Pagina de capa

Brupo: P5G2
Marta Cruz, MEC: 119572
Pedro Martins, MEC: 119916

Introdução

Este projeto visa o desenvolvimento de uma base de dados para uma plataforma de gestão de dados pessoais nas piscinas municipais, com o objetivo de resolver a atual dificuldade verificada nas piscinas de Ílhavo e da Gafanha da Nazaré: a inexistência de um sistema eficiente para consulta de saldo do cartão, registo de entradas e marcação de sessões.

Para esse efeito, a plataforma permitirá aos utilizadores consultar o saldo do respetivo cartão, realizar carregamentos diretamente e efetuar marcações de sessões, com acesso a um conjunto alargado de opções e informações disponibilizadas pelo sistema.

A interface foi desenvolvida com foco na perspectiva do utilizador, tendo em conta que o pretendido seria integrar a plataforma com o sistema administrativo já existente nas piscinas.

Análise de Requisitos

Numa fase inicial recolhemos os seguintes requisitos funcionais:

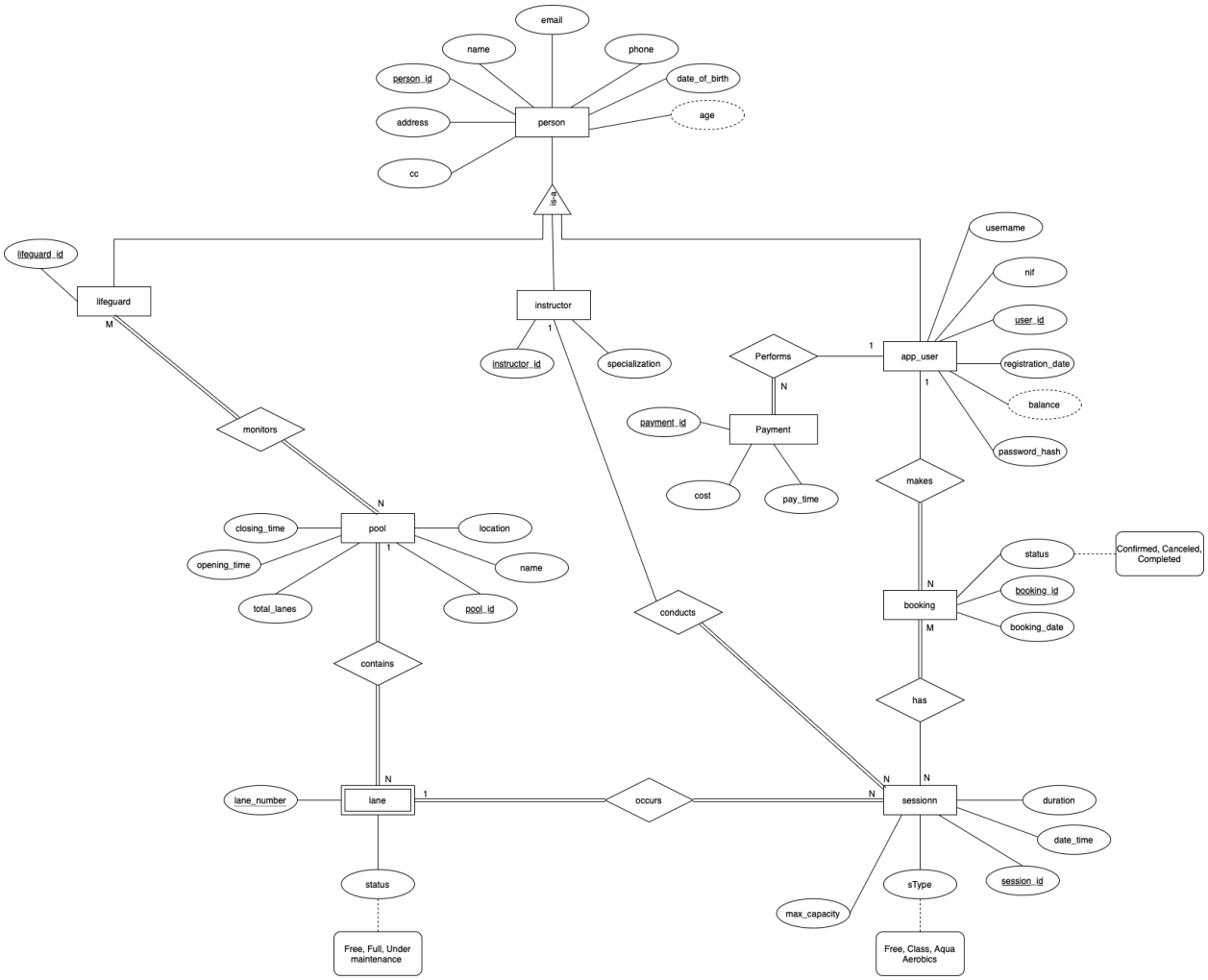
1. O sistema deve permitir o registo de utilizadores, cada um identificado pelo seu número do cartão de cidadão, morada, nome, e-mail, número de telefone, data de nascimento, idade, número de utilizador, nif, data de registo e um cartão com saldo associado.
2. As piscinas são compostas por pistas, cada uma identificada por um número e estado (livre, ocupada, em manutenção).
3. Cada piscina é monitorizada por 1 ou mais nadadores-salvadores identificados pelo seu número do cartão de cidadão, morada, nome, e-mail, número de telefone, data de nascimento, idade e número único atribuído pela autoridade marítima nacional.
4. Para carregar o seu cartão o utilizador deve realizar 1 ou vários pagamentos identificados pelo seu identificador único e montante.
5. Para poderem usufruir das piscinas municipais os utilizadores devem fazer marcações, que são identificadas pelo seu identificador único, data para quando se vai realizar e estado (completa, confirmada, cancelada) das sessões que querem participar. Uma sessão é identificada pela sua duração, hora e data em que vai ser realizada, um identificador único, uma capacidade máxima e um tipo (natação livre, aula de natação, hidroginástica).
6. Cada sessão do tipo aula ou hidroginástica é conduzida por um instrutor, identificado pelo seu número do cartão de cidadão, morada, nome, e-mail, número de telefone, data de nascimento, idade, número de instrutor e especialização.
7. Cada sessão ocorre numa pista.

