Lab: Introduction to Databases

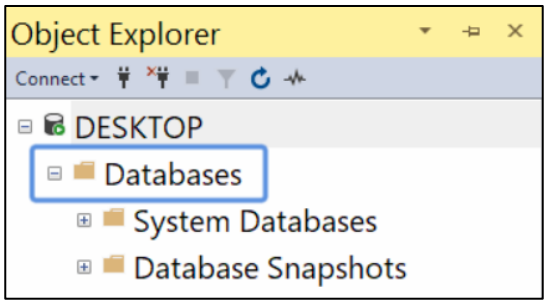
In this lab, we will create a **"Bank" database** in **SQL Server**, using **MS SQL Server Management Studio**. We will create **tables** and **fill them with data**, create **views**, **functions**, **procedures** and **triggers**. This tutorial is a part of the "Databases Basics - MS SQL Server" course @ SoftUni.

Before starting this tutorial, make sure you've followed the SQL Server installation guide.

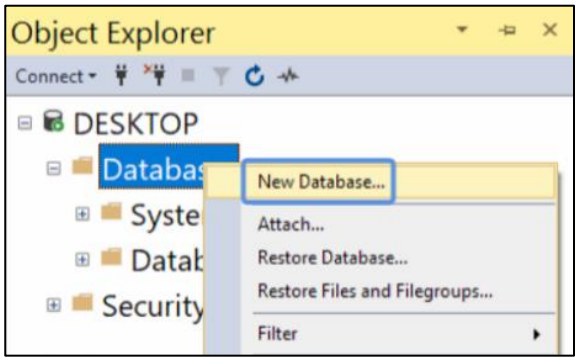
1. Create a Database

Create a new **database** named **Bank** using the **MS SQL Server Management Studio GUI**.

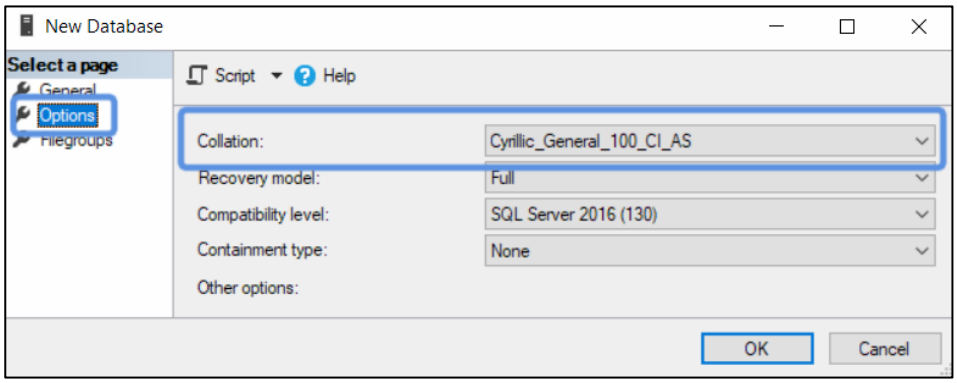
Step 1. Right click on **Databases** in the **Object Explorer**:



Step 2. Choose **New Database** from the drop-down menu:

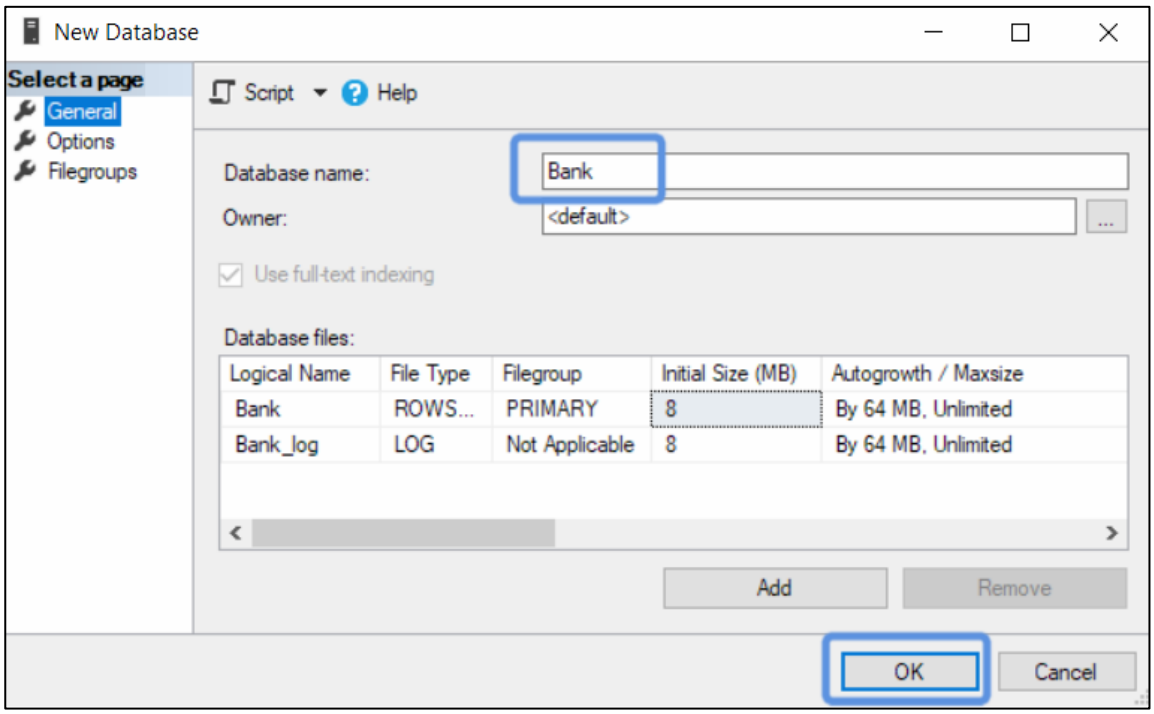


Step 3. A popup window will open. Go to **Options** and change the **Collation** to **`Cyrillic_General_100_CI_AS`**:



The reason we do this is so that **Cyrillic characters** show up properly.

Then go back to **General**, type in "**Bank**" as the **Database name** and click [**OK**]:



2. Create Tables

Step 1. Using an **SQL query**, create table **`Clients`** with the following properties:

- **`Id`** – unique **number** for every client (**auto-incremented**, **primary key**)
- **`FirstName`** – the **name** of the user, which will be no more than **50 Unicode characters** (**Cannot** be **`null`**).
- **`LastName`** – has the **same properties** as **`FirstName`**

```
CREATE TABLE Clients (
   Id INT PRIMARY KEY IDENTITY,
   FirstName NVARCHAR(50) NOT NULL,
   LastName NVARCHAR(50) NOT NULL
)
```

Step 2. Create table **`AccountType`** with:

- **`Id`** – unique number for every type. (Auto incremented, primary key)
- **`Name`** – the name of the account type, no longer than **50 Unicode characters** (**Cannot** be **`null`**)

Important: Don't forget to select the query you want to run before clicking Execute (F5) if you have multiple queries!

```
CREATE TABLE AccountTypes (
   Id INT PRIMARY KEY IDENTITY,
   [Name] NVARCHAR(50) NOT NULL
)
```

Step 3. Create table **Accounts** with:

- **`Id`** - **unique number** for every user. (**Auto incremented**, **primary key**)
- **`AccountTypeId`** – references the **AccountTypes** table (**foreign key**)
- **`Balance`** – **`decimal`** data type with up to 15 digits including 2 after the decimal point and a default value of 0 (Not null)
- **`ClientId`** – references the **Clients** table (**foreign key**)

```
CREATE TABLE Accounts (
   Id INT PRIMARY KEY IDENTITY,
   AccountTypeId INT FOREIGN KEY REFERENCES AccountTypes(Id),
   Balance DECIMAL(15, 2) NOT NULL DEFAULT(0),
   ClientId INT FOREIGN KEY REFERENCES Clients(Id)
)
```

3. Insert Sample Data into Database

We need some data to work with, so let's use **INSERT INTO (…) VALUES (…)** queries to fill our tables:

```
INSERT INTO Clients (FirstName, LastName) VALUES
('Gosho', 'Ivanov'),
('Pesho', 'Petrov'),
('Ivan', 'Iliev'),
('Merry', 'Ivanova')

INSERT INTO AccountTypes (Name) VALUES
('Checking'),
('Savings')

INSERT INTO Accounts (ClientId, AccountTypeId, Balance)
VALUES
(1, 1, 175),
(2, 1, 275.56),
(3, 1, 138.01),
(4, 1, 40.30),
(4, 2, 375.50)
```

4. Create a Function

Now let's create a **function**, which **calculates** the **total balance** from **all accounts** of a single **client**. Functions in **SQL** receive **parameters**, complete certain actions with them and **always** return a **result**. Our **function** will receive an **`int`**, called **`@ClientID`** and return a **`DECIMAL`**. It could look like this:

```
CREATE FUNCTION f_CalculateTotalBalance (@ClientID INT)
RETURNS DECIMAL(15, 2)
BEGIN
  DECLARE @result AS DECIMAL(15, 2) = (
    SELECT SUM(Balance)
    FROM Accounts WHERE ClientId = @ClientID
  )
  RETURN @result
END
```

Now try and **select** the **function**, giving it an existing **client ID** as the **parameter**, example for **client ID ➔ 4**:

```
SELECT dbo.f_CalculateTotalBalance(4) AS Balance
```

Notice the **`dbo.`** before the function name – this is the name of the **schema** which we **must** type when calling **functions**.

