.id == 5 ||
                    reservation.end_date <= startDate || reservation.init_date >= endDate))
               )
        .Select(h => new
        {
            h.id_house,
            h.name,
            h.price,
            h.priceyear,
            h.rooms,
            h.doorNumber,
            h.propertyAssessment,
            h.guestsNumber,
            h.road,
            h.sharedRoom,
            PostalCode = new
            {
                h.PostalCode.postalCode,
                h.PostalCode.concelho,
                h.PostalCode.district
            },
            Images = h.Images.Select(img => new
            {
                img.image,
                img.formato
            }).ToList()
        })
        .OrderBy(h => h.id_house)
        .ToListAsync();

    return houses.ToList<object>();
}
```

*Figura 23 - Função para obter casas disponíveis do Repositório das Casas*

Foram criadas também algumas funções personalizadas, como é o caso da função "AvailableHouses" do Repositório das Casas, que nos permite obter apenas as casas disponíveis para reserva.

```
1 referência
public async Task<bool> CreateReservation(Reservation reservation)
{
    await _context.AddAsync(reservation);
    return Save();
}
```

*Figura 24 - Função de Criar Reserva do Repositório das Reservas*

```
public bool UpdateReservation(Reservation reservation)
{

    _context.Update(reservation);
    return Save();
}
```

*Figura 25 - Função de Alterar Reserva do Repositório das Reservas*

```
public bool DeleteReservation(Reservation reservation)
{
    _context.Remove(reservation);
    return Save();
}
```

*Figura 26 - Função de Apagar Reserva do Repositório das Reservas*

```
public bool Save()
{
    var saved = _context.SaveChanges();
    return saved > 0 ? true : false;
}

5 referências
public bool ReservationExists(int reservationid)
{
    return _context.Reservations.Any(r => r.id_reservation == reservationid);
}
```

*Figura 27 - Funções de Salvar e Verificar se a Reserva Existe do Repositório das Reservas*

Temos também para cada Repositório, as funções de Criar, Alterar e Apagar, que nos permitem fazer os respetivos comandos no conteúdo da base de dados, e as funções para salvar as alterações e para verificar se a reserva existe na base de dados.

```
public async Task<bool> ValidateReservationDate(Reservation reservation)
{
    var reservations = await _context.Reservations
        //.Where(r => r.House == reservation.House && r.ReservationStates.state != 1)
        .Where(r => r.House == reservation.House && r.ReservationStates.id == 2)
        .ToListAsync();

    foreach (Reservation r in reservations)
    {
        if (r.init_date < reservation.end_date && r.end_date > reservation.init_date)
        {
            return false; // As datas sobrep◆em, logo n◆o est◆o dispon◆veis
        }
    }

    return true; // As datas est◆o dispon◆veis
}
```

*Figura 28 - Função de Validar datas da reserva no Repositório das Reservas*

No Repositório das reservas, temos também outras funções que serão usadas nos controladores, como a função “ValidateReservationDate” que irá verificar se já existe uma reserva nas datas definidas posteriormente.

```
1 referência
public async Task<double?> CalculateTotalPrice(Reservation reservation)
{
    double? priceNight = reservation.House.price;

    DateTime checkInDate = reservation.init_date;
    DateTime checkOutDate = reservation.end_date;

    TimeSpan difference = checkOutDate - checkInDate;
    int nights = (int)Math.Ceiling(difference.TotalDays);


    double? total = priceNight * nights;

    return total;
}
```

A função “CalculateTotalPrice” que irá fazer o cálculo do preço total da reserva efetuada pelo utilizador, consoante o número de dias escolhidos.

# 14 - API – Controladores

Os controladores serão os responsáveis por receber as requisições HTTP vindas das rotas e determinar qual ação deve ser executada. Estes vão processar os dados recebidos, interagir com os Repositórios e retornar respostas apropriadas.

Dentro dos controladores, temos métodos para cada tipo de requisição HTTP, como GET, POST, PUT e DELETE.

Os controladores irão fornecer uma camada intermediária entre as rotas e os modelos, permitindo que as regras de negócio sejam implementadas de forma eficiente e modular.

## 14.1 - API AuthenticationEzBooking

### 14.1.1 - Login

Para a criação do login, criamos um Dto para que seja selecionado apenas o email e password do utilizador:

```
using System.ComponentModel.DataAnnotations;

namespace AuthenticationEzBooking.DTO
{
    5 references
    public class AuthDto
    {
        [Required(ErrorMessage = "O campo Email é obrigatório.")]
        4 references
        public string email { get; set; }
        [Required(ErrorMessage = "O campo Password é obrigatório.")]
        3 references
        public string password { get; set; }
    }
}
```

*Figura 29 - Dto para autenticação*

Este método recebe os dados do login e tenta aceder ao token do utilizador caso este já tenha um token. Caso o utilizador já tenha um token antes de fazer o login, este será revocado para que não seja utilizado novamente.

O método partirá para a validação dos dados do utilizador, caso estes sejam validados, o método irá criar um token após a recolha de alguns dados do utilizador e irá atualizar o mesmo, inserindo o seu token na base de dados.

Por fim, o método responde com os dados do utilizador e o respetivo token gerado.

```
[HttpPost("login")]
[ProducesResponseType(400)]
[ProducesResponseType(401)]
0 references
public async Task<ActionResult<string>> Login([FromBody] AuthDto login)
{
    var existingToken = _authRepo.GetTokenByEmail(login.email);

    if(!string.IsNullOrEmpty(existingToken) && !_authRepo.TokenIsRevoked(existingToken)) {
        _authRepo.RevokeToken(existingToken);
    }
    var getuser = _authRepo.GetUserByEmail(login.email);

    if (getuser == null || !BCrypt.Net.BCrypt.Verify(login.password, getuser.password))
    {
        return BadRequest("Dados inválidos");
    }
    else
    {
        int userId = _authRepo.GetUserIdbyEmail(getuser.email);
        bool userStatus = _authRepo.GetStatusById(userId);


        int getUserType = _authRepo.GetUserTypeById(userId);
        string token = _authRepo.CreateToken(userId, getUserType);
        _authRepo.UpdateUserToken(userId, token);

        var response = new LoginResponse
        {
            UserId = userId,
            UserType = getUserType,
            Token = token,
            Status = userStatus
        };
        return Ok (response);
    }
}
```

*Figura 30 – Login*

### 14.1.2 - Logout

O logout do utilizador consistirá na revocação do seu token, pois caso o token do mesmo seja revocado, este não terá acesso a métodos que necessitam autenticação:

```
[HttpPost("logout")]
[AuthAuthorize]
[ProducesResponseType(200)]
[ProducesResponseType(400)]
0 references
public async Task<ActionResult> Logout()
{
    var token = Request.Headers["Authorization"].ToString().Replace("Bearer ", "");

    if (_authRepo.IsTokenValid(token))
    {
        _authRepo.RevokeToken(token);
    }
    else
        return BadRequest("Acesso não autorizado: Token inválido");

    return Ok("Logout bem sucedido");

}
```

*Figura 31 - Logout*

Este método acede ao token que está localizado no header, valida o token e caso este seja validado, partirá para a revocação do token.

## 14.2 - API EzBooking

### 14.2.1 - Métodos “GET”

Em cada controlador, foram definidos os métodos “GET” para obter o conteúdo das bases de dados, como no seguinte exemplo no controlador das Casas:

```
/// <summary>
/// Obtém todas as Casas.
/// </summary>
/// <returns>Uma lista de Casas.</returns>
/// <response code="200">Retorna uma lista de Casas.</response>
[HttpGet("All")]
[AuthAuthorize]
[AdminAuthorize]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
0 referências
public async Task<ActionResult<IEnumerable<House>>> GetAllHouses()
{
    var houses = await _houseRepo.GetAllHouses();

    if (houses == null || houses.Count == 0)
    {
        return NotFound("Nenhuma casa encontrada."); //404
    }

    return Ok(houses);
}
```

*Figura 32 - Método para obter todas as casas no Controlador das Casas*

Com o método “GetAllHouses” no controlador das casas, podemos obter uma lista com todas as casas na base de dados, através da chamada do método “GetAllHouses” no Repositório das Casas.