Com base nestes requisitos identificamos as seguintes entidades:

1. Person
2. App_user
3. Payment
4. Instructor
5. Lifeguard
6. Pool
7. Lane
8. Session
9. Booking

DER - Diagrama Entidade Relacionamento

Versão final

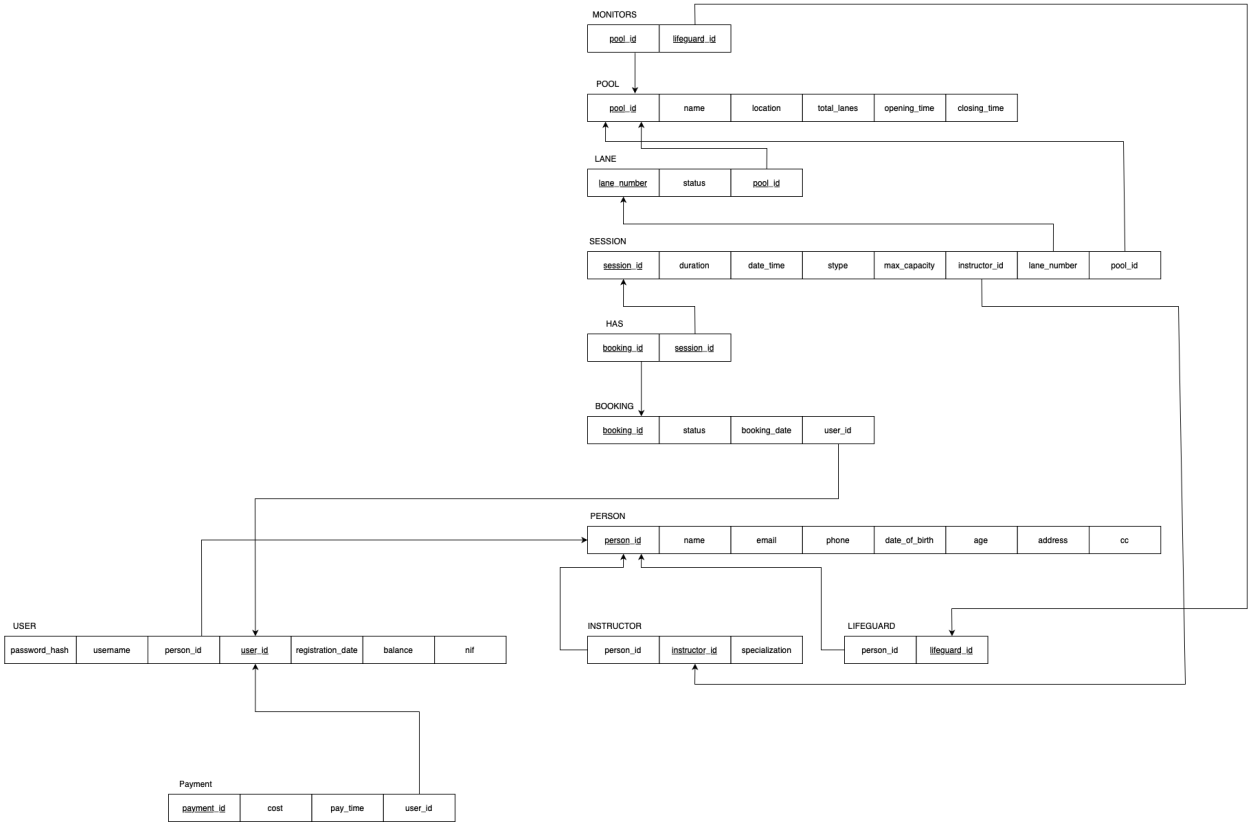


APFE

Em relação ao DER apresentado na proposta inicial, a versão final sofreu algumas alterações significativas. Substituímos a chave primária da entidade *Person* por um atributo *person_id*, após concluirmos que a utilização do número de cartão de cidadão (CC) para identificação de indivíduos não é permitida por lei. Na entidade *App_User*, foram adicionados os atributos *username* e *password_hash* para suportar a autenticação na plataforma. Por fim, foi incluído o atributo *pay_time*na entidade *Payment*, de forma a registar o momento exato de cada transação.

ER - Esquema Relacional

Versão final



APFE

Relativamente ao modelo ER proposto, refletimos as alterações efetuadas no DER, nomeadamente a atualização das chaves estrangeiras na hierarquia — substituindo o *cc* por *person_id* — bem como a adição dos novos atributos introduzidos nas respetivas entidades.

SQL DDL - Data Definition Language

TODO inserir a imagem do DDL

Código:

```
-- Create schema
create schema municipal;
go

-- Create Tables
create table municipal.person (
   person_id int identity(1,1),
   cc varchar(12),
   name varchar(30),
   email varchar(30),
   phone int,
   date_of_birth date,
   age int,
   address varchar(100),

   primary key (person_id)
);

create table municipal.app_user (
   user_id int identity(1,1),
   person_id int unique not null,
   registration_date datetime2 default current_timestamp,
   balance decimal(10, 2),
   nif int,
   username varchar(30) unique not null,
   password_hash varchar(255) not null,

   foreign key (person_id) references municipal.person(person_id),
   primary key (user_id)
);

create table municipal.payment (
   payment_id int identity(1,1),
```

```
    cost decimal(10, 2),
    user_id int not null,
    pay_time datetime2 default current_timestamp,

    foreign key (user_id) references municipal.app_user(user_id),
    primary key (payment_id)
);

create table municipal.instructor (
    instructor_id int identity(1,1),
    person_id int not null,
    specialization varchar(30),

    foreign key (person_id) references municipal.person(person_id),
    primary key (instructor_id)
);

create table municipal.lifeguard (
    lifeguard_id int identity(1,1),
