# Lab: Intro to Databases

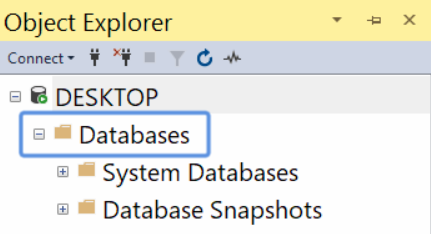
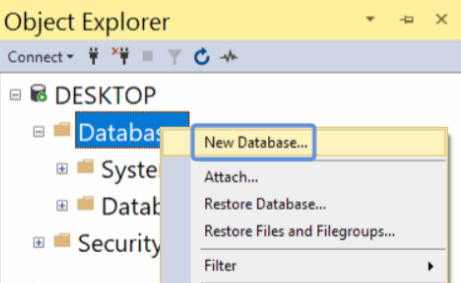
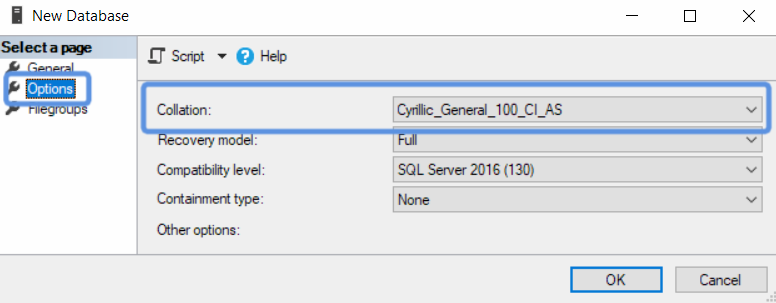
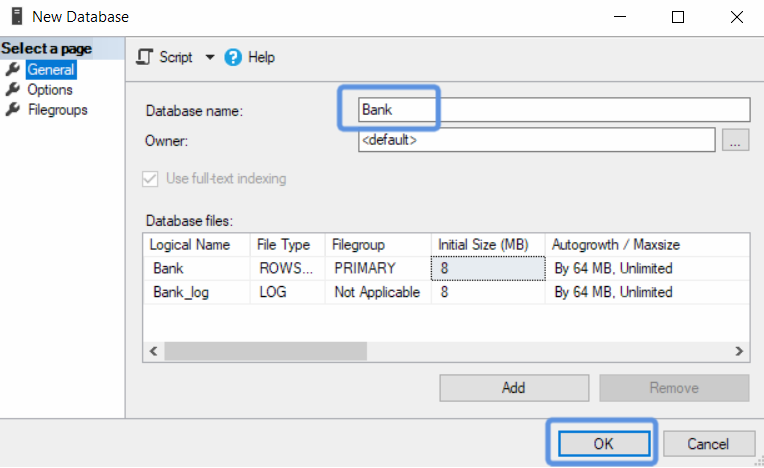
In this lab, we will create a **“Bank” database** in **SQL Server**, using **MS SQL Server Management Studio**. We will create **tables** and **fill them with data**, create **views**, **functions**, **procedures** and **triggers**.

Before starting this tutorial, make sure you’ve followed the youtube videos I provided last week

(See database channel in teams).

## Create a Database

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

1. Right click on **Databases** in the **Object Explorer**:  
   
2. Choose **New Database** fromthe drop-down menu:  
   
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**]:  
   

## Create Tables

NOTE: if you get any errors while typing your queries, try this:



1. Using an **SQL query**,create table Clientswith 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  ) |

1. Create table AccountTypewith:

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

***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  ) |

1. Create table **Accounts** with:

* Id - **unique number** for every user. (**Auto incremented**, **primary key**)
* AccountTypeId– references the **AccountTypes** table (**foreign key**)
* Balance – decimaldata 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)  ) |

## Insert Example Data into our Database

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

|  |
| --- |
| INSERT INTO Clients (FirstName, LastName) VALUES  ('Greta', 'Andersson'),  ('Peter', 'Pettersson'),  ('Mel', 'Gibson'),  ('Maria', 'Danielsson')  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) |

## Create a simple View

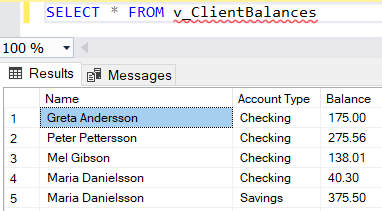
Next, we’re going to create a **view**, which displays the **owner**, **type** and **balance** of **each account** in our database:

|  |
| --- |
| CREATE VIEW v\_ClientBalances AS  SELECT (FirstName + ' ' + LastName) AS [Name],  (AccountTypes.Name) AS [Account Type], Balance  FROM Clients  JOIN Accounts ON Clients.Id = Accounts.ClientId  JOIN AccountTypes ON AccountTypes.Id = Accounts.AccountTypeId |

We concatenated FirstNameand LastNameinto a **new column**, called **Name** and took the **Account Type** and

**Balance** for **every account** by **joining** the tables Clients**,** Accounts and AccountTypes to access **data from all of them**. Joining tables will be explained in more detail later into our course.

Now, if we **select** everything (\*) from this view we will receive this:



## 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**.

## 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 SELECTstatement:

|  |
| --- |
| 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 |

## 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 andinserted 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:

