# Creating a Web API with SQL backing store

## Lab Overview

In this lab session, you will create a local SQL database, populate it with a dataset sourced from the internet, before writing a simple data access layer and exposing the data over a HTTP-enabled API.

## Prerequisites

* Visual Studio 2015, SQL LocalDB

## Time Estimate

* 30 minutes

## Table of Contents

[Creating a Web API with SQL backing store 1](#_Toc438555085)

[Lab Overview 1](#_Toc438555086)

[Prerequisites 1](#_Toc438555087)

[Time Estimate 1](#_Toc438555088)

[Table of Contents 1](#_Toc438555089)

[Exercise 1: Importing data 2](#_Toc438555090)

[Exercise 2: Querying the DB in a data access layer 2](#_Toc438555091)

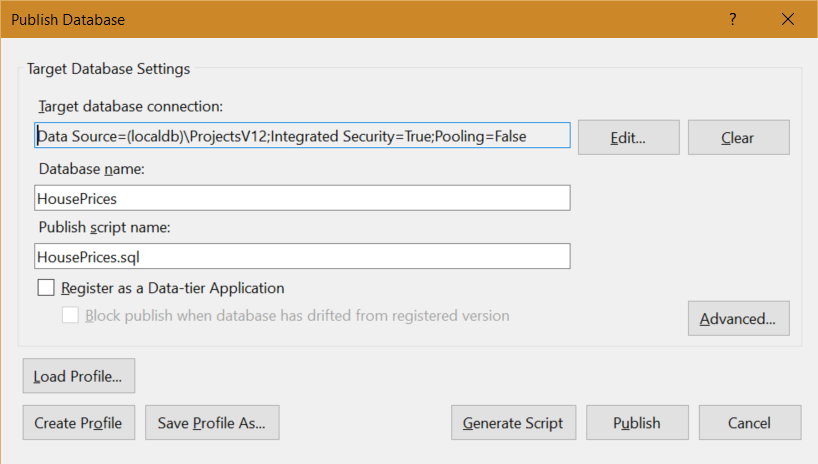
[Exercise 3: Hook the DAL into the web layer. 3](#_Toc438555092)

[Summary 3](#_Toc438555093)

## Exercise 1: Importing data

In this exercise, we will quickly import an external data set from the internet into a local SQL database. If you do not have a version of SQL on your machine, you can quickly install SQL LocalDB (<https://msdn.microsoft.com/en-us/library/hh510202.aspx>).

1. Open the WebDemo solution.
2. Build the solution, ensuring that it builds. Note that we are using Paket rather than NuGet in this solution; we’ve added a custom msbuild target so that it will download the appropriate NuGet packages on building the solution, but you can also install the **Paket Visual Studio** extension if you wish from the Visual Studio Gallery (<https://visualstudiogallery.msdn.microsoft.com/>) to give you greater control and see details from Paket in the Output window.
3. Publish the HousePrices database project to your LocalDB installation by choosing the “Publish” option when right-clicking the project.



1. Open the DataImport.fsx from the Scripts folder.
2. Execute the first part which downloads a data file in CSV format from the UK Land Registry.
3. Once it has downloaded, close and re-open the script.
4. Execute the code for stages 2 and 3. These will connect to the CSV file and your SQL database using type providers. Ensure that you have updated the SQL connection string as appropriate.
5. Execute stage 4. This will convert every row from the CSV file into a data row to insert on the DB, and then perform a bulk insert to save all rows to the DB. Depending on your machine, this may take a few seconds to complete!

Notice how the CSV file does not contain row headers. In such a case, we can simply supply the column headers ourselves inside the CSV Type Provider declaration.

## Exercise 2: Querying the DB in a data access layer

In this exercise, we will quickly create some simple wrapper functions that will encapsulate the use of the type providers and return some simple data structures. We will implement this code in the Suave project, but if you prefer, you can do it within the WebAPI one – the implementation for the controller method will be similar to what will happen later in this example.

1. Open the DataAccess.fs file in the Suave folder.
2. Update the DbConnection literal to point to your database.
3. Amend the FindByTownPrice query to select the Price, DateOfTransfer, Street, TownCity and PostCode columns from the top 50 rows from the dbo.PropertySale table ordered descending by Price.
4. Add parameterized filters on the SQL query to search by TownCity and between a min and max Price.

To create parameters within SQL queries for the SQL Type Provider, use @Param e.g. WHERE Price > @MaxPrice. This will be exposed within the generated methods that the TP creates.

## Exercise 3: Hook the DAL into the web layer.

In this exercise, we will tie in our DAL to the Web layer in order to return JSON data to the caller.

If you preferred to use WebAPI, you will need to create a new controller method with a route taking in the Town, Min Price and Max Price, and then port the record and findHouseSales function from below. Remember to convert the result from Async<T> to Task<T>!

1. Go to the Program.fs and find the findHouseSales function.
2. You will need to update the callsite to FindByTownPrice’s AsyncExecute. If your DAL was written correctly, it should now need three arguments – the town, min price and max price. Populate them from the arguments supplied on the function.
3. Update the mapping code in that function to correctly map from the provided type returned by the DAL, to the HousePriceResult record type defined above.

In F#, types generated by a Type Provider are known as *Provided Types*. Provided Types are only available within F#, *and only at compile time*. This means that (a) we cannot reflect over them, and (b) that they are unsuitable for exposing to cross-platform APIs e.g. web layers. In this case, it means we create a formal record type which is used as an explicit contract to expose to the outside world, and we map between the generated type and this contract.

1. Notice the path in the handleRoute function below, which uses this findHouseSales function in conjunction with a simple helper method that “lifts” the output into a WebPart function, and at the same time converting the output into JSON.
2. Launch the Suave application. In your browser navigate to localhost:8083/housePrice/London/500000/515000.
3. You should receive a number of results in JSON.
4. Try setting breakpoints to walk through the code and see what is happening.
5. Try changing the route definition using pathScan – notice that the %s %d formats etc. are strongly typed and will not compile if the signature of the findHouseSales does not match.

## Summary

In this lab, you rapidly constructed a SQL database using publicly available data using a combination of type providers. Then, you created a strongly-typed query to that data to search for specific data. Finally, you connected this to a web layer in order to expose the data to consumers.