The ORM Cookbook

In October of 2016, InfoQ published a series of articles on the repository pattern in .NET. To illustrate the concepts three ORMs were demonstrated: Entity Framework, Dapper, and Chain.

A criticism of the articles was that it didn’t include many people’s favorite ORM. So as a follow up, this GitHub repository was created to expand on that idea and create a shared “cookbook” of design patterns for any or all of the .NET ORMs. Contributions are welcome.

When contributing recipes, please keep in mind the best practices for the ORM you are working with as novices may be using the code without fully understanding it.

Original InfoQ Articles

- Implementation Strategies for the Repository Pattern with Entity Framework, Dapper, and Chain
- Advanced Use Cases for the Repository Pattern in .NET

Presentation

The ORM Cookbook Repository

Each ORM is presented as its own xUnit test project. The actual recipes are in the Models and Repositories folder.

To ensure each ORM is "playing by the rules", a shared set of tests will be used. The tests are arranged into "use cases" classes. Each ORM can opt in for a given use case by inheriting from the appropriate use case class. This can be done multiple times if the ORM wishes to demonstrate alternate patterns.

Each use case has a matching markdown file in which the code samples can be added along with any relevant explanations. When possible, use Projbook notation to inline code samples. This will prevent the code samples from getting out of sync with the documentation.

If you build the “Documentation” project, the cookbook will be compiled as a website or PDF file.

Use Case: Single Model Repositories

This use case demonstrates CRUD operations on a single model mapped to a class.

Prototype Repository

This is the interface that every example repository will implement.

```
    public interface IEmployeeClassificationRepository
    {
        EmployeeClassification GetByKey(int employeeClassificationKey);

        EmployeeClassification FindByName(string employeeClassificationName);

        IList<EmployeeClassification> GetAll();

        int Create(EmployeeClassification classification);
        void Update(EmployeeClassification classification);
        void Delete(EmployeeClassification classification);
        void Delete(int employeeClassificationKey);
    }
```

The class `EmployeeClassification` is defined as such:

```
public interface IEmployeeClassification
{
    int EmployeeClassificationKey { get; set; }
    string EmployeeClassificationName { get; set; }
}
```

ADO.NET

With ADO.NET, the model does not actually participate in database operations so it needs no adornment.

```
public class EmployeeClassification : Recipes.Models.IEmployeeClassification
{
    public int EmployeeClassificationKey { get; set; }

    public string EmployeeClassificationName { get; set; }
}
```

The repository methods use raw SQL strings. All other ORMs internally generate the same code.

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{

    readonly string m_ConnectionString;
    public EmployeeClassificationRepository(string connectionString)
    {
        m_ConnectionString = connectionString;
    }

    public int Create(EmployeeClassification classification)
    {
        var sql = @"INSERT INTO HR.EmployeeClassification (EmployeeClassificationName)
                    OUTPUT Inserted.EmployeeClassificationKey
                    VALUES(@EmployeeClassificationName )";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", classification.EmployeeClassificationName);
                return (int)cmd.ExecuteScalar();
            }
        }
    }

    public void Delete(int employeeClassificationKey)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", employeeClassificationKey);
                cmd.ExecuteNonQuery();
            }
        }
    }

    public void Delete(EmployeeClassification classification)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", classification.EmployeeClassificationKey);
                cmd.ExecuteNonQuery();
            }
        }
    }

    public EmployeeClassification FindByName(string employeeClassificationName)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationName = @EmployeeClassificationName;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", employeeClassificationName);
                using (var reader = cmd.ExecuteReader())
                {
                    if (!reader.Read())
                        return null;

                    return new EmployeeClassification()
                    {
                        EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                        EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                    };
                }
            }
        }
    }

    public IList<EmployeeClassification> GetAll()
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName FROM HR.EmployeeClassification ec;";

        var result = new List<EmployeeClassification>();