```
/// <summary>
/// Obtém Uma Casa
/// </summary>
/// <param name="id" example ="1">O ID da Casa</param>
/// <returns>Uma Casa.</returns>
[HttpGet("{id}")]
[ProducesResponseType(200)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
0 referências
public async Task<ActionResult<House>> GetHouseById(int id)
{
    var house = await _houseRepo.GetHouseByIdDetail(id);

    if (house == null)
    {
        return NotFound("Casa não encontrada."); // Código 404 se a casa não for encontrada.
    }

    if (id <= 0)
    {
        return BadRequest("ID inválido."); // Código 400 se o ID for inválido.
    }

    return Ok(house);
}
```

*Figura 33 - Método para obter uma casa no Controlador das Casas*

Com o método “GetHouseById” no controlador das casas, podemos obter uma casa presente na base de dados, através do seu id que é passado por parâmetro, e chamada do método “GetHouseByIdDetail” no Repositório das Casas.

```
/// <summary>
/// Obtém Reservas de uma Casa
/// </summary>
/// <param name="id" example ="1">O ID da Casa</param>
/// <returns>Uma Casa.</returns>
[HttpGet("Reservations/{id}")]
[AuthAuthorize]
[AdvertiserAuthorize]
[ProducesResponseType(200)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
0 referências
public async Task<ActionResult<IEnumerable<Reservation>>> GetReservationsFromHouseById(int id)
{

    if (id <= 0)
    {
        return BadRequest("ID inválido."); // Código 400 se o ID for inválido.
    }
    var house = await _houseRepo.GetHouseReservations(id);

    if (house == null)
    {
        return NotFound("Casa não encontrado.");
    }

    return Ok(house ?? new List<Reservation>());
}
```

*Figura 34 - Método para obter as reservas de uma casa no Controlador das Casa*

Com o método “GetReservationsFromHouseById” no controlador das casas, podemos obter as reservas de uma casa, através do seu id que é passado por parâmetro, e chamada do método “GetHouseReservations” no Repositório das Casas.

```
[HttpGet("CodDoor/{id}")]
[ProducesResponseType(200)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
0 referências
public async Task<ActionResult<House>> GetCodDoorHouseById(int id)
{
    var house = await _houseRepo.GetHouseById(id);

    if (house == null)
    {
        return NotFound("Casa não encontrada."); // Código 404 se a casa não for encontrada.
    }

    var codHouse = _houseRepo.GetCodDoorHouseById(id);

    if (codHouse == -1)
    {
        return NotFound("Casa não tem código."); // Código 404 se a casa não for encontrada.
    }

    if (id <= 0)
    {
        return BadRequest("ID inválido."); // Código 400 se o ID for inválido.
    }

    return Ok(codHouse);
}
```

*Figura 35- Obtem Código da Porta da Casa*

Com método “GetCodDoorHouseById” no controlador, obtemos o código da porta da casa para o utilizador introduzir o código para abrir a casa quando chega a mesma.

### 14.2.2 - Métodos “POST”

Foram definidos os métodos “POST” para cada controlador. No seguinte exemplo temos o método “CreateHouse” para criar uma casa nova:

```
//CREATES
/// <summary>
/// Cria uma Casa
/// </summary>
/// <returns>Cria uma Casa.</returns>
[HttpPost]
[AuthAuthorize]
[ProducesResponseType(201)]
[ProducesResponseType(409)]
0 referências
public async Task<IActionResult> CreateHouse([FromBody] House house)
{
    if (house == null)
    {
        return BadRequest("Dados inválidos");
    }

    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }


    if (house.price != null && house.priceyear != null)
        return BadRequest("A casa so pode ter preço mensal ou anual");

