Coalesce Documentation

IntelliTect

Mar 22, 2019

Coalesce

Designed to help you quickly build amazing web applications, Coalesce is a rapid-development code generation framework, created by IntelliTect and built on top of:

- ASP.NET Core
- EF Core
- TypeScript
- Knockout

CHAPTER 1

What do I do?

You are responsible for the interesting parts of your application:

- Data Model
- Business Logic
- External Integrations
- Page Content
- Site Design
- Custom Scripting

CHAPTER 2

What is done for me?

Coalesce builds the part of your application that are mundane and monotonous to build:

- Client side Knockout *TypeScript ViewModels* that mirror your data model for both *lists* and *individual objects*. Utilize these to rapidly build out your applications various pages.
- APIs to interact with your models via endpoints like List, Get, Save, and more.
- *Out-of-the-box bindings* for common controls like dates, selecting objects via drop downs, enums, etc. Drop-downs support searching and paging automatically.
- A complete set of admin pages are built, allowing you to read, create, edit, and delete data straight away without writing any additional code.

CHAPTER 3

Getting Started

3.1 Creating a Project

The quickest and easiest way to create a new Coalesce application is to check out the starter project: https://github.com/IntelliTect/Coalesce.Starter. This project already contains all the needed projects, dependencies, and configuration. In PowerShell:

```
mkdir MyProject
cd MyProject
git clone --depth 1 https://github.com/IntelliTect/Coalesce.Starter.git .
./RenameProject.ps1
```

At this point, you can open up the solution in Visual Studio and run your application. However, your application won't do much without a data model, so you will probably want to do the following before running:

- Create an initial *Data Model* by adding EF entity classes to the data project and the corresponding `DbSet<>` properties to `AppDbContext`. You will notice that the starter project includes a single model, `ApplicationUser`, to start with. Feel free to change this model or remove it entirely. Read *EF Entity Models* for more information about creating a data model.
- Run `dotnet ef migrations add Init` (Init can be any name) in the data project to create an initial database migration.
- Run Coalesce's code generation by either:
  - Running `dotnet coalesce` in the web project's root directory
  - Running the `coalesce` gulp task in the Task Runner Explorer

You're now at a point where you can start creating your own controllers (or utilizing the generated `partial` controllers) to serve your own pages with your own scripts.

3.2 Building Pages & Features

Lets say we've created a *model* called `Person` like so, and we've ran code generation with `dotnet coalesce`:

```
namespace MyApplication.Data.Models
{
    public class Person
    {
        public int PersonId { get; set; }
        public string Name { get; set; }
        public DateTimeOffset? BirthDate { get; set; }
    }
}
```

We can create a details page for a Person by creating:

A controller in `src/MyApplication.Web/Controllers/PersonController.cs`:

```
namespace MyApplication.Web.Controllers
{
    public partial class PersonController
    {
        public IActionResult Details() => View();
    }
}
```

A view in `src/MyApplication.Web/Views/Person/Details.cshtml`:

```
<h1>Person Details</h1>

<div data-bind="with: person">
    <dl class="dl-horizontal">
        <dt>Name </dt>
        <dd data-bind="text: name"></dd>

        <dt>Date of Birth </dt>
        <dd data-bind="moment: birthDate, format: 'MM/DD/YYYY hh:mm a'"></dd>
    </dl>
</div>

@section Scripts
{
<script src="~/js/person.details.js"></script>
<script>
    $(function () {
        var vm = new MyApplication.PersonDetails();
        ko.applyBindings(vm);
        vm.load();
    });
</script>
}
```

And a script in `src/MyApplication.Web/Scripts/person.details.ts`:

```
/// <reference path="viewmodels.generated.d.ts" />

module MyApplication {
```

(continues on next page)

(continued from previous page)

```
    export class PersonDetails {
        public person = new ViewModels.Person();

        load() {
            var id = Coalesce.Utilities.GetUrlParameter("id");
            if (id != null && id != '') {
                this.person.load(id);
            }
        }
    }
}
```

With these pieces in place, we now have a functioning page that will display details about a person. We can start up the application and navigate to `/Person/Details?id=1` (assuming a person with ID 1 exists - if not, navigate to `/Person/Table` and create one).

From this point, one can start adding more fields, more features, and more flair to the page. Check out all the other documentation in the sidebar to see what else Coalesce has to offer.

3.2.1 EF Entity Models

Overview

Models are the core business objects of your application - they serve as the fundamental representation of data in your application. The design of your models is very important. In Entity Framework Core, data models are just Plain Old CLR Objects (POCOs).

Building a Data Model

To start building your data model that Coalesce will generate code for, follow the best practices for EF Core.

Guidance on this topic is available in abundance in the Entity Framework Core documentation.

Don't worry about querying or saving data when you're just getting started - Coalesce will provide a lot of that functionality for you, and it is very easy to customize what Coalesce offers later. To get started, just build your POCOs and `DbContext` classes. Annotate your `DbContext` class with `[Coalesce]` so that Coalesce will discover it and generate code based off of your context for you.

Before you start building, you are highly encouraged to read the sections below. The linked pages explain in greater detail what Coalesce will build for you for each part of your data model.

Properties

Read *Properties* for an outline of the different types of properties that you may place on your models and the code that Coalesce will generate for each of them.

Attributes

Coalesce provides a number of C# attributes that can be used to decorate your model classes and their properties in order to customize behavior, appearance, security, and more. Coalesce also supports a number of annotations from `System.ComponentModel.DataAnnotations`.

Read *Attributes* to learn more.

Methods

You can place both static and interface methods on your model classes. Any public methods annotated with *[Coalesce]* will have a generated API endpoint and corresponding generated TypeScript members for calling this API endpoint. Read *Methods* to learn more.

Customizing CRUD Operations

Once you've got a solid data model in place, its time to start customizing the way that Coalesce will *read* your data, as well as the way that it will handle your data when processing *creates*, *updates*, and *deletes*.

Data Sources

The method by which you can control what data the users of your application can access through Coalesce's generated APIs is by creating custom data sources. These are classes that allow complete control over the way that data is retrieved from your database and provided to clients. Read *Data Sources* to learn more.

Behaviors

Behaviors in Coalesce are to mutating data as data sources are to reading data. Defining a behaviors class for a model allows complete control over the way that Coalesce will create, update, and delete your application's data in response to requests made through its generated API. Read *Behaviors* to learn more.

3.2.2 External Types

In Coalesce, any type which is connected to your data model but is not directly part of it is considered to be an "external type".

The collection of external types for a data model looks like this:

1. Take all of the api-served types in your data model. This includes *EF Entity Models* and *Custom DTOs*.
2. Take all of the property types, method parameters, and method return types of these types.
3. Any of these types which are not primitives and not database-mapped types are external types.
4. For any external type, any of the property types which qualify under the above rules are also external types.

> **Warning:** Be careful when using types that you do not own for properties and method returns in your data model. When Coalesce generates external type ViewModels and DTOs, it will not stop until it has exhausted all paths that can be reached by following public property types and method returns.
>
> In general, you should only expose types that you have created so that you will always have full control over them. Mark any properties you don't wish to expose with *[InternalUse]*, or make those members non-public.

Generated Code

For each external type found in your application's model, Coalesce will generate:

- A *Generated DTO*
- A *TypeScript Model*

Example Data Model

For example, in the following scenario, the following classes are considered as external types:

- `PluginMetadata`, exposed through a getter-only property on `ApplicationPlugin`.
- `PluginResult`, exposed through a method return on `ApplicationPlugin`.

`PluginHandler` is not because it not exposed by the model, neither directly nor through any of the other external types.

```
public class AppDbContext : DbContext {
    public DbSet<Application> Applications { get; set; }
    public DbSet<ApplicationPlugin> ApplicationPlugins { get; set; }
}

public class Application {
    public int ApplicationId { get; set; }
    public string Name { get; set; }
    public ICollection<ApplicationPlugin> Plugins { get; set; }
}

public class ApplicationPlugin {
    public int ApplicationPluginId { get; set; }
    public int ApplicationId { get; set; }
    public Application Application { get; set; }

    public string TypeName { get; set; }

    private PluginHandler GetInstance() =>
        ((PluginHandler)Activator.CreateInstance(Type.GetType(TypeName)));

    public PluginMetadata Metadata => GetInstance().GetMetadata();

    public PluginResult Invoke(string action, string data) => GetInstance().
↪Invoke(Application, action, data);
}

public abstract class PluginHandler {
    public abstract PluginMetadata GetMetadata();
    public abstract PluginResult Invoke(Application app, string action, string data);
}

public abstract class PluginMetadata {
    public bool Name { get; set; }
    public string Version { get; set; }
    public ICollection<string> Actions { get; set; }
}

public abstract class PluginResult {
    public bool Success { get; set; }
    public string Message { get; set; }
}
```

Loading & Serialization

External types have slightly different behavior when undergoing serialization to be sent to the client. Unlike database-mapped types which are subject to the rules of *Include Tree*, external types ignore the Include Tree when being mapped

to DTOs for serialization. Read *Include Tree/External Type Caveats* for a more detailed explanation of this exception.

3.2.3 Custom DTOs

In addition to the generated *Generated DTOs* that Coalesce will create for you, you may also create your own implementations of an `IClassDto`. These types are first-class citizens in Coalesce - you will get a full suite of features surrounding them as if they were entities. This includes generated API Controllers, Admin Views, and full *TypeScript ViewModels* and *TypeScript ListViewModels*.

Contents

The difference between a Custom DTO and the underlying entity that they represent is as follows:

- The only time your custom DTO will be served is when it is requested directly from one of the endpoints on its generated controller. It will not be used when making a call to an API endpoint that was generated from an entity.
- When mapping data from your database, or unmapping data from the client, the DTO itself must manually map all properties, since there is no corresponding *Generated DTO*. Attributes like *[DtoIncludes] & [DtoExcludes]* and property-level security through *Security Attributes* have no effect on custom DTOs, since those attribute only affect what get generated for *Generated DTOs*.

Creating a Custom DTO

To create a custom DTO, define a class that implements `IClassDTO<T>`, where `T` is an EF Core POCO, and annotate it with `[Coalesce]` . Add any *Properties* to it just as you would add *model properties* to a regular EF model.

Next, ensure that one property is annotated with `[Key]` so that Coalesce can know the primary key of your DTO in order to perform database lookups and keep track of your object uniquely in the client-side TypeScript.

Now, populate the required `MapTo` and `MapFrom` methods with code for mapping from and to your DTO, respectively (the methods are named with respect to the underlying entity, not the DTO). Most properties probably map one-to-one in both directions, but you probably created a DTO because you wanted some sort of custom mapping - say, mapping a collection on your entity with a comma-delimited string on the DTO. This is also the place to perform any user-based, role-based, property-level security. You can access the current user on the `IMappingContext` object.

```
[Coalesce]
public class CaseDto : IClassDto<Case>
{
    [Key]
    public int CaseId { get; set; }

    public string Title { get; set; }

    [Read]
    public string AssignedToName { get; set; }

    public void MapTo(Case obj, IMappingContext context)
    {
        obj.Title = Title;
```

(continues on next page)

(continued from previous page)

```
    }

    public void MapFrom(Case obj, IMappingContext context = null, IncludeTree tree =␣
↪null)
    {
        CaseId = obj.CaseKey;
        Title = obj.Title;
        if (obj.AssignedTo != null)
        {
            AssignedToName = obj.AssignedTo.Name;
        }
    }
}
```

Warning: Custom DTOs do not utilize property-level *Security Attributes* nor *[DtoIncludes] & [DtoExcludes]*, since these are handled in the *Generated DTOs*. If you need property-level security or trimming, you must write it yourself in the `MapTo` and `MapFrom` methods.

If you have any child objects on your DTO, you can invoke the mapper for some other object using the static `Mapper` class. Also seen in this example is how to respect the *Include Tree* when mapping entity types; however, respecting the `IncludeTree` is optional. Since this DTO is a custom type that you've written, if you're certain your use cases don't need to worry about object graph trimming, then you can ignore the `IncludeTree`. If you do ignore the `IncludeTree`, you should pass `null` to calls to `Mapper` - don't pass in the incoming `IncludeTree`, as this could cause unexpected results.

```
using IntelliTect.Coalesce.Mapping;

[Coalesce]
public class CaseDto : IClassDto<Case>
{
    public int ProductId { get; set; }
    public Product Product { get; set; }
    ...

    public void MapFrom(Case obj, IMappingContext context = null, IncludeTree tree =␣
↪null)
    {
        ProductId = obj.ProductId;

        if (tree == null || tree[nameof(this.Product)] != null)
            Product = Mapper.MapToDto<Product, ProductDtoGen>(obj.Product, context,␣
↪tree?[nameof(this.Product)]
        ...
    }
}
```

Using Custom DataSources and Behaviors

When you create a custom DTO, it will use the *Standard Data Source* and *Standard Behaviors* just like any of your regular *EF Entity Models*. If you wish to override this, your custom data source and/or behaviors MUST be declared in one of the following ways:

1. As a nested class of the DTO. The relationship between your data source or behaviors and your DTO will be

picked up automatically.

```
[Coalesce]
public class CaseDto : IClassDto<Case>
{
    [Key]
    public int CaseId { get; set; }

    public string Title { get; set; }

    ...

    public class MyCaseDtoSource : StandardDataSource<Case, AppDbContext>
    {
        ...
    }
}
```

2. With a `[DeclaredFor]` attribute that references the DTO type:

```
[Coalesce]
public class CaseDto : IClassDto<Case>
{
    [Key]
    public int CaseId { get; set; }

    public string Title { get; set; }

    ...
}

[Coalesce, DeclaredFor(typeof(CaseDto))]
public class MyCaseDtoSource : StandardDataSource<Case, AppDbContext>
{
    ...
}
```

3.2.4 Services

In a Coalesce, you are fairly likely to end up with a need for some API endpoints that aren't closely tied with your regular data model. While you could stick static *Methods* on one of your entities, this solution just leads to a jumbled mess of functionality all over your data model that doesn't belong there.

Instead, Coalesce allows you to generate API Controllers and a TypeScript client from a service. A service, in this case, is nothing more than a C# class or an interface with methods on it, annotated with `[Coalesce,Service]`. An implementation of this class or interface must be injectable from your application's service container, so a registration in Startup.cs is needed.

The instance methods of these services conform exactly to the specifications outlined in *Methods* with a few exceptions:

- TypeScript functions for invoking the endpoint have no `reload: boolean` parameter.
- Instance methods don't operate on an instance of some model with a known key, but instead on an injected instance of the service.