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            using (var cmd = new SqlCommand(sql, con))
            {
                using (var reader = cmd.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        result.Add(new EmployeeClassification()
                        {
                            EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                            EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                        });
                    }
                    return result;
                }
            }
        }
    }

    public EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", employeeClassificationKey);
                using (var reader = cmd.ExecuteReader())
                {
                    if (!reader.Read())
                        return null;

                    return new EmployeeClassification()
                    {
                        EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                        EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                    };
                }
            }
        }
    }
    public void Update(EmployeeClassification classification)
    {
        var sql = @"UPDATE HR.EmployeeClassification
                    SET EmployeeClassificationName = @EmployeeClassificationName
                    WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", classification.EmployeeClassificationKey);
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", classification.EmployeeClassificationName);
                cmd.ExecuteNonQuery();
            }
        }
    }
}
```

Dapper

Dapper is essentially just ADO.NET with some helper methods to reduce the amount of boilerplate code.

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{

    readonly string m_ConnectionString;
    public EmployeeClassificationRepository(string connectionString)
    {
        m_ConnectionString = connectionString;
    }

    public int Create(EmployeeClassification classification)
    {
        var sql = @"INSERT INTO HR.EmployeeClassification (EmployeeClassificationName)
                    OUTPUT Inserted.EmployeeClassificationKey
                    VALUES (@EmployeeClassificationName )";
        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            return con.ExecuteScalar<int>(sql, classification);
        }
    }

    public EmployeeClassification FindByName(string employeeClassificationName)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationName = @EmployeeClassificationName;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();

            return con.QuerySingle<EmployeeClassification>(sql, new { EmployeeClassificationName = employeeClassificationName }
        }
    }

    public void Delete(int employeeClassificationKey)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            con.Execute(sql, new { EmployeeClassificationKey = employeeClassificationKey });
        }
    }

    public void Delete(EmployeeClassification classification)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            con.Execute(sql, classification);
        }
    }

    public IList<EmployeeClassification> GetAll()
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName FROM HR.EmployeeClassification ec;";

        var result = new List<EmployeeClassification>();

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            return con.Query<EmployeeClassification>(sql).ToList();
        }
    }

    public EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            return con.QuerySingle<EmployeeClassification>(sql, new { EmployeeClassificationKey = employeeClassificationKey });
        }
    }

    public void Update(EmployeeClassification classification)
    {
        var sql = @"UPDATE HR.EmployeeClassification
                    SET EmployeeClassificationName = @EmployeeClassificationName
                    WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            con.Open();
            con.Execute(sql, classification);
        }
    }
}
```

Tortuga Chain

Strictly speaking, Chain can use the same models as ADO.NET and Dapper so long as the column and property names match. However, it is more convenient to tag the class with what table it refers to.

```
[Table("HR.EmployeeClassification")]
public class EmployeeClassification : Recipes.Models.IEmployeeClassification
{
    public int EmployeeClassificationKey { get; set; }

    public string EmployeeClassificationName { get; set; }
}
```

Without the Table attribute, the table name will have to be specified in every call in the repository.

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{
    const string TableName = "HR.EmployeeClassification";
    readonly SqlServerDataSource m_DataSource;
    public EmployeeClassificationRepository(SqlServerDataSource dataSource)
    {
        m_DataSource = dataSource;
    }

    public int Create(EmployeeClassification classification)
    {
        return m_DataSource.Insert(classification).ToInt32().Execute();
    }

    public void Delete(int employeeClassificationKey)
    {
        m_DataSource.DeleteByKey(TableName, employeeClassificationKey).Execute();
    }

    public void Delete(EmployeeClassification classification)
    {
        m_DataSource.Delete(classification).Execute();
    }

    public EmployeeClassification FindByName(string employeeClassificationName)
    {
        return m_DataSource.From(TableName, new { EmployeeClassificationName = employeeClassificationName }).ToObject<EmployeeC
    }

    public IList<EmployeeClassification> GetAll()
    {
        return m_DataSource.From(TableName).ToCollection<EmployeeClassification>().Execute();
    }

    public EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        return m_DataSource.GetByKey(TableName, employeeClassificationKey).ToObject<EmployeeClassification>().Execute();
    }

    public void Update(EmployeeClassification classification)
    {
        m_DataSource.Update(classification).Execute();
    }
}
```

Entity Framework

To use Entity Framework, one needs to create a DbContext class. Here is a minimal example:

```
public partial class OrmCookbook : DbContext
{
    public OrmCookbook()
        : base("name=OrmCookbook")
    {
    }

    public virtual DbSet<EmployeeClassification> EmployeeClassifications { get; set; }

}
```

The model requires some annotations so that Entity Framework knows what table it applies to and what the primary key is.

