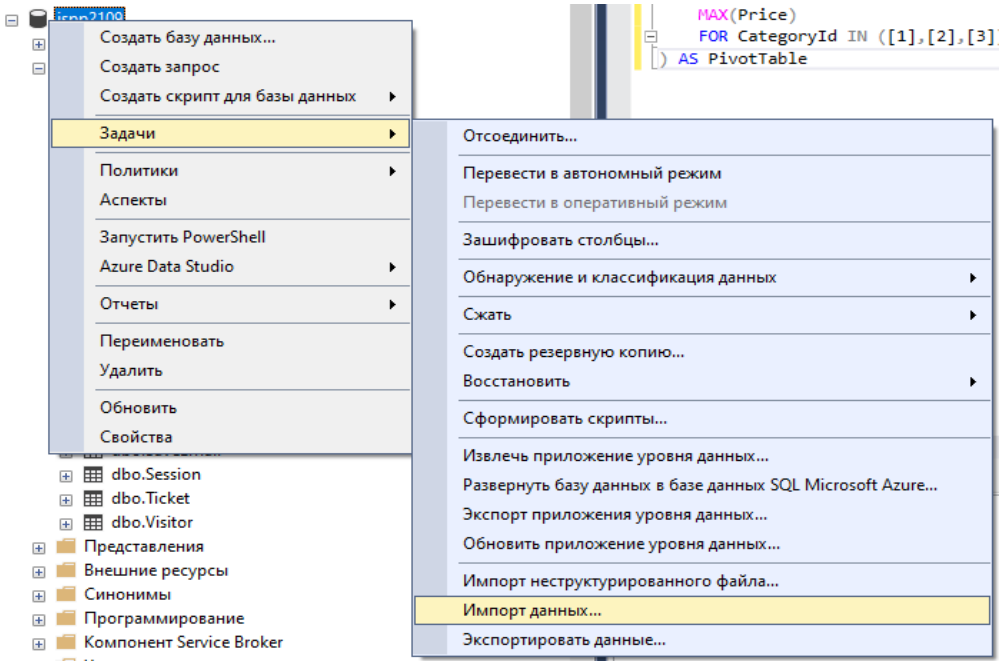
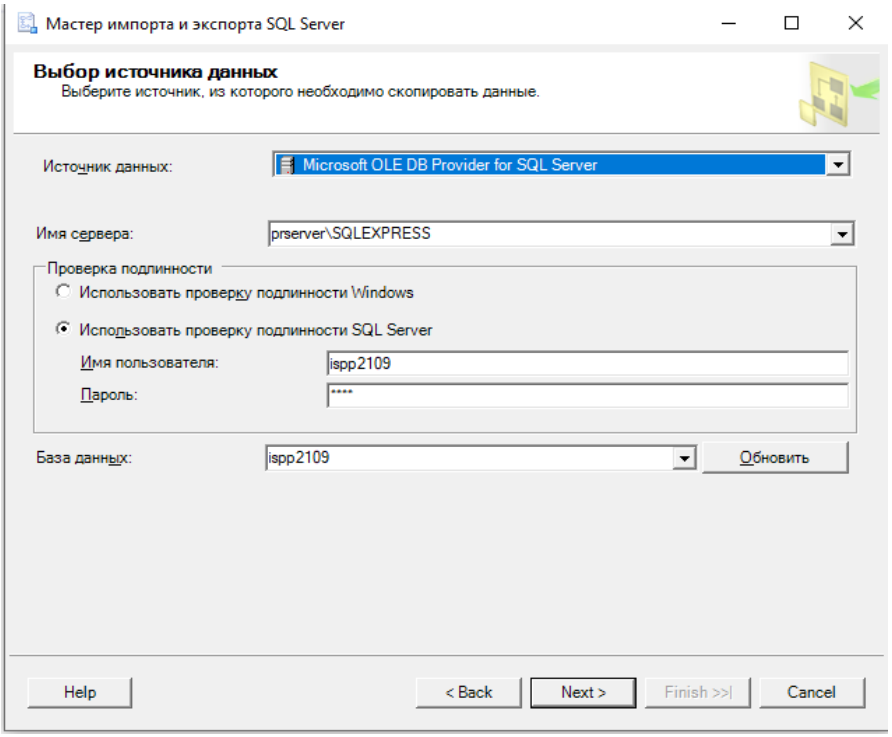
ИМПОРТ ДАННЫХ

Экспорт – сохранение файлов DB во внешние файлы
Импорт – это загрузка банных в DB из внешнего файла



При экспорте – сервер источник
При импорте – сервер приёмник
Для экспорта



Стандартные типы фалов для экспорта и импорта:

- Excel (xls/xlsx)
- Flat File (csv/tsv/txt)
- XML
- Json

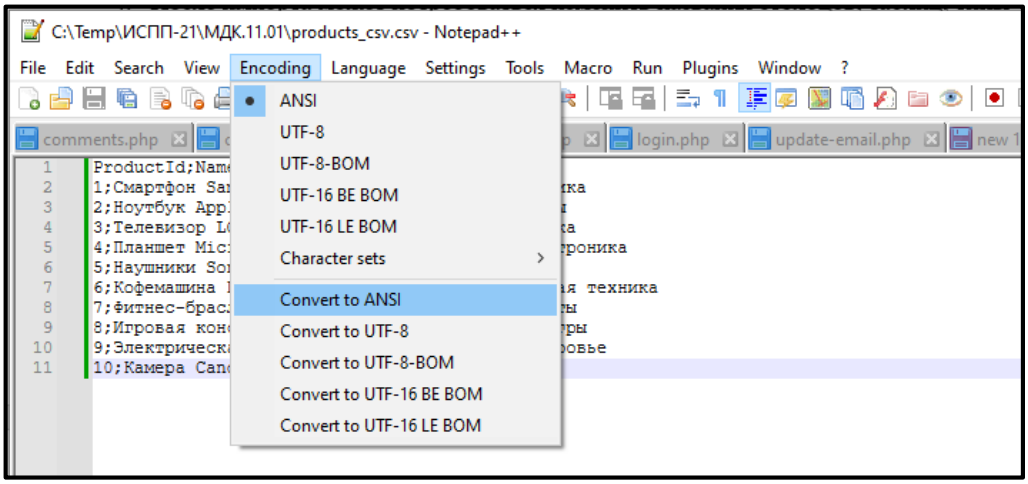
Excel

1. Данные каждой таблицы на отдельном листе

2. xls – формат 97-2003
3. xlsx – формат 2007-…
   Если файл для импорта в формате xlsx то лучше его пересохранить в xls
4. После импорта может потребоваться изменить названия столбцов, таблиц, типы данных, и обязательность, добавить PK и FK

Flat File – таблица, которая хранится в текстовом формате. Может быть открыт в табличном редакторе.

1. csv (comma separated values)
2. tsv (tab separated values)
3. txt
   Символы разделители: , ; пробел / (табуляция)
   Разделители строк: (Enter) \n
   Ограничители текста: ‘текст’ “текст”
4. При ошибке загрузки строковых данных надо поменять с DT_STR на DT_TEXT
5. Смена кодировки выполняется в NOTEPAD++ или Блокнот



6. Типы данных лучше настраивать в проекте
7. Можно пересохранить csv как xls

XML – расширяемый язык разметки. Можно создавать свои теги и атрибуты.
Особенности:

- Похож на XAML, HTML
- ROOT – корневой тег
  <тег1>…</тег1>
  <тег2/>
- В открывающемся теге могут быть указаны атрибуты
  <тег1 атр1= «знач1» атр2= «знач2»...>
- Названия тегов и атрибутов не могут содержать пробелов и спец символы
- Между тегами могут быть значения и другие теги

Пример экспорта XML файла:

Общая форма:

```
SELECT *
FROM Таблица
FOR XML PATH('Название узла/строки'), ROOT('Название корневого узла/таблица')
столбец AS '@имяАтрибута'
XML PATH
```

Пример:

```
SELECT *
FROM Game
FOR XML PATH('Game'), ROOT('Games')
```

Пример импорта с помощью XML:

1. Объявление переменной
```
DECLARE @xml XML = '...';
```
2. Общая формула:
```
SELECT
    Объект.value('Имя тэга столбца[1]', 'тип данных') AS ProductId,
    ...
FROM @xml.nodes('/Тэг корневой/Тэг объекта') AS Таблица(Объект);
```
3. Пример:
```
DECLARE @xml XML = '
<Products>
    <Product>
        <ProductId>1</ProductId>
        <Name>Смартфон Samsung Galaxy S21</Name>
        <Price>699.99</Price>
        <Category>Электроника</Category>
    </Product>
    <Product>
        <ProductId>2</ProductId>
        <Name>Ноутбук Apple MacBook Pro</Name>
        <Price>1299.99</Price>
        <Category>Компьютеры</Category>
    </Product>
</Products>';

SELECT
    product.value('ProductId[1]', 'INT') AS ProductId,
    product.value('Name[1]', 'NVARCHAR(100)') AS Name,
    product.value('Price[1]', 'DECIMAL(10,2)') AS Price,
    product.value('Category[1]', 'NVARCHAR(100)') AS Category
FROM @xml.nodes('/Products/Product') AS t(product)
```
4. Пример использования атрибута
```
/*SELECT
    Объект.value('Имя тэга столбца[1]', 'тип данных') AS Столбец,
    ...
    Объект.value('../@атрибут родительского тэга', 'тип данных') AS Столбец,
    ...
FROM @xml.nodes('/Тэг корневой/Тэг родительский/Тэг объекта') AS Таблица(Объект);*/
```

```
DECLARE @xml XML = '
<Products>
     <Category name="Электроника">
        <Product attr="1">
            <ProductId>1</ProductId>
            <Name>Смартфон Samsung Galaxy S21</Name>
            <Price>699.99</Price>
        </Product>
        <Product>
            <ProductId>4</ProductId>
            <Name>Планшет Microsoft Surface Pro 7</Name>
            <Price>799.99</Price>
        </Product>
    </Category>
    <Category name="Компьютеры">
        <Product>
            <ProductId>2</ProductId>
            <Name>Ноутбук Apple MacBook Pro</Name>
            <Price>1299.99</Price>
        </Product>
    </Category>
</Products>';

SELECT
    product.value('ProductId[1]', 'INT') AS ProductId,
    product.value('Name[1]', 'NVARCHAR(100)') AS Name,
    product.value('Price[1]', 'DECIMAL(10,2)') AS Price,
    product.value('@attr', 'INT') AS attr,
    category.value('@name', 'NVARCHAR(100)') AS Category
FROM @xml.nodes('/Products/Category') AS t1(category)
    CROSS APPLY t1.category.nodes('Product') AS t2(product)
```

```
SELECT
    C.value('ProductId[1]', 'INT') AS ProductId,
    C.value('Name[1]', 'NVARCHAR(100)') AS Name,
    C.value('Price[1]', 'DECIMAL(10, 2)') AS Price,
    C.value('@attr', 'INT') AS attr,
    C.value('../@name', 'NVARCHAR(100)') AS Category
FROM @xml.nodes('/Products/Category/Product') AS T(C);
```

Пример с использованием CROSS APPLY

```
/*SELECT
    ДочернийОбъект.value('Имя тэга столбца[1]', 'тип данных') AS Столбец,
    ...
    РодительскийОбъект.value('@атрибут родительского тэга', 'тип данных') AS Столбец,
    ...
FROM @xml.nodes('/Тэг корневой/Тэг родительский') AS Таблица1(РодительскийОбъект)
    CROSS APPLY Таблица1.РодительскийОбъект.nodes('Тэг дочерний') AS Таблица2(ДочернийОбъект)*/
```

Пример с разными названиями родительских тегов:

```
DECLARE @xml XML = '
<Products>
     <Электроника>
        <Product>
            <ProductId>1</ProductId>
            <Name>Смартфон Samsung Galaxy S21</Name>
            <Price>699.99</Price>
        </Product>
        <Product>
            <ProductId>4</ProductId>
            <Name>Планшет Microsoft Surface Pro 7</Name>
            <Price>799.99</Price>
        </Product>
    </Электроника>
    <Компьютеры>
        <Product>
            <ProductId>2</ProductId>
            <Name>Ноутбук Apple MacBook Pro</Name>
            <Price>1299.99</Price>
        </Product>
    </Компьютеры>
</Products>';
```