Generated Code

For each external type found in your application's model, Coalesce will generate:

- An API controller with endpoints that correspond to the service's instance methods.
- A TypeScript client containing the members outlined in *Methods* for invoking these endpoints.

Example Service

An example of a service might look something like this:

```
[Coalesce, Service]
public interface IWeatherService
{
    WeatherData GetWeather(string zipCode);
}
```

```
public class WeatherData
{
    public double TempFahrenheit { get; set; }

    // ... Other properties as desired
}
```

With an implementation:

```
public class WeatherService : IWeatherService
{
    public WeatherService(AppDbContext db)
    {
        this.db = db;
    }

    public WeatherData GetWeather(string zipCode)
    {
        // Assuming some magic HttpGet method that works as follows...
        var response = HttpGet("http://www.example.com/api/weather/" + zipCode);
        return response.Body.SerializeTo<WeatherData>();
    }

    public void MethodThatIsntExposedBecauseItIsntOnTheExposedInterface() {  }
}
```

And a registration:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCoalesce<AppDbContext>();
        services.AddScoped<IWeatherService, WeatherService>();
    }
}
```

While it isn't required that an interface for your service exist - you can generate directly from the implementation, it is highly recommended that an interface be used. Interfaces increase testability and reduce risk of accidentally changing the signature of a published API, among other benefits.

3.2.5 Properties

Models in a Coalesce application are just EF Core POCOs. The properties defined on your models should fit within the constraints of EF Core.

Coalesce currently has a few more restrictions than what EF Core allows, but hopefully over time some of these restrictions can be relaxed as Coalesce grows in capability.

Property Varieties

The following kinds of properties may be declared on your models.

Primary Key To work with Coalesce, your model must have a single property for a primary key. By convention, this property should be named the same as your model class with `Id` appended to that name, but you can also annotate a property with `[Key]` to denote it as the primary key.

Foreign Keys & Reference Navigation Properties While a foreign key may be declared on your model using only the EF OnModuleBuilding method to specify its purpose, Coalesce won't know what the property is a key for. Therefore, foreign key properties should always be accompanied by a reference navigation property, and vice versa.

In cases where the foreign key is not named after the navigation property with "`Id`" appended, the `[ForeignKeyAttribute]` may be used on either the key or the navigation property to denote the other property of the pair, in accordance with the recommendations set forth by EF Core's Modeling Guidelines.

Collection Navigation Properties Collection navigation properties can be used in a straightforward manner. In the event where the inverse property on the other side of the relationship cannot be determined, `[InversePropertyAttribute]` will need to be used. EF Core provides documentation on how to use this attribute. Errors will be displayed at generation time if an inverse property cannot be determined without the attribute. We recommend recommended that you declare the type of collection navigations as `ICollection<T>`.

Non-mapped POCOs Properties of a type that are not on your `DbContext` will also have corresponding properties generated on the *TypeScript ViewModels* typed as *TypeScript External ViewModels*, and the values of such properties will be sent with the object to the client when requested.

See *External Types* for more information.

Value Types Strings (although not actually a value type), any numeric type, enums, booleans, DateTimes, and DateTimeOffsets are all supported by Coalesce, as well as the nullable versions of all of these.

Getter-only Properties Any property that only has a getter will also have a corresponding property generated in the *TypeScript ViewModels*, but won't be sent back to the server during any save actions.

If such a property is defined as an auto-property, the `[NotMapped]` attribute should be used to prevent EF Core from attempting to map such a property to your database.

Other Considerations

For any of the kinds of properties outlined above, the following rules are applied:

Attributes Coalesce provides a number of *Attributes*, and supports a number of other .NET attributes, that allow for further customization of your model.

Security Properties will not be sent to the client and/or will be ignored if received by the client if authorization checks against any property-level *Security Attributes* present fail. This security is handled by the *Generated DTOs*.

Loading & Serialization The *Default Loading Behavior*, any functionality defined in *Data Sources*, and *[DtoIncludes] & [DtoExcludes]* may also restrict which properties are sent to the client when requested.

NotMapped While Coalesce does not do anything special for the `[NotMapped]` attribute, it is still and important attribute to keep in mind while building your model, as it prevents EF Core from doing anything with the property.

3.2.6 Attributes

Coalesce provides a number of C# attributes that can be used to decorate your model classes and their properties in order to customize behavior, appearance, security, and more. Coalesce also supports a number of annotations from `System.ComponentModel.DataAnnotations`.

Coalesce Attributes

Visit each link below to learn about the attribute that Coalesce provides that can be used to decorate your models.

[ClientValidation]

The `[IntelliTect.Coalesce.DataAnnotations.ClientValidation]` attribute is used to control the behavior of client-side model validation and to add additional client-only validation parameters. Database validation is available via standard `System.ComponentModel.DataAnnotations` annotations. These propagate to the client as validations in TypeScript via Knockout-Validation and prevent saves from going to the server. Additionally, there is general server side validation for cases like duplicate names or other validations requiring database lookups.

Contents

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    [ClientValidation(IsRequired = true, AllowSave = true)]
    public string FirstName { get; set; }

    [ClientValidation(IsRequired = true, AllowSave = false, MinLength = 1, MaxLength
↪= 100)]
    public string LastName { get; set; }
}
```

Properties

Behavioral Properties

`public bool AllowSave { get; set; };` If set to `true`, any client validation errors on the property will not prevent saving on the client. This includes **all** client-side validation, including null-checking for required foreign keys and other validations that are implicit. This also includes other explicit validation from `System.ComponentModel.DataAnnotations` annotations.

Instead, validation errors will be treated only as warnings, and will be available through the `warnings: KnockoutValidationErrors` property on the TypeScript ViewModel.

Tip: Use `AllowSave = true` to allow partially complete data to still be saved, protecting your user from data loss upon navigation while still hinting to them that they are not done filling out data.

`public string ErrorMessage { get; set; }` Set an error message to be used if any client validations fail

Pass-Through Properties

The following properties all map directly to Knockout-Validation properties.

```
public bool IsRequired { get; set; }
public double MinValue { get; set; } = double.MaxValue;
public double MaxValue { get; set; } = double.MinValue;
public double MinLength { get; set; } = double.MaxValue;
public double MaxLength { get; set; } = double.MinValue;
public double Step { get; set; }
public string Pattern { get; set; }
public bool IsEmail { get; set; }
public bool IsPhoneUs { get; set; }
public bool IsDate { get; set; }
public bool IsDateIso { get; set; }
public bool IsNumber { get; set; }
public bool IsDigit { get; set; }
```

The following properties are outputted to TypeScript unquoted. If you need to assert equality to a string, wrap the value you set to this property in quotes. Other literals (numerics, bools, etc) need no wrapping.

```
public string Equal { get; set; }
public string NotEqual { get; set; }
```

Custom Knockout Validation Extenders

Used together to specify a custom Knockout-Validation property.

It will be emitted into the TypeScript as `this.extend({ CustomName: CustomValue })`. Neither value will be quoted in the emitted TypeScript - add quotes to your value as needed to generate valid TypeScript.

```
public string CustomName { get; set; }
public string CustomValue { get; set; }
```

[Coalesce]

Used to mark a type or member for generation by Coalesce.

Some types and members will be implcitly included in generation - for example, all types represented by a `DbSet<T>` on a `DbContext` that has a `[Coalesce]` attribute will automatically be included. Properties on these types will also be generated for unless explicitly excluded, since this is by far the most common usage scenario in Coalesce.

On the other hand, *Methods* on these types will not have endpoints generated unless they are explicitly annotated with `[Coalesce]` to avoid accidentally exposing methods that were perhaps not intended to be exposed.

The documentation pages for types and members that require/accept this attribute will state as such. An exhaustive list of all valid targets for `[Coalesce]` will not be found on this page.

[Controller]

Allows for control over the generated MVC Controllers.

Currently only controls over the API controllers are present, but additional properties may be added in the future.

This attribute may be placed on any type from which an API controller is generated, including *EF Entity Models*, *Custom DTOs*, and *Services*.

Example Usage

```
[Controller(ApiRouted = false, ApiControllerSuffix = "Gen", ApiActionsProtected =␣
↪true)]
public class Person
{
    public int PersonId { get; set; }

    ...
}
```

Properties

`public bool ApiRouted { get; set; } = true;` Determines whether or not a `[Route]` annotation will be placed on the generated API controller. Set to `false` to prevent emission of the `[Route]` attribute.

Use cases include:

- Defining your routes through IRouteBuilder in Startup.cs instead
- Preventing API controllers from being exposed by default.
- Routing to your own custom controller that inherits from the generated API controller in order to implement more granular or complex authorization logic.

`public string ApiControllerName { get; set; } = null;` If set, will determine the name of the generated API controller.

Takes precedence over the value of `ApiControllerSuffix`.

public string ApiControllerSuffix { get; set; } = null; If set, will be appended to the default name of the API controller generated for this model.

Will be overridden by the value of `ApiControllerName` if it is set.

public bool ApiActionsProtected { get; set; } = false; If true, actions on the generated API controller will have an access modifier of `protected` instead of `public`.

In order to consume the generated API controller, you must inherit from the generated controller and override each desired generated action method via hiding (i.e. use `public new ...`, not `public override ...`).

Note: If you inherit from the generated API controllers and override their methods without setting `ApiActionsProtected = true`, all non-overriden actions from the generated controller will still be exposed as normal.

[ControllerAction]

Specifies the HTTP method/verb to use in the generated API controller for the model method. If this attribute is excluded or no method is specified, the controller action will use POST. Note that using the GET method will cause all method parameters to be returned as URL parameters which are not encrypted.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }
    public string {get; set; }

    [Coalesce]
    [ControllerAction(Method = HttpMethod.Get)]
    public static long PersonCount(AppDbContext db, string␣
↪lastNameStartsWith = "")
    {
        return db.People.Count(f => f.LastName.
↪StartsWith(lastNameStartsWith));
    }
}
```

Properties

public HttpMethod Method { get; set; } 1 The HTTP method to use on the generated API Controller.

Enum values are:

- `HttpMethod.Post` Use the POST method.
- `HttpMethod.Get` Use the GET method.
- `HttpMethod.Put` Use the PUT method.
- `HttpMethod.Delete` Use the DELETE method.

- `HttpMethod.Patch` Use the PATCH method.

[CreateController]

By default an API and View controller are both created. This allows for supressing the creation of either or both of these.

Example Usage

```
[CreateController(view: false, api: true)]
public class Person
{
    public int PersonId { get; set; }

    ...
}
```

Properties

`public bool WillCreateView { get; set; }` 1

`public bool WillCreateApi { get; set; }` 2

[DateType]

Specifies whether a DateTime type will have a date and a time, or only a date.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    [DateType(DateTypeAttribute.DateTypes.DateOnly)]
    public DateTimeOffset? BirthDate { get; set; }
}
```

Properties

`public DateTypes DateType { get; set; }` **1** The type of date the property represents.

Enum values are:

- `DateTypeAttribute.DateTypes.DateTime` Subject is both a date and time.
- `DateTypeAttribute.DateTypes.DateOnly` Subject is only a date with no significant time component.

[DefaultOrderBy]

Allows setting of the default manner in which the data returned to the client will be sorted. Multiple fields can be used to sort an object by specifying an index.

This affects the sort order both when requesting a list of the model itself, as well as when the model appears as a child collection off of a navigation property of another object.

In the first case (a list of the model itself), this can be overridden by setting the `orderBy` or `orderByDescending` property on the TypeScript `ListViewModel` - see *TypeScript ListViewModels*.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    public int DepartmentId { get; set; }

    [DefaultOrderBy(FieldOrder = 0, FieldName = nameof(Department.Order))]
    public Department Department { get; set; }

    [DefaultOrderBy(FieldOrder = 1)]
    public string LastName { get; set; }
}
```

```
public class LoginHistory
{
    public int LoginHistoryId {get; set;}

    [DefaultOrderBy(OrderByDirection = DefaultOrderByAttribute.
↪OrderByDirections.Descending)]
    public DateTime Date {get; set;}
}
```

Properties

`public int FieldOrder { get; set; }` 1 Specify the index of this field when sorting by multiple fields.

Lower-valued properties will be used first; higher-valued properties will be used as a tiebreaker (i.e. `.ThenBy(...)`).

`public OrderByDirections OrderByDirection { get; set; }` 2 Specify the direction of the ordering for the property.

Enum values are:

- `DefaultOrderByAttribute.OrderByDirections.Ascending`
- `DefaultOrderByAttribute.OrderByDirections.Descending`

`public string FieldName { get; set; }` When using the `DefaultOrderByAttribute` on an object property, specifies the field on the object to use for sorting. See the first example above.

[DtoIncludes] & [DtoExcludes]

Allows for easily controlling what data gets set to the client. When requesting data from the generated client-side list view models, you can specify an `includes` property on the ViewModel or ListViewModel.

For more information about the includes string, see *Includes String*.

When the database entries are returned to the client they will be trimmed based on the requested includes string and the values in `DtoExcludes` and `DtoIncludes`.

> **Caution:** These attributes are **not security attributes** - consumers of your application's API can set the includes string to any value when making a request.
>
> Do not use them to keep certain data private - use the *Security Attributes* family of attributes for that.

It is important to note that the value of the includes string will match against these attributes on *any* of your models that appears in the object graph being mapped to DTOs - it is not limited only to the model type of the root object.

Example Usage

```
public class Person
{
    // Don't include CreatedBy when editing - will be included for all other views
    [DtoExcludes("Editor")]
    public AppUser CreatedBy { get; set; }

    // Only include the Person's Department when :ts:`includes == "details"` on the TypeScript ViewModel.
    [DtoIncludes("details")]
    public Department Department { get; set; }

    // LastName will be included in all views
    public string LastName { get; set; }
}

public class Department
{
    [DtoIncludes("details")]
    public ICollection<Person> People { get; set; }
}
```

In TypeScript:

```
var personList = new ListViewModels.PersonList();
personList.includes = "Editor";
personList.load(() => {
    // objects in personList.items will not contain CreatedBy nor Department objects.
});
```

```
var personList = new ListViewModels.PersonList();
personList.includes = "details";
personList.load(() => {
```

(continues on next page)

(continued from previous page)

```
    // objects in personList.items will be allowed to contain both CreatedBy␣
↪and Department objects. Department will be allowed to include its other␣
↪Person objects.
});
```

Properties

`public string ContentViews { get; set; }` 1 A comma-delimited list of values of `includes` on which to operate.

For `DtoIncludes`, this will be the values of `includes` for which this property will be allowed to be serialized and sent to the client.

Important: `DtoIncludes` does not ensure that specific data will be loaded from the database. Only data loaded into current EF DbContext can possibly be returned from the API. See *Data Sources* for more information.

For `DtoExcludes`, this will be the values of `includes` for which this property will **never** be serialized and sent to the client.

[Execute]

Controls permissions for executing of a static or instance method through the API.

For other security controls, see *Security Attributes*.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    [Coalesce, Execute(Roles = "Payroll,HR")]
    public void GiveRaise(int centsPerHour) {
        ...
    }

    ...
}
```

Properties

`public string Roles { get; set; }` A comma-separated list of roles which are allowed to execute the method.

