Saturn - Data Access

Table of Contents

Introduction

Saturn makes use of the Data Access Application Block included in the Microsoft patterns & practices Enterprise Library 5.0. Full class library documentation is available here and guidance and tutorials are available here.

Saturn encapsulates DAAB interaction in repositories. See Martin Fowler's Patterns of Enterprise Application Architecture and MSDN's The Repository Pattern for more information.

The DAAB includes support for both stored procedures and inline SQL statements but Saturn repositories only interact with stored procedures. They do so through the SprocAccessor class which is explained in the Returning Data as Objects for Client Side Querying section of the guidance.

How to use SprocAccessor

Using SprocAccessor is simple. There are 6 built-in extension methods that allow you to call a stored procedure in one line of code:

```
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, params object[] parameterValues)
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, IParameterMapper parameterMapper, params object[]
parameterValues)
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, IRowMapper<TResult> rowMapper, params object[]
parameterValues)
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, IParameterMapper parameterMapper,
IRowMapper<TResult> rowMapper, params object[] parameterValues)
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, IResultSetMapper<TResult> resultSetMapper, params
object[] parameterValues)
    IEnumerable<TResult> ExecuteSprocAccessor<TResult>(this Database database, string procedureName, IParameterMapper parameterMapper,
IResultSetMapper<TResult> resultSetMapper, params object[] parameterValues)
```

In the simplest case, you only need to provide the stored procedure name:

```
    public List<SpotType> GetAll()
    {
        return Database.ExecuteSprocAccessor<SpotType>("dbo.SpotType_GetAll").ToList();
    }
```

Note that the `ExecuteSprocAccessor` methods return an `IEnumerable` which will not execute against the database until is is enumerated. You should explicitly enumerate it (e.g. with a `for` loop or a call to `ToList` before returning from your repository. This will also automatically close the underlying `IDataReader` and `DbConnection`.

Parameters are provided to the stored procedure in the `parameterValues` params list and are automatically mapped in order to the parameters of the stored procedure:

```
    public List<Spot> GetByScheduledProgramId(int scheduledProgramId)
    {
        return Database.ExecuteSprocAccessor<Spot>("dbo.Spot_GetByScheduledProgramId", scheduledProgramId).ToList();
    }
```

More complex parameter mappings, such as when you have an entity and need to map each parameter to a property of the entity, can be achieved using an `IParameterMapper` - see Defining Parameter Mappers for more information or review `Nine.Saturn.Services.Core.Infrastructure.SaveEntityParameterMapper<TEntity>` for an example.

Mapping the results

NOTE: If your proc result set and object members match then you don't need to map. Mapping will happen automatically.

The records returned from your stored procedure will be returned as a new instance of `TResult` and you can control this mapping using either an `IRowMapper` or `IResultSetMapper` (Building Output Mappers).
The default `IRowMapper` that will be used if you don't specify one will map each property on the entity to a column in the record with the same name as the property. If no column is found with the same name as the property, an exception will be thrown and you will have to define an `IRowMapper` to instruct the `SprocAccessor` how to map that property (or to ignore it).

IRowMapper

An `IRowMapper` is the safest and easiest way to provide custom column mappings from your `SprocAccessor` results to `C#` classes. The fluent interface on `MapBuilder<TResult>` quickly creates an `IRowMapper` for you. Start with `MapAllProperties` or `MapNoProperties` and add mappings for specific properties, before finishing with a call to `Build`. `MapAllProperties` sets up mappings for all properties of `TResult` and allows you to override some or all of them, whereas `MapNoProperties` starts with a blank slate. If you design your entities and stored procedures sensibly, `MapAllProperties` should take care of most of your mappings for you, leaving only a few custom mappings to provide:

```
    IRowMapper<Spot> spotRowMapper = MapBuilder<Spot>.MapAllProperties()
            .DoNotMap(s => s.TimeStamp)
            .Map(s => s.Id).ToColumn("SpotId")
            .Map(s => s.Status).ToColumnAsEnum<Spot, SpotStatus>("StatusId")
            .Map(s => s.RequestedPosition).ToColumnAsEnum("RequestedPositionId")
            .Map(s => s.Type).WithFunc(row => _spotTypeRepository.GetOne(st => st.Code == (string) row["TypeCode"]))
            .Build();
```

Once set up you pass the `IRowMapper` to your `ExecuteSprocAccessor` call and it takes care of the rest:

```
List<Spot> spots = Database.ExecuteSprocAccessor<Spot>("dbo.Spot_GetByBookingId", spotRowMapper, bookingId).ToList();
```

IResultSetMapper

NOTE: Please avoid this technique unless you really do require it.