    person_id int not null,

    foreign key (person_id) references municipal.person(person_id),
    primary key (lifeguard_id)
);

create table municipal.pool (
    pool_id int identity(1,1),
    name varchar(30),
    location varchar(100),
    total_lanes int,
    opening_time time,
    closing_time time,

    primary key (pool_id)
);

create table municipal.monitors (
    pool_id int,
    lifeguard_id int,
```

```
   foreign key (pool_id) references municipal.pool(pool_id),
                    foreign       key       (lifeguard_id)       references
municipal.lifeguard(lifeguard_id),
   primary key (pool_id, lifeguard_id)
);

create table municipal.lane (
   lane_number int,
   status varchar(30),
   pool_id int,

   foreign key (pool_id) references municipal.pool(pool_id),
   primary key (pool_id, lane_number)
);

create table municipal.sessionn (
   session_id int identity(1,1),
   duration int,
   date_time datetime,
   sType varchar(30),
   max_capacity int,
   instructor_id int,
   lane_number int,
   pool_id int,

                    foreign       key       (instructor_id)       references
municipal.instructor(instructor_id),
    foreign key (pool_id, lane_number) references municipal.lane(pool_id,
lane_number),
   primary key (session_id)
);

create table municipal.booking (
   booking_id int identity(1,1),
   status varchar(30),
   booking_date date,
   user_id int,

   foreign key (user_id) references municipal.app_user(user_id),
   primary key (booking_id)
```

```
);

create table municipal.has (
   booking_id int,
   session_id int,

   foreign key (booking_id) references municipal.booking(booking_id),
   foreign key (session_id) references municipal.sessionn(session_id),
primary key (booking_id, session_id)
);




-- Deleted records tables
create table municipal.deletes_app_user (
   user_id int,
   person_id INT,
   registration_date DATETIME2,
   balance INT,
   nif INT,
   username VARCHAR(30),
   password_hash VARCHAR(255),
   deleted_at DATETIME2 DEFAULT CURRENT_TIMESTAMP
)

create table municipal.deletes_payment (
   payment_id INT,
   cost DECIMAL(10, 2),
   user_id INT,
   deleted_at DATETIME2 DEFAULT CURRENT_TIMESTAMP
)

create table municipal.deletes_booking (
   booking_id INT,
   status VARCHAR(30),
   booking_date DATE,
   user_id INT,
   deleted_at DATETIME2 DEFAULT CURRENT_TIMESTAMP
)
```

```
create table municipal.deletes_has (
   booking_id INT,
   session_id INT,
   deleted_at DATETIME2 DEFAULT CURRENT_TIMESTAMP
)
```