```
[Table("HR.EmployeeClassification")]
public partial class EmployeeClassification
{
    [Key]
    public int EmployeeClassificationKey { get; set; }

    [StringLength(30)]
    public string EmployeeClassificationName { get; set; }
}
```

The context and model can be generated for you from the database using Entity Framework's "Code First" tooling. (The name "code first" doesn't literally mean the code has to be written before the database. Rather, it really means that you are not using EDMX style XML files.)

Finally, there is the repository itself:

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{
    public virtual int Create(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            context.EmployeeClassifications.Add(classification);
            context.SaveChanges();
            return classification.EmployeeClassificationKey;
        }
    }

    public virtual void Delete(int employeeClassificationKey)
    {
        using (var context = new OrmCookbook())
        {
            var temp = context.EmployeeClassifications.Find(employeeClassificationKey);
            if (temp != null)
            {
                context.EmployeeClassifications.Remove(temp);
                context.SaveChanges();
            }
        }
    }

    public virtual void Delete(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            var temp = context.EmployeeClassifications.Find(classification.EmployeeClassificationKey);
            if (temp != null)
            {
                context.EmployeeClassifications.Remove(temp);
                context.SaveChanges();
            }
        }
    }

    public virtual EmployeeClassification FindByName(string employeeClassificationName)
    {
        using (var context = new OrmCookbook())
        {
            return context.EmployeeClassifications.Where(ec => ec.EmployeeClassificationName == employeeClassificationName).Sin
        }
    }

    public virtual IList<EmployeeClassification> GetAll()
    {
        using (var context = new OrmCookbook())
        {
            return context.EmployeeClassifications.ToList();
        }
    }

    public virtual EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        using (var context = new OrmCookbook())
        {
            return context.EmployeeClassifications.Find(employeeClassificationKey);
        }
    }

    public virtual void Update(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            var temp = context.EmployeeClassifications.Find(classification.EmployeeClassificationKey);
            temp.EmployeeClassificationName = classification.EmployeeClassificationName;
            context.SaveChanges();
        }
    }
}
```

Note that the repository methods are not normally virtual. This was done so that they could be overridden with better implementations as shown below.

Entity Framework Intermediate

The data access patterns in Entity Framework can be quite inefficient, so to reduce unnecessary database calls you can modify the code as shown below.

```
public class EmployeeClassificationRepository_Intermediate : EmployeeClassificationRepository
{
    public override void Delete(int employeeClassificationKey)
    {
        using (var context = new OrmCookbook())
        {
            context.Database.ExecuteSqlCommand("DELETE FROM HR.EmployeeClassification WHERE EmployeeClassificationKey = @p0", e
        }
    }

    public override void Delete(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            context.Database.ExecuteSqlCommand("DELETE FROM HR.EmployeeClassification WHERE EmployeeClassificationKey = @p0", c
        }
    }

    public override void Update(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            context.Entry(classification).State = EntityState.Modified;
            context.SaveChanges();
        }
    }
}
```

NHibernate

NHibernate is one of the oldest ORMs for the .NET Framework. Based on Java's Hibernate, it heavily relies on XML configuration files and interfaces.

The models are interesting in that every property needs to be virtual. Without this, you'll get a runtime error.

```
public class EmployeeClassification : IEmployeeClassification
{
    public virtual int EmployeeClassificationKey { get; set; }

    public virtual string EmployeeClassificationName { get; set; }
}
```

Instead of attributes, a mapping file is used to associate the model with a database table.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                   assembly="Recipes.NHibernate"
                   namespace="Recipes.NHibernate.Models">

  <class name="EmployeeClassification" table="EmployeeClassification" schema="HR">

    <id name="EmployeeClassificationKey" >
      <generator class="native"/>
    </id>
    <property name="EmployeeClassificationName" />
  </class>

</hibernate-mapping>
```

The NHibernate documentation recommends create a session factory helper using this pattern:

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{
    public int Create(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Save(classification);
            session.Flush();
            return classification.EmployeeClassificationKey;
        }
    }

    public void Delete(int employeeClassificationKey)
    {

        using (ISession session = NHibernateHelper.OpenSession())
        {
            var temp = session.Get<EmployeeClassification>(employeeClassificationKey);
            session.Delete(temp);
            session.Flush();
        }
    }

    public void Delete(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Delete(classification);
            session.Flush();
        }
    }

    public EmployeeClassification FindByName(string employeeClassificationName)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            return session
                .CreateCriteria(typeof(EmployeeClassification))
                .Add(Restrictions.Eq("EmployeeClassificationName", employeeClassificationName))
                .List<EmployeeClassification>()
                .SingleOrDefault();
        }
    }

    public IList<EmployeeClassification> GetAll()
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            return session
                .CreateCriteria(typeof(EmployeeClassification))
                .List<EmployeeClassification>();
        }
    }

    public EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        using (ISession session = NHibernateHelper.OpenSession())
            return session.Get<EmployeeClassification>(employeeClassificationKey);
    }

    public void Update(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Update(classification);
            session.Flush();
        }
    }
}
```