    PostalCode existingPostalCode = _postalCodeRepo.GetPostalCodeById(house.PostalCode.postalCode);

    if (existingPostalCode != null && _houseRepo.PostalCodePropertyExists(existingPostalCode.postalCode, house.propertyAssessment))
        return StatusCode(409, "Já Existe uma casa com esse artigo matricial nesse codigo postal");

    if (existingPostalCode == null)
    {
        existingPostalCode = new PostalCode
        {
            postalCode = house.PostalCode.postalCode,
            concelho = house.PostalCode.concelho,
            district = house.PostalCode.district,
        };
        _postalCodeRepo.CreatePostalCode(existingPostalCode);
    }

    house.PostalCode = existingPostalCode;

    StatusHouse status = _statusHouseRepo.GetStatusHouseById(1);
    house.StatusHouse = status;

    var userId = HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

    User userLogged = _userRepo.GetUser(int.Parse(userId));
    house.User = userLogged;


    await _houseRepo.CreateHouse(house);

    return CreatedAtAction("CreateHouse", new { id = house.id_house }, house);
}
```

*Figura 36 - Método para criar uma Casa no Controlador das Casas*

Ao utilizar a rota para criar a casa, o controlador irá executar este método e assim criar uma casa nova com os parâmetros inseridos pelo utilizador. Esta função irá obter o “id” do utilizador logado, e irá também fazer a verificação do Código postal para obter um código postal já existente ou efetuar a criação de um novo Código postal. A casa será assim criada com o estado de “id” um, que corresponde na base de dados ao estado pendente, ficando assim pendente para a aprovação do Administrador. A criação da casa fica então concluída através da chamada da função “CreateHouse” do Repositório das Casas.

### 14.2.3 - Métodos “PUT”

Foram definidos os métodos “PUT” para cada controlador. No seguinte exemplo temos o método “UpdateHouse” para editar uma casa existente:

```
/// <summary>
/// Altera Dados de uma casa.
/// </summary>
/// <param name="id" example ="1">O ID da Casa</param>
/// <returns>Uma lista de Casas.</returns>
[HttpPut("{id}")]
[ProducesResponseType(400)]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
0 referências
public async Task<IActionResult> UpdateHouse(int id, [FromBody] House house)
{
    if (house == null)
        return BadRequest(ModelState);

    if (!_houseRepo.HouseExists(id))
        return NotFound();

    if (!ModelState.IsValid)
        return BadRequest();


    PostalCode existingPostalCode = _postalCodeRepo.GetPostalCodeById(house.PostalCode.postalCode);

    if (existingPostalCode == null)
    {
        existingPostalCode = new PostalCode
        {
            postalCode = house.PostalCode.postalCode,
            concelho = house.PostalCode.concelho,
            district = house.PostalCode.district,
        };
        _postalCodeRepo.CreatePostalCode(existingPostalCode);
    }

    StatusHouse status = _statusHouseRepo.GetStatusHouseById(1);
    var rhouse = await _houseRepo.GetHouseById(id);
    rhouse.name = house.name;
    rhouse.price = house.price;
    rhouse.priceyear = house.priceyear;
    rhouse.codDoor = house.codDoor;
    rhouse.floorNumber = house.floorNumber;
    rhouse.doorNumber = house.doorNumber;
    rhouse.guestsNumber = house.guestsNumber;
    rhouse.rooms = house.rooms;
    rhouse.propertyAssessment = house.propertyAssessment;
    rhouse.road = house.road;
    rhouse.sharedRoom = house.sharedRoom;
    rhouse.StatusHouse = status;
    rhouse.PostalCode = existingPostalCode;


    bool codeChanged = rhouse.propertyAssessment != house.propertyAssessment;
    if (existingPostalCode != null && codeChanged && _houseRepo.PostalCodePropertyExists(existingPostalCode.postalCode, house.propertyAssessment))
        return StatusCode(409, "Já Existe uma casa com esse artigo matricial nesse codigo postal");