`public SecurityPermissionLevels PermissionLevel { get; set; } = SecurityPermissionLeve` The level of access to allow for the action for the method.

Enum values are:

- `SecurityPermissionLevels.AllowAll` Allow all users to perform the action for the attribute, including users who are not authenticated at all.
- `SecurityPermissionLevels.AllowAuthorized` Allow only users who are members of the roles specified on the attribute to perform the action. If no roles are specified on the attribute, then all authenticated users are allowed (no anonymous access).
- `SecurityPermissionLevels.DenyAll` Deny the action to all users, regardless of authentication status or authorization level. If `DenyAll` is used, no API endpoint for the action will be generated.

[Hidden]

Mark an property as hidden from the edit, List or All areas.

> **Caution:** This attribute is **not a security attribute** - it only affects the HTML of the generated views. It has no impact on data visibility in the API.
>
> Do not use it to keep certain data private - use the *Security Attributes* family of attributes for that.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    [Hidden(HiddenAttribute.Areas.All)]
    public int? IncomeLevelId { get; set; }
}
```

Properties

`public Areas Area { get; set; }` 1 The areas in which the property should be hidden.

Enum values are:

- `HiddenAttribute.Areas.All` Hide from all generated views
- `HiddenAttribute.Areas.List` Hide from generated list views only (Table/Cards)
- `HiddenAttribute.Areas.Edit` Hide from generated editor only (CreateEdit)

[Inject]

Used to mark a method parameter for dependency injection from the application's `IServiceProvider`.

See *Methods* for more.

This gets translated to a `Microsoft.AspNetCore.Mvc.FromServicesAttribute` in the generated API controller's action.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public string GetFullName([Inject] ILogger<Person> logger)
    {
        logger.LogInformation(0, "Person " + PersonId + "'s full name was␣
↪requested");
        return FirstName + " " + LastName";
    }
}
```

[InternalUse]

Used to mark a property or method for internal use. Internal Use members are:

- Not exposed via the API.
- Not present in the generated TypeScript view models.
- Not present nor accounted for in the generated C# DTOs.
- Not present in the generated editor or list views.

Effectively, an Internal Use member is invisible to Coalesce.

> Note that this only needs to be used on members that are public. Non-public members are always invisible to Coalesce.

Example Usage

In this example, `Color` is the property exposed to the API, but `ColorHex` is the property that maps to the database that stores the value. A helper method also exists for the color generation, but needs no attribute to be hidden since methods must be explicitly exposed with *[Coalesce]*.

If no color is saved in the database (the user hasn't picked a color), one is deterministically created.

```
public class ApplicationUser
{
    public int ApplicationUserId { get; set; }

    [InternalUse]
    public string ColorHex { get; set; }

    [NotMapped]
    public string Color
    {
        get => ColorHex ?? GenerateColor(ApplicationUserId).ToRGBHexString();
        set => ColorHex = value;
    }

    public static HSLColor GenerateColor(int? seed = null)
```

(continues on next page)

(continued from previous page)

```
    {
        var random = seed.HasValue ? new Random(seed.Value) : new Random();
        return new HSLColor(random.NextDouble(), random.Next(40, 100) / 100d,
↪ random.Next(25, 65) / 100d);
    }
}
```

[ListText]

When a dropdown list is used to select a related object, this controls the text shown in the dropdown by default. When using these dropdown, only the key and this field are returned as search results by the API.

The property with this attribute will also be used as the displayed text for reference navigation properties when they are displayed as text using the *Computed Text Properties* properties on the *TypeScript ViewModels*.

If this attribute is not used, and a property named "Name" exists on the model, that property will be used. Otherwise, the primary key will be used.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    [ListText]
    [Hidden]
    [NotMapped]
    public string Name => FirstName + " " + LastName
}
```

[LoadFromDataSource]

Specifies that the targeted model instance method should load the instance it is called on from the specified data source when invoked from an API endpoint. By default, whatever the default data source for the model's type will be used.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }
    public string {get; set; }

    [Coalesce, LoadFromDataSource(typeof(WithoutCases))]
    public void ChangeSpacesToDashesInName()
    {
        FirstName = FirstName.Replace(" ", "-");
    }
}
```

Properties

public Type DataSourceType { get; set; } 1 The DataSource to load the instance object from.

[ManyToMany]

Used to specify a Many to Many relationship. Because EF core does not support automatic intermediate mapping tables, this field is used to allow for direct reference of the many-to-many collections from the ViewModel.

The named specified in the attribute will be used as the name of a collection of the objects on the other side of the relationship in the generated TypeScript.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [ManyToMany("Appointments")]
    public ICollection<PersonAppointment> PersonAppointments { get; set; }
}
```

Properties

public string CollectionName { get; } 1 The name of the collection that will contain the set of objects on the other side of the many-to-many relationship.

[Search]

Coalesce supports searching through the generated API in its various implementations, including the generated list views (Table & Cards), in Select2 dropdowns, and directly through the TypeScript ListViewModels' `search` property.

The `search` parameter of the API can also be formatted as `PropertyName:SearchTerm` in order to search on an arbitrary property of a model. For example, a value of `Nickname:Steve-o` for a search term would search the `Nickname` property, even through it is not marked as searchable using this attribute.

By default, the system will search any field with the name 'Name'. If this doesn't exist, the ID is used as the only searchable field. Once you place the `Search` attribute on one or more properties on a model, only those annotated properties will be searched.

The following types can be searched:

Strings String fields will be searched based on the `SearchMethod` property on the attribute. See below.

Numeric Types If the input is numeric, numeric fields will be searched for the exact value.

Enums If the input is a valid name of an enum value for an enum property and that property is searchable, rows will be searched for the exact value.

Dates If the input is a parsable date, rows will be searched based on that date.

Date search will do its best to guess at the user's intentions:

- Various forms of year/month combos are supported, and if only a year/month is inputted, it will look for all dates in that month, e.g. "Feb 2017" or "2016-11".
- A date without a time (or a time of exactly midnight) will search the entire day, e.g. "2017/4/18".
- A date/time with minutes and seconds equal to 0 will search the entire hour, e.g. "April 7, 2017 11 AM".

Tip: When searching on date properties, you should almost always set `IsSplitOnSpaces = false` on the `Search` attribute. This allows natural inputs like "July 21, 2017" to search correctly. Otherwise, only non-whitespaced date formats will work, like "2017/21/07".

Reference Navigation Properties When a reference navigation property is marked with `[Search]`, searchable properties on the referenced object will also be searched. This behavior will go up to two levels away from the root object, and can be controlled with the `RootWhitelist` and `RootBlacklist` properties on the `[Search]` attribute that are outlined below.

Collection Navigation Properties When a collection navigation property is marked with `[Search]`, searchable properties on the child objects will also be searched. This behavior will go up to two levels away from the root object, and can be controlled with the `RootWhitelist` and `RootBlacklist` properties on the `[Search]` attribute that are outlined below.

> **Warning:** Searches on collection navigation properties usually don't translate well with EF Core, leading to potentially degraded performance. Use this feature cautiously.

Example Usage

```
public class Person
{
    public int PersonId { get; set; }

    [Search]
    public string FirstName { get; set; }

    [Search]
    public string LastName { get; set; }

    [Search(IsSplitOnSpaces = false)]
    public string BirthDate { get; set; }

    public string Nickname { get; set; }

    [Search(RootWhitelist = nameof(Person))]
    public ICollection<Address> Addresses { get; set; }
}
```

Properties

`public bool IsSplitOnSpaces { get; set; } = true;` If set to true (the default), each word in the search terms will be searched for in each searchable field independently, and a row will only be considered a match if each word in the search term is a match on at least one searchable property where `IsSplitOnSpaces == true`

This is useful when searching for a full name across two or more fields. In the above example, using `IsSplitOnSpaces = true` would provide more intuitive behavior since it will search both first name and last name for each word entered into the search field. But, you probably shouldn't be doing that in the first place.

`public SearchMethods SearchMethod { get; set; } = SearchMethods.BeginsWith;` For string properties, specifies whether the value of the field will be checked using `Contains` or using `BeginsWith`.

Note that standard database indexing can be used to speed up `BeginsWith` searches.

`public string RootWhitelist { get; set; } = null;` A comma-delimited list of model class names that, if set, will prevent the targeted property from being searched unless the root object of the API call was one of the specified class names.

`public string RootBlacklist { get; set; } = null;` A comma-delimited list of model class names that, if set, will prevent the targeted property from being searched if the root object of the API call was one of the specified class names.

Security Attributes

Coalesce provides a collection of four attributes which can provide class-level (and property-level, where appropriate) security controls over the generated API.

Properties

`public string Roles { get; set; }` **1** A comma-delimited list of roles that are authorized to take perform the action represented by the attribute. If the current user belongs to any of the listed roles, the action will be allowed.

The string set for this property will be outputted as an `[Authorize(Roles="RolesString")]` attribute on generated API controller actions.

`public SecurityPermissionLevels PermissionLevel { get; set; }` **2** The level of access to allow for the action for **class-level security** only. Has no effect for property-level security.

Enum values are:

- `SecurityPermissionLevels.AllowAll` Allow all users to perform the action for the attribute, including users who are not authenticated at all.
- `SecurityPermissionLevels.AllowAuthorized` Allow only users who are members of the roles specified on the attribute to perform the action. If no roles are specified on the attribute, then all authenticated users are allowed (no anonymous access).
- `SecurityPermissionLevels.DenyAll` Deny the action to all users, regardless of authentication status or authorization level. If `DenyAll` is used, no API endpoint for the action will be generated.

Class vs. Property Security

Important: There are important differences between class-level security and property-level security, beyond the usage of the attributes themselves. In general, class-level security is implemented in the generated API Controllers as `[Authorize]` attributes on the generated actions. Property security attributes are implemented in the *Generated DTOs*.

Implementations

Read

Controls permissions for reading of objects and properties through the API.

For **property-level** security only, if a `[Read]` attribute is present without an `[Edit]` attribute, the property is read-only.

Example Usage

```
[Read(Roles = "Management", PermissionLevel = SecurityPermissionLevels.
↪AllowAuthorized)]
public class Employee
{
    public int EmployeeId { get; set; }

    [Read("Payroll")]
    public string LastFourSsn { get; set; }

    ...
}
```

Edit

Controls permissions for editing of objects and properties through the API.

For **property-level** security only, if a `[Read]` attribute is present, one of its roles must be fulfilled in addition to the roles specified (if any) for the `[Edit]` attribute..

Example Usage

```
[Edit(Roles = "Management,Payroll", PermissionLevel = SecurityPermissionLevels.
↪AllowAuthorized)]
public class Employee
{
    public int EmployeeId { get; set; }
```

(continues on next page)

(continued from previous page)

```
    [Read("Payroll,HumanResources"), Edit("Payroll")]
    public string LastFourSsn { get; set; }

    ...
}
```

Create

Controls permissions for deletion of an object of the targeted type through the API.

Example Usage

```
[Create(Roles = "HumanResources", PermissionLevel = SecurityPermissionLevels.
↪AllowAuthorized)]
public class Employee
{
    ...
}
```

Delete

Controls permissions for deletion of an object of the targeted type through the API.

Example Usage

```
[Delete(Roles = "HumanResources,Management", PermissionLevel =␣
↪SecurityPermissionLevels.AllowAuthorized)]
public class Employee
{
    ...
}
```

Execute

A separate attribute for controlling method execution exists. Its documentation may be found on the *[Execute]* page.

[SelectFilter]

Specify a property to restrict dropdown menus by. Values presented will be only those where the value of the foreign property matches the value of the local property.

The local property name defaults to the same value of the foreign property.

Additionally, in place of a `LocalPropertyName` to check against, you may instead specify a static value using `StaticPropertyValue` to filter by a constant.

Important: This attribute only affects generated HTML - it does not enforce any relational rules in your data.

Example Usage

In this example, a dropdown for `EmployeeRank` created using `@Knockout.SelectForObject` in cshtml files will only present possible values of `EmployeeRank` which are valid for the `EmployeeType` of the `Employee`.