An `IResultSetMapper` should be used **only** when you want finer-grained control over the construction of your `TResult` entities or if your entity does not map one-one with a record in your results (e.g. if you are constructing an aggregate root). To do this you need a class which implements `IResultSetMapper<TResult>` which will require a method `IEnumerable<TReuslt> MapSet(IDataReader)`. In this method, you will have to iterate over the reader and construct your results.

Note: don't forget to close the `IDataReader` when you are done with it!

```
public List<Booking> GetBookings()
{
    return Database.ExecuteSprocAccessor<Booking>("Booking_GetAll", new BookingResultSetMapper()).ToList();
}

private class BookingResultSetMapper : IResultSetMapper<Booking>
{
    public IEnumerable<Booking> MapSet(IDataReader reader)
    {
        List<Booking> bookings = new List<Booking>();
        Booking booking = null;
        using (reader)
        {
            while (reader.Read())
            {
                int bookingId = (int) reader["BookingId"];
                if (booking == null || booking.Id != bookingId)
                {
                    booking = new Booking() { Id = bookingId, Spots = new List<Spot>() };
                    bookings.Add(booking);
                }
                booking.Spots.Add(new Spot()
                                 {
                                     Id = (int) reader["SpotId"],
                                     Status = (SpotStatus) ((int) reader["StatusId"])
                                 });
            }
        }
        return bookings;
    }
}
```

If you want to get really creative, the two techniques can be combined:

```
private class BookingResultSetMapper : IResultSetMapper<Booking>
{
    private readonly IRowMapper<Booking> _bookingRowMapper;
    private readonly IRowMapper<Spot> _spotRowMapper;

    public BookingResultSetMapper(IRowMapper<Booking> bookingRowMapper, IRowMapper<Spot> spotRowMapper)
    {
        _bookingRowMapper = bookingRowMapper;
        _spotRowMapper = spotRowMapper;
    }

    public IEnumerable<Booking> MapSet(IDataReader reader)
    {
        List<Booking> bookings = new List<Booking>();
        Booking booking = null;
        using (reader)
        {
            while (reader.Read())
            {
                int bookingId = (int) reader["BookingId"];
                if (booking == null || booking.Id != bookingId)
                {
                    booking = _bookingRowMapper.MapRow(reader);
                    bookings.Add(booking);
                }
                booking.Spots.Add(_spotRowMapper.MapRow(reader));
            }
        }
        return bookings;
    }
}
```

Shortcomings of the DAAB

The SprocAccessor has few shortcomings that have had to be worked around in order to get the functionality we want in Saturn:

- SprocAccessor does not allow you to set the CommandTimeout of the DbCommand used to execute the stored procedure.
- When a SprocAccessor is called from the IRowMapper being used by another SprocAccessor, an `InvalidOperationException` (There is already an open DataReader associated with this Command which must be closed first) is thrown.

Setting the CommandTimeout

The `DatabaseExtensionMethods` class in `Nine.Saturn.Services.Core.Infrastructure` defines some additional overloads for the 6 built-in `ExecuteSprocAccessor` extension methods that allow you to specify the `CommandTimeout` for a stored procedure call. It's as simple as updating your `ExecuteSprocAccessor` call to:

```
public IList<SpotType> GetAll()
```

```
    {
        return Database.ExecuteSprocAccessor<SpotType>(Config.CommandTimeout, "dbo.SpotType_GetAll").ToList();
    }
```

Calling a SprocAccessor from an IRowMapper (This section is under review - Stonie)

NOTE: this situation is not a straightforward as it appears. Will be addressed by PBI 5303.