SQL DML - Data Manipulation Language

Listar todas as sessões:

```
     SELECT s.session_id, s.sType, s.date_time,
            p.name AS instructor_name,
            pl.name AS pool_name,
            s.lane_number, l.status AS lane_status
     FROM municipal.sessionn s
     JOIN municipal.instructor i ON s.instructor_id = i.instructor_id
     JOIN municipal.person p ON i.person_id = p.person_id
     JOIN municipal.pool pl ON s.pool_id = pl.pool_id
      JOIN municipal.lane l ON s.pool_id = l.pool_id AND s.lane_number =
l.lane_number
     WHERE s.date_time > GETDATE()
     ORDER BY s.date_time;
```

Obter os detalhes duma sessão:

```
     SELECT s.session_id, s.duration, s.date_time, s.sType,
            s.max_capacity, s.instructor_id, p.name AS instructor_name,
            s.lane_number, l.status AS lane_status,
            s.pool_id, pl.name AS pool_name
     FROM municipal.sessionn s
     JOIN municipal.instructor i ON s.instructor_id = i.instructor_id
     JOIN municipal.person p ON i.person_id = p.person_id
     JOIN municipal.pool pl ON s.pool_id = pl.pool_id
      JOIN municipal.lane l ON s.pool_id = l.pool_id AND s.lane_number =
l.lane_number
     WHERE s.session_id = ?;
```

Obter os detalhes duma sessão:

```
    SELECT b.booking_id, b.status, b.booking_date, b.user_id,
           s.session_id, pmt.cost,
           s.sType, s.date_time,
           pl.name, s.lane_number, l.status,
           per.name, s.max_capacity,
           (SELECT COUNT(*) FROM municipal.has h2
           JOIN municipal.booking b2 ON h2.booking_id = b2.booking_id
           WHERE h2.session_id = s.session_id) AS booked_count
    FROM municipal.booking b
    JOIN municipal.has h ON b.booking_id = h.booking_id
    JOIN municipal.sessionn s ON h.session_id = s.session_id
    JOIN municipal.instructor i ON s.instructor_id = i.instructor_id
    JOIN municipal.person per ON i.person_id = per.person_id
    JOIN municipal.pool pl ON s.pool_id = pl.pool_id
     JOIN municipal.lane l ON s.pool_id = l.pool_id AND s.lane_number =
l.lane_number
    LEFT JOIN municipal.payment pmt ON pmt.user_id = b.user_id
    WHERE b.booking_id = ?
```