Finally there is the repository itself.

```
public class EmployeeClassificationRepository : IEmployeeClassificationRepository<EmployeeClassification>
{
    public int Create(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Save(classification);
            session.Flush();
            return classification.EmployeeClassificationKey;
        }
    }

    public void Delete(int employeeClassificationKey)
    {

        using (ISession session = NHibernateHelper.OpenSession())
        {
            var temp = session.Get<EmployeeClassification>(employeeClassificationKey);
            session.Delete(temp);
            session.Flush();
        }
    }

    public void Delete(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Delete(classification);
            session.Flush();
        }
    }

    public EmployeeClassification FindByName(string employeeClassificationName)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            return session
                .CreateCriteria(typeof(EmployeeClassification))
                .Add(Restrictions.Eq("EmployeeClassificationName", employeeClassificationName))
                .List<EmployeeClassification>()
                .SingleOrDefault();
        }
    }

    public IList<EmployeeClassification> GetAll()
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            return session
                .CreateCriteria(typeof(EmployeeClassification))
                .List<EmployeeClassification>();
        }
    }

    public EmployeeClassification GetByKey(int employeeClassificationKey)
    {
        using (ISession session = NHibernateHelper.OpenSession())
            return session.Get<EmployeeClassification>(employeeClassificationKey);
    }

    public void Update(EmployeeClassification classification)
    {
        using (ISession session = NHibernateHelper.OpenSession())
        {
            session.Update(classification);
            session.Flush();
        }
    }
}
```

The rules on when you need to call `Flush` are complex. In some cases it will be called for you implicitly, but as a general rule you need to invoke it before leaving a block that includes modifications.

Use Case: Asynchronous Repositories

This use case demonstrates the use of asynchronous CRUD operations. Asynchronous database calls are generally preferable, as they allow for more throughput in server applications and prevent UI blocking in GUI applications.

Prototype Repository

The prototype repository based on our “Single Model Repositories” use case, with a slight modification to the signatures.

```
    public interface IEmployeeClassificationAsynchronousRepository
    {
        Task<EmployeeClassification> GetByKeyAsync(int employeeClassificationKey);

        Task<EmployeeClassification> FindByNameAsync(string employeeClassificationName);

        Task<IList<EmployeeClassification>> GetAllAsync();

        Task<int> CreateAsync(EmployeeClassificationclassification);
        Task UpdateAsync(EmployeeClassificationclassification);
        Task DeleteAsync(EmployeeClassificationclassification);
        Task DeleteAsync(int employeeClassificationKey);
    }
```

No changes are needed for the model in any of these examples.

ADO.NET

To make an ADO.NET repository asynchronous, simply add `await` and `Async` in the appropriate places.

```
public class EmployeeClassificationAsynchronousRepository : IEmployeeClassificationAsynchronousRepository<EmployeeClassificatio
{

    readonly string m_ConnectionString;
    public EmployeeClassificationAsynchronousRepository(string connectionString)
    {
        m_ConnectionString = connectionString;
    }

    public async Task<int> CreateAsync(EmployeeClassification classification)
    {
        var sql = @"INSERT INTO HR.EmployeeClassification (EmployeeClassificationName)
                    OUTPUT Inserted.EmployeeClassificationKey
                    VALUES(@EmployeeClassificationName )";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", classification.EmployeeClassificationName);
                return (int)await cmd.ExecuteScalarAsync();
            }
        }
    }

    public async Task DeleteAsync(EmployeeClassification classification)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", classification.EmployeeClassificationKey);
                await cmd.ExecuteNonQueryAsync();
        }
    }

    public async Task DeleteAsync(int employeeClassificationKey)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", employeeClassificationKey);
                await cmd.ExecuteNonQueryAsync();
            }
        }
    }


    public async Task<EmployeeClassification> FindByNameAsync(string employeeClassificationName)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationName = @EmployeeClassificationName;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", employeeClassificationName);
                using (var reader = await cmd.ExecuteReaderAsync())
                {
                    if (!await reader.ReadAsync())
                        return null;

                    return new EmployeeClassification()
                    {
                        EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                        EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                    };
                }
            }
        }
    }

    public async Task<IList<EmployeeClassification>> GetAllAsync()
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName FROM HR.EmployeeClassification ec;";

        var result = new List<EmployeeClassification>();

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            using (var cmd = new SqlCommand(sql, con))
            {
                using (var reader = await cmd.ExecuteReaderAsync())
                {
                    while (await reader.ReadAsync())
                    {
                        result.Add(new EmployeeClassification()
                        {
                            EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                            EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                        });
                    }
                    return result;
                }
            }
        }
    }


    public async Task<EmployeeClassification> GetByKeyAsync(int employeeClassificationKey)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", employeeClassificationKey);
                using (var reader = await cmd.ExecuteReaderAsync())
                {
                    if (!await reader.ReadAsync())
                        return null;

                    return new EmployeeClassification()
                    {
                        EmployeeClassificationKey = reader.GetInt32(reader.GetOrdinal("EmployeeClassificationKey")),
                        EmployeeClassificationName = reader.GetString(reader.GetOrdinal("EmployeeClassificationName"))
                    };
                }
            }
        }
    }


    public async Task UpdateAsync(EmployeeClassification classification)
    {
        var sql = @"UPDATE HR.EmployeeClassification
                    SET EmployeeClassificationName = @EmployeeClassificationName
                    WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            using (var cmd = new SqlCommand(sql, con))
            {
                cmd.Parameters.AddWithValue("@EmployeeClassificationKey", classification.EmployeeClassificationKey);
                cmd.Parameters.AddWithValue("@EmployeeClassificationName", classification.EmployeeClassificationName);
                await cmd.ExecuteNonQueryAsync();
            }
        }
    }
}
```

Dapper

To make a Dapper repository asynchronous, simply add `await` and `Async` in the appropriate places.

```
public class EmployeeClassificationAsynchronousRepository : IEmployeeClassificationAsynchronousRepository<EmployeeClassificatio
{

    readonly string m_ConnectionString;
    public EmployeeClassificationAsynchronousRepository(string connectionString)
    {
        m_ConnectionString = connectionString;
    }

    public async Task<int> CreateAsync(EmployeeClassification classification)
    {
        var sql = @"INSERT INTO HR.EmployeeClassification (EmployeeClassificationName)
```

```
                    OUTPUT Inserted.EmployeeClassificationKey
                    VALUES (@EmployeeClassificationName )";
        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            return await con.ExecuteScalarAsync<int>(sql, classification);
        }
    }

    public async Task DeleteAsync(EmployeeClassification classification)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            await con.ExecuteAsync(sql, classification);
        }
    }

    public async Task DeleteAsync(int employeeClassificationKey)
    {
        var sql = @"DELETE HR.EmployeeClassification WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            await con.ExecuteAsync(sql, new { EmployeeClassificationKey = employeeClassificationKey });
        }
    }

    public async Task<EmployeeClassification> FindByNameAsync(string employeeClassificationName)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationName = @EmployeeClassificationName;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();

            return await con.QuerySingleAsync<EmployeeClassification>(sql, new { EmployeeClassificationName = employeeClassific
        }
    }

    public async Task<IList<EmployeeClassification>> GetAllAsync()
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName FROM HR.EmployeeClassification ec;";

        var result = new List<EmployeeClassification>();

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            return (await con.QueryAsync<EmployeeClassification>(sql)).ToList();
        }
    }


    public async Task<EmployeeClassification> GetByKeyAsync(int employeeClassificationKey)
    {
        var sql = @"SELECT ec.EmployeeClassificationKey, ec.EmployeeClassificationName
                    FROM HR.EmployeeClassification ec
                    WHERE ec.EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            return await con.QuerySingleAsync<EmployeeClassification>(sql, new { EmployeeClassificationKey = employeeClassifica
        }
    }
    public async Task UpdateAsync(EmployeeClassification classification)
    {
        var sql = @"UPDATE HR.EmployeeClassification
                    SET EmployeeClassificationName = @EmployeeClassificationName
                    WHERE EmployeeClassificationKey = @EmployeeClassificationKey;";

        using (var con = new SqlConnection(m_ConnectionString))
        {
            await con.OpenAsync();
            await con.ExecuteAsync(sql, classification);
        }
    }
}
```

