INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Licenciatura em Engenharia Informática

Trabalho Prático – Base de Dados para Site de Aluguer de Alojamentos Locais

Tierri Fernando de Coito Ferreira

2023

Resumo

O presente relatório foi realizado no âmbito do Trabalho Prático da Unidade Curricular de Bases de Dados II no curso de Engenharia Informática da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Beja. Pretende-se demonstrar o desenvolvimento de uma base de dados em *SQL Server* de uma plataforma exclusivamente para o aluguer de alojamentos locais em qualquer parte do mundo.

O Trabalho Prático realizado é constituído pela base de dados em si no formato de *full backup*, os *scripts* para a construção da mesma, um gerador de dados realizado em *Node.js* e o presente relatório.

Palavras-chave: aluguer de alojamentos locais, bases de dados, *SQL Server*, gerador de dados, *Node.js*, *T-SQL*

Abstract

This report has been made as an integral part of the Practical Project in the Databases II field for the Computer Science Degree in the School of Technology and Management of the Polytechnic Institute of Beja. It focuses on demonstrating the development of a SQL Server database of an online platform exclusively for the rental of housings in any part of the world.

The Practical Project contains the database itself, in full backup format, the scripts for its construction, a data generator written in Node.js and this report.

Keywords: housing rental, databases, SQL Server, data generator, Node.js, T-SQL

Índice

Lista de Figuras

1. Introdução

O presente relatório foi realizado no âmbito do Trabalho Prático, que consiste na realização de uma base de dados para o aluguer exclusivo de alojamentos locais.

Para a base de dados, foi utilizado o *SQL Server*, tendo como linguagem para os scripts o *Transact-SQL*, ou simplesmente *T-SQL*.

Em relação à inserção de dados, para a facilitação da visualização da base de dados da forma menos abstrata possível, foi desenvolvido um gerador de dados em *Node.js* [1] (que usa *JavaScript* como linguagem) para a geração de dados fictícios na base de dados. Em cada tabela, com a exceção das tabelas para moderação, foram inseridos 10.000 dados fictícios.

Este relatório encontra-se organizado da seguinte forma:

- Na secção 2, demonstra-se a estrutura da base de dados;
- Na secção 3, são enumerados os *stored procedures* e *triggers* criados e respetivamente demonstrados;
- A secção 4 é dedicada à segurança;
- Na secção 5 enumeram-se alguns casos possíveis de catástrofe e como a base de dados está preparada para os mesmos;
- Na secção 6 são demonstrados alguns casos de uso da base de dados com as respetivas consultas e funcionalidades da mesma;
- A secção 7 é dedicada à *performance* da base de dados;
- Na secção 8 demonstra-se a geração de dados no gerador.

2. Estrutura da Base de Dados

A base de dados é constituída por 20 tabelas. A Figura 2.1 representa a mesma num diagrama E-R (Entidade-Relação).

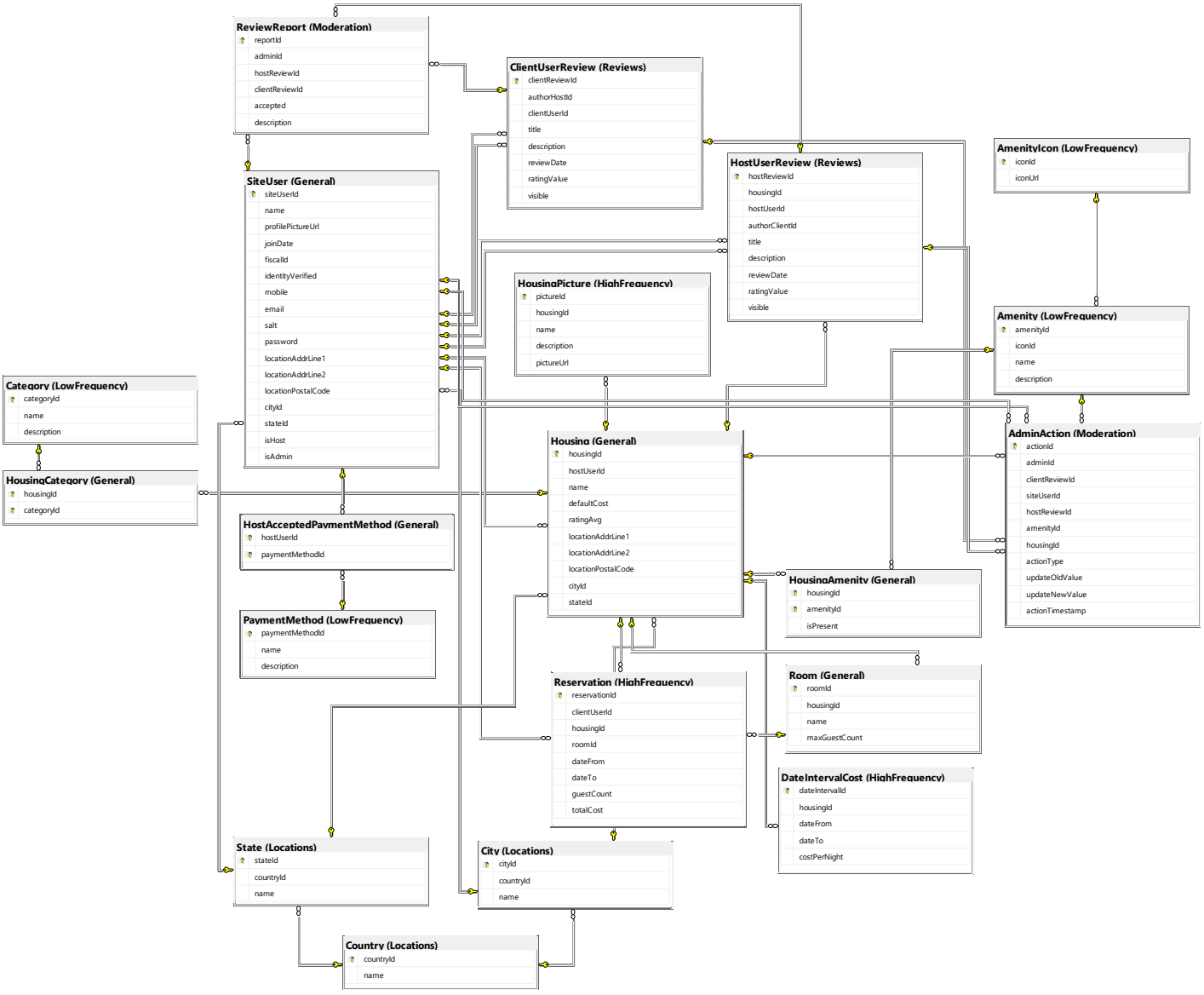

Figura 2.1 – Diagrama E-R da base de dados.

Existe uma divisão da base de dados por *schemas*, que coincidem com a localização das bases de dados no âmbito dos respetivos *Filegroups*. Os *schemas* criados são: *Locations, General, High Frequency, Low Frequency, Reviews* e *Moderation*.

As tabelas principais da base de dados são:

- *SiteUser,* que representa qualquer utilizador da plataforma, nomeadamente clientes (*clients*), anfitriões (*hosts*) e administradores (*admins*);
- *Housing*, que contém todos os alojamentos locais do Site.

Efetuar uma reserva implicará a criação de um registo na tabela *Reservations*, que é a responsável pelo relacionamento N-N entre as tabelas *SiteUser* e *Housing*, uma vez que, claramente, um cliente poderá fazer mais que uma reserva em diversos alojamentos locais.

Efetivamente, cada alojamento local poderá ter mais do que uma reserva a ocorrer ao mesmo tempo no caso de estar dividido por diferentes quartos (ou mesmo divisões independentes). Nesse caso, a tabela *Room* (quartos) será sempre usada em todas as reservas, e consequentemente em todos os alojamentos locais.

A nível aplicacional, para termos práticos, haverá uma componente que tratará da criação de "quartos" ou "divisões" automaticamente no caso de um alojamento local apenas acolher uma reserva de cada vez, tendo apenas um quarto ou divisão aplicável.

Para a criação de cada tabela os respetivos atributos são mencionados na sintaxe e os *scripts* completos podem ser encontrados no ficheiro ***sql/create_tables.sql***. Este ficheiro também contém a criação de todos os *schemas*, que devem ser criados antes das tabelas.

2.1. Demonstração de uma tabela (*SiteUser*)

A Figura 2.2 contém um exemplo de um *script* de criação de uma tabela da base de dados, neste caso de *SiteUser*.

```
CREATE TABLE General.SiteUser (
    siteUserId BIGINT IDENTITY CONSTRAINT PK_SiteUser PRIMARY KEY,
    [name] NVARCHAR(64) NOT NULL,
    profilePictureUrl NVARCHAR(256),
    joinDate DATETIME NOT NULL DEFAULT GETDATE(),
    fiscalId BIGINT NOT NULL DEFAULT 999999990,
    identityVerified BIT NOT NULL DEFAULT 0,
    mobile NVARCHAR(32) NOT NULL,
    email NVARCHAR(64) NOT NULL,
    salt NVARCHAR(128) NOT NULL,
    [password] NVARCHAR(512) NOT NULL,
    locationAddrLine1 NVARCHAR(64) NOT NULL,
    locationAddrLine2 NVARCHAR(64),
    locationPostalCode NVARCHAR(16) NOT NULL,
    cityId INT NOT NULL,
    stateId INT,
    isHost BIT NOT NULL DEFAULT 0,
    isAdmin BIT NOT NULL DEFAULT 0,

    FOREIGN KEY (cityId) REFERENCES Locations.City(cityId),
    FOREIGN KEY (stateId) REFERENCES Locations.State(stateId)
) ON tg1_General
```

Figura 2.2 – Exemplo de *script* de criação de tabela para *SiteUser*.