Obter todas as reservas de um utilizador:

```
          select b.booking_id, b.status, b.booking_date,
                        s.session_id,  s.date_time,  s.sType,  s.duration,
s.max_capacity,
              i.instructor_id, p.name as instructor_name,
              s.lane_number, s.pool_id
          from municipal.booking b
          join municipal.has h on b.booking_id = h.booking_id
          join municipal.sessionn s on h.session_id = s.session_id
                    join  municipal.instructor  i  on  s.instructor_id  =
i.instructor_id
          join municipal.person p on i.person_id = p.person_id
          where b.user_id = ?
          and s.date_time > GETDATE()
```

Normalização

TODO inserir a imagem do DDL

Índices e Views

Usamos uma view e um índice para manter a integridade dos dados, mais especificamente para certificar que um utilizador não consegue criar mais do que uma reserva sobre uma sessão.
Código:

```
-- Helper view to prevent data duplication
go
create view municipal.unique_user_session
with schemabinding
as
   select b.user_id, h.session_id
   from municipal.has h
   join municipal.booking b on h.booking_id = b.booking_id
go

-- Enforce uniqueness
create unique clustered index UQ_User_session
on municipal.unique_user_session(user_id, session_id)
```

SQL Programming

Stored Procedures

Stored procedure para criar uma entidade reserva:

```
-- Create a booking
create procedure municipal.createBooking

   -- Return parameters explanations
   -- 0 => Success
   -- 1 => Session not found
   -- 2 => Session full
   -- 3 => Duplicate booking
```

```
    -- 4 => Unexpected
    -- 5 => User not found
    -- 6 => User doesn't have enough money
    -- 7 => Unknown session type

    -- Input params
    @user_id int,
    @session_id int
as
begin
    set nocount on;
    begin try
        begin transaction;

        -- Set high isolation level to prevent concurrency issues
        set transaction isolation level serializable;

        -- 1. Validate that session exists
        if not exists (
            select 1
            from municipal.sessionn
            where session_id = @session_id
        )
        begin
            rollback transaction;
            set transaction isolation level read committed;
            return 1; -- Session not found
        end;


        -- 2. Check session capacity
          declare @max_capacity int, @sType varchar(30), @current_bookings
int;

        select @max_capacity = max_capacity, @sType = sType
        from municipal.sessionn
        where session_id = @session_id;

        select @current_bookings = count(*)
        from municipal.has h
```

```
    join municipal.booking b on h.booking_id = b.booking_id
    where h.session_id = @session_id;

    if @current_bookings >= @max_capacity
    begin
        rollback transaction;
        set transaction isolation level read committed;
        return 2; -- Session full
    end;


    -- 3. Check for existing booking
    if exists (
        select 1
        from municipal.unique_user_session
        where user_id = @user_id and session_id = @session_id
    )
    begin
        rollback transaction;
        set transaction isolation level read committed;
        return 3; -- Duplicate booking
    end;


    -- 4. Check if the user exists and get balance
    declare @user_balance decimal(10, 2);

    select @user_balance = balance
    from municipal.app_user
    where user_id = @user_id;

    if @user_balance is null
    begin
        rollback transaction;
        set transaction isolation level read committed;
        return 5; -- User not found
    end;


    -- 5. Check if the user ain't broke (if balance is enough)
```

```
declare @session_price decimal(10, 2);

if @sType = 'Free'
    set @session_price = 1 -- 1€ entrance fee

else if @sType = 'Aerobics'
    set @session_price = 5 -- 5€ Aerobics

else if @sType = 'Class'
    set @session_price = 3 -- 3€ Class

else
begin
    rollback transaction;
    set transaction isolation level read committed;
    return 7; -- Unknown session type
end;

if @user_balance - @session_price < 0
begin
    rollback transaction;
    set transaction isolation level read committed;
    return 6; -- Broke ass
end;


-- 6. Deduct from account (create negative payment)
insert into municipal.payment (cost, user_id)
values (-@session_price, @user_id);


-- 7. Create booking
declare @booking_id int;

insert into municipal.booking (status, booking_date, user_id)
values ('confirmed', cast(getdate() as date), @user_id)

set @booking_id = scope_identity();

-- 8. Link booking to session
```

```
        insert into municipal.has (booking_id, session_id)
        values (@booking_id, @session_id);

        commit transaction;

        -- Reset isolation level back to read committed
        set transaction isolation level read committed;

        return 0; -- Success
    end try
    begin catch
        if @@trancount > 0
            rollback transaction;

        -- Reset isolation level back to read committed
        set transaction isolation level read committed;

        -- Handle other errors (those that weren't already)
        return 4; -- Unexpected
    end catch;

end;
go
```