    _houseRepo.UpdateHouse(rhouse);

    return Ok();
}
```

*Figura 37 - Método para Editar uma casa existente no controlador das casas*

Ao utilizar a rota para editar a casa, o controlador irá executar este método e assim editar uma casa existente, com os parâmetros inseridos pelo utilizador. Esta função irá receber por parâmetro o “id” da casa a editar, irá verificar se esta casa existe na base dados e irá também verificar se já existe um Código postal igual na base de dados, e se não existir irá criar um novo. A alteração da casa fica então concluída através da chamada da função “UpdateHouse” do Repositório das Casas.

### 14.2.4 - Métodos “DELETE”

Foram definidos os métodos “DELETE” para cada controlador. No seguinte exemplo temos o método “DeleteHouse” para apagar uma casa existente:

```
//DELETE
/// <summary>
/// Apaga uma Casa
/// </summary>
/// <param name="id" example ="1">O ID da Casa</param>
/// <returns>Exclui uma Casa.</returns>
[HttpDelete("{id}")]
[ProducesResponseType(400)]
[ProducesResponseType(200)]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
0 referências
public async Task<IActionResult> DeleteHouse(int id)
{
    if (!_houseRepo.HouseExists(id))
    {
        return NotFound();
    }

    var houseToDelete = await _houseRepo.GetHouseById(id);

    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    if (!_houseRepo.DeleteHouse(houseToDelete))
    {
        ModelState.AddModelError("", "Erro ao eliminar a Casa");
    }

    var responseMessage = $"A casa com o ID {id} foi eliminada com sucesso.";
    return Ok(responseMessage);
}
```

*Figura 38 - Método para Apagar uma Casa no controlador das casas*

Ao utilizar a rota para apagar a casa, o controlador irá executar este método e assim apagar uma casa existente. Esta função irá receber por parametro o “id” da casa a apagar, irá verificar se esta casa existe na base dados, e se esta for encontrada é apagada. A remoção da casa fica então concluida através da chamada da função “DeleteHouse” do Repositório das Casas.

### 14.2.5 - Métodos “PUT” Auxiliares

Para além destes métodos padrão, que estão presentes em todos os controladores, foram também utilizados alguns métodos auxiliares para cada controlador.

No seguinte exemplo temos o método “PUT” chamado “UpdateStateHouse” para alterar o estado da casa para aprovado:

```
/// <summary>
/// Altera Estado da casa para Aprovado.
/// </summary>
/// <param name="id" example ="1">O ID da Casa</param>
/// <returns>Uma Casa com estado atualizado.</returns>
[HttpPut("state/{id}")]
[ProducesResponseType(400)]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
0 referências
public async Task<IActionResult> UpdateStateHouse(int id)
{

    if (!_houseRepo.HouseExists(id))
        return NotFound();

    if (!ModelState.IsValid)
        return BadRequest();

    StatusHouse status = _statusHouseRepo.GetStatusHouseById(2);
    var rhouse = await _houseRepo.GetHouseById(id);
    rhouse.StatusHouse = status;
    _houseRepo.UpdateHouse(rhouse);

    return Ok(rhouse);
}
```

*Figura 39 - Método para Alterar o Estado da Casa para aprovado do controllador das casas*

Este método será utilizado pelo Administrador da aplicação para aprovar uma Casa.

Ao utilizar a rota para este método, o controlador irá executar este método e alterar o estado da casa para aprovado. Esta função irá receber por parâmetro o “id” da casa a ser editada, irá verificar se esta casa existe na base dados, e se esta for encontrada será alterado o estado da mesma para aprovado. A alteração da casa fica então concluída através da chamada da função “UpdateHouse” do Repositório das Casas, que também é utilizado aqui mas onde apenas é alterado o estado da casa como pretendido.

No seguinte exemplo temos o método “PUT” chamado “SoftDeleteUser” para alterar o estado do Utilizador para desativado, de forma a desativar o utilizador na aplicação:

```
/// <summary>
/// Altera estado do utilizador
/// </summary>
/// <param name="userId" example="1">ID do utilizador</param>
/// <returns></returns>
[HttpPut("{userId}/Deactivate")]
[AuthAuthorize]
[AdminAuthorize]
[ProducesResponseType(400)]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
0 referências
public IActionResult SoftDeleteUser(int userId)
{

    var existingUser = _userRepo.GetUser(userId);

    if (existingUser == null)
    {
        return NotFound();
    }

    if (existingUser.status == false)
        existingUser.status = true;
    else
        existingUser.status = false;



    bool updated = _userRepo.UpdateUser(existingUser);

    if (updated)
    {
        return Ok("Estado do utilizador atualizado com sucesso!");
    }
    else
    {
        ModelState.AddModelError("", "Something went wrong updating owner");
        return StatusCode(500, ModelState);
    }
}
```

*Figura 40 - Método para alterar o estado do Utilizador para desativado*

Este método será utilizado pelo Administrador da aplicação para desativar um Utilizador.

Ao utilizar a rota para este método, o controlador irá executar este método e alterar o estado do utilizador para desativado. Esta função irá receber por parâmetro o “id” do utilizador a ser editado, irá verificar se este utilizador existe na base dados, e se este for encontrado será alterado o estado para “false”, já que a variável “status” é um “bool”, desativando assim o utilizador. A alteração do utilizador fica então concluída através da chamada da função “UpdateUser” do Repositório dos Utilizadores, que também é utilizado aqui mas onde apenas é alterado o estado do utilizador como pretendido.

No seguinte exemplo temos o método “PUT” chamado “SoftDeleteReservation” para alterar o estado da Reserva para Cancelada:

```
/// <summary>
/// Cancela Reserva
/// </summary>
/// <param name="reservationId" example="1">ID da Reserva</param>
/// <returns></returns>
[HttpPut("{reservationId}/Deactivate")]
[AuthAuthorize]
[ProducesResponseType(400)]
[ProducesResponseType(204)]
[ProducesResponseType(404)]
0 referências
public async Task<IActionResult> SoftDeleteReservation(int reservationId)
{
    var existingReservation = await _reservationRepo.GetReservationById(reservationId);

    if (existingReservation == null)
        return BadRequest(ModelState);

    if (!_reservationRepo.ReservationExists(existingReservation.id_reservation))
        return NotFound();

    if (!ModelState.IsValid)
        return BadRequest();


    if (existingReservation.ReservationStates.id == 1 || existingReservation.ReservationStates.id == 2)
    {
        if (existingReservation != null && DateTime.Now > existingReservation.init_date && DateTime.Now < existingReservation.end_date)
        {

            ReservationStates status = _reservationStatesRepo.GetReservationStatesById(5);
            //var nreservation = await _reservationRepo.GetReservationById(reservation.id_reservation);
            existingReservation.ReservationStates = status;
            existingReservation.end_date = DateTime.Now;

            _reservationRepo.UpdateReservation(existingReservation);

            var respayment = _paymentRepo.GetPaymentReservation(existingReservation.id_reservation);

            if (respayment != null)
            {
                respayment.state = _paymentStateRepo.GetPaymentState(3);
                respayment.paymentValue = respayment.paymentValue * 0.5f;

                _paymentRepo.UpdatePayment(respayment);
            }

        }
        else if (existingReservation != null)
        {
            ReservationStates status = _reservationStatesRepo.GetReservationStatesById(5);
            var nreservation = await _reservationRepo.GetReservationById(existingReservation.id_reservation);
            nreservation.ReservationStates = status;
            nreservation.end_date = DateTime.Now;

            _reservationRepo.UpdateReservation(nreservation);

            var respayment = _paymentRepo.GetPaymentReservation(existingReservation.id_reservation);

            if (respayment != null)
            {
                respayment.state = _paymentStateRepo.GetPaymentState(3);