5. Create Procedures

Next, we'll create a **procedure** that creates a **new account** for an **existing client**. Just like functions, **procedures** receive **parameters**, but **do not return** results. Our **procedure** will receive **`@ClientID`** and **`@AccountTypeID`** as **parameters** and will look like this:

```
CREATE PROC p_AddAccount @ClientId INT, @AccountTypeId INT AS
INSERT INTO Accounts (ClientId, AccountTypeId)
VALUES (@ClientId, @AccountTypeId)
```

Now we can **create** a new savings **account** for our **client** with **ID = 2** like this:

```
p_AddAccount 2, 2
```

After you **execute the procedure** a couple of times, don't forget to **check** if an account is **added correctly**, using a **SELECT** statement:

```
SELECT * FROM Accounts
```

Let's create **two** more **procedures** to **deposit** and **withdraw** money from the **accounts**.

Deposit Procedure

The **deposit procedure** will always **add** our **input amount** to the **current balance**:

```
CREATE PROC p_Deposit @AccountId INT, @Amount DECIMAL(15, 2) AS
UPDATE Accounts
SET Balance += @Amount
WHERE Id = @AccountId
```

Withdraw Procedure

The **withdraw procedure** will **subtract** the given **amount** of money from the account **if the balance is enough** and **return an error message if it isn't**:

```
CREATE PROC p_Withdraw @AccountId INT, @Amount DECIMAL(15, 2) AS
BEGIN
  DECLARE @OldBalance DECIMAL(15, 2)
  SELECT @OldBalance = Balance FROM Accounts WHERE Id = @AccountId
  IF (@OldBalance - @Amount >= 0)
  BEGIN
    UPDATE Accounts
    SET Balance -= @Amount
    WHERE Id = @AccountId
  END
  ELSE
  BEGIN
    RAISERROR('Insufficient funds', 10, 1)
  END
END
```

6. Create Transactions Table and a Trigger

Our bank will need a way to **record transactions** done by its **clients**, so we are now going to create a **new table** and a **trigger**, which will **automatically** record the **date**, **time** and **amount transferred** into the table.

We will name the table **`Transactions`** and it will have:

- **Id** – unique **number** for every **transaction**. (**auto-incremented**, **primary key**)
- **AccountId** – references the **Accounts** table (**foreign key**)
- **OldBalance** – the balance **before** the transaction
- **NewBalance** – the balance **after** the transaction
- **Amount** – the amount transferred (**calculated** column)
- **DateTime** – the date and time of the transaction (as **datetime2** data type)

Let's create the **table**:

```
CREATE TABLE Transactions (
   Id INT PRIMARY KEY IDENTITY,
   AccountId INT FOREIGN KEY REFERENCES Accounts(Id),
   OldBalance DECIMAL(15, 2) NOT NULL,
   NewBalance DECIMAL(15, 2) NOT NULL,
   Amount AS NewBalance - OldBalance,
   [DateTime] DATETIME2
)
```

Now we can create our **trigger**, which will run whenever the **Accounts** table is **updated** by the **procedures** (or regular **UPDATE** statements), like this:

```
CREATE TRIGGER tr_Transaction ON Accounts
AFTER UPDATE
AS
   INSERT INTO Transactions (AccountId, OldBalance, NewBalance, [DateTime])
   SELECT inserted.Id, deleted.Balance, inserted.Balance, GETDATE() FROM inserted
   JOIN deleted ON inserted.Id = deleted.Id
```

We used the **built in deleted** and **inserted** tables to get the **OldBalance** and **NewBalance** values.

Next, let's do some **transactions**, which should **run** our **trigger**:

```
p_Deposit 1, 25.00
GO

p_Deposit 1, 40.00
GO

p_Withdraw 2, 200.00
GO

p_Deposit 4, 180.00
GO
```

And finally, let's take a look at our **Transactions table** to make sure our **trigger** is working:

```
SELECT * FROM Transactions
```

The result should be something like this:

Id	AccountId	OldBalance	NewBalance	Amount	Time
1	1	175.00	200.00	25.00	2017-09-14 00:38:48.9833333
2	1	200.00	240.00	40.00	2017-09-14 00:38:48.9833333
3	2	275.56	75.56	-200.00	2017-09-14 00:38:48.9866667
4	4	40.30	220.30	180.00	2017-09-14 00:38:48.9866667