Outros stored procedures desenvolvidos:

- CreateUser
- CreateInstructor
- CreateLifeguard
- MakePayment
- CreateMonitors
- CreateLane
- createSession
- createBooking
- deleteUser
- deleteBooking

Triggers

```
-- Trigger to prevent duplicate bookings
create trigger municipal.trg_prevent_duplicate_booking
```

```
on municipal.has
instead of insert
as
begin

   if exists (
       select 1
       from inserted i
       join municipal.booking new_booking
           on i.booking_id = new_booking.booking_id
       join municipal.booking existing_booking
           on new_booking.user_id = existing_booking.user_id
       join municipal.has h
           on h.booking_id = existing_booking.booking_id
           and h.session_id = i.session_id
   )

   begin
       throw 51000, 'User already has a booking for this session.', 1;
       return;
   end;

   -- If there isn't a duplicate, proceed with the insert
   insert into municipal.has (booking_id, session_id)
   select booking_id, session_id from inserted;

end;
go


-- Trigger to make update balance upon a new payment entity
create trigger municipal.UpdateBalanceOnPayment
on municipal.payment
after insert
as
begin
   set nocount on;

   -- For each user, calculate the sum of all costs
   -- in the inserted batch
```

```
    if exists
    (
        select 1
        from
        (
            select
                i.user_id,
                sum(i.cost) as total_cost
            from inserted as i
            group by i.user_id
        ) as NewTotals
        join municipal.app_user as u
            on NewTotals.user_id = u.user_id
        where u.balance + NewTotals.total_cost < 0
    )
    begin
        rollback transaction;
        throw 51000, 'Transaction would result in negative balance', 1;
        return;
    end;


    -- No negative-balance violations, then update the users
    ; with NewTotals as
    (
        select
            i.user_id,
            sum(i.cost) as total_cost
        from inserted as i
        group by i.user_id
    )
    update u
    set u.balance = u.balance + nt.total_cost
    from municipal.app_user as u
    join NewTotals as nt
        on u.user_id = nt.user_id;
end;
go
```

UDFs

```
-- Search session based on many different parameters
create function municipal.SearchSessions (
   @sType varchar(30) = null,
   @instructor_name varchar(30) = null,
   @duration_min int = null,
   @duration_max int = null,
   @search_date date = null
)
returns table
as
return (
   select
       s.session_id,
       s.duration,
       s.date_time,
       s.sType,
       s.max_capacity,
       i.instructor_id,
       p.name as instructor_name,
       s.lane_number,
       po.pool_id,
       po.name as pool_name,
       l.status as lane_status
   from municipal.sessionn s
   left join municipal.instructor i on s.instructor_id = i.instructor_id
   left join municipal.person p on i.person_id = p.person_id
    join municipal.lane l on s.pool_id = l.pool_id and s.lane_number =
l.lane_number
   join municipal.pool po on s.pool_id = po.pool_id
   where
       (@sType is null or s.sType = @sType) -- Apply filters if they exist
       and (@instructor_name is null or p.name like '%' + @instructor_name
+ '%')
       and (
           (@duration_min is null and @duration_max is null) or
                   (s.duration between coalesce(@duration_min, 0) and
coalesce(@duration_max, 2147483647))
       )
```

```
            and (@search_date is null or cast(s.date_time as date) =
@search_date)
);
go


-- Payment history
create function municipal.PaymentHistory (
   @user_id int
)
returns table
as
return (
   select
       payment_id,
       cost,
       pay_time
   from municipal.payment
   where user_id = @user_id
);
go


-- Bookings the user has
create function municipal.UserBookings (
   @user_id int
)
returns table
with schemabinding
as
return
(
   select
       b.booking_id,
       b.status,
       b.booking_date,
       h.session_id,
       s.date_time as session_datetime,
       s.sType as session_type,
       s.duration as session_duration,
```

```
        s.max_capacity as session_capacity,
        s.lane_number as session_lane,
        s.pool_id as session_pool,
        p.name as instructor_name
    from municipal.booking as b
    join municipal.has as h
        on b.booking_id = h.booking_id
    join municipal.sessionn as s
        on h.session_id = s.session_id
    join municipal.instructor as i
        on s.instructor_id = i.instructor_id
    join municipal.person as p
        on i.person_id = p.person_id
    where b.user_id = @user_id
        and s.date_time > getdate() -- Only future sessions
);
go
```