This scenario arises when you need to map a property of an object to an instance from another repository which executes against the same database:

```
    public IList<Spot> GetByScheduledProgramId(int scheduledProgramId)
    {
        return Database.ExecuteSprocAccessor<Spot>("dbo.Spot_GetByScheduledProgramId", GetRowMapper(), scheduledProgramId).ToList();
    }

    private IRowMapper<Spot> GetRowMapper()
    {
        return MapBuilder<Spot>.MapAllProperties()
            .Map(s => s.Type).WithFunc(row => _spotTypeRepository.GetOne(st => st.Code == (string) row["TypeCode"]))
            .Build();
    }
```

An `InvalidOperationException` will be thrown because the call to `_spotTypeRepository.GetOne` is executed while the `IDataReader` created by `ExecuteSprocAccessor("dbo.Spot_GetByScheduledProgramId")` is still enumerating (and therefore not closed). The fix for this is not as straightforward as the `CommandTimeout` fix and requires a few more steps. First you need to change your `.config` file Autofac registration of the database component from:

```
    <component type="Microsoft.Practices.EnterpriseLibrary.Data.Sql.SqlDatabase, Microsoft.Practices.EnterpriseLibrary.Data"
               service="Microsoft.Practices.EnterpriseLibrary.Data.Database, Microsoft.Practices.EnterpriseLibrary.Data"
               instance-scope="single-instance" name="SaturnCampaignsService">
```

To:

```
    <!-- Use the SaturnSqlDatabase rather than the SqlDatabase to allow commands to be forcibly executed on new connections -->
    <component type="Nine.Saturn.Services.Core.Infrastructure.Data.SaturnSqlDatabase, Nine.Saturn.Services.Core"
             service="Microsoft.Practices.EnterpriseLibrary.Data.Database, Microsoft.Practices.EnterpriseLibrary.Data"
             instance-scope="single-instance" name="SaturnCampaignsService">
```

Next, on the **inner** repository (the one called **from** the IRowMapper) you need to add this to your `ExecuteSprocAccessor` call:

```
    public IList<SpotType> GetAll()
    {
        return Database.ExecuteSprocAccessor<SpotType>(ForceNewConnection, "dbo.SpotType_GetAll").ToList();
    }
```

`ForceNewConnection` is a public property on `SqlRepository` which your repository should already inherit from. The final step is to ensure that `ForceNewConnection` is set to true at the correct time (before the `IRowMapper` is used in the **outer** repository):

```
    public IList<Spot> GetByScheduledProgramId(int scheduledProgramId)
    {
        return Database.ExecuteSprocAccessor<Spot>("dbo.Spot_GetByScheduledProgramId", GetRowMapper(), scheduledProgramId).ToList();
    }

    private IRowMapper<Spot> GetRowMapper()
    {
        // If _spotTypeRepository allows it, force it to execute it's Get on a new connection
        SqlRepository sqlRepository = _spotTypeRepository as SqlRepository;
        if (sqlRepository != null)
        {
            sqlRepository.ForceNewConnection = true;
        }
        return MapBuilder<Spot>.MapAllProperties()
            .Map(s => s.Type).WithFunc(row => _spotTypeRepository.GetOne(st => st.Code == (string) row["TypeCode"]))
            .Build();
    }
```

Other Important Points

Case Sensitivity

Since parameter mappers and row mappers use reflection to map parameter names and result set field names to CLR properties, case sensitivity is important.

Using a camel-case parameter name or result set field name, although case-insensitive in SQL, will result in a failure to map to the equivalent Pascal-cased C# class property.

Maintaining this case-sensitivity for mapping resolution is important. It is valid for a C# class to define properties that differ only by case (even by accident) e.g. `Id` vs `ID`. If the mapper was case-insensitive, there would be the potential for non-deterministic mapping resolution depending on whether the parameter or result set field is named *`[Id]`* or *`[ID]`* respectively.

*** VS2010 Database Projects will not detect and hence not deploy a change if ONLY CAPITALISATION of stored procedure parameters, etc. differs.**