Вместо названия тега используется **local-name(..)**

```
SELECT
    product.value('ProductId[1]', 'INT') AS ProductId,
    product.value('Name[1]', 'NVARCHAR(100)') AS Name,
    product.value('Price[1]', 'DECIMAL(10,2)') AS Price,
    product.value('@attr', 'INT') AS attr,
    product.value('local-name(..)', 'NVARCHAR(100)') AS Category
FROM @xml.nodes('/Products/*/Product') AS t1(product);
```

Вместо родительского тега пишется *
Для считывания можно использовать

```
sp_xml_preparedocument

DECLARE @hDoc INT; -- handle
EXEC sp_xml_preparedocument @hDoc OUTPUT, @xml;

SELECT *
FROM OPENXML(@hDoc, '/Products/Category/Product')
WITH
(
    ProductId INT 'ProductId',
    Name NVARCHAR(100) 'Name',
    Price DECIMAL(10, 2) 'Price',
    Category NVARCHAR(100) '../@name'
);
```

Json – для представления данных объекта в виде текстового файла
Данные в квадратных скобках - это массив
Пример заполнения файла

```
][
]    {
        "ProductId": 1,
        "Name": "Смартфон Samsung Galaxy S21",
        "Price": 699.99,
        "Category": "Электроника"
    },
]    {
        "ProductId": 2,
        "Name": "Ноутбук Apple MacBook Pro",
        "Price": 1299.99,
        "Category": "Компьютеры"
    },
-]
```

В роли значения может массив или другой объект

Пример

```
SELECT *
FROM Game
FOR JSON PATH, INCLUDE_NULL_VALUES
```

Для импорта используется OPENJSON
Чтоб получить доступ к значению

```
'$.атрибут'
```

```
DECLARE @json NVARCHAR(MAX) = '
[
    {
        "ProductId": 1,
        "Name": "Смартфон Samsung Galaxy S21",
        "Price": 699.99,
        "Category": "Электроника"
    },
    {
        "ProductId": 2,
        "Name": "Ноутбук Apple MacBook Pro",
        "Price": 1299.99,
        "Category": "Компьютеры"
    },
    {
        "ProductId": 3,
        "Name": "Телевизор LG OLED55CX",
        "Price": 1499.99,
        "Category": "Бытовая техника"
    },
```

```sql
SELECT *
FROM OPENJSON(@json)
WITH
(
    ProductId INT '$.ProductId',
    Name NVARCHAR(100) '$.Name',
    Price DECIMAL(10, 2) '$.Price',
    Category NVARCHAR(100) '$.Category'
);
```

OPENJSON – позволяет использовать данные json формата как источник данных.

```sql
OPENJSON(данные  json)
```

WITH() – позволяет описать структуру данных

```sql
WITH
(
 ИмяСтолбца ТипДанных '$путькданным'
)

$.свойствоОбъекта
$.вложенныйОбъект.свойствоВложенногоОбъекта
```

JSON_VALUE() – возвращает в виде строки значение

```sql
JSON_VALUE(данные  json, '$путь') AS ИмяСтолбца
```

JSON_QUERY() – возвращает данные объекта или массива

```sql
JSON_QUERY(данные  json, '$путь') AS ИмяСтолбцаJson
```

Если у объекта одно из значений массив, то надо использовать CROSS APPLY

```sql
--1 Пример вывода данных используя OPENJSON
SELECT *
FROM OPENJSON(@json)


--2 Пример вывода данных используя WITH
SELECT *
FROM OPENJSON(@json)
WITH
(
    ProductId INT '$.ProductId',
    Name NVARCHAR(50) '$.Name',
    dimensions NVARCHAR(50) '$.dimensions.width'

);
```

```sql
--3 Пример вывода данных используя JSON_VALUE
SELECT
    JSON_VALUE(value, '$.ProductId') AS ProductId,
    JSON_VALUE(value, '$.Name') AS Name
FROM OPENJSON(@json)

--4 Пример вывода данных используя JSON_VALUE, JSON_QUERY
SELECT
    JSON_VALUE(value, '$.ProductId') AS ProductId,
    JSON_VALUE(value, '$.Name') AS Name,
    JSON_QUERY(value, '$.dimensions') AS dimensionsJson,
    JSON_VALUE(value, '$.dimensions.width') AS width,
    JSON_VALUE(value, '$.dimensions.height') AS height,
    JSON_VALUE(value, '$.dimensions.depth') AS depth
FROM OPENJSON(@json)

--5 Пример вывода данных с массивом CROSS APPLY
SELECT
    JSON_VALUE(category.value, '$.CategoryName') AS CategoryName,
    JSON_VALUE(product.value, '$.ProductId') AS ProductId,
    JSON_VALUE(product.value, '$.Name') AS Name
FROM OPENJSON(@json) AS category
    CROSS APPLY
OPENJSON(category.value, '$.Products') AS product;

--6 Пример вывода данных с массивом CROSS APPLY при помощи WITH
SELECT CategoryName, ProductId, Name
FROM OPENJSON(@json)
    WITH
    (
        CategoryName NVARCHAR(50) '$.CategoryName',
        jsonArray NVARCHAR(MAX) '$.Products' AS JSON
    )
    CROSS APPLY
        OPENJSON(jsonArray)
        WITH
        (
            ProductId INT '$.ProductId',
            Name NVARCHAR(50) '$.Name'
        );
```