```
public class Employee
{
    public int EmployeeId { get; set; }
    public int EmployeeTypeId { get; set; }
    public EmployeeType EmployeeType { get; set; }
    public int EmployeeRankId { get; set; }

    [SelectFilter(ForeignPropertyName = nameof(EmployeeRank.EmployeeTypeId),␣
↪LocalPropertyName = nameof(Employee.EmployeeTypeId))]
    public EmployeeRank EmployeeRank { get; set; }
}

public class EmployeeRank
{
    public int EmployeeRankId { get; set; }
    public int EmployeeTypeId { get; set; }
    public EmployeeType EmployeeType { get; set; }
}
```

```
<div>
    @(Knockout.SelectForObject<Models.Employee>(e => e.EmployeeRank))
</div>
```

Properties

`public string ForeignPropertyName { get; set; }` The name of the property on the foreign object to filter against.

`public string LocalPropertyName { get; set; }` The name of another property belonging to the class in which this attribute is used. The results of select lists will be filtered to match this value.

Defaults to the value of `ForeignPropertyName` if not set.

public string LocalPropertyObjectName { get; set; } If specified, the `LocalPropertyName` will be resolved from the property by this name that resides on the local object.

This allows for querying against properties that are one level away from the current object.

public string StaticPropertyValue { get; set; } A constant value that the foreign property will be filtered against. This string must be parsable into the foreign property's type to have any effect. If this is set, `LocalPropertyName` will be ignored.

[TypeScriptPartial]

If defined on a model, a typescript file will be generated in ./Scripts/Partials if one does not already exist. This 'Partial' TypeScript file contains a class which inherits from the generated TypeScript ViewModel. The partial class has the same name as the generated ViewModel would normally have, and the generated ViewModel is renamed to `"<ClassName>Partial"`.

This behavior allows you to extend the behavior of the generated TypeScript view models with your own properties and methods for defining more advanced behavior on the client. One of the most common use cases of this is to define additional Knockout `ComputedObservable` properties for information that is only useful in the browser - for example, computing a css class based on data in the object.

Example Usage

```
[TypeScriptPartial]
public class Employee
{
    public int EmployeeId { get; set; }

    ...
}
```

Properties

public string BaseClassName { get; set; } If set, overrides the name of the generated ViewModel which becomes the base class for the generated 'Partial' TypeScript file.

ComponentModel Attributes

Coalesce also supports a number of the built-in `System.ComponentModel.DataAnnotations` attributes and will use these to shape the generated code.

Display

The displayed name name of a property, as well as the order in which it appears in generated views, can be set via the `[Display]` attribute. By default, properties will be displayed in the order in which they are defined in their class.

DisplayName

The displayed name name of a property can also be set via the `[DisplayName]` attribute.

Required

Properties with `[Required]` will generate Knockout-Validation extenders. See *[ClientValidation]*.

Range

Properties with `[Range]` will generate Knockout-Validation extenders. See *[ClientValidation]*.

MinLength

Properties with `[MinLength]` will generate Knockout-Validation extenders. See *[ClientValidation]*.

MaxLength

Properties with `[MaxLength]` will generate Knockout-Validation extenders. See *[ClientValidation]*.

ForeignKey

Normally, Coalesce figures out which properties are foreign keys, but if you don't use standard EF naming conventions then you'll need to annotate with `[ForeignKey]` to help out both EF and Coalesce. See the Entity Framework Relationships documentation for more.

InverseProperty

Sometimes, Coalesce (and EF, too) can have trouble figuring out what the foreign key is supposed to be for a collection navigation property. See the Entity Framework Relationships documentation for details on how and why to use `[InverseProperty]`.

NotMapped

Model properties that aren't mapped to the database should be marked with `[NotMapped]` so that Coalesce doesn't try to load them from the database when *searching* or carrying out the *Default Loading Behavior*.

3.2.7 Methods

Any public methods you place on your POCO classes that are annotated with the *[Coalesce]* will get built into your TypeScript ViewModels and ListViewModels, and API endpoints will be created for these methods to be called. Both instance methods and static methods are supported. Additionally, any instance methods on *Services* will also have API endpoints and TypeScript generated.

Contents

Parameters

The following parameters can be added to your methods:

Primitives & Dates Primitive values (numerics, strings, booleans, enums) and dates (`DateTime`, `DateTimeOffset`, and nullable variants) are accepted as parameters to be passed from the client to the method call.

Objects Any object types may be passed to the method call. These may be existing *EF Entity Models* or *External Types*. When invoking the method on the client, the object's properties will only be serialized one level deep. If an object parameter has additional child object properties, they will not be included in the invocation of the method - only the object's primitive & date properties will be deserialized from the client.

`<YourDbContext> db` If the method has a parameter of the same type as your DbContext class, the current DbContext will be passed to the method call. For *Services* which don't have a defined backing EF context, this is treated as having an implicit `[Inject]` attribute.

`ClaimsPrincipal user` If the method has a parameter of type ClaimsPrincipal, the current user will be passed to the method call.

`[Inject] <anything>` If a parameter is marked with the *[Inject]* attribute, it will be injected from the application's `IServiceProvider`.

`out IncludeTree includeTree` If the method has an `out IncludeTree includeTree` parameter, then the `IncludeTree` that is passed out will be used to control serialization. See *Generated DTOs* and *Include Tree* for more information. If the method returns an `IQueryable`, this will supercede the include tree obtained from inspecting the query.

Return Values

You can return virtually anything from these methods:

Primitives & Dates Any primitive data types may be returned - `string`, `int`, etc.

Model Types Any of the types of your models may be returned. The generated TypeScript for calling the method will use the generated TypeScript ViewModels of your models to store the returned value.

If the return type is the same as the type that the method is defined on, and the method is not static, then the results of the method call will be loaded into the calling TypeScript object.

Custom Types Any custom type you define may also be returned from a method. Corresponding TypeScript ViewModels will be created for these types. See *External Types*.

> **Warning:** When returning custom types from methods, be careful of the types of their properties. As Coalesce generates the TypeScript ViewModels for your *External Types*, it will also generate ViewModels for the types of any of its properties, and so on down the tree. If a type is encountered from the FCL/BCL or another package that your application uses, these generated types will get out of hand extremely quickly.
>
> Mark any properties you don't want generated on these TypeScript ViewModels with the *[InternalUse]* attribute, or give them a non-public access modifier. Whenever possible, don't return types that you don't own or control.

`ICollection<T>` or `IEnumerable<T>` Collections of any of the above valid return types above are also valid return types. IEnumerables are useful for generator functions using `yield`. `ICollection` is highly suggested over `IEnumerable` whenever appropriate, though.

`IQueryable<T>` Queryables of the valid return types above are valid return types. The query will be evaluated, and Coalesce will attempt to pull an *Include Tree* from the queryable to shape the response. When *Include Tree* functionality is needed to shape the response but an `IQueryable<>` return type is not feasible, an `out IncludeTree includeTree` parameter will do the trick as well.

`IntelliTect.Coalesce.Models.ItemResult<T>` or `ItemResult` An `ItemResult<T>` of any of the valid return types above, including collections, is valid. The `WasSuccessful` and `Message` properties on the result object will be sent along to the client to indicate success or failure of the method. The type `T` will be mapped to the appropriate DTO object before being serialized as normal.

`IntelliTect.Coalesce.Models.ListResult<T>` A `ListResult<T>` of any of the non-collection types above, is valid. The `WasSuccessful` `Message`, and all paging information on the result object will be sent along to the client. The type `T` will be mapped to the appropriate DTO objects before being serialized as normal.

The class created for the method in TypeScript will be used to hold the paging information included in the ListResult. See below for more information about this class.

Security

You can implement role-based security on a method by placing the *[Execute]* on the method. Placing this attribute on the method with no roles specified will simply require that the calling user be authenticated.

Security for instance methods is also controlled by the data source that loads the instance - if the data source can't provide an instance of the requested model, the method won't be executed.

Generated TypeScript

For each method you define, a class will be created on the corresponding TypeScript ViewModel (instance methods) or ListViewModel (static methods) that contains the properties and functions for interaction with the method. This class is accessible through a static property named after the method. An instance of this class will also be created on each instance of its parent - this instance is in a property with the camel-cased name of the method.

Here's an example for a method called Rename that takes a single parameter 'string name' and returns a string.

```
public string Rename(string name)
{
    FirstName = name;
    return FullName; // Return the new full name of the person.
}
```

Method-specific Members

public static Rename = class Rename extends Coalesce.ClientMethod<Person, string> { ... }
Declaration of class that provides invocation methods and status properties for the method.

public readonly rename = new Person.Rename(this) Default instance of the method for easy calling of the method without needing to manually instantiate the class.

public invoke: (name: string, callback: (result: string) => void = null, reload: boole
Function that takes all the method parameters and a callback. If `reload` is true, the ViewModel or ListViewModel that owns the method will be reloaded after the call is complete, and only after that happens will the callback be called.

The following members are only generated for methods with arguments:

public static Args = class Args { public name: KnockoutObservable<string> = ko.observable
Class with one observable member per method argument for binding method arguments to user input.

public args = new Rename.Args() Default instance of the args class.

public invokeWithArgs: (args = this.args, callback?: (result: string) => void, reload:
Function for invoking the method using the args class. The default instance of the args class will be used if none is provided.

public invokeWithPrompts: (callback: (result: string) => void = null, reload: boolean =
Simple interface using browser `prompt()` input boxes to prompt the user for the required data for the method call. The call is then made with the data provided.

Base Class Members

public result: KnockoutObservable<string> Observable that will contain the results of the method call after it is complete.

public rawResult: KnockoutObservable<Coalesce.ApiResult> Observable with the raw, deserialized JSON result of the method call. If the method call returns an object, this will contain the deserialized JSON object from the server before it has been loaded into ViewModels and its properties loaded into observables.

public isLoading: KnockoutObservable<boolean> Observable boolean which is true while the call to the server is pending.

public message: KnockoutObservable<string> If the method was not successful, this contains exception information.

public wasSuccessful: KnockoutObservable<boolean> Observable boolean that indicates whether the method call was successful or not.

ListResult<T> Method Members

public page: KnockoutObservable<number> Page number of the results.

public pageSize: KnockoutObservable<number> Page size of the results.

public pageCount: KnockoutObservable<number> Total number of possible result pages.

public totalCount: KnockoutObservable<number> Total number of results.

Instance Methods

Instance methods generate the members above on the TypeScript ViewModel.

The instance of the model will be loaded using the data source specified by an attribute `[LoadFromDataSource(typeof(MyDataSource))]` if present. Otherwise, the model instance will be loaded using the default data source for the POCO's type. If you have a *Custom Data Source* annotated with `[DefaultDataSource]`, that data source will be used. Otherwise, the *Standard Data Source* will be used.

Static Methods

Static methods are created as functions on the TypeScript ListViewModel. All of the same members that are generated for instance methods are also generated for static methods.

If a static method returns the type that it is declared on, it will also be generated on the TypeScript ViewModel of its class.

```
public static ICollection<string> NamesStartingWith(string characters, AppDbContext␣
↪db)
{
    return db.People.Where(f => f.FirstName.StartsWith(characters)).Select(f => f.
↪FirstName).ToList();
}
```

Method Annotations

Methods can be annotated with attributes to control API exposure and TypeScript generation. The following attributes are available for model methods. General annotations can be found on the *Attributes* page.

[Coalesce] The *[Coalesce]* attribute causes the method to be exposed via a generated API controller. This is not needed for methods defined on an interface marked with `[Service]` - Coalesce assumes that all methods on the interface are intended to be exposed. If this is not desired, create a new, more restricted interface with only the desired methods to be exposed.

[ControllerAction(Method = HttpMethod)] The *[ControllerAction]* attribute controls how this method is exposed via HTTP. By default all controller method actions use the POST HTTP method. This behavior can be overridden with this attribute to use GET, POST, PUT, DELETE, or PATCH HTTP methods. Keep in mind that when using the GET method, all parameters are sent as part of the URL, so the typical considerations with sensitive data in a query string applies.

[Execute(string roles)] The *[Execute]* attribute specifies which roles can execute this method from the generated API controller.

[Hidden(Areas area)] The *[Hidden]* attribute allows for hiding this method on the admin pages both for list/card views and the editor.

[LoadFromDataSource(Type dataSourceType)] The *[LoadFromDataSource]* attribute specifies that the targeted model instance method should load the instance it is called on from the specified data source when invoked from an API endpoint. By default, whatever the default data source for the model's type will be used.

3.2.8 Data Sources

In Coalesce, all data that is retrieved from your database through the generated controllers is done so by a data source. These data sources control what data gets loaded and how it gets loaded. By default, there is a single generic data source that serves all data for your models in a generic way that fits many of the most common use cases - the *Standard Data Source*.

In addition to this standard data source, Coalesce allows you to create custom data sources that provide complete control over the way data is loaded and serialized for transfer to a requesting client. These data sources are defined on a per-model basis, and you can have as many of them as you like for each model.

Contents

Defining Data Sources

By default, each of your models that Coalesce exposes will expose the standard data source (`IntelliTect.Coalesce.StandardDataSource<T, TContext>`). This data source provides all the standard functionality one would expect - paging, sorting, searching, filtering, and so on. Each of these component pieces is implemented in one or more virtual methods, making the `StandardDataSource` a great place to start from when implementing

your own data source. To suppress this behavior of always exposing the raw `StandardDataSource`, create your own custom data source and annotate it with `[DefaultDataSource]`.

To implement your own custom data source, you simply need to define a class that implements `IntelliTect.Coalesce.IDataSource<T>`. To expose your data source to Coalesce, either place it as a nested class of the type `T` that you data source serves, or annotate it with the `[Coalesce]` attribute. Of course, the easiest way to create a data source that doesn't require you to re-engineer a great deal of logic would be to inherit from `IntelliTect.Coalesce.StandardDataSource<T, TContext>`, and then override only the parts that you need.

```
public class Person
{
    [DefaultDataSource]
    public class IncludeFamily : StandardDataSource<Person, AppDbContext>
    {
        public IncludeFamily(CrudContext<AppDbContext> context) :␣
↪base(context) { }

        public override IQueryable<Person> GetQuery(IDataSourceParameters␣
↪parameters)
            => Db.People
            .Where(f => User.IsInRole("Admin") || f.CreatedById == User.
↪GetUserId())
            .Include(f => f.Parents).ThenInclude(s => s.Parents)
            .Include(f => f.Cousins).ThenInclude(s => s.Parents);
    }
}

[Coalesce]
public class NamesStartingWithA : StandardDataSource<Person, AppDbContext>
{
    public NamesStartingWithA(CrudContext<AppDbContext> context) :␣
↪base(context) { }

    public override IQueryable<Person> GetQuery(IDataSourceParameters␣
↪parameters)
        => Db.People.Include(f => f.Siblings).Where(f => f.FirstName.
↪StartsWith("A"));
}
```

The structure of the `IQueryable` built by the various methods of `StandardDataSource` is used to shape and trim the structure of the DTO as it is serialized and sent out to the client. One may also override method `IncludeTree GetIncludeTree(IQueryable<Person> query, IDataSourceParameters parameters)` to control this explicitly. See *Include Tree* for more information on how this works.

Warning: If you create a custom data source that has custom logic for securing your data, be aware that the default implementation of `StandardDataSource` (or your custom default implementation - see below) is still exposed unless you annotate one of your custom data sources with `[DefaultDataSource]`. Doing so will replace the default data source with the annotated class for your type `T`.

Dependency Injection

All data sources are instantiated using dependency injection and your application's `IServiceProvider`. As a result, you can add whatever constructor parameters you desire to your data sources as long as a value for them can be resolved from your application's services. The single parameter to the `StandardDataSource` is resolved in

this way - the `CrudContext<TContext>` contains the common set of objects most commonly used, including the `DbContext` and the `ClaimsPrincipal` representing the current user.

Consuming Data Sources

The TypeScript ViewModels and ListViewModels have a property called `dataSource`. These properties accept an instance of a `Coalesce.DataSource<T>`. Generated classes that satisfy this relationship for all the data sources that were defined in C# may be found in the `dataSources` property on an instance of a ViewModel or ListViewModel, or in `ListViewModels.<ModelName>DataSources`

```
var viewModel = new ViewModels.Person();
viewModel.dataSource = new viewModel.dataSources.IncludeFamily();
viewModel.load(1);

var list = new ListViewModels.PersonList();
list.dataSource = new list.dataSources.NamesStartingWith();
list.load();
```

Standard Parameters

All methods on `IDataSource<T>` take a parameter that contains all the client-specified parameters for things paging, searching, sorting, and filtering information. Almost all overridable methods on `StandardDataSource` are also passed the relevant set of parameters.

Custom Parameters

On any data source that you create, you may add additional properties annotated with `[Coalesce]` that will then be exposed as parameters to the client. These property parameters are currently restricted to primitives (numeric types, strings) and dates (DateTime, DateTimeOffset). Property parameter primitives may be expanded to allow for more types in the future.

```
[Coalesce]
public class NamesStartingWith : StandardDataSource<Person, AppDbContext>
{
    public NamesStartingWith(CrudContext<AppDbContext> context) :␣
↪base(context) { }

    [Coalesce]
    public string StartsWith { get; set; }

    public override IQueryable<Person> GetQuery(IDataSourceParameters␣
↪parameters)
        => Db.People.Include(f => f.Siblings)
        .Where(f => string.IsNullOrWhitespace(StartsWith) ? true : f.
↪FirstName.StartsWith(StartsWith));
}
```

The properties created on the TypeScript objects are observables so they may be bound to directly. In order to automatically reload a list when a data source parameter changes, you must explicitly subscribe to it:

```
var list = new ListViewModels.PersonList();
var dataSource = new list.dataSources.NamesStartingWith();
dataSource.startsWith("Jo");
```

(continues on next page)

(continued from previous page)

```
dataSource.subscribe(list); // Optional - call to enable automatic reloading.
list.dataSource = dataSource;
list.load();
```

Standard Data Source

The standard data source, `IntelliTect.Coalesce.StandardDataSource<T, TContext>`, contains a significant number of properties and methods that can be utilized and/or overridden at your leisure.

Default Loading Behavior

When an object or list of objects is requested, the default behavior of the the `StandardDataSource` is to load all of the immediate relationships of the object (parent objects and child collections), as well as the far side of many-to-many relationships. This can be suppressed by settings `includes = "none"` on your TypeScript *ViewModel* or *ListViewModel* when making a request.

In most cases, however, you'll probably want more or less data than what the default behavior provides. You can achieve this by overriding the `GetQuery` method, outlined below.

Properties

The following properties are available for use on the `StandardDataSource`

`CrudContext<TContext> Context` The object passed to the constructor that contains the set of objects needed by the standard data source, and those that are most likely to be used in custom implementations.

`TContext Db` An instance of the db context that contains a `DbSet<T>` for the entity served by the data source.

`ClaimsPrincipal User` The user making the current request.

`int MaxSearchTerms` The max number of search terms to process when interpreting a search term word-by-word. Override by setting a value in the constructor.

`int DefaultPageSize` The page size to use if none is specified by the client. Override by setting a value in the constructor.

`int MaxPageSize` The maximum page size that will be served. By default, client-specified page sizes will be clamped to this value. Override by setting a value in the constructor.

Method Overview

The standard data source contains 19 different methods which can be overridden in your derived class to control its behavior.

These methods often call one another, so overriding one method may cause some other method to no longer be called. The hierarchy of method calls, ignoring any logic or conditions contained within, is as follows:

```
GetMappedItemAsync
    GetItemAsync
        GetQuery
```

(continues on next page)

(continued from previous page)

```
        GetIncludeTree
    TransformResults

GetMappedListAsync
    GetListAsync
        GetQuery
        ApplyListFiltering
            ApplyListPropertyFilters
                ApplyListPropertyFilter
            ApplyListSearchTerm
        GetListTotalCountAsync
        ApplyListSorting
            ApplyListClientSpecifiedSorting
            ApplyListDefaultSorting
        ApplyListPaging
        GetIncludeTree
    TrimListFields
    TransformResults

GetCountAsync
    GetQuery
    ApplyListFiltering
        ApplyListPropertyFilters
            ApplyListPropertyFilter
        ApplyListSearchTerm
    GetListTotalCountAsync
```

Method Details

All of the methods outlined above can be overridden. A description of each of the non-interface inner methods is as follows:

`GetQuery` The method is the one that you will most commonly be override in order to implement custom query logic. From this method, you could:

- Specify additional query filtering such as row-level security or soft-delete logic. Or, restrict the data source entirely for users or whole roles by returning an empty query.
- Include additional data using EF's `.Include()` and `.ThenInclude()`.
- Add additional edges to the serialized object graph using Coalesce's `.IncludedSeparately()` and `.ThenIncluded()`.

Note: When `GetQuery` is overridden, the *Default Loading Behavior* is overridden as well. To restore this behavior, use the `IQueryable<T>.IncludeChildren()` extension method to build your query.

`GetIncludeTree` Allows for explicitly specifying the *Include Tree* that will be used when serializing results obtained from this data source into DTOs. By default, the query that is build up through all the other methods in the data source will be used to build the include tree.

`CanEvalQueryAsynchronously` Called by other methods in the standard data source to determine whether or not EF Core async methods will be used to evaluate queries. This may be globally disabled when bugs like https://github.com/aspnet/EntityFrameworkCore/issues/9038 are present in EF Core.

`ApplyListFiltering` A simple wrapper that calls `ApplyListPropertyFilters` and `ApplyListSearchTerm`.

`ApplyListPropertyFilters` For each value in `parameters.Filter`, invoke `ApplyListPropertyFilter` to apply a filter to the query.

`ApplyListPropertyFilter` Given a property and a client-provided string value, perform some filtering on that property.

- Dates with a time component will be matched exactly.
- Dates with no time component will match any dates that fell on that day.
- Strings will match exactly unless an asterisk is found, in which case they will be matched with `string.StartsWith`.
- Enums will match by string or numeric value. Multiple comma-delimited values will create a filter that will match on any of the provided values.
- Numeric values will match exactly. Multiple comma-delimited values will create a filter that will match on any of the provided values.

`ApplyListSearchTerm` Applies filters to the query based on the specified search term. See *[Search]* for a detailed look at how searching works in Coalesce.

`ApplyListSorting` If any client-specified sort orders are present, invokes `ApplyListClientSpecifiedSorting`. Otherwise, invokes `ApplyListDefaultSorting`.

`ApplyListClientSpecifiedSorting` Applies sorting to the query based on sort orders specified by the client. If the client specified `"none"` as the sort field, no sorting will take place.

`ApplyListDefaultSorting` Applies default sorting behavior to the query, including behavior defined with use of `[DefaultOrderBy]` in C# POCOs, as well as fallback sorting to `"Name"` or primary key properties.

`ApplyListPaging` Applies paging to the query based on incoming parameters. Provides the actual page and pageSize that were used as out parameters.

`GetListTotalCountAsync` Simple wrapper around invoking `.Count()` on a query.

`TransformResults` Allows for transformation of a result set after the query has been evaluated. This will be called for both lists of items and for single items. This can be used for things like populating non-mapped properties on a model. This method is only called immediately before mapping to a DTO - if the data source is serving data without mapping (e.g. when invoked by *Behaviors*) to a DTO, this will not be called..

> **Warning:** It is STRONGLY RECOMMENDED that this method does not modify any database-mapped properties, as any such changes could be inadvertently persisted to the database.

`TrimListFields` Performs trimming of the fields of the result set based on the parameters given to the data source. Can be overridden to forcibly disable this, override the behavior to always trim specific fields, or any other functionality desired.

Replacing the Standard Data Source

You can, of course, create a custom base data source that all your custom implementations inherit from. But, what if you want to override the standard data source across your entire application, so that `StandardDataSource<,>` will never be instantiated? You can do that too!

Simply create a class that implements `IEntityFrameworkDataSource<,>` (the `StandardDataSource<,>` already does - feel free to inherit from it), then register it at application startup like so:

```
public class MyDataSource<T, TContext> : StandardDataSource<T, TContext>
    where T : class, new()
    where TContext : DbContext
{
    public MyDataSource(CrudContext<TContext> context) : base(context)
    {
    }

    ...
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCoalesce(b =>
    {
        b.AddContext<AppDbContext>();
        b.UseDefaultDataSource(typeof(MyDataSource<,>));
    });
```

Your custom data source must have the same generic type parameters - `<T, TContext>`. Otherwise, the Microsoft.Extensions.DependencyInjection service provider won't know how to inject it.

3.2.9 Behaviors

> *In a CRUD system, creating, updating, and deleting are considered especially different from reading. In Coalesce, the dedicated classes that perform these operations are derivatives of a special interface known as the* `IBehaviors<T>`*. These are their stories.*

Coalesce separates out the parts of your API that read your data from the parts that mutate it. The read portion is performed by *Data Sources*, and the mutations are performed by behaviors. Like data sources, there exists a standard set of behaviors that Coalesce provides out-of-the-box that cover the most common use cases for creating, updating, and deleting objects in your data model.

Also like data sources, these functions can be easily overridden on a per-model basis, allowing complete control over the ways in which your data is mutated by the APIs that Coalesce generates. However, unlike data sources which can have as many implementations per model as you like, you can only have one set of behaviors.

Contents

Defining Behaviors

By default, each of your models that Coalesce exposes will utilize the standard behaviors (`IntelliTect.Coalesce.StandardBehaviors<T, TContext>`) for the out-of-the-box API endpoints that Coalesce provides. These behaviors provide a set of create, update, and delete methods for an EF Core `DbContext`, as well as a plethora of virtual methods that make the `StandardBehaviors` a great base class for your custom implementations. Unlike data sources which require an annotation to override the Coalesce-provided standard class, the simple presence of an explicitly declared set of behaviors will suppress the standard behaviors.

Note: When you define a set of custom behaviors, take note that these are only used by the standard set of API endpoints that Coalesce always provides. They will not be used to handle any mutations in any *Methods* you write for your models.

To create your own behaviors, you simply need to define a class that implements `IntelliTect.Coalesce.IBehaviors<T>`. To expose your behaviors to Coalesce, either place it as a nested class of the type `T` that your behaviors are for, or annotate it with the `[Coalesce]` attribute. Of course, the easiest way to create behaviors that doesn't require you to re-engineer a great deal of logic would be to inherit from `IntelliTect.Coalesce.StandardBehaviors<T, TContext>`, and then override only the parts that you need.

```
public class Case
{
    public int CaseId { get; set; }
    public int OwnerId { get; set; }
    public bool IsDeleted { get; set; }
    ...
}

[Coalesce]
public class CaseBehaviors : StandardBehaviors<Case, AppDbContext>
{
    public Behaviors(CrudContext<AppDbContext> context) : base(context) { }

    public override ItemResult BeforeSave(SaveKind kind, Case oldItem, Case␣
↪item)
    {
        // Allow admins to bypass all validation.
        if (User.IsInRole("Admin")) return true;

        if (kind == SaveKind.Update && oldItem.OwnerId != item.OwnerId)
            return "The owner of a case may not be changed";

        // This is a new item, OR its an existing item and the owner isn't␣
↪being modified.
        if (item.CreatedById != User.GetUserId())
            return "You are not the owner of this item."

        return true;
    }

    public override ItemResult BeforeDelete(Case item)
        => User.IsInRole("Manager") ? true : "Unauthorized";

    public override Task ExecuteDeleteAsync(Case item)
    {
        // Soft delete the item.
```

(continues on next page)

(continued from previous page)

```
        item.IsDeleted = true;
        return Db.SaveChangesAsync();
    }
}
```

Dependency Injection

All behaviors are instantiated using dependency injection and your application's `IServiceProvider`. As a result, you can add whatever constructor parameters you desire to your behaviors as long as a value for them can be resolved from your application's services. The single parameter to the `StandardBehaviors` is resolved in this way - the `CrudContext<TContext>` contains the common set of objects most commonly used, including the `DbContext` and the `ClaimsPrincipal` representing the current user.

Standard Behaviors

The standard behaviors, `IntelliTect.Coalesce.StandardBehaviors<T, TContext>`, contains a significant number of properties and methods that can be utilized and/or overridden at your leisure.

Properties

`CrudContext<TContext> Context` The object passed to the constructor that contains the set of objects needed by the standard behaviors, and those that are most likely to be used in custom implementations.

`TContext Db` An instance of the db context that contains a `DbSet<T>` for the entity handled by the behaviors

`ClaimsPrincipal User` The user making the current request.

`IDataSource<T> OverrideFetchForUpdateDataSource` A data source that, if set, will override the data source that is used to retrieve the target of an update operation from the database. The incoming values will then be set on this retrieved object. Null by default; override by setting a value in the constructor.

`IDataSource<T> OverridePostSaveResultDataSource` A data source that, if set, will override the data source that is used to retrieve a newly-created or just-updated object from the database after a save. The retrieved object will be returned to the client. Null by default; override by setting a value in the constructor.

`IDataSource<T> OverrideFetchForDeleteDataSource` A data source that, if set, will override the data source that is used to retrieve the target of an delete operation from the database. The retrieved object will then be deleted. Null by default; override by setting a value in the constructor.

`IDataSource<T> OverridePostDeleteResultDataSource` A data source that, if set, will override the data source that is used to retrieve the target of an delete operation from the database after it has been deleted. If an object is able to be retrieved from this data source, it will be sent back to the client. This allows soft-deleted items to be returned to the client when the user is able to see them. Null by default; override by setting a value in the constructor.

Method Overview

The standard behaviors implementation contains many different methods which can be overridden in your derived class to control functionality.

These methods often call one another, so overriding one method may cause some other method to no longer be called. The hierarchy of method calls, ignoring any logic or conditions contained within, is as follows:

```
SaveAsync
    DetermineSaveKind
    GetDbSet
    ValidateDto
    MapIncomingDto
    BeforeSave
    AfterSave

DeleteAsync
    BeforeDelete
    ExecuteDeleteAsync
        GetDbSet
    AfterDelete
```

Method Details

All of the methods outlined above can be overridden. A description of each of the methods is as follows:

`SaveAsync` Save the given item. This is the main entry point for saving, and takes a DTO as a parameter. This method is responsible for performing mapping to your EF models and ultimately saving to your database. If it is required that you access properties from the incoming DTO in this method, a set of extension methods `GetValue` and `GetObject` are available on the DTO for accessing properties that are mapped 1:1 with your EF models.

`DetermineSaveKind` Given the incoming DTO on which Save has been called, examine its properties to determine if the operation is meant to be a create or an update operation. Return this distinction along with the key that was used to make the distinction.

This method is called outside of the standard data source by the base API controller to perform role-based security on saves at the controller level.

`GetDbSet` Fetch a `DbSet<T>` that items can be added to (creates) or remove from (deletes).

`ValidateDto` Provides a chance to validate the properties of the DTO object itself, as opposed to the properties of the model after the DTO has been mapped to it in `BeforeSave`. A number of extension methods on `IClassDto<T>` can be used to access the value of the properties of *Generated DTOs*. For behaviors on *Custom DTOs* where the DTO type is known, simply cast to the correct type.

`MapIncomingDto` Map the properties of the incoming DTO to the model that will be saved to the database. By default, this will call the `MapTo` method on the DTO, but if more precise control is needed, the `IClassDto<T>` extension methods or a cast to a known type can be used to get specific values. If all else fails, the DTO can be reflected upon.

`BeforeSave` Provides an easy way for derived classes to intercept a save attempt and either reject it by returning an unsuccessful result, or approve it by returning success. The incoming item can also be modified at will in this method to override changes that the client made as desired.

`AfterSave` Provides an easy way for derived classes to perform actions after a save operation has been completed. Failure results returned here will present an error to the client, but will not prevent

modifications to the database since changes have already been saved at this point. This method can optionally modify or replace the item that is sent back to the client after a save by setting `ref T item` to another object or to null. Setting `ref IncludeTree includeTree` will override the *Include Tree* used to shape the response object.

> **Warning:** Setting `ref T item` to null will prevent the new object from being returned - be aware that this can be harmful in create scenarios since it prevents the client from recieving the primary key of the newly created item. If autoSave is enabled on the client, this could cause a large number of duplicate objects to be created in the database, since each subsequent save by the client will be treated as a create when the incoming object lacks a primary key.

`DeleteAsync` Deletes the given item.

`BeforeDelete` Provides an easy way to intercept a delete request and potentially reject it.

`ExecuteDeleteAsync` Performs the delete action aginst the database. The implementation of this method removes the item from its corresponding `DbSet<T>`, and then calls `Db.SaveChangesAsync()`.

Overriding this allows for changing this row-deletion implementation to something else, like setting of a soft delete flag, or copying the data into another archival table before deleting.

`AfterDelete` Allows for performing any sort of cleanup actions after a delete has completed. If the item was still able to be retrieved from the database after the delete operation completed, this method allows lets you modify or replace the item that is sent back to the client by setting `ref T item` to another object or to null. Setting `ref IncludeTree includeTree` will override the *Include Tree* used to shape the response object.

Replacing the Standard Behaviors

You can, of course, create a custom base behaviors class that all your custom implementations inherit from. But, what if you want to override the standard behaviors across your entire application, so that `StandardBehaviors<,>` will never be instantiated? You can do that too!

Simply create a class that implements `IEntityFrameworkBehaviors<,>` (the `StandardBehaviors<,>` already does - feel free to inherit from it), then register it at application startup like so:

```
public class MyBehaviors<T, TContext> : StandardBehaviors<T, TContext>
    where T : class, new()
    where TContext : DbContext
{
    public MyBehaviors(CrudContext<TContext> context) : base(context)
    {
    }

    ...
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCoalesce(b =>
    {
        b.AddContext<AppDbContext>();
        b.UseDefaultBehaviors(typeof(MyBehaviors<,>));
    });
```

Your custom behaviors class must have the same generic type parameters - `<T, TContext>`. Otherwise, the Microsoft.Extensions.DependencyInjection service provider won't know how to inject it.

3.2.10 Code Generation

The primary function of Coalesce is as a code generation framework. Below, you find an overview of the different components of Coalesce's code generation features.

Contents

Running Code Generation

Coalesce's code generation is ran via a dotnet CLI tool, `dotnet coalesce`. In order to invoke this tool, you must have the appropriate references to the package that provides it in your .csproj file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

    ...

    <ItemGroup>
        <PackageReference Include="IntelliTect.Coalesce.Knockout" Version="..
↪." />
    </ItemGroup>

    <ItemGroup>
        <DotNetCliToolReference Include="IntelliTect.Coalesce.Tools" Version=
↪"..." />
    </ItemGroup>
</Project>
```

CLI Options

All configuration of the way that Coalesce interacts with your projects, including locating, analyzing, and producing generated code, is done in a json configuration file, `coalesce.json`. Read more about this file at *Code Generation Configuration*.

There are a couple of extra options which are only available as CLI parameters to `dotnet coalesce`. These options do not affect the behavior of the code generation - only the behavior of the CLI itself.

--debug When this flag is specified when running `dotnet coalesce`, Coalesce will wait for a debugger to be attached to its process before starting code generation.

-v|--verbosity <level> Set the verbosity of the output. Options are `trace`, `debug`, `information`, `warning`, `error`, `critical`, and `none`.

Generated Code

Below you will find a brief overview of each of the different pieces of code that Coalesce will generate for you.

TypeScript

Coalesce generates a number of different types of TypeScript classes to support your data through the generated API.

ViewModels One view model class is generated for each of your EF Database-mapped POCO classes. These models contain fields for your model *Properties*, and functions and other members for your model *Methods*. They also contain a number of standard fields & functions inherited from `BaseViewModel` which offer basic loading & saving functionality, as well as other handy utility members for use with Knockout.

See *TypeScript ViewModels* for more details.

List ViewModels One ListViewModel is generated for each of your EF Database-mapped POCO classes. These classes contain functionality for loading sets of objects from the server. They provide searching, paging, sorting, and filtering functionality.

See *TypeScript ListViewModels* for more details.

External Type ViewModels Any types which are accessible through your Database-mapped POCO classes, either through one of its getter-only *Properties* or return value from one of its *Methods*, will have a corresponding TypeScript ViewModel generated for it. These ViewModels only provide a `KnockoutObservable` field for each property on the C# class.

see *TypeScript External ViewModels* for more details.

C# DTOs

For each of your EF Database-mapped POCO classes, a C# DTO class is created. These classes are used to hold the data that will be serialized and sent to the client, as well as data that has been received from the client before it has been mapped back to your EF POCO class.

See *Generated DTOs* for more information.

API Controllers

For each of your EF Database-mapped POCO classes, an API controller is created in the `/Api/Generated` directory of your web project. These controllers provide a number of endpoints for interacting with your data.

View Controllers

For each of your EF Database-mapped POCO classes, a controller is created in the `/Controllers/Generated` directory of your web project. These controllers provide routes for the generated admin views.

As you add your own pages to your application, you should add additional partial classes in the `/Controllers` that extend these generated partial classes to expose those pages.

Admin Views

For each of your EF Database-mapped POCO classes, a number of views are generated to provide administrative-level access to your data.

Table Provides a basic table view with sorting, searching, and paging of your data.

TableEdit Provides the table view, but with inline editing in the table.

Cards Provides a card-based view of your data with searching and paging.

CreateEdit Provides an editor view which can be used to create new entities or edit existing ones.

EditorHtml Provides a minimal amount of HTML to display an editor for the object type. This is used by the `showEditor` method on the generated TypeScript ViewModels.

3.2.11 Generated DTOs

Data Transfer Objects, or DTOs, allow for transformations of data from the data store into a format more suited for transfer and use on the client side. This often means trimming properties and flattening structures to provide a leaner over-the-wire experience. Coalesce aims to support this as seamlessly as possible.

Coalesce supports two types of DTOs:

- DTOs that are automatically generated for each POCO database object. These are controlled via *Attributes* on the POCO. These are outlined below.
- DTOs that you create with IClassDto and create unique ViewModels. These are outlined at *Custom DTOs*.

Automatically Generated DTOs

Every class that is exposed through Coalesce's generated API will have a corresponding DTO generated for it. These DTOs are used to shuttle data back and forth to the client. They are generated classes that have nullable versions of all the properties on the POCO class.

[DtoIncludes] & [DtoExcludes] and the *Includes String* infrastructure can be used to indicate which properties should be transferred to the client in which cases, and *Include Tree* is used to dictate how these DTOs are constructed from POCOs retrieved from the database.

3.2.12 TypeScript ViewModels

For each database-mapped type in your model, Coalesce will generate a TypeScript class that provides a multitude of functionality for interacting with the data on the client.

These ViewModels are dependent on Knockout, and are designed to be used directly from Knockout bindings in your HTML. All data properties on the generated model are Knockout observables.

Base Members

`includes: string` String that will be passed to the server when loading and saving that allows for data trimming via C# Attributes. See *Includes String* for more information.

`isChecked: KnockoutObservable<boolean>` Flag to use to determine if this item is checked. Only provided for convenience.

`isSelected: KnockoutObservable<boolean>` Flag to use to determine if this item is selected. Only provided for convenience.

isEditing: KnockoutObservable<boolean> Flag to use to determine if this item is being edited. Only provided for convenience.

toggleIsEditing () => void Toggles the `isEditing` flag.

isExpanded: KnockoutObservable<boolean> Flag to use to determine if this item is expanded. Only provided for convenience.

toggleIsExpanded: () => void Toggles the `isExpanded` flag.

isVisible: KnockoutObservable<boolean> Flag to use to determine if this item is shown. Only provided for convenience.

toggleIsSelected () => void Toggles the `isSelected` flag.

selectSingle: (): boolean Sets isSelected(true) on this object and clears on the rest of the items in the parent collection.

isDirty: KnockoutObservable<boolean> Dirty Flag. Set when a value on the model changes. Reset when the model is saved or reloaded.

isLoaded: KnockoutObservable<boolean> True once the data has been loaded.

isLoading: KnockoutObservable<boolean> True if the object is loading.

isSaving: KnockoutObservable<boolean> True if the object is currently saving.

isThisOrChildSaving: KnockoutComputed<boolean> Returns true if the current object, or any of its children, are saving.

load: id: any, callback?: (self: T) => void): JQueryPromise<any> | undefined Loads the object from the server based on the id specified. If no id is specified, the current id, is used if one is set.

loadChildren: callback?: () => void) => void Loads any child objects that have an ID set, but not the full object. This is useful when creating an object that has a parent object and the ID is set on the new child.

loadFromDto: data: any, force?: boolean, allowCollectionDeletes?: boolean) => void Loads this object from a data transfer object received from the server.

- `force` - Will override the check against isLoading that is done to prevent recursion.
- `allowCollectionDeletes` - Set true when entire collections are loaded. True is the default. In some cases only a partial collection is returned, set to false to only add/update collections.

deleteItem: callback?: (self: T) => void): JQueryPromise<any> | undefined Deletes the object without any prompt for confirmation.

deleteItemWithConfirmation: callback?: () => void, message?: string): JQueryPromis Deletes the object if a prompt for confirmation is answered affirmatively.

errorMessage: KnockoutObservable<string> Contains the error message from the last failed call to the server.

onSave: callback: (self: T) => void): boolean Register a callback to be called when a save is done. Returns `true` if the callback was registered, or `false` if the callback was already registered.

saveToDto: () => any Saves this object into a data transfer object to send to the server.

save: callback?: (self: T) => void): JQueryPromise<any> | boolean | undefined Saves the object to the server and then calls a callback. Returns false if there are validation errors.

parent: any Parent of this object, if this object was loaded as part of a hierarchy.

parentCollection: KnockoutObservableArray<T> Parent of this object, if this object was loaded as part of list of objects.

editUrl: KnockoutComputed<string> URL to a stock editor for this object.

showEditor: callback?: any): JQueryPromise<any> Displays an editor for the object in a modal dialog.

validate: (): boolean Triggers any validation messages to be shown, and returns a bool that indicates if there are any validation errors.

validationIssues: any ValidationIssues returned from the server when trying to persist data

warnings: KnockoutValidationErrors List of warnings found during validation. Saving is still allowed with warnings present.

errors: KnockoutValidationErrors List of errors found during validation. Any errors present will prevent saving.

Model-Specific Members

Configuration A static configuration object for configuring all instances of the ViewModel's type is created, as well as an instance configuration object for configuring specific instances of the ViewModel. See (see *ViewModel Configuration*) for more information.

```
public static coalesceConfig: Coalesce.ViewModelConfiguration<Person>
    = new Coalesce.ViewModelConfiguration<Person>(Coalesce.
↪GlobalConfiguration.viewModel);

public coalesceConfig: Coalesce.ViewModelConfiguration<Person>
    = new Coalesce.ViewModelConfiguration<Person>(Person.coalesceConfig);
```

DataSources For each of the *Data Sources* for a model, a class will be added to a namespace named `ListViewModels.<ClassName>DataSources`. This namespace can always be accessed on both `ViewModel` and `ListViewModel` instances via the `dataSources` property, and class instances can be assigned to the `dataSource` property.

```
public dataSources = ListViewModels.PersonDataSources;
public dataSource: DataSource<Person> = new this.dataSources.Default();
```

Data Properties For each exposed property on the underlying EF POCO, a `KnockoutObservable<T>` property will exist on the TypeScript model. For navigation properties, these will be typed with the corresponding TypeScript ViewModel for the other end of the relationship. For collections (including collection navigation properties), these properties will be `KnockoutObservableArray<T>` objects.

```
public personId: KnockoutObservable<number> = ko.observable(null);
public fullName: KnockoutObservable<string> = ko.observable(null);
public gender: KnockoutObservable<number> = ko.observable(null);
public companyId: KnockoutObservable<number> = ko.observable(null);
public company: KnockoutObservable<ViewModels.Company> = ko.
↪observable(null);
public addresses: KnockoutObservableArray<ViewModels.Address> = ko.
↪observableArray([]);
public birthDate: KnockoutObservable<moment.Moment> = ko.
↪observable(moment());
```

Computed Text Properties For each reference navigation property and each Enum property on your POCO, a `KnockoutComputed<string>` property will be created that will provide the text to display for that property. For navigation properties, this will be the property on the class annotated with *[ListText]*.

```
public companyText: () => string;
public genderText: () => string;
```

Collection Navigation Property Helpers For each collection navigation property on the POCO, the following members will be created:

- A method that will add a new object to that collection property. If `autoSave` is specified, the auto-save behavior of the new object will be set to that value. Otherwise, the inherited default will be used (see *ViewModel Configuration*)

```
public addToAddresses: (autoSave?: boolean) => ViewModels.
↪Address;
```

- A `KnockoutComputed<string>` that evaluates to a relative url for the generated table view that contains only the items that belong to the collection navigation property.

```
public addressesListUrl: KnockoutComputed<string>;
```

Reference Navigation Property Helpers For each reference navigation property on the POCO, the following members will be created:

- A method that will call `showEditor` on that current value of the navigation property, or on a new instance if the current value is null.

```
public showCompanyEditor: (callback?: any) => void;
```

Instance Method Members For each *Instance Method* on your POCO, the members outlined in *Methods - Generated TypeScript* will be created.