Tortuga Chain

In Chain, calls to `.Execute()` are replaced with `.ExecuteAsync()`.

```
public class EmployeeClassificationAsynchronousRepository : IEmployeeClassificationAsynchronousRepository<EmployeeClassificatio
{
    const string TableName = "HR.EmployeeClassification";
    readonly SqlServerDataSource m_DataSource;
    public EmployeeClassificationAsynchronousRepository(SqlServerDataSource dataSource)
    {
        m_DataSource = dataSource;
    }

    public async Task<int> CreateAsync(EmployeeClassification classification)
    {
        return await m_DataSource.Insert(classification).ToInt32().ExecuteAsync();
    }

    public async Task DeleteAsync(int employeeClassificationKey)
    {
        await m_DataSource.DeleteByKey(TableName, employeeClassificationKey).ExecuteAsync();
    }

    public async Task DeleteAsync(EmployeeClassification classification)
    {
        await m_DataSource.Delete(classification).ExecuteAsync();
    }

    public async Task<EmployeeClassification> FindByNameAsync(string employeeClassificationName)
    {
        return await m_DataSource.From(TableName, new { EmployeeClassificationName = employeeClassificationName }).ToObject<Emp
    }

    public async Task<IList<EmployeeClassification>> GetAllAsync()
    {
        return await m_DataSource.From(TableName).ToCollection<EmployeeClassification>().ExecuteAsync();
    }

    public async Task<EmployeeClassification> GetByKeyAsync(int employeeClassificationKey)
    {
        return await m_DataSource.GetByKey(TableName, employeeClassificationKey).ToObject<EmployeeClassification>().ExecuteAsyn
    }

    public async Task UpdateAsync(EmployeeClassification classification)
    {
        await m_DataSource.Update(classification).ExecuteAsync();
    }
}
```

Entity Framework

To make an Entity Framework repository asynchronous, you need to import the `System.Data.Entity` namespace. This creates the async version of the LINQ extension methods needed.

```
public class EmployeeClassificationAsynchronousRepository : IEmployeeClassificationAsynchronousRepository<EmployeeClassificatio
{

    public async Task<int> CreateAsync(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            context.EmployeeClassifications.Add(classification);
            await context.SaveChangesAsync();
            return classification.EmployeeClassificationKey;
        }
    }

    public async Task DeleteAsync(int employeeClassificationKey)
    {
        using (var context = new OrmCookbook())
        {
            var temp = await context.EmployeeClassifications.FindAsync(employeeClassificationKey);
            if (temp != null)
            {
                context.EmployeeClassifications.Remove(temp);
                await context.SaveChangesAsync();
            }
        }
    }

    public async Task DeleteAsync(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            var temp = await context.EmployeeClassifications.FindAsync(classification.EmployeeClassificationKey);
            if (temp != null)
            {
                context.EmployeeClassifications.Remove(temp);
                await context.SaveChangesAsync();
            }
        }
    }

    public async Task<EmployeeClassification> FindByNameAsync(string employeeClassificationName)
    {
        using (var context = new OrmCookbook())
        {
            return await context.EmployeeClassifications.Where(ec => ec.EmployeeClassificationName == employeeClassificationNam
        }
    }

    public async Task<IList<EmployeeClassification>> GetAllAsync()
    {
        using (var context = new OrmCookbook())
        {
            return await context.EmployeeClassifications.ToListAsync();
        }
    }

    public async Task<EmployeeClassification> GetByKeyAsync(int employeeClassificationKey)
    {
        using (var context = new OrmCookbook())
        {
            return await context.EmployeeClassifications.FindAsync(employeeClassificationKey);
        }
    }


    public async Task UpdateAsync(EmployeeClassification classification)
    {
        using (var context = new OrmCookbook())
        {
            var temp = await context.EmployeeClassifications.FindAsync(classification.EmployeeClassificationKey);
            temp.EmployeeClassificationName = classification.EmployeeClassificationName;
            await context.SaveChangesAsync();
        }
    }
}
```

NHibernate

NHibernate does not support asynchronous calls. To ensure UI responsiveness, use `Task.Run` calls to create awaitable tasks that run on a background thread. (Do not use `Task.Run` in a server scenario.)