Como podemos observar na Figura 2.2, *siteUserId* é a chave primária da tabela.

No atributo *joinDate*, aquando de cada inserção, normalmente não é necessário mencionar o seu valor, uma vez que o valor por defeito definido é a data atual (ou seja, a data precisamente do momento em que a inserção está a ser efetuada à base de dados).

Em relação ao NIF (*fiscalId*), no caso de o mesmo não ser definido, deve ser o NIF de consumidor final, que tem como valor "999999990". Assim, o valor na base de dados nunca é nulo.

No que toca ao atributo *identityVerified* (identidade verificada), este apenas se aplica a *hosts* (anfitriões). Este atributo tem como significado se o anfitrião verificou a sua identidade com um administrador da plataforma, para que possa haver confiança entre clientes e anfitriões no site. Este atributo deve ser sempre 0 (*false*) durante a criação de cada utilizador, mesmo que seja anfitrião, e a verificação deve ser feita estritamente pelo *stored procedure* respetivo, que pode ser observado na secção 3.

Para que haja segurança na autenticação de todos os utilizadores e no caso de ataque à aplicação ou mesmo base de dados, as palavras-passe são todas guardadas em *hash*, o *NVARCHAR* tendo consequentemente como tamanho 512 bytes. Isto significa que primeiro, é impossível reverter o valor de um *hash* no valor original, portanto se, em caso catastrófico de ataque, um atacante tivesse acesso às palavras-passe, não teria o valor original das mesmas, e segundo, todos os valores destas palavras-passe têm o mesmo tamanho, porque um *hash* gera sempre uma *string* de igual tamanho, dependendo do algoritmo.

Para acrescer à segurança destas palavras passe, ainda existe o atributo *salt*, que traduz-se diretamente a "sal". Em termos informais, pode-se descrever este atributo como um "condimento" da palavra-passe, que tem como intuito a não-criação de um *hash* diretamente a partir da palavra-passe original. Isto porque, hipoteticamente, se um utilizador utilizar a mesma palavra-passe em vários sites e a mesma fosse comprometida num, se os mesmos atacantes conseguissem encontrar o *hash* da palavra-passe guardado nesta base de dados e tiver o mesmo valor que o *hash* cujo valor original foi comprometido, os atacantes saberiam que a mesma palavra-passe foi utilizada nesta plataforma. Assim, ao guardar o valor da palavra-passe do utilizador, este será acrescido do "sal", um valor de 128 bytes aleatórios gerado durante a criação do mesmo utilizador.

Por exemplo, se a palavra-passe de um utilizador for "p4ssw0rd", o mesmo será criado no nível aplicacional, na base de dados será enviado o "sal" gerado naquele momento (por exemplo, "3dac5948128bdae60bf8c26aa209a6f16857d636fab385d335b9862fb2e36a1d885ab87698561bf959718444f968ce48604dff07510250f5a06518aeb70d26f5") e o valor de *hash* será o resultado do que o algoritmo processará, que tem como *input* esta *string:* "3dac5948128bdae60bf8c26aa209a6f16857d636fab385d335b9862fb2e36a1d885ab87698561bf959718444f968ce48604dff07510250f5a06518aeb70d26f5p4ssw0rd". Note-se no final da mesma que a palavra-passe original é concatenada a este "sal". O resultado do *hash* em SHA-256 é o seguinte:
"5596ed7be2756cc84d34f62af11582850067312bea84b37481382575b3b9b730". Este valor será o valor final guardado no atributo *password*.

2.2. Moradas e Localizações

Em relação aos valores relacionados a moradas e localizações de utilizadores e alojamentos locais, os mesmos encontram-se repartidos nas tabelas que pertencem ao *schema Locations*, e nas tabelas *Housing* e *SiteUser*.

As tabelas que pertencem ao mesmo *schema* são *Country* (país), *State* (estado) e *City* (cidade). Em relação aos outros atributos em moradas, os mesmos são iguais nas tabelas *Housing* e *SiteUser* e são *locationAddrLine1, locationAddrLine2* e *locationPostalCode*. Estes atributos são normalmente únicos, e se não o são, raramente se repetem em casos extraordinários. Devido a esse facto, não pertencem a nenhuma tabela com esse intuito, e são guardados em *VARCHAR*. Em relação ao país, estado e cidade, estes são guardados em chave estrangeira. O país é guardado nas tabelas de estado e cidade, uma vez que o país nunca muda a partir do momento que o utilizador/alojamento local se encontra(m) numa certa cidade ou estado.

A Figura 2.3 demonstra a criação destas tabelas.

```
CREATE TABLE Locations.Country (
    countryId INT IDENTITY CONSTRAINT PK_Country PRIMARY KEY,
    [name] NVARCHAR(64) NOT NULL
) ON tg1_Locations

CREATE TABLE Locations.State (
    stateId INT IDENTITY CONSTRAINT PK_State PRIMARY KEY,
    countryId INT NOT NULL,
    [name] NVARCHAR(64) NOT NULL,

    FOREIGN KEY (countryId) REFERENCES Locations.Country(countryId)
) ON tg1_Locations

CREATE TABLE Locations.City (
    cityId INT IDENTITY CONSTRAINT PK_City PRIMARY KEY,
    countryId INT NOT NULL,
    [name] NVARCHAR(64) NOT NULL,

    FOREIGN KEY (countryId) REFERENCES Locations.Country(countryId)
) ON tg1_Locations
```

Figura 2.3 – Criação das tabelas do *schema Locations*.

2.3. Funcionamento de *AdminAction*

A tabela *AdminAction* no *schema Moderation,* como referido anteriormente, representa os *logs* das ações feitas por administradores da plataforma. A estrutura da tabela com o respetivo código de criação da mesma pode ser observada na Figura 2.4.

```
CREATE TABLE Moderation.AdminAction (
    actionId BIGINT IDENTITY CONSTRAINT PK_AdminAction PRIMARY KEY,
    adminId BIGINT NOT NULL,
    clientReviewId BIGINT,
    siteUserId BIGINT,
    hostReviewId BIGINT,
    amenityId INT,
    housingId BIGINT,
    actionType BIT NOT NULL, -- 0: update, 1: delete
    updateOldValue NVARCHAR(512),
    updateNewValue NVARCHAR(512),
    actionTimestamp DATETIME NOT NULL DEFAULT GETDATE(),
    FOREIGN KEY (adminId) REFERENCES General.SiteUser(siteUserId),
    FOREIGN KEY (clientReviewId) REFERENCES Reviews.ClientUserReview(clientReviewId),
    FOREIGN KEY (siteUserId) REFERENCES General.SiteUser(siteUserId),
    FOREIGN KEY (hostReviewId) REFERENCES Reviews.HostUserReview(hostReviewId),
    FOREIGN KEY (amenityId) REFERENCES LowFrequency.Amenity(amenityId),
    FOREIGN KEY (housingId) REFERENCES General.Housing(housingId)
) ON tg1_Moderation
```

Figura 2.4 – Criação da tabela *AdminAction*.

As chaves estrangeiras (exceto *adminId*) têm como referência todas as tabelas em que é possível guardar um *log* de uma certa ação.

Ao guardar um *log*, apenas uma chave estrangeira (se aplicável) deve ser preenchida. O resto dos atributos deve estar nulo (NULL). Estas ações são enviadas a nível aplicacional uma vez que guardar o ID do administrador sempre que há uma alteração em qualquer tabela será impossível sem fazer o mesmo parte de todas estas tabelas, o que não é prático de todo.

Quando ocorre uma ação, esta pode ser uma atualização (edição) ou uma eliminação. Se é uma atualização, o código será 0, portanto o valor de *actionType* será o mesmo. Caso contrário, será 1. No caso anterior, o valor antigo será guardado em *updateOldValue* e o novo em *updateNewValue*. Apesar de valores muitas vezes não serem *strings* (VARCHARs), estas ações são meramente uma fonte de informação interna que apenas serve para consulta de alguma alteração que tenha sido feita por um administrador da plataforma.

Dependendo da forma como o nível aplicacional é feito, poderá ser adicionado um atributo *observations* do tipo VARCHAR que conterá ou o nome do atributo que for alterado (se necessário), ou até alguma observação do administrador. No entanto, como isto depende muito da forma como o nível aplicacional é implementado, será deixado para essa fase.

2.4. Filegroups das Tabelas

A base de dados está dividida por 6 *schemas* com os respetivos *filegroups*. Apesar de o *SQL Server* não suportar a definição de *filegroups* especificamente para cada *schema*, a criação das tabelas respeitou a estrutura de *schemas* de modo que cada *filegroup* corresponda aos mesmos.

A secção 5 abordará com mais detalhe a estrutura física da base de dados em relação a discos e a metodologia RAID usada.

O *filegroup* “tg1_Primary” é o *filegroup* principal e não se encontra dividido.

Em relação aos *filegroups* de *schemas*, o nome dos mesmos é dado pelo prefixo "tg1_", seguido do nome do *schema*, como por exemplo, "tg1_General". O sufixo é depois "_*n*", *n* sendo o número do *filegroup*, se este estiver dividido em mais que uma parte. Nos casos dos *filegroups* que não necessitam de divisão, o sufixo será "_Main", como em "tg1_Locations_Main".

O *filegroup* do *schema General* contém as tabelas principais da base de dados. Estas têm muito mais leitura que escrita, e, portanto, estão separadas do *schema High Frequency*, que como o nome indica, é o *schema* de alta frequência de escritas.

Classificações (*Reviews*) também têm um número elevado de leituras e escritas, e são então colocados no seu próprio *schema*.