                _paymentRepo.UpdatePayment(respayment);
            }
        }
    }
    else
    {
        return BadRequest("Reserva não existe!");
    }


    return Ok("Reserva Cancelada com sucesso!");
}
```

*Figura 41 - Método para alterar o estado da Reserva para cancelada*

Este método será utilizado pelo Utilizador da aplicação, de forma a cancelar a Reserva efetuada pelo mesmo. Ao utilizar a rota para este método, o controlador irá executar este método e alterar o estado da Reserva para Cancelada. Esta função irá receber por parametro o “id” da reserva a ser editada, irá verificar se esta reserva existe na base dados, e se esta for

encontrada será alterado o estado para cancelada, cancelando assim a reserva do utilizador. A alteração da reserva fica então concluida através da chamada da função "UpdateReservation" do Repositório das Reservas, que também é utilizado aqui, mas onde apenas é alterado o estado da reserva como pretendido.

## 15 - Serviço de Atualização automática de Reservas

Para a API atualizar o estado das reservas para concluída, foi necessário criar um serviço que corre na API em segundo plano.

```
2 referências
public class ReservationUpdateService : BackgroundService
{
    private readonly IServiceProvider _serviceProvider;

    0 referências
    public ReservationUpdateService(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    0 referências
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using (var scope = _serviceProvider.CreateScope())
            {
                var reservationRepo = scope.ServiceProvider.GetRequiredService<ReservationRepo>();

                // Chame o método de atualização de reservas aqui
                await reservationRepo.UpdateStateReservationsByDates();

                // Intervalo entre as verificações
                await Task.Delay(TimeSpan.FromDays(1), stoppingToken); // Execução diária
                //await Task.Delay(TimeSpan.FromMinutes(2), stoppingToken);
            }
        }
    }
}
```

*Figura 42 - Serviço de Atualização automática de Reservas*

Este serviço chamado "ReservationUpdateService" irá executar diariamente e irá obter as reservas, e verificar o estado e as datas das mesmas, para proceder à sua atualização.

```
1 referência
public async Task UpdateStateReservationsByDates()
{
    var now = DateTime.UtcNow;

    var expiredReservations = await _context.Reservations
        .Include(r => r.ReservationStates)
        .Where(r => r.ReservationStates.id == 2 && r.end_date <= now)
        .ToListAsync();

    var completeState = await _context.ReservationStates
        .FirstOrDefaultAsync(rs => rs.id == 3);

    if (completeState == null)
    {
        // Caso o estado "Completa" não exista, pode criar um aqui
        completeState = new ReservationStates { state = "Completa" };
        await _context.ReservationStates.AddAsync(completeState);
        await _context.SaveChangesAsync();
    }

    foreach (var reservation in expiredReservations)
    {
        reservation.ReservationStates = completeState;
    }

    await _context.SaveChangesAsync();
}
```

*Figura 43 - Função para o Serviço de Atualizar as Reservas*

A função "UpdateStateReservationsByDates" que consta no Repositório das Reservas e que será utilizada pelo Serviço de atualização automático das Reservas, obtém as reservas da base de dados e verifica a sua data final, se a data final for superior à data atual e o estado da reserva for "Aprovada (USO)", o estado será alterado para "Completa", uma vez que a reserva já foi concluída.

Este serviço será executado no arquivo "program.cs" ao executar a API, através do código:

```
// Servico de atualizacao de reservas
builder.Services.AddSingleton<IHostedService, ReservationUpdateService>();
```

*Figura 44 - Código de Execução do Serviço*

# 16 - Conclusão

No decorrer deste projeto, tivemos a oportunidade de aplicar de forma prática os conhecimentos adquiridos ao longo de várias Unidades Curriculares, tanto no semestre atual quanto nos anteriores. Essa experiência foi fundamental para integrar e aplicar conceitos que abarcam não apenas a teoria, mas também na prática. Ao longo do desenvolvimento, utilizamos e consolidamos conhecimentos que se revelaram essenciais para ampliar nossas habilidades no contexto de criação de código, especialmente no âmbito do backend da aplicação. A construção de uma API em C# com .NET Core representou um desafio significativo, mas ao mesmo tempo uma oportunidade valiosa para expandir nosso conhecimento.

Em suma, este projeto foi mais do que a aplicação de conhecimentos teóricos, foi uma oportunidade para aprimorar habilidades práticas e desenvolver um entendimento mais profundo das nuances do desenvolvimento de backend de uma aplicação.