Use Case: Model with Single Child

In this use case, the Department model consits of a Deparment object with a single child object of type Division. For the purpose of this use case, the repository cannot create new Division records when saving the Department.

The model is defined with these two interfaces:

```
namespace Recipes.Models
{
    /// <summary>
    /// This model shows a Department with a FK represented as a child object.
    /// </summary>
    /// <typeparam name="TDivision">The type of the t division.</typeparam>
    public interface IDepartment<TDivision> where TDivision : IDivision
    {
        int DepartmentKey { get; set; }
        string DepartmentName { get; set; }
        TDivision Division { get; set; }
    }
}
```

```
namespace Recipes.Models
{
    public interface IDivision
    {
        int DivisionKey { get; set; }
        string DivisionName { get; set; }

    }


}
```

ADO.NET

Nothing interesting here, as the data is just manually mapped.

Tortuga Chain

Chain heavily relies on the use of views. If you wish to populate a child object, then its fields should be represented in the view using a normal join.

In the model, the `Decompose` attribute is applied to a property to indicate that it should be populated from the parent's result set.

```
[Decompose]
public Division Division { get; set; }
```

Decomposed properties do not participate in Insert/Update operations. To handle this, you need to "pull up" the FK using this syntax.

```
public int? DivisionKey
{
    get { return Division?.DivisionKey; }
}
```

As mentioned above, you need to read from the view (called HR.DepartmentDetail in this case) in order to fetch the data needed to populate the child object. For inserts and updates, you will still refer to the underlying table.

```
public Department GetByKey(int departmentKey)
{
    return m_DataSource.From("HR.DepartmentDetail", new { DepartmentKey = departmentKey }).ToObject<Department>().Execute();
}
```

Dapper

Like Chain, Dapper can decompose result sets into parent-child models. This is called "Multi Mapping". It requires a function to indicate how the parent and child are related, plus a parameter to indicate which column is the foreign Key.

```
public IList<Department> GetAll()
{
    using (var context = new OrmCookbook())
    {
        return context.Departments.Include(d => d.Division).ToList();
    }
}
```

Unlike the other ORMs, Dapper has a hard limit of no more than 6 child objects. When using more than one child object, pass a comma separated list to the `splitOn` parameter.

Entity Framework

The model used in Entity Framework expects the Division property to be vitual. If it isn't, then lazy loading will not be enabled.

```
public Division Division { get; set; }
```

A limitation of EF is that for every child object you expose, you also need to expose the matching foreign Key.

```
public int DivisionKey { get; set; }
```

When you save the record, EF's context handling becomes a problem. You need to set the "Entry State" for the division object so that EF knows that it came from a different context. Otherwise EF will try to insert a new row for Division, which will result in a runtime error.

```
public int Create(Department department)
{
    using (var context = new OrmCookbook())
    {
        context.Departments.Add(department);
        context.Entry(department.Division).State = EntityState.Unchanged;
        context.SaveChanges();
        return department.DepartmentKey;
    }
}
```

When performing an update, you don't need to touch the Entry State. Instead you need to update the DivsionKey on the Department object to match the DivsionKey on the Division object. Since that is an implementation detail, we are doing it inside the repository itself.

```
public void Update(Department department)
{
    department.DivisionKey = department.Division.DivisionKey;

    using (var context = new OrmCookbook())
    {
        context.Entry(department).State = EntityState.Modified;
        context.SaveChanges();
    }
}
```

With reads, the `Include` clause is necessary. Without if one of two things will happen:

- If Division is virtual, lazy loading will be tiggered. Since the context will already be closed by the time that happens, an exception will be thrown.
- If Division is not virtual, the property will be null. This will most likely lead to a `NullReferenceException`, as logically this should always have a value.

```
public IList<Department> GetAll()
{
    using (var context = new OrmCookbook())
    {
        return context.Departments.Include(d => d.Division).ToList();
    }
}
```

```
public Department GetByKey(int departmentKey)
{
    using (var context = new OrmCookbook())
    {
        return context.Departments.Include(d => d.Division).Where(d => d.DepartmentKey == departmentKey).SingleOrDefault();
    }
}
```

Note that the `Find` shortcut is not compatible with the `Include` clause, so you have to write the statement longform.