O *schema Low Frequency* tem, pelo contrário, um número muito menos elevado de escritas, apesar de ter muitas leituras, sendo para dados que raramente se alteram, como as comodidades, os métodos de pagamento, entre outros.

Por fim, o *schema* de moderação (*Moderation*) contém as tabelas de denúncias de classificações e de *logs* de ações de um administrador.

A secção 6 demonstrará com mais detalhe os índices criados para melhorar a performance de leitura ou manter equilíbrio entre leitura/escrita.

3. *Stored Procedures* e *Triggers*

Nesta base de dados, foram criados 7 *triggers* e 4 *procedures*.

Os *procedures*, de modo geral, têm o objetivo de tornar certas operações menos complexas, para que não haja a necessidade de repetir o mesmo raciocínio para as mesmas.

Em relação aos *triggers*, estes fazem parte do funcionamento da base de dados e são cruciais para o funcionamento da mesma, assim como o próprio funcionamento da plataforma a nível aplicacional. Por exemplo, para saber o valor da média do valor das classificações de alojamentos locais, em vez de calculá-la sempre que um alojamento é consultado, este terá como atributo a média em si, que será atualizada sempre que uma classificação é feita, alterada ou até removida.

3.1. Stored Procedures

3.1.1. Verificar anfitrião

No *procedure verifyHostUser*, temos o intuito de tornar um *SiteUser* verificado. No entanto, isto apenas é aplicável a anfitriões, portanto precisamos de fazer duas verificações: se o *user* existe, e se é anfitrião. Podemos observar o código que implementa este *procedure* na Figura 3.1.

```
CREATE PROCEDURE verifyHostUser
    @hostUserId BIGINT
AS BEGIN
    IF NOT EXISTS (SELECT * FROM General.SiteUser WHERE siteUserId = @hostUserId) BEGIN
        RAISERROR ('This user does not exist.', 16, 1)
        RETURN
    END ELSE IF (SELECT isHost FROM General.SiteUser WHERE siteUserId = @hostUserId) = 0 BEGIN
        RAISERROR ('Verification only applies to host users - the selected user is not a host.', 16, 1)
        RETURN
    END

    UPDATE General.SiteUser SET identityVerified = 1 WHERE siteUserId = @hostUserId
END
GO
```

Figura 3.1 – Implementação de *verifyHostUser*.

Erros são lançados quando qualquer uma das condições é falsa. Se for utilizador e anfitrião, então a verificação é efetuada.

3.1.2. Encontrar quarto para certa quantidade de pessoas

Este *stored procedure* encontra um quarto num dado alojamento local, para uma certa quantidade de pessoas. A forma como o cálculo é efetuado é encontrar o quarto com menos capacidade possível para esta quantidade de pessoas. Assim, não é selecionado um quarto demasiado grande, e tem a capacidade para este número simultaneamente.

O código pode ser observado na Figura 3.2.

```
CREATE PROCEDURE findRoomForGuestAmount
    @housingId BIGINT,
    @amount TINYINT,
    @roomId BIGINT OUT
AS BEGIN
    IF NOT EXISTS (SELECT * FROM General.Housing WHERE housingId = @housingId) BEGIN
        RAISERROR ('This housing does not exist.', 16, 1)
        RETURN
    END

    -- Obter o quarto com a menor capacidade que seja maior ou igual à quantidade de hóspedes.
    SELECT TOP 1 @roomId = roomId FROM General.Room WHERE housingId = @housingId AND maxGuestCount >= @amount
        ORDER BY maxGuestCount ASC
END
GO
```

Figura 3.2 – Implementação de *findRoomForGuestAmount*.

Inicialmente, como é óbvio, precisamos de ter a certeza que o alojamento local existe. Depois, é feita uma consulta de todos os quartos deste alojamento local onde o número de pessoas é menor ou igual à capacidade máxima do quarto. Idealmente, será igual, mas se não houver um quarto com essa capacidade, será um quarto maior. No pior dos cenários, o *procedure* retornará um valor nulo, por não encontrar nenhum resultado nesta consulta.

A forma como seleciona-se a opção com a menor capacidade possível é através da ordenação pela mesma e obtenção do primeiro registo encontrado, uma vez que a ordenação é feita ascendentemente, tendo o menor no primeiro registo do *result set*.

3.1.3. Calcular o preço para certas datas

Para saber o preço de cada noite num alojamento local, não basta obter o preço normal de cada noite, mas sim verificar também se a mesma data está inserida num intervalo de época com preços diferentes, na tabela *DateIntervalCost*. A figura 3.3 mostra a implementação deste *procedure*.

```
CREATE PROCEDURE calculatePriceForDates
    @housingId BIGINT,
    @startDate DATETIME,
    @endDate DATETIME,
    @roomId BIGINT,
    @guestCount TINYINT,

    @finalPrice MONEY OUT
AS BEGIN
    IF @startDate > @endDate BEGIN
        RAISERROR ('Start date must be before end date.', 16, 1)
        RETURN
    END

    DECLARE @toReturn MONEY = 0
    -- Obter preço por noite com base na tabela DateIntervalCost, iterando por datas.
    WHILE @startDate <= @endDate BEGIN
        IF EXISTS (SELECT * FROM HighFrequency.DateIntervalCost WHERE @startDate
                BETWEEN dateFrom AND dateTo AND housingId = @housingId) BEGIN
            -- Somar ao preço final o preço da data atual.
            SELECT @toReturn = @toReturn + MAX(costPerNight) FROM HighFrequency.DateIntervalCost
                WHERE @startDate BETWEEN dateFrom AND dateTo AND housingId = @housingId
        END ELSE BEGIN
            SELECT @toReturn = @toReturn + defaultCost FROM General.Housing WHERE housingId = @housingId
        END
        SET @startDate = DATEADD(DAY, 1, @startDate)
    END

    -- Retornar.
    SET @finalPrice = @toReturn * @guestCount
END
GO
```

Figura 3.3 – Implementação de *calculatePriceForDates*.

Os parâmetros passados são o ID do alojamento local, a data de início, a data de fim (inclusive), o ID do quarto (após seleção do mesmo usando *findRoomForGuestAmount*) e o número de hóspedes.

Primeiro, obviamente a data de fim deve ser após a data de início e isso é verificado.

Depois, itera-se o intervalo de datas e soma-se ao custo final o preço de cada noite, verificando se existe preço especial ou não em cada noite. Se, por alguma razão, houver colisão entre intervalos de preços especiais, o preço mais alto é o considerado, daí ao uso de MAX() na consulta a *DateIntervalCost*.

Usa-se a função *DATEADD()* para adicionar um dia (*DAY*) a *@startDate*, que está a ser iterada até ser igual a *@endDate.*

3.1.4. Obter capacidade máxima de um alojamento local em todos os quartos

Para obter a capacidade máxima em todos os quartos, apenas basta fazer a consulta de todos os quartos no alojamento local desejado, usando a função MAX() no atributo *maxGuestCount* do quarto. A Figura 3.4 representa esta consulta.

```
CREATE PROCEDURE maxGuestCount
    @housingId BIGINT,
    @maxGuestCount TINYINT OUT
AS BEGIN
    IF NOT EXISTS (SELECT * FROM General.Housing WHERE housingId = @housingId) BEGIN
        RAISERROR ('This housing does not exist.', 16, 1)
        RETURN
    END

    SELECT @maxGuestCount = MAX(maxGuestCount) FROM General.Room WHERE housingId = @housingId
END
GO
```

Figura 3.4 – Implementação de *maxGuestCount*.

3.2. *Triggers*

3.2.1. Atualizar classificação média

O *triggers* responsável pela atualização da classificação média é executado quando um registo de *HostUserReview* é atualizado, inserido ou removido. Sempre que isto acontece, *ratingAvg* em *Housing* é atualizado com o valor da nova média de *ratingValue* em todos os *reviews* que estão direcionados a este alojamento local. A Figura 3.5 contém a implementação deste *trigger.*

```
CREATE TRIGGER updateRating ON Reviews.HostUserReview AFTER INSERT, UPDATE, DELETE AS BEGIN
    UPDATE General.Housing SET ratingAvg = (SELECT AVG(CAST(ratingValue as FLOAT)) from Reviews.HostUserReview)
        WHERE General.Housing.housingId = (SELECT housingId FROM inserted)
END
GO
```

Figura 3.5 – Implementação de *updateRating*.

3.2.2. Verificação de ações de administradores

Quando é criada uma ação de administradores, pelo menos uma tabela deve ser referenciada na alteração, porque senão esta ação não tem conteúdo e não faz sentido. Assim, é verificado se nos valores inseridos pelo menos um valor das chaves estrangeiras aplicáveis não é nulo. A Figura 3.6 mostra como o mesmo é implementado.

```
CREATE TRIGGER verifyAdminAction ON Moderation.AdminAction AFTER INSERT, UPDATE AS BEGIN
    IF (SELECT isAdmin FROM General.SiteUser WHERE siteUserId = (SELECT adminId FROM inserted)) = 0 BEGIN
        RAISERROR ('Admin actions can''t be executed by non-admins!', 16, 1)
        ROLLBACK
        RETURN
    END

    -- Verificar que pelo menos uma das chaves estrangeiras n@o @ nula
    DECLARE @clientReviewId BIGINT,
            @siteUserId BIGINT,
            @hostReviewId BIGINT,
            @amenityId INT,
            @housingId BIGINT

    SELECT @clientReviewId = clientReviewId,
           @siteUserId = siteUserId,
           @hostReviewId = hostReviewId,
           @amenityId = amenityId,
           @housingId = housingId FROM inserted

    IF @clientReviewId IS NULL AND @siteUserId IS NULL
        AND @hostReviewId IS NULL AND @amenityId IS NULL
        AND @housingId IS NULL BEGIN
        RAISERROR ('Admin actions must at least reference one entity!', 16, 1)
        ROLLBACK
    END
END
GO
```

Figura 3.6 – Implementação de *verifyAdminAction*.

3.2.3. Atualizar visibilidade de uma classificação quando denúncia é aceite e verificação de denúncias

Quando é feita uma denúncia, a mesma tem de ser aceite por um administrador. Assim, quando esta denúncia é aceite, é prático que a base de dados já faça com que o que foi denunciado deixe de ser visível, uma vez que é o que sempre acontece. A Figura 3.7 demonstra como o mesmo é feito.

```
CREATE TRIGGER updateVisibilityOnReportAccepted ON Moderation.ReviewReport AFTER UPDATE AS BEGIN
    IF (SELECT accepted FROM inserted) = 1 AND (SELECT accepted FROM deleted) IS NULL BEGIN
        IF (SELECT adminId FROM inserted) IS NULL BEGIN
            RAISERROR ('You must supply an admin who accepted the report.', 16, 1)
            ROLLBACK
            RETURN
        END

        IF (SELECT clientReviewId FROM inserted) IS NOT NULL BEGIN
            UPDATE Reviews.ClientUserReview SET visible = 0 WHERE clientReviewId = (SELECT clientReviewId FROM inserted)
            UPDATE Moderation.ReviewReport SET accepted = 1, adminId = (SELECT adminId FROM inserted)
                WHERE clientReviewId = (SELECT clientReviewId FROM inserted)
        END ELSE IF (SELECT hostReviewId FROM inserted) IS NOT NULL BEGIN
            UPDATE Reviews.HostUserReview SET visible = 0 WHERE hostReviewId = (SELECT hostReviewId FROM inserted)
            UPDATE Moderation.ReviewReport SET accepted = 1, adminId = (SELECT adminId FROM inserted)
                WHERE hostReviewId = (SELECT hostReviewId FROM inserted)
        END
    END
END
GO
```

Figura 3.7 – Implementação de *updateVisibilityOnReportAccepted*.

Podemos observar que esta atualização é feita tanto para classificações de clientes como de anfitriões. A verificação é feita e o aplicável é atualizado. Também, para quaisquer outras denúncias feitas ao mesmo comentário, as mesmas serão aceites automaticamente se esta denúncia também o for.

Outro *trigger* relacionado com o anterior é *verifyReport*, que apenas serve para validações básicas de uma denúncia, como estados ilícitos dos atributos da tabela *ReviewReport*.

3.2.4. Prevenção de eliminação de ações de administradores e de verificação manual de anfitriões

Estes *triggers*, denominados respetivamente de *denyAdminActionDelete* e *denyVerifyUserManually*, servem para prevenir a alteração dos registos destas tabelas de forma ilícita, uma vez que ambas estas tabelas têm uma forma específica de representação de dados que não é tão simples de modo a que seja suficiente a estrutura normal de uma tabela em SQL.

A prevenção da eliminação de ações de administradores é aplicável a qualquer utilizador que não seja "*server*", ou seja, qualquer utilizador que não seja o servidor em si (que é bloqueado apenas ao servidor de produção ou ao gerador de dados). Isto para que administradores não consigam, caso algum tenha acesso à base de dados, eliminar *logs* de algo que tenham feito e não queiram mostrar, para prevenir abuso.

A verificação de utilizadores deve sempre ser feita pelo *stored procedure*, e caso contrário, o *trigger* irá prevenir esta alteração.

3.2.5. Cálculo automático de preço na criação de uma reserva

Este *trigger*, denominado de *autoPriceSet*, é talvez o *trigger* mais importante desta base de dados, e responsável pelo cálculo automático do custo total de uma reserva para todos os hóspedes e durante o intervalo todo. Este cálculo é feito pelo *procedure* mencionado acima em **3.1.3**, *calculatePriceForDates*. A Figura 3.8 mostra a forma como este *trigger* foi implementado.

```
CREATE TRIGGER autoPriceSet ON HighFrequency.Reservation AFTER INSERT AS BEGIN
    IF (SELECT totalCost FROM inserted) IS NOT NULL BEGIN
        RAISERROR ('Total cost must be NULL when inserting a reservation.', 16, 1)
        ROLLBACK
        RETURN
    END

    DECLARE @housingId BIGINT,
            @startDate DATETIME,
            @endDate DATETIME,
            @roomId BIGINT,
            @guestCount TINYINT,
            @totalCost MONEY
    SELECT @housingId = housingId FROM inserted
    SELECT @startDate = dateFrom FROM inserted
    SELECT @endDate = dateTo FROM inserted
    SELECT @roomId = roomId FROM inserted
    SELECT @guestCount = guestCount FROM inserted
    EXEC calculatePriceForDates @housingId, @startDate, @endDate, @roomId, @guestCount, @totalCost OUT

    UPDATE Reservation SET totalCost = @totalCost
        WHERE reservationId = (SELECT reservationId FROM inserted)
END
```

Figura 3.8 – Implementação de *calculatePriceForDates*.

4. Segurança

No que toca à segurança desta base de dados, em vários aspetos a mesma é atingida. A mesma é feita através de utilizadores da base de dados, e outros níveis que não estão diretamente ligados ao *SQL Server* mas que são importantes também.

4.1. Utilizadores da base de dados

Nesta base de dados, existem dois utilizadores. *server*, que é o utilizador do servidor, que está bloqueado ao endereço IP do servidor em si (ver secção 4.2) e *operator*, que é a conta ligeiramente mais limitada para operadores da base de dados em que não há acesso às tabelas da moderação e *dbo*.

Na Figura 4.1 podemos observar os comandos de criação destes utilizadores.

```
-- server é o utilizador que vai ser usado ao nível aplicacional.
CREATE LOGIN [server] WITH PASSWORD = 'server', CHECK_POLICY = OFF
CREATE USER [server] FROM LOGIN [server]
GRANT CONTROL ON DATABASE::tg1 TO [server]
ALTER SERVER ROLE [sysadmin] ADD MEMBER [server]

-- operator é o utilizador para operações de manutenção da base de dados.
CREATE LOGIN [operator] WITH PASSWORD = '0p3r4t0r!'
CREATE USER [operator] FROM LOGIN [operator]
```

Figura 4.1 – Criação dos utilizadores.

No utilizador *server*, *CHECK_POLICY* apenas é desligado para que a palavra-passe permaneça simples neste contexto. No entanto, em produção, é óbvio que será melhor ter uma palavra-passe mais segura. De qualquer modo, este utilizador apenas pode ser acedido pelo utilizador.

São dadas permissões máximas no que toca ao controlo da base de dados *tg1*, que é a base de dados deste projeto. O *role sysadmin* também é dado a este utilizador, para dar permissões administrativas ao mesmo.

O operador, pelo contrário, é criado com alguns limites nas suas permissões. A Figura 4.2 inclui as restrições e permissões dadas a este utilizador.

```
DENY CONTROL ON SCHEMA::dbo TO [operator]
GRANT CONTROL ON SCHEMA::General TO [operator]
GRANT CONTROL ON SCHEMA::HighFrequency TO [operator]
GRANT CONTROL ON SCHEMA::Locations TO [operator]
GRANT CONTROL ON SCHEMA::LowFrequency TO [operator]
GRANT CONTROL ON SCHEMA::Reviews TO [operator]
DENY CONTROL ON SCHEMA::Moderation TO [operator]
GO
```

Figura 4.2 – Permissões dos operadores.

Como referido anteriormente, os *schemas Moderation* e *dbo* são restringidos dos operadores. Os restantes *schemas* são, no entanto, permitidos com permissões máximas.

4.2. *Triggers* aplicados à segurança

Para aumentar a segurança, no *user server* foi aplicada uma regra adicional de restrição de IPs. A Figura 4.3 demonstra como este *trigger* foi implementado.

```
CREATE TRIGGER blockServerLogins ON ALL SERVER FOR LOGON AS BEGIN
    -- Alterar 127.0.0.1 para IP aplicável do servidor que acede à base de dados.
    IF ORIGINAL_LOGIN() = 'server' AND (SELECT client_net_address FROM sys.dm_exec_connections
            WHERE session_id = @@SPID) != '127.0.0.1' BEGIN
        ROLLBACK
    END
END
GO
```

Figura 4.3 – Implementação do *trigger* de *LOGON blockServerLogins*

Este *trigger* é especial uma vez que nem é do tipo DML (o tipo de todos os restantes *triggers* desta base de dados) nem DDL, mas sim um *trigger* de *LOGON*. Este *trigger* executa sempre que um utilizador faz *login* na base de dados, e é possível preveni-lo com *ROLLBACK*, tal como num trigger normal.

Neste *trigger*, usa-se ORIGINAL_LOGIN(), que verifica qual o utilizador atual, e o IP atual, que se encontra na tabela de sistema *dm_exec_connections*, que guarda as conexões para cada *session id*, que pode ser obtido através da variável global @@SPID. O IP utilizado neste caso é 127.0.0.1, que é o mesmo que *localhost*, mas em produção deverá ser mudado para o IP real do servidor (que terá de ser estático). Se o IP não for igual, a sessão será cancelada, e o *login* não será sucedido.

Outros *triggers* usados também complementam para a segurança da base de dados, mas os mesmos já foram descritos na secção anterior.

4.3. Outras medidas de segurança

A segurança não deve ser somente feita a nível da base de dados. Deve ser feita a todos os níveis em que a base de dados é chamada, incluindo no próprio servidor onde está alojada.

Começa por ter sempre uma *firewall* configurada. Apenas devem conseguir conectar-se à base de dados os operadores ou o servidor. Assim, a *firewall* terá todos os endereços de IP das redes que podem-se conectar à base de dados. Esta *firewall* poderá ser alterada por acesso remoto ao servidor, por SSH, no caso de haver a necessidade de conectar-se à base de dados a partir de um IP dinâmico.