Enum Members For each `enum` property on your POCO, the following will be created:

- A static array of objects with properties `id` and `value` that represent all the values of the enum.

```
public genderValues: Coalesce.EnumValue[] = [
    { id: 1, value: 'Male' },
    { id: 2, value: 'Female' },
    { id: 3, value: 'Other' },
];
```

- A TypeScript enum that mirrors the C# enum directly. This enum is in a sub-namespace of `ViewModels` named the same as the class name.

```
export namespace Person {
    export enum GenderEnum {
        Male = 1,
        Female = 2,
        Other = 3,
    };
}
```

3.2.13 TypeScript ListViewModels

In addition to *TypeScript ViewModels* for interacting with instances of your data classes in TypeScript, Coalesce will also generated a List ViewModel for loading searched, sorted, paginated data from the server.

These ListViewModels, like the ViewModels, are dependent on Knockout and are designed to be used directly from Knockout bindings in your HTML.

Base Members

The following members are defined on `BaseListViewModel<>` and are available to the ListViewModels for all of your model types:

`modelKeyName: string` Name of the primary key of the model that this list represents.

`includes: string` String that is used to control loading and serialization on the server. See *Includes String* for more information.

`items: KnockoutObservableArray<TItem>` The collection of items that have been loaded from the server.

`addNewItem: (): TItem` Adds a new item to the items collection.

`deleteItem: (item: TItem): JQueryPromise<any>` Deletes an item and removes it from the items collection.

`queryString: string` Query string to append to the API call when loading the list of items. If `query` is non-null, this value will not be used. See *below* for more information about `query`.

`search: KnockoutObservable<string>` Search criteria for the list. This can be easily bound to with a text box for easy search behavior. See *[Search]* for a detailed look at how searching works in Coalesce.

`isLoading: KnockoutObservable<boolean>` True if the list is loading.

`isLoaded: KnockoutObservable<boolean>` True once the list has been loaded.

`load: (callback?: any): JQueryPromise<any>` Load the list using current parameters for paging, searching, etc Result is placed into the items property.

`message: KnockoutObservable<string>` If a load failed, this is a message about why it failed.

`getCount: (callback?: any): JQueryPromise<any>` Gets the count of items without getting all the items. Result is placed into the count property.

`count: KnockoutObservable<number>` The result of getCount(), or the total on this page.

`totalCount: KnockoutObservable<number>` Total count of items, even ones that are not on the page.

`nextPage: (): void` Change to the next page.

`nextPageEnabled: KnockoutComputed<boolean>` True if there is another page after the current page.

`previousPage: (): void` Change to the previous page.

`previousPageEnabled: KnockoutComputed<boolean>` True if there is another page before the current page.

`page: KnockoutObservable<number>` Page number. This can be set to get a new page.

`pageCount: KnockoutObservable<number>` Total page count

pageSize: KnockoutObservable<number> Number of items on a page.

orderBy: KnockoutObservable<string> Name of a field by which this list will be loaded in ascending order.

If set to "none", default sorting behavior, including behavior defined with use of [DefaultOrderBy] in C# POCOs, is suppressed.

orderByDescending: KnockoutObservable<string> Name of a field by which this list will be loaded in descending order.

orderByToggle: (field: string): void Toggles sorting between ascending, descending, and no order on the specified field.

csvUploadUi: (callback?: () => void): void Prompts to the user for a file to upload as a CSV.

downloadAllCsvUrl: KnockoutComputed<string> Returns URL to download a CSV for the current list with all items.

Model-Specific Members

Configuration A static configuration object for configuring all instances of the ListViewModel's type is created, as well as an instance configuration object for configuring specific instances of the ListViewModel. See (see *ViewModel Configuration*) for more information.

```
public static coalesceConfig = new Coalesce.ListViewModelConfiguration
↪<PersonList, ViewModels.Person>(Coalesce.GlobalConfiguration.
↪listViewModel);

public coalesceConfig = new Coalesce.ListViewModelConfiguration
↪<PersonList, ViewModels.Person>(PersonList.coalesceConfig);
```

Filter Object For each exposed value type instance property on the underlying EF POCO, a property named `filter` will have a property declaration generated for that property. If the `filter` object is set, requests made to the server to retrieve data will be passed all the values in this object via the URL's query string. These parameters will filter the resulting data to only rows where the parameter values match the row's values. For example, if `filter.companyId` is set to a value, only people from that company will be returned.

```
public filter: {
    personId?: string
    firstName?: string
    lastName?: string
    gender?: string
    companyId?: string
} = null;
```

```
var list = new ListViewModels.PersonList();
list.filter = {
    lastName: "Erickson",
};
list.load();
```

These parameters all allow for freeform string values, allowing the server to implement any kind of filtering logic desired. The *Standard Data Source* will perform simple equality checks, but also the following:

- Enum properties may have a filter that contains either enum names or integer values. There may be a single such value, or multiple, comma-delimited values where the actual value may match any of the filter values.
- The same goes for numeric properties - you can specify a comma-delimited list of numbers to match on any of those values.
- Date properties can specify an exact time, or a date with no time component. In the latter case, any times that fall within that day will be matched.

Static Method Members For each exposed *Static Method* on your POCO, the members outlined in *Methods - Generated TypeScript* will be created.

DataSources For each of the *Data Sources* on the class, a corresponding class will be added to a namespace named `ListViewModels.<ClassName>DataSources`. This namespace can always be accessed on both `ViewModel` and `ListViewModel` instances via the `dataSources` property, and class instances can be assigned to the `dataSource` property.

```
module ListViewModels {
    export namespace PersonDataSources {

        export class WithoutCases extends Coalesce.DataSource<ViewModels.
↪Person> { }
        export const Default = WithoutCases;

        export class NamesStartingWithAWithCases extends Coalesce.
↪DataSource<ViewModels.Person> { }

        /** People whose last name starts with B or c */
        export class BorCPeople extends Coalesce.DataSource<ViewModels.
↪Person> { }
    }

    export class PersonList extends Coalesce.BaseListViewModel
↪<PersonList, ViewModels.Person> {
        public dataSources = PersonDataSources;
        public dataSource: PersonDataSources = new this.dataSources.
↪Default();
    }
}
```

3.2.14 TypeScript External ViewModels

For all *External Types* in your model, Coalesce will generate a TypeScript class that provides a barebones representation of that type's properties.

These ViewModels are dependent on Knockout, and are designed to be used directly from Knockout bindings in your HTML. All data properties on the generated model are Knockout observables.

Base Members

The TypeScript ViewModels for external types do not have a common base class, and do not have any of the behaviors or convenience properties that the regular *TypeScript ViewModels* for database-mapped classes have.

Model-Specific Members

Data Properties For each exposed property on the underlying EF POCO, a `KnockoutObservable<T>` property will exist on the TypeScript model. For POCO properties, these will be typed with the corresponding TypeScript ViewModel for the other end of the relationship. For collections, these properties will be `KnockoutObservableArray<T>` objects.

```
public personId: KnockoutObservable<number> = ko.observable(null);
public fullName: KnockoutObservable<string> = ko.observable(null);
public gender: KnockoutObservable<number> = ko.observable(null);
public companyId: KnockoutObservable<number> = ko.observable(null);
public company: KnockoutObservable<ViewModels.Company> = ko.
↪observable(null);
public addresses: KnockoutObservableArray<ViewModels.Address> = ko.
↪observableArray([]);
public birthDate: KnockoutObservable<moment.Moment> = ko.
↪observable(moment());
```

Computed Text Properties For each Enum property on your POCO, a `KnockoutComputed<string>` property will be created that will provide the text to display for that property.

```
public genderText: () => string;
```

3.2.15 ViewModel Configuration

A crucial part of the generated TypeScript ViewModels that Coalesce creates for you is the hierarchical configuration system that allows coarse-grained or fine-grained control over their behaviors.

Hierarchy

The configuration system has four levels where configuration can be performed, structured as follows:

Root Configuration

`Coalesce.GlobalConfiguration`
`Coalesce.GlobalConfiguration.app`

The root configuration contains all configuration properties which apply to class category (*TypeScript ViewModels*, *TypeScript ListViewModels*, and *Services*). The `app` property contains global app configuration that exists independent of any models. Then, for each class kind, the following are available:

Root ViewModel/ListViewModel Configuration

`Coalesce.GlobalConfiguration.viewModel`
`Coalesce.GlobalConfiguration.listViewModel`
`Coalesce.GlobalConfiguration.serviceClient`

Additional root configuration objects exist, one for each class kind. These configuration objects govern behavior that applies to only objects of these types. Root configuration *can* be overridden using these objects, although the practicality of doing so is dubious.

Class Configuration

```
ViewModels.<ClassName>.coalesceConfig
ListViewModels.<ClassName>List.coalesceConfig
Services.<ServiceName>Client.coalesceConfig
```

Each class kind has a static property named `coalesceConfig` that controls behavior for all instances of that class.

Instance Configuration

```
instance.coalesceConfig
```

Each instance of these classes also has a `coalesceConfig` property that controls behaviors for that instance only.

Evaluation

All configuration properties are Knockout `ComputedObservable<T>` objects. These observables behave like any other observable - call them with no parameter to obtain the value, call with a parameter to set their value.

Whenever a configuration property is read from, it first checks its own configuration object for the value of that property. If the explicit value for that configuration object is null, the parent's configuration will be checked for a value. This continues until either a value is found or the root configuration object is reached.

When a configuration property is given a value, that value is established on that configuration object only. Any dependent configuration objects will not be modified, and if those dependent configuration objects already have a value for that property, their existing value will be used unless that value is later set to null.

To obtain the raw value for a specific configuration property, call the `raw()` method on the observable: `model.coalesceConfig.autoSaveEnabled.raw()`.

Available Properties & Defaults

The following configuration properties are available. Their default values are also listed.

Root Configuration

These properties on `Coalesce.GlobalConfiguration` are available to both ViewModelConfiguration, ListViewModelConfiguration, and ServiceClientConfiguration.

baseApiUrl - `"/api"` The relative url where the API may be found.

baseViewUrl - `""` The relative url where the admin views may be found.

showFailureAlerts - `true` Whether or not the callback specified for `onFailure` will be called or not.

onFailure - `(obj, message) => alert(message)` A callback to be called when a failure response is received from the server.

onStartBusy - `obj => Coalesce.Utilities.showBusy()` A callback to be called when an AJAX request begins.

onFinishBusy - `obj => Coalesce.Utilities.hideBusy()` A callback to be called when an AJAX request completes.

App Configuration

These properties on `Coalesce.GlobalConfiguration.app` are not hierarchical - they govern the entire Coalesce application:

select2Theme - `null` The theme parameter to select2's constructor when called by Coalesce's select2 *Knockout Bindings*.

ViewModelConfiguration

saveTimeoutMs - `500` Time to wait after a change is seen before auto-saving (if `autoSaveEnabled` is true). Acts as a debouncing timer for multiple simultaneous changes.

autoSaveEnabled - `true` Determines whether changes to a model will be automatically saved after `saveTimeoutMs` milliseconds have elapsed.

autoSaveCollectionsEnabled - `true` Determines whether or not changes to many-to-many collection properties will automatically trigger a save call to the server or not.

showBusyWhenSaving - `false` Whether to invoke `onStartBusy` and `onFinishBusy` during saves.

loadResponseFromSaves - `true` Whether or not to reload the ViewModel with the state of the object received from the server after a call to `.save()`.

validateOnLoadFromDto - `true` Whether or not to validate the model after loading it from a DTO from the server. Disabling this can improve performance in some cases.

setupValidationAutomatically - `true` Whether or not validation on a ViewModel should be setup in its constructor, or if validation must be set up manually by calling `viewModel.setupValidation()`. Turning this off can improve performance in read-only scenarios.

onLoadFromDto - `null` An optional callback to be called when an object is loaded from a response from the server. Callback will be called after all properties on the ViewModel have been set from the server response.

initialDataSource = `null` The dataSource (either an instance or a type) that will be used as the initial dataSource when a new object of this type is created. Not valid for global configuration; recommended to be used on class-level configuration. E.g. `ViewModels.MyModel.coalesceConfig.initialDataSource(MyModel.dataSources.MyDataSource);`

ListViewModelConfiguration

No special configuration is currently available for ListViewModels.

ServiceClientConfiguration

No special configuration is currently available for ServiceClients.

3.2.16 Knockout Bindings

Coalesce provides a number of knockout bindings that make common model binding activities much easier.

Editors Note: On this page, some bindings are split into their requisite HTML component with their `data-bind` component listed immediately after. Keep this in mind when reading.

Contents

Input Bindings

select2Ajax

```
<select data-bind="..."></select>
select2Ajax: personId, url: '/api/Person/list', idField: 'personId',
textField: 'Name', object: person, allowClear: true
```

Creates a select2 dropdown using the specified url and fields that can be used to select an object from the endpoint specified. Additional complimentary bindings include:

idField (required) The name of the field on each item in the results of the AJAX call which contains the ID of the option. The value of this field will be set on the observable specified for the main `select2Ajax` binding.

textField (required) The name of the field on each item in the results of the AJAX call which contains the text to be displayed for each option.

url (required) The Coalesce List API url to call to populate the contents of the dropdown.

pageSize The number of items to request in each call to the server.

format A string containing the substring `{0}`, which will be replaced with the text value of an option in the dropdown list when the option is displayed.

selectionFormat A string containing the substring `{0}`, which will be replaced with the text value of the selected option of the dropdown list.

object An observable that holds the full object corresponding to the foreign key property being bound to. If the selected value changes, this will be set to null to avoid representation of incorrect data (unless `setObject` is used - see below).

setObject If true, the observable specified by the `object` binding will be set to the selected data when an option is chosen in the dropdown. Binding `itemViewModel` is required if this binding is set.