ISJSON() – позволяет проверить что данные в столбце в формате json

```sql
ISJSON(столбец) > 0
```

Если в столбце хранятся данные в формате json то столбец должен быть типа NVARCHAR(MAX)

```sql
--7  Пример проверик данных на формат Json
SELECT *, ISJSON(jsonArray), ISJSON(CategoryName)
FROM OPENJSON(@json)
    WITH
    (
        CategoryName NVARCHAR(50) '$.CategoryName',
        jsonArray NVARCHAR(MAX) '$.Products' AS JSON
    )
    CROSS APPLY
        OPENJSON(jsonArray)
        WITH
        (
            ProductId INT '$.ProductId',
            Name NVARCHAR(50) '$.Name'
        );
```

Результаты | Сообщения

CategoryName	jsonArray	ProductId	Name	(Отсутствует имя столбца)	(Отсутствует имя столбца)
Электроника	[{ "ProductId": 1, "Name": "Смарт...	1	Смартфон Samsung Galaxy S21	1	0
Электроника	[{ "ProductId": 1, "Name": "Смарт...	4	Планшет Microsoft Surface Pro 7	1	0
Компьютеры	[{ "ProductId": 2, "Name": "Ноутбу...	2	Ноутбук Apple MacBook Pro	1	0

В MySQL Workbench доступен экспорт и импорт в формате json и csv

Импорт данных M:M

Структура файла txt

```
PhotoId;FileName;Tags
1;1.jpg;summer,sea
2;2.png;vacation,sea
3;3.png;
4;4.jpg;summer
```

```sql
--Создаем таблицу Tags
CREATE TABLE Tag
(
    TagId INT IDENTITY NOT NULL,
    TagName VARCHAR(50) NOT NULL,
    PRIMARY KEY (TagId)
)
INSERT INTO Tag(TagName) -- вставка в существующую таблицу
SELECT DISTINCT TRIM(value)
FROM photoTags
    CROSS APPLY STRING_SPLIT(tags, ',')
WHERE TRIM(value) <> ''
```

У данных есть Id поэтому таблицу можно создать в SELECT

```sql
--Создаем таблицу Photo
SELECT PhotoId, FileName
INTO Photo --создали новую таблицу(нужно добавить в неё PK)
FROM photoTags
```

```sql
--создаем таблицу PhotoTag (PhotoId, TagId)
SELECT PhotoId, TagId
INTO PhotoTag -- надо добавить PK(PhotoId, TagId) и FK(PhotoId, TagId)
FROM Tag JOIN (
    SELECT PhotoId, TRIM(value) AS Tag
    FROM photoTags
        CROSS APPLY STRING_SPLIT(tags, ',')
    ) AS t ON Tag.TagName = t.Tag;
```

В DB можно хранить путь к файлу или сам файл в формате VARBINARY(MAX) / BLOB

Хранение в DB:
+ За сохранность файла отвечает СУБД
- Растет размер DB и резервной копии

Хранение файлов отдельно:
+ DB весит меньше
+ Можно легко открыть файл
- За хранение файлов отвечает Файловая Система
Файл загружается как массив byte[]

```
byte[] fileData = file.ReadAllBytes("путь к файлу");
if (fileData.Length < 10240) // до 10 КБ
{
    // использование fileData
}
```