Outra alternativa é tornar a *firewall* mais flexível, permitindo uma região em vez de um endereço IP específico. Mas essa parte será mais relevante para o administrador do sistema.

Existem mais medidas muito importantes a nível aplicacional. Uma delas é tratar dos comandos executados a nível aplicacional no caso de haver *input* do utilizador final, para prevenir um *exploit* denominado por *SQL Injection*, que consiste na alteração do comando enviado por código SQL malicioso. Isto previne-se através do tratamento do comando através de comandos preparados (*Prepared Statements*). Estes podem ser observados na secção 8, quando efetuarmos a ligação à base de dados no gerador.

5. Casos de Catástrofe, Resolução e Backups

Numa base de dados, em produção, há quase uma obrigação de haver uma alternativa e uma resposta a qualquer problema que possa ocorrer. Por isso, devem-se fazer backups e manter a base de dados organizada em mais que um disco para que quando haja a ocorrência da perda de um disco, outro terá os mesmos dados em paralelo, sem haver nenhuma perda.

5.1. Backups

Para os backups, esta base de dados fará a cada 15 minutos um backup de *transaction log* [2], todos os dias um backup diferencial e semanalmente um *full* backup.

No *SQL Server Management Studio*, o mesmo pode ser feito através do *Maintenance Plan Wizard*, que serve para criar um plano de manutenção, neste caso para backups, com a possibilidade de fazê-lo de X em X tempo. Neste caso, serão feitos três planos, uma vez que todos terão execuções diferentes em termos de intervalo de tempo.

Antes disso, é necessário ativar este sistema. Para tal, os seguintes comandos [3] têm de ser usados, como demonstra a Figura 5.1.

```
EXEC SP_CONFIGURE 'SHOW ADVANCE', 1
GO
RECONFIGURE WITH OVERRIDE
GO
EXEC SP_CONFIGURE 'AGENT XPs', 1
GO
RECONFIGURE WITH OVERRIDE
GO
```

Figura 5.1 – Comandos a executar para ativar planos de manutenção.

As Figuras 5.2, 5.3 e 5.4 mostram como o mesmo pode ser feito. O que for alterado está destacado a vermelho.

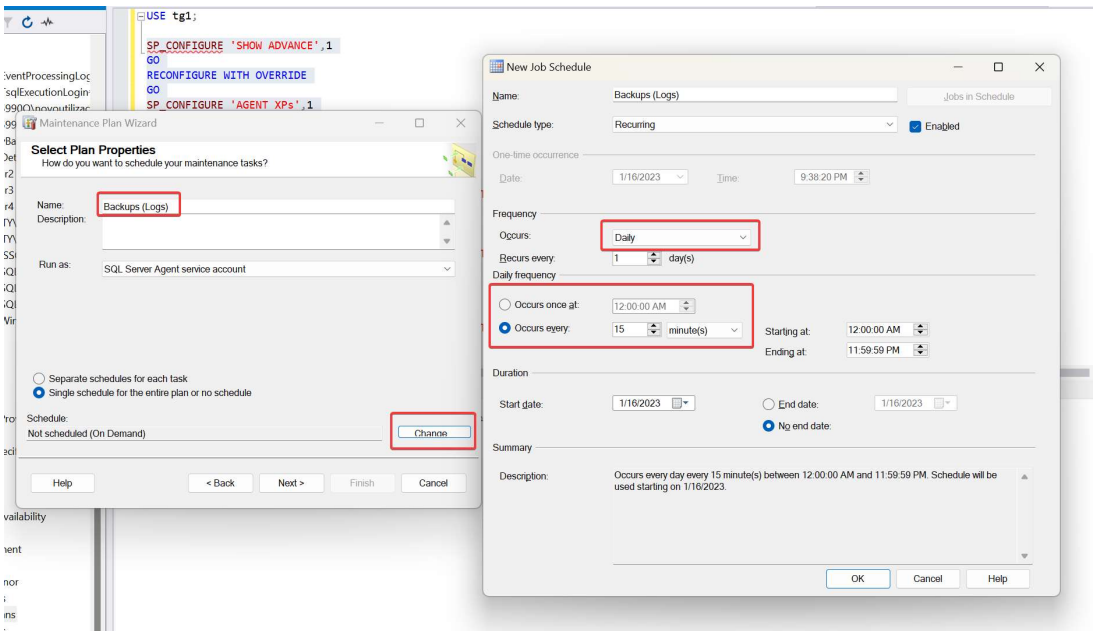

Figura 5.2 – Invocação do gestor de manutenção e planeamento do backup de *logs*.

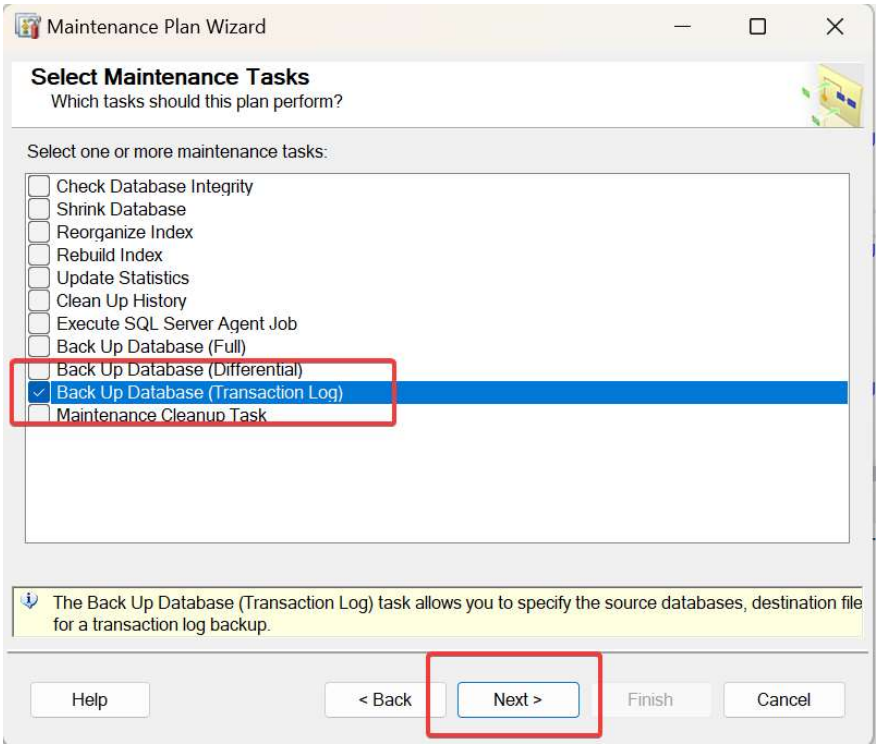

Figura 5.3 – Seleção do tipo de ação (Backup da base de dados, *Logs*).

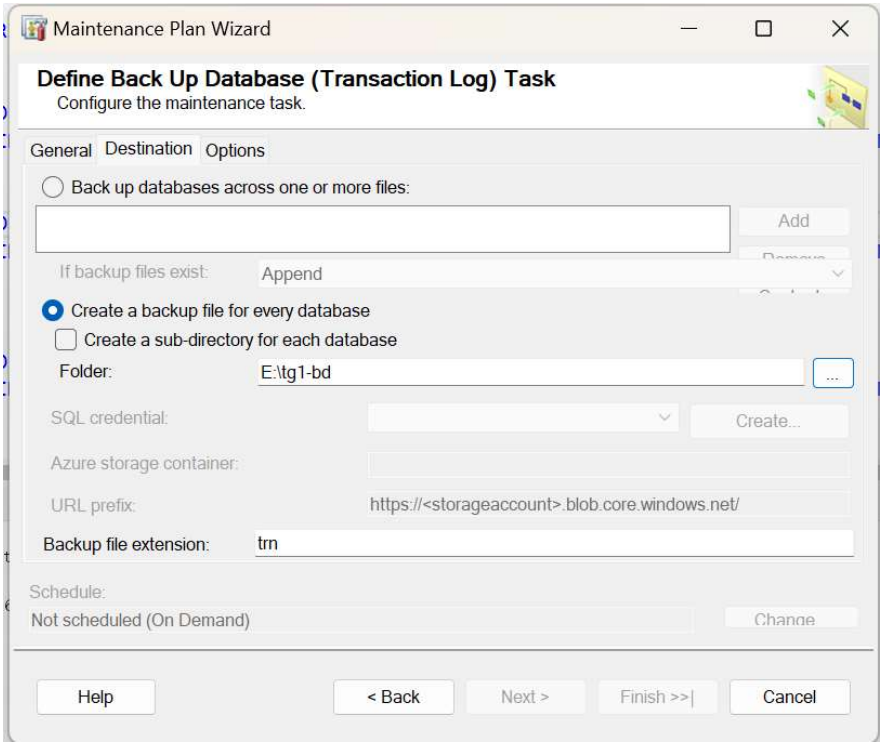

Figura 5.4 – Seleção do destino do backup.

Após a seleção da base de dados e selecionar "*Finish*", a janela demonstrada na Figura 5.5 aparecerá.

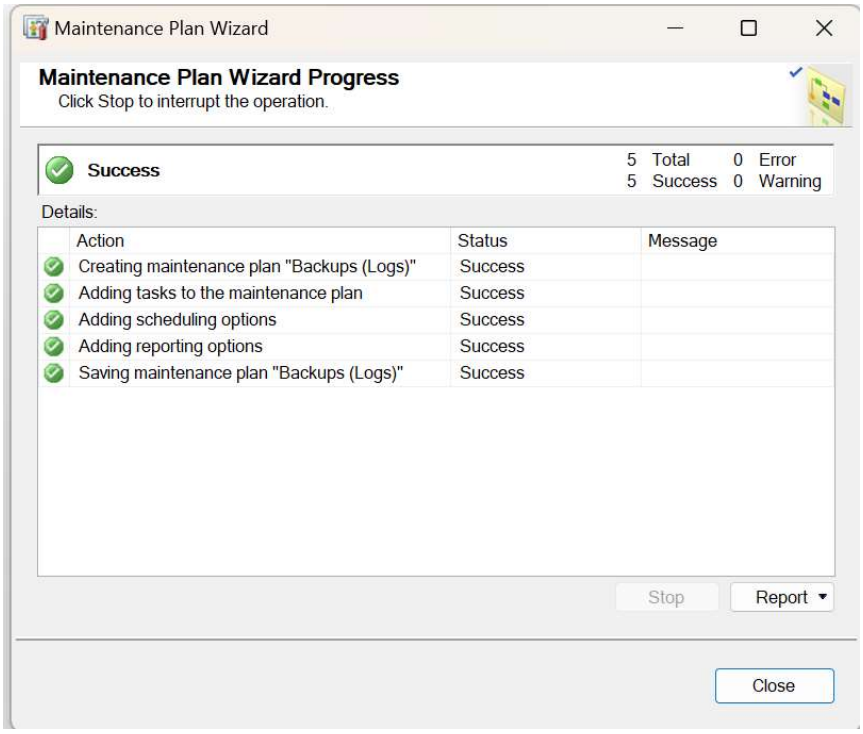

Figura 5.5 – Plano criado com sucesso.

O mesmo deverá ser repetido com os restantes tipos de backup, com as alterações respetivas, como o intervalo de tempo de execução e o tipo de backup.

Após a criação dos mesmos, a diretoria *Maintenance Plans* terá o aspeto que pode ser observado na Figura 5.6.

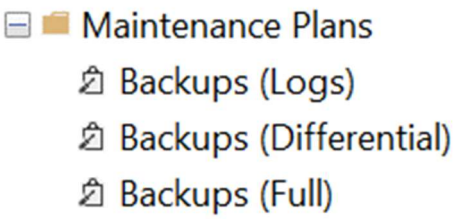

Figura 5.6 – Conteúdo da diretoria *Maintenance Plans*.

5.2. Estrutura física da base de dados

Obviamente, *backups* não são suficientes. É necessário haver mais robustez na estrutura da base de dados, tendo a necessidade de dividir a mesma em vários discos, assim como a espelhagem dos mesmos.

Por isso, em produção, os ficheiros dos *filegroups* da base de dados devem estar organizados em discos espelhados por RAID 10. Os filegroups serão distribuídos da seguinte forma:

- **Discos A**: tg1_Primary, tg1_General_1, tg1_Locations_Main, tg1_High_Frequency_1, tg1_High_Frequency_2;
- **Discos B**: tg1_General_2, tg1_High_Frequency_3, tg1_High_Frequency_4, tg1_Low_Frequency_Main;
- **Discos C**: tg1_General_3, tg1_Reviews_1, tg1_High_Frequency_5, tg1_Moderation_1 tg1_Logs_1;
- **Discos D**: tg1_General_4, tg1_Reviews_2, tg1_High_Frequency_6, tg1_Moderation_2, tg1_Logs_2.

Estes discos serão agrupados em números, sendo espelhados três vezes, resultando em doze discos no total. No final, teremos os seguintes discos:

- A1, B1, C1, D1;
- A2, B2, C2, D2;
- A3, B3, C3, D3.

Na estrutura atual da base de dados, nos scripts de criação da mesma, o caminho usado é sempre o mesmo. Isto é porque, em desenvolvimento, não é necessário (obviamente) usar RAID de todo.

Por fim, os backups também serão armazenados, nos **Discos A**. Estes discos devem ter maior capacidade que os restantes tipos. A capacidade variará consoante o número de utilizadores da plataforma. Para não manter todos os dados no mesmo local, também será boa prática enviar *full backups* a discos noutra localização física.

5.3. Casos de Catástrofe e Resolução

Em caso de catástrofes, umas mais severas que outras, a base de dados estará preparada para remediar as situações. Como esta plataforma não se trata de nenhum sistema crítico, os *logs* têm backups feitos de 15 em 15 minutos, portanto poderá haver uma perda de dados até 15 minutos no caso de corrupção das bases de dados.

Este acontecimento é, no entanto, raro. O que é mais provável de acontecer é a perda de um disco em específico.

Por exemplo, imaginando que o disco B2 queima, ainda haverá a possibilidade de busca de dados dos discos B1 e B3, não havendo interrupções ao funcionamento da base de dados. Nesse acontecimento, apenas substitui-se o disco B2, e configura-se de modo a que o mesmo tenha os dados de B1/B3 espelhados de novo.

No caso mais extremo possível, que é a destruição total de todo o hardware, poderá haver a salvaguarda de backups numa outra localização, podendo então proceder ao restauro através da mesma.

6. Casos de uso da Base de Dados e Consultas

Nesta secção não se apresentarão todos os casos de uso, mas sim os principais. Muitos já foram mencionados através de *stored procedures* ou até *triggers*, mas nesta secção serão descritos com um detalhe ligeiramente maior.

Esta secção não contém *scripts*. Apenas explica os casos de uso e alguns passos que podem ser menos claros. Para *scripts*, o gerador de dados já inclui a maioria do que é necessário para inserir dados na base de dados.

6.1. Registo de um utilizador

Primeiro, deve-se saber qual o tipo de utilizador. Se é administrador, o mesmo normalmente será criado manualmente, pelo menos numa versão inicial da plataforma. Senão, ou é anfitrião, ou é cliente.

O URL da foto de perfil (*profilePictureUrl*) poderá ser nulo. A nível aplicacional, o valor por defeito será um URL para a foto de perfil *default*, mas para guardar espaço na base de dados e evitar redundância, será mantido o valor como nulo.

Durante a inserção do utilizador, devem ser gerados 128 bytes aleatórios e transformá-los em hexadecimal (por exemplo) para usar como *sal* (como referido na secção **2.1**). O *hash* usado para palavras-passe poderá ser SHA-256 ou SHA-512, mas o mais recomendável em produção será um *hash* mais lento, para evitar a possibilidade de *brute force*.

6.2. Alojamentos com apenas um quarto

Como já referido anteriormente, alojamentos que apenas conseguem acolher um cliente (com ou sem grupo de outras pessoas) terão, na base de dados, apenas um quarto. No entanto, como também já foi referido, não será prático integrar essa funcionalidade a nível aplicacional, sendo o mais recomendável a criação automática de quartos se o utilizador define que apenas consegue acolher um cliente no seu alojamento local.

Isto pode ser feito através de uma seleção simples durante a criação de um alojamento da plataforma, um para um alojamento local com mais que um quarto, e outro para um alojamento com apenas um quarto, que não terá um editor de quartos diferentes.

6.3. Criação de uma reserva

Para criar uma reserva, não é necessário calcular o preço. Apenas insere-se à tabela de reservas o ID do utilizador do cliente, o ID do alojamento, o ID do quarto (que pode ser buscado através do *stored procedure findRoomForGuestAmount*), a data de início, a data de fim e o número de hóspedes total. O custo será calculado automaticamente no *trigger*.

6.4. Épocas especiais com custos diferentes

É possível ter custos diferentes num alojamento baseado num certo intervalo de datas. Por exemplo, se na época balnear um certo alojamento local é mais caro, poderá ser criado um registo em *DateIntervalCost* para cada quarto aplicável (se aplicável a todos, o *front-end* poderá ajudar o utilizador com o mesmo) com o novo custo respetivo.

No caso de um quarto ter um custo maior que outro durante todo o ano, poderá ser feito um registo na mesma tabela mas com a data de início a 1 de janeiro e data de fim a 31 de dezembro. Neste caso, será também melhor o nível aplicacional gerir estas datas de forma automática como funcionalidade extraordinária.

6.5. Filtragem e ordenação de alojamentos para um certo utilizador

Os alojamentos têm diversos filtros possíveis, como a classificação média, a localização, a capacidade máxima (que pode ser obtida pelo *stored procedure maxGuestCount*), entre outros.

Para a classificação média, a mesma é calculada pelo *trigger* mencionado na secção dos *triggers*, e guardada com todas as casas decimais, sem arredondamentos. Isto para que a ordenação seja feita com um pouco mais de precisão no caso de haver diferença na média entre alojamentos e o valor arredondado ser igual, para que seja uma ordenação mais justa. No entanto, a nível de *front-end*, é óbvio que deve-se mostrar ao utilizador uma ou duas casas decimais na média da classificação, e não todas as que a base de dados guarda.

Qualquer outro filtro ou ordenação pode ser feito(a) através de consultas simples SQL. Não é necessário chamar nenhum *stored procedure*.

7. *Performance* da Base de Dados

Para melhorar o desempenho da base de dados, podemos criar alguns índices nas colunas mais usadas nas cláusulas *WHERE*.

Note-se que apenas serão criados índices *NONCLUSTERED* uma vez que todas as tabelas desta base de dados têm chaves primárias com índices *CLUSTERED*, com auto-incremento.

Os índices criados podem ser observados através do *script* na Figura 7.1.

```
-- Índices non-clustered para velocidades de leitura mais rápidas.
CREATE NONCLUSTERED INDEX cities_index ON Locations.City ([name])
CREATE NONCLUSTERED INDEX states_index ON Locations.[State] ([name])
CREATE NONCLUSTERED INDEX countries_index ON Locations.Country ([name])

CREATE NONCLUSTERED INDEX housings_index ON General.Housing (hostUserId, cityId, stateId, ratingAvg)
CREATE NONCLUSTERED INDEX room_index ON General.Room (housingId, maxGuestCount)

CREATE NONCLUSTERED INDEX dateintervals_index ON HighFrequency.DateIntervalCost (dateFrom, dateTo)

CREATE NONCLUSTERED INDEX reviewreports_index ON Moderation.ReviewReport (hostReviewId, clientReviewId)
```

Figura 7.1 – Criação de índices para melhorar *performance* de consultas.

Os nomes das cidades, estados e países serão sempre obtidos a partir dos seus nomes e não pela identificação, devido à pesquisa dos mesmos. Assim, será melhor guardar os mesmos num índice.

Em relação a alojamentos, os alojamentos poderão ter filtros ou ordenações baseadas em cidades, estados, classificações ou por anfitrião. Assim, estes atributos também estarão num índice.

No que toca aos quartos, estes são obtidos na maioria das vezes a partir da chave estrangeira do alojamento, ou então quando procura-se um quarto através da capacidade, pelo que será necessário adicionar estes dois atributos a um índice também.

Os intervalos de custos são quase sempre consultados pelas datas, portanto um índice também é criado para estes atributos.

Por fim, as denúncias quando aceites tornam outras denúncias no mesmo comentário aceites igualmente, portanto será uma boa ideia também criar um índice das chaves estrangeiras das mesmas.

8. Geração de Dados Fictícios

Para a geração de dados fictícios, como mencionado na introdução do presente relatório, foi criada uma aplicação de linha de comandos em Node.js que gera dados aleatórios, com contexto, para todas as tabelas da base de dados exceto para as tabelas de moderação.´

Este gerador de dados tem secções que utilizam concorrência. Apesar de o Node.js (e o JavaScript em geral) ser *single-threaded*, existe uma forma de utilizar *threads* na mesma, usando *worker_threads* [4].

A razão pelo qual houve necessidade para o mesmo foi o quão lenta a inserção se tornou para a quantidade de dados gerados. Assim, a inserção pode ser muito mais rápida.

8.1. Conexão à base de dados

No gerador de dados, que pode ser encontrado nos ficheiros em **gen/**, podemos encontrar o ficheiro ***config.json*** que contém a configuração para a ligação à base de dados, assim como o número de threads a usar na inserção de dados, quando aplicável.

O objeto *dbConfig* neste ficheiro JSON contém o *user*, a *password*, o *hostname* (*server*), a base de dados (*database*), um valor booleano obrigatório para permitir o certificado do servidor e ligar-se sem problemas, e uma configuração para o *pool*, que contém múltiplas conexões à base de dados para melhorar a *performance*.

O código para a conexão à mesma pode ser observado na Figura 8.1.

```
mssql.connect(dbConfig, async (err) => {
  const pool = new mssql.ConnectionPool(dbConfig)
  await pool.connect()

  if (err) {
    console.error(err)
  } else {
    console.info('Connected to the SQL Server database.')
  }
```

Figura 8.1 – Ligação à base de dados por Node.js.

Note-se que *dbConfig* importa o ficheiro JSON, que é feito através de um *import*.

8.2. Inicialização do Gerador

Para inicializar o gerador, será necessário apagar todos os dados primeiro. Como os dados estão todos interligados por chaves estrangeiras, haverá sempre conflito. Para evitar isso, começa-se por eliminar os *constraints* existentes, de forma oposta à ordem de criada (uma vez que a ordem de criação é à base de dependência nas chaves estrangeiras). Para tal, usa-se a seguinte *query*, na Figura 8.2:

```
for (const r of reg.reverse()) {
  if (r.enabled) {
    for (const tableName of r.tableNames) {
      console.log(tableName)
      const request = new mssql.Request()
      request.input('tableName', mssql.VarChar, tableName)
      await request.query(`
      ALTER TABLE ${tableName} NOCHECK CONSTRAINT ALL
      DELETE FROM ${tableName}
      ALTER TABLE ${tableName} WITH CHECK CHECK CONSTRAINT ALL
      IF (OBJECTPROPERTY(OBJECT_ID(@tableName), 'TableHasIdentity') = 1) DBCC CHECKIDENT (@tableName, RESEED, 0)
    `)
    }
  }
}
```

Figura 8.2 – Remoção de *constraints* e dados das tabelas.

A variável *reg* contém o registo de todos os ficheiros que são responsáveis por uma secção da base de dados a ser criada. Estas secções contêm várias tabelas que estão relacionadas com um contexto em específico. No total existem 11 secções. É usada a função *reverse()* de *arrays* para que a ordem de criação das secções seja a oposta ao eliminar os dados e *constraints* das tabelas.

Quando este processo termina, será chamado o "injetor", que inicializa a inserção de todos os dados e chama todas as "secções" sequencialmente.

8.3. Chamada e Funcionamento do "Injetor"

A Figura 8.3 mostra a implementação da função *registrations()*, que retornará, quando pronta (uma vez que é assíncrona) todas estas "secções" (ou registos) de injeção.

```
export async function registrations() {
  const rawFiles = readdirSync('./src/io')
  const files = (await Promise.all(rawFiles.filter((file) => file.endsWith('.js'))
    .sort((a, b) => getIndex(a) - getIndex(b))
    .map(async (file) => ({ fileName: file, mod: await import(`./io/${file}`) })))
  )).filter((file) => file.mod.enabled)

  return files.map(({ fileName, mod }) => ({
    type: fileName,
    insert: mod.insert,
    multithread: mod.multithread,
    tableNames: mod.tableNames,
    enabled: mod.enabled,
    insertSinglethread: mod.insertSinglethread,
    amountOfDataToInsert: mod.amountOfDataToInsert,
    iterableDataStatement: mod.iterableDataStatement,
    iterableDataPrimaryKey: mod.iterableDataPrimaryKey, // Chave primária da tabela que será iterada.
  }))
}
```

Figura 8.3 – Função *registrations()*.

Esta função lê a diretoria ***gen/src/io*,** que contém todos os ficheiros que representam estas 11 secções. Cada uma será então retornada como um *array* de objetos que podemos observar no retorno da chamada de *map()*. O atributo *insert* é na verdade a função que fará a inserção de dados em cada secção, que está aplicada de acordo ao contexto aplicável em cada uma das mesmas. Podemos dizer que uma espécie de polimorfismo foi aplicada aqui, apesar de não ser usada orientação a objetos diretamente. De qualquer modo, o mesmo é ligeiramente desnecessário nesta linguagem, da forma como este gerador está escrito.

Esta função será chamada para dar como parâmetro o resultado à função *inject()*, que é chamada no passo anterior, mesmo após a eliminação de dados e *constraints*. O conteúdo da mesma pode ser observado na Figura 8.4.

```
export async function inject(reg) {
  for (let i = 0; i < reg.length; i++) {
    await _inject(i + 1, reg.length, reg[i].type, config.threadAmount)
  }
}
```

Figura 8.4 – Função *inject(reg)*.

A chamada a *_inject* é uma chamada "interna", que será então a função que se encontra no gestor de *threads*, que necessita da informação que *reg* contém. No entanto, como estas *threads* não conseguem partilhar informação muito complexa (como funções), temos de partilhar apenas o nome do ficheiro (*type*), que pertence aos objetos retornados em *registrations(),* para depois então cada *threads* chamar *registrations()* novamente para obter a informação que necessita por si só.

Na Figura 8.5 podemos observar o conteúdo da função *_inject* (aqui denominada por *inject*).

```
if (isMainThread) {
  inject = (j, total, type, workerAmount) => {
    const now = new Date().getTime()
    return new Promise((resolve) => {
      if (getIndex(type) > 1) {
        process.stdout.write(`\r(${j}/${total}) Generating ${_formatFileName(type)}...                                    `)
      }
      const workers = new Array(workerAmount)
      let workersFinished = 0
      for (let i = 0; i < workers.length; i++) {
        const currentWorker = i
        workers[i] = new Worker(fileURLToPath(import.meta.url), { workerData: { i, workerAmount, type, total }, argv: process.argv })
        const onDone = () => {
          process.stdout.write(`\r(${j}/${total}) Generating ${_formatFileName(type)}... Done in ${new Date().getTime() - now}ms.
          workers.forEach((worker) => worker.terminate())
          resolve()
        }
        workers[i].on('error', (error) => {
          console.error(error)
        })
        workers[i].on('exit', (code) => {
          if (code !== 0) return console.error((`Worker ${currentWorker} stopped with exit code ${code}.`))
          else workersFinished++
          if (workersFinished === workers.length) onDone()
        })
        workers[i].on('message', (message) => {
          if (message === 'start') workers.forEach((worker) => worker.postMessage('start'))
        })
      }
    })
  }
} else {
```

Figura 8.5 – Conteúdo de *_inject()*.

A verificação inicial de *isMainThread* é porque já estamos num ficheiro que é um *worker*, ou pelo menos tornar-se-á num. Estes *workers* (ou *threads*) executam o ficheiro JavaScript atual, e temos de fazer essa verificação com essa variável importada do próprio módulo *worker_threads*. Portanto, se estamos na *threads* principal, então inicializamos tudo, incluindo os *workers*.

Começamos por definir um *array* de *workers* que terá o tamanho da configuração mencionada anteriormente. Este número varia consoante o *hardware*. **Atenção:** aumentar o número de *threads* nem sempre aumenta a *performance*, especialmente se o *hardware* não tem um número de núcleos maior que este número. O número por defeito é 12 uma vez que o meu

processador tem a possibilidade de executar 16 *threads* em paralelo, e foi o número com que obtive os melhores resultados em tempos de CPU.

Quando a execução termina, o evento 'exit' será chamado em cada *worker*. Para saber se todos os *workers* terminaram, simplesmente contamos quantos terminaram com sucesso. Se o número de *workers* que terminou é igual ao número total dos mesmos, então *onDone()* é chamado, que resolve a promessa e continua a execução para a próxima secção no injetor.

Em relação ao que cada *worker* executa, estes distribuem os dados que devem inserir entre si. No caso de haver preferência de tornar uma inserção em *single-thread* (porque há dependência de dados inseridos ou devido a problemas de concorrência), apenas um *worker* fará a inserção, e o resto será dado como terminado imediatamente.

8.4. Exemplo de implementação de uma secção

Para a geração de *SiteUsers*, foram usados vários *datasets*, que estão disponíveis na bibliografia. Na Figura 8.6, podemos observar a geração de todos os atributos de *SiteUser*.

```
const name = randomName()
// 70% dos utilizadores têm foto de perfil.
const profilePictureUrl = Math.random() > 0.3 ? '/profiles/' + randomBytes(32).toString('hex') + '.png' : null
// 5% dos utilizadores são anfitriões.
const isHost = Math.random() > 0.95
// 0.01% dos utilizadores são administradores.
const isAdmin = Math.random() > 0.9995
// Emails aleatórios compostos pelo nome, um número aleatório e servidor de email aleatório.
const email = randomEmail(name)
// 60% dos anfitriões tem identidade verificada.
const identityVerified = isHost && Math.random() > 0.4
// Número de telemóvel americano aleatório.
const mobile = '+1555' + Math.random().toString().slice(2, 9)
// Sal do hash da password de 128 bytes.
const salt = randomBytes(64).toString('hex')
// Hash da password.
const unhashedPassword = salt + randomBytes(128).toString('hex')
const password = createHash('sha512').update(unhashedPassword).digest().toString('hex')
// Se o utilizador for anfitrião, tem um número de contribuinte fiscal aleatório, senão tem 70% de probabilidade de não ter.
const fiscalId = isHost
  ? Math.random().toString().slice(2, 11)
  : Math.random() > 0.3 ? '999999990' : Math.random().toString().slice(2, 11)

const { locationAddrLine1, locationAddrLine2, cityId, stateId, locationPostalCode } = await genAddress(mssql, request)
```

Figura 8.6 – Geração de todos os atributos de *SiteUser*.

As variáveis aqui criadas têm exatamente os mesmos nomes que os respetivos atributos.

Os atributos *name*, *email* e todos os atributos vindos de *genAddress()* são gerados à parte. Nomes são gerados a partir do *dataset* de primeiros nomes e apelidos, e moradas são geradas a partir de moradas aleatórias com o auxílio de alguns *datasets* também. O código completo pode ser observado na pasta ***gen/io*** e ***/gen/io/common***.

8.5. Uso de *PreparedStatements* para executar inserções

Apesar de este gerador não conter nenhum *input* de utilizador nem ser usado em produção, continua sempre a ser boa prática o uso de PreparedStatements que previne a injeção de SQL malicioso a *queries* SQL. Nas Figuras 8.7, 8.8 e 8.9, podemos observar como os mesmos são executados.

```
const ps = new mssql.PreparedStatement(pool)
```

Figura 8.7 – Inicialização do PreparedStatement.

```
ps.input('name', mssql.NVarChar)
ps.input('profilePictureUrl', mssql.NVarChar)
ps.input('isHost', mssql.Bit)
ps.input('isAdmin', mssql.Bit)
ps.input('email', mssql.NVarChar)
ps.input('identityVerified', mssql.Bit)
ps.input('mobile', mssql.NVarChar)
ps.input('password', mssql.NVarChar)
ps.input('salt', mssql.NVarChar)
ps.input('locationAddrLine1', mssql.NVarChar)
ps.input('locationAddrLine2', mssql.NVarChar)
ps.input('cityId', mssql.Int)
ps.input('stateId', mssql.Int)
ps.input('locationPostalCode', mssql.NVarChar)
ps.input('fiscalId', mssql.NVarChar)


await ps.prepare(`
INSERT INTO General.SiteUser (name, profilePictureUrl, isHost,
  isAdmin, email, identityVerified, mobile,
  password, salt, locationAddrLine1,
  locationAddrLine2, cityId, stateId,
  locationPostalCode, fiscalId)
VALUES (@name, @profilePictureUrl, @isHost, @isAdmin,
  @email, @identityVerified, @mobile, @password, @salt,
  @locationAddrLine1, @locationAddrLine2, @cityId,
  @stateId, @locationPostalCode, @fiscalId)
)
```

Figura 8.8 – Preparação do *statement* com as variáveis respetivas.

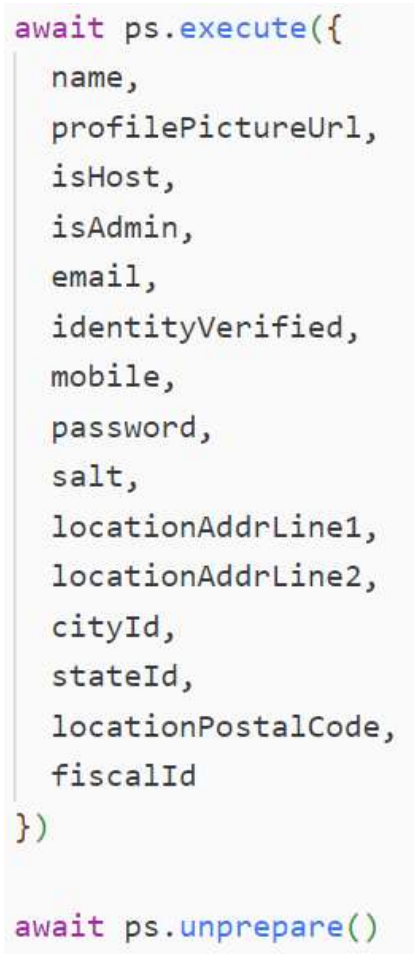

Figura 8.9 – Execução e fecho do PreparedStatement.

O fecho do *PreparedStatement* é essencial, uma vez que temos, no caso da minha execução, 11 outros *threads* em concorrência a comunicar com a base de dados, e o *pool* terá demasiadas conexões se não fecharmos todos estes que estão a ser criados. A certo momento, a conexão deixará se funcionar.

8.6. Execução do Gerador

Por fim, para executar o gerador, simplesmente executa-se "**npm start"** numa linha de comandos no nível de ***gen/***, confirma-se a eliminação de dados, e espera-se que os dados gerem. Existe uma interface de linha de comandos intuitiva que indicará o progresso com um detalhe considerável.

9. Conclusões

Para concluir o presente relatório e o projeto desenvolvido, posso afirmar que o mesmo foi extremamente importante para o meu desenvolvimento académico não só nos conceitos desta unidade curricular, mas como em qualquer conceito relacionado com o desenvolvimento de *software*, em especial no treino de desenvolvimento de *Node.js* e conexões a bases de dados SQL.

Aprendi como podia efetuar concorrência por *threads* em Node.js para acelerar o gerador de dados, o que foi muito interessante para mim, apesar de não estar relacionado com a unidade curricular de todo, mas também que uma base de dados tem uma capacidade muito mais envolvente que simplesmente "guardar dados", com as funcionalidades que o T-SQL do *SQL Server* nos oferece.

Também trabalhei com vários tipos de *datasets* e na geração de dados que pode-se tornar útil no futuro, se precisar de preencher uma tabela rapidamente com dados numa base de dados que estiver a criar, ou mesmo para criar dados fictícios com qualquer fim.

Em relação aos conteúdos mais específicos desta unidade curricular de Bases de Dados II, posso afirmar que fiquei a saber muito mais sobre bases de dados e que nunca pensei que houvesse tantos conceitos que fossem importantes nas mesmas, como a melhoria de *performance*, a divisão de ficheiros em *filegroups* das tabelas, a existência de *schemas*, a segurança com utilizadores, os *triggers* que têm imensa utilidade e que dariam muito mais trabalho a fazer a nível aplicacional, e os *procedures* que facilitam algumas consultas e lógica a quem está a trabalhar a nível aplicacional também.

Bibliografia

[1] (2023) Node.js: API reference documentation. [Online]. Disponível: https://nodejs.org/en/docs/

[2] (2023) Microsoft Learn: Apply Transaction Log Backups (SQL Server). [Online]. Disponível: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/apply-transaction-log-backups-sql-server?view=sql-server-ver16

[3] (2023) Dave, Pinal, "SQL Server – FIX – Agent XPs Component is Turned Off as Part of the Security Configuration for this Server". [Online]. Disponível: https://blog.sqlauthority.com/2016/06/13/sql-server-fix-agent-xps-component-turned-off-part-security-configuration-server/

[4] (2023) Node.js v19.4.0 documentation: Worker threads. [Online]. Disponível: https://nodejs.org/api/worker_threads.html#:~:text=Workers%20(threads)%20are%20useful%20for,cluster%20%2C%20worker_threads%20can%20share%20memory.

[5] (2023) DataHub: Major Cities of the World. [Online]. Disponível: https://datahub.io/core/world-cities

[6] (2023) Goodwin, Ross, "All the surnames from the 2000 U.S. Census, plus all first names occurring > 4 times in Census data from 1883 forward". [Online]. Disponível: https://github.com/rossgoodwin/american-names

[7] (2023) Shafrir, Michael, "US states in JSON form". [Online]. Disponível: https://gist.github.com/mshafrir/2646763

[8] (2023) Opendatasoft: Airbnb – Listings. [Online]. Disponível: https://public.opendatasoft.com/explore/dataset/airbnb-listings/api/?disjunctive.host_verifications&disjunctive.amenities&disjunctive.features&rows=100