Additionally, requests to the API to populate the dropdown will request the entire object, as opposed to only the two fields specified for `idField` and `textField` like is normally done when this binding is missing or set to false. To override this behavior and continue requesting only specific fields even when `setObject` is true, add `fields=field1,field2,...` to the query string of the `url` binding.

itemViewModel A reference to the class that represents the type of the object held in the `object` observable. This is used when constructing new objects from the results of the API call. Not used if `setObject` is false or unspecified. For example, `setObject: true, itemViewModel: ViewModels.Person`

selectOnClose Directly maps to select2 option `selectOnClose`

allowClear Whether or not to allow the current select to be set to null. Directly maps to select2 option `allowClear`

placeholder Placeholder when nothing is selected. Directly maps to select2 option `placeholder`

openOnFocus If true, the dropdown will open when tabbed to. Browser support may be incomplete in some versions of IE.

cache Controls caching behavior of the AJAX request. Defaults to false. Seems to only affect IE - Chrome will never cache JSON ajax requests.

select2AjaxMultiple

```
<select multiple="multiple" data-bind="..."></select>
select2AjaxMultiple: people, url: '/api/Person/list', idField: 'personId',
textField: 'Name', itemViewModel: ViewModels.PersonCase
```

Creates a select2 multi-select input for choosing objects that participate as the foreign object in a many-to-many relationship with the current object. The primary `select2AjaxMultiple` binding takes the collection of items that make up the foreign side of the relationship. This is NOT the collection of the join objects (a.k.a. middle table objects) in the relationship.

Additional complimentary bindings include:

idField (required) The name of the field on each item in the results of the AJAX call which contains the ID of the option. The value of this field will be set as the key of the foreign object in the many-to-many relationship.

textField (required) The name of the field on each item in the results of the AJAX call which contains the text to be displayed for each option.

url (required) The Coalesce List API url to call to populate the contents of the dropdown. In order to only receive specific fields from the server, add `fields=field1,field2,...` to the query string of the url, ensuring that at least the `idField` and `textField` are included in that collection.

itemViewModel (required) A reference to the class that represents the types in the supplied collection. For example, a many-to-many between `Person` and `Case` objects where `Case` is the object being bound to and `Person` is the type represented by a child collection, the correct value is :ts:`ViewModels.Person`. This is used when constructing new objects representing the relationship when a new item is selected.

pageSize The number of items to request in each call to the server.

format A string containing the substring `{0}`, which will be replaced with the text value of an option in the dropdown list when the option is displayed.

selectionFormat A string containing the substring `{0}`, which will be replaced with the text value of the selected option of the dropdown list.

selectOnClose Directly maps to select2 option `selectOnClose`

allowClear Whether or not to allow the current select to be set to null. Directly maps to select2 option `allowClear`

placeholder Placeholder when nothing is selected. Directly maps to select2 option `placeholder`

openOnFocus If true, the dropdown will open when tabbed to. Browser support may be incomplete in some versions of IE.

cache Controls caching behavior of the AJAX request. Defaults to false. Seems to only affect IE - Chrome will never cache JSON ajax requests.

select2AjaxText

```
<select data-bind="..."></select>
select2AjaxText: schoolName, url: '/api/Person/SchoolNames'
```

Creates a select2 dropdown against the specified url where the url returns a collection of string values that are potential selection candidates. The dropdown also allows the user to input any value they choose - the API simply serves suggested values.

url The url to call to populate the contents of the dropdown. This should be an endpoint that returns one of the following:

- A raw `string[]`
- An object that conforms to `{ list: string[] }`
- An object that conforms to `{ object: string[] }`
- An object that conforms to `{ list: { [prop: string]: string } }` where the value given to `resultField` is a valid property of the returned objects.

- An object that conforms to `{ object: { [prop: string]: string } }` where the value given to `resultField` is a valid property of the returned objects.

The url will also be passed a `search` parameter and a `page` parameter appended to the query string. The chosen endpoint is responsible for implementing this functionality. Page size is expected to be some fixed value. Implementer should anticipate that the requested page may be out of range.

The cases listed above that accept arrays of objects (as opposed to arrays of strings) require that the `resultField` binding is also used. These are designed for obtaining string values from objects obtained from the standard `list` endpoint.

resultField If provided, specifies a field on the objects returned from the API to pull the string values from. See examples in `url` above.

selectOnClose Directly maps to select2 option `selectOnClose`

openOnFocus If true, the dropdown will open when tabbed to. Browser support may be incomplete in some versions of IE.

allowClear Whether or not to allow the current select to be set to null. Directly maps to select2 option `allowClear`

placeholder Placeholder when nothing is selected. Directly maps to select2 option `placeholder`

cache Controls caching behavior of the AJAX request. Defaults to false. Seems to only affect IE - Chrome will never cache JSON ajax requests.

select2

```
<select data-bind="..."></select>
select2: personId
```

Sets up a basic select2 dropdown on an HTML select element. Dropdown contents should be populated through other means - either using stock Knockout bindings or server-side static contents (via cshtml).

selectOnClose Directly maps to select2 option `selectOnClose`

openOnFocus If true, the dropdown will open when tabbed to. Browser support may be incomplete in some versions of IE.

allowClear Whether or not to allow the current select to be set to null. Directly maps to select2 option `allowClear`

placeholder Placeholder when nothing is selected. Directly maps to select2 option `placeholder`

datePicker

```
<div class="input-group date">
    <input data-bind="datePicker: birthDate" type="text" class="form-control
↪" />
    <span class="input-group-addon">
        <span class="fa fa-calendar"></span>
    </span>
</div>
```

Creates a date/time picker for changing a `moment.Moment` property. The control used is bootstrap-datetimepicker

preserveDate If true, the date portion of the `moment.Moment` object will be preserved by the date picker. Only the time portion will be changed by user input.

preserveTime If true, the time portion of the `moment.Moment` object will be preserved by the date picker. Only the date portion will be changed by user input.

format Specify the moment-compatible format string to be used as the display format for the text value shown on the date picker. Defaults to `M/D/YY h:mm a`. Direct pass-through to bootstrap-datetimepicker.

sideBySide if true, places the time picker next to the date picker, visible at the same time. Direct pass-through to corresponding bootstrap-datetimepicker option.

stepping Direct pass-through to corresponding bootstrap-datetimepicker option.

timeZone Direct pass-through to corresponding bootstrap-datetimepicker option.

keyBinds Override key bindings of the date picker. Direct pass-through to corresponding bootstrap-datetimepicker option. Defaults to `{ left: null, right: null, delete: null }`, which disables the default binding for these keys.

updateImmediate If true, the datePicker will update the underlying observable on each input change. Otherwise, the observable will only be changed when the datePicker loses focus (on `blur`).

saveImmediately

```
<div data-bind="with: product">
    <input type="text" data-bind="textValue: description, saveImmediately:␣
↪true" />
</div>
```

When used in a context where `$data` is a `Coalesce.BaseViewModel`, that object's `saveTimeoutMs` configuration property (see *ViewModel Configuration*) will be set to `0` when the element it is placed on gains focus. This value will be reverted to its previous value when the element loses focus. This will cause any changes to the object, including any observable bound as input on the element, to trigger a save immediately rather than after a delay (defaults to 500ms).

delaySave

```
<div data-bind="with: product">
    <input type="text" data-bind="textValue: description, delaySave: true" />
</div>
```

When used in a context where `$data` is a `Coalesce.BaseViewModel`, that object's `autoSaveEnabled` configuration property (see *ViewModel Configuration*) will be set to `false` when the element it is placed on gains focus. This will cause any changes to the object, including any observable bound as input on the element, to not trigger auto saves while the element has focus. When the element loses focus, the `autoSaveEnabled` flag will be reverted to its previous value and an attempt will be made to save the object.

Display Bindings

tooltip

```
tooltip:  tooltipText
```

```
tooltip: {title: note, placement: 'bottom', animation: false}
```

Wrapper around the Bootstrap tooltip component. Binding can either be simply a string (or observable string), or it can be an object that will be passed directly to the Bootstrap tooltip component.

fadeVisible

```
fadeVisible: isVisible
```

Similar to the Knockout `visible`, but uses jQuery `fadeIn/fadeOut` calls to perform the transition.

slideVisible

```
slideVisible: isVisible
```

Similar to the Knockout `visible`, but uses jQuery `slideIn/slideOut` calls to perform the transition.

moment

```
<span data-bind="moment: momentObservable"></span>
moment: momentObservable
moment: momentObservable, format: 'MM/DD/YYYY hh:mm a'
```

Controls the text of the element by calling the `format` method on a moment object.

momentFromNow

```
<span data-bind="momentFromNow: momentObservable"></span>
momentFromNow: momentObservable
momentFromNow: momentObservable, shorten: true
```

Controls the text of the element by calling the `fromNow` method on a moment object. If shorten is true, certain phrases will be slightly shortened.

Utility Bindings

let

```
let: {variableName: value}
```

The let binding is a somewhat common construct used in Knockout applications, but isn't part of Knockout itself. It effectively allows the creation of variables in the binding context, allowing complex statements which may be used multiple times to be aliased for both clarity of code and better performance.

```
<div class="item">
    <!-- ko let: { showControls: $data.isEditing() || $parent.
↪editingChildren() } -->
    <button data-bind="click: $root.editItem, visible: showControls">Edit</
↪button>
    <span data-bind="text: name"></span>
    <button data-bind="click: $root.deleteItem, visible: showControls">Delete
↪</button>
    <!-- /ko -->
</div>
```

Knockout Binding Defaults

These are static properties on `IntelliTect.Coalesce.Knockout.Helpers.Knockout` you can assign to somewhere in the app lifecycle startup to change the default markup generated server-side when using `@Knockout.*` methods to render Knockout bindings in your `.cshtml` files. Currently, there are defaults for the Bootstrap grid system width of `<label>` and `<input>` tags, as well as default formats for the date pickers.

The date/time picker properties can be coupled with `DateTimeOffset` model properties to display time values localized for the current user's locale. If you want to make the localization static, simply include a script block in your `_Layout.cshtml` or in a specific view that sets the default for Moment.js:

```
<script>
    moment.tz.setDefault("America/Chicago");
</script>
```

Note: This needs to happen *after* Moment is loaded, but *before* the bootstrap-datetimepicker script is loaded.

DefaultLabelCols

```
public static int DefaultLabelCols { get; set; } = 3;
```

The default number of Bootstrap grid columns a field label should span across.

DefaultInputCols

```
public static int DefaultInputCols { get; set; } = 9;
```

The default number of Bootstrap grid columns a form input should span across.

DefaultDateFormat

```
public static string DefaultDateFormat { get; set; } = "M/D/YYYY";
```

Sets the default date-only format to be used by all date/time pickers. This only applies to models with a date-only *[DateType]* attribute.

DefaultTimeFormat

```
public static string DefaultTimeFormat { get; set; } = "h:mm a";
```

Sets the default time-only format to be used by all date/time pickers. This only applies to models with a time-only *[DateType]* attribute.

DefaultDateTimeFormat

```
public static string DefaultDateTimeFormat { get; set; } = "M/D/YYYY h:mm a"
```

Sets the default date/time format to be used by all date/time pickers. This only applies to `DateTimeOffset` model properties that do not have a limiting *[DateType]* attribute.

Note: `DefaultDateFormat`, `DefaultTimeFormat` and `DefaultDateTimeFormat` all take various formatting strings from the Moment.js library. A full listing can be found on the Moment website.

3.2.17 Include Tree

When Coalesce maps from the your POCO objects that are returned from EF Core queries, it will follow a structure called an `IncludeTree` to determine what relationships to follow and how deep to go in re-creating that structure in the mapped DTOs.

Contents

Purpose

Without an `IncludeTree` present, Coalesce will map the entire object graph that is reachable from the root object. This can often spiral out of control if there aren't any rules defining how far to go while turning this graph into a tree.

For example, suppose you had the following model with a many-to-many relationship (key properties omitted for brevity):

```
public class Employee
{
    [ManyToMany("Projects")]
    public ICollection<EmployeeProject> EmployeeProjects { get; set;␣
↪}

    public static IQueryable<Employee>␣
↪WithProjectsAndMembers(AppDbContext db, ClaimsPrincipal user)
    {
        // Load all projects of an employee, as well as all members␣
↪of those projects.
        return db.Employees.Include(e => e.EmployeeProjects)
                           .ThenInclude(ep => ep.Project.
↪EmployeeProjects)
                           .ThenInclude(ep => ep.Employee);
    }
}

public class Project
{
    [ManyToMany("Employees")]
    public ICollection<EmployeeProject> EmployeeProjects { get; set;␣
↪}
}

public class EmployeeProject
{
    public Employee Employee { get; set; }
    public Project Project { get; set; }
}
```

Now, imagine that you have five employees and five projects, with every employee being a member of every project (i.e. there are 25 EmployeeProject rows).

Your client code makes a call to the Coalesce-generated API to load Employee #1 using the custom data source:

```
var employee = new ViewModels.Employee();
employee.dataSource = new employee.dataSources.
↪WithProjectsAndMembers();
employee.load(1);
```

If you're already familiar with the fact that an `IncludeTree` is implicitly created in this scenario, then imagine for a moment that this is not the case (if you're not familiar with this fact, then keep reading!).

After Coalesce has called your *Data Sources* and evaluated the EF IQueryable returned, there are now 35 objects loaded into the current `DbContext` being used to handle this request - the 5 employees, 5 projects, and 25 relationships.

To map these objects to DTOs, we start with the root (employee #1) and expand outward from there until the entire object graph has been faithfully re-created with DTO objects, including all navigation

properties.

The root DTO object (employee #1) then eventually is passed to the JSON serializer by ASP.NET Core to formulate the response to the request. As the object is serialized to JSON, the only objects that are not serialized are those that were already serialized as an ancestor of itself. What this ultimately means is that the structure of the serialized JSON with our example scenario ends up following a pattern like this (the vast majority of items have been omitted):

```
Employee#1
    EmployeeProject#1
        Project#1
            EmployeeProject#6
                Employee#2
                    EmployeeProject#7
                        Project#2
                            ... continues down through all remaining␣
↪employees and projects.
                        ...
            EmployeeProject#11
                Employee#3
            ...
    EmployeeProject#2
        Project#2
    ...
```

See how the structure includes the EmployeeProjects of Employee#2? We didn't write our custom data source calls to `.Include` in such a way that indicated that we wanted the root employee, their projects, the employees of those projects, and then **the projects of those employees**. But, because the JSON serializer blindly follows the object graph, that's what gets serialized. It turns out that the depth of the tree increases on the order of `O(n^2)`, and the total size increases on the order of `Ω(n!)`.

This is where `IncludeTree` comes in. When you use a custom data source like we did above, Coalesce automatically captures the structure of the calls to `.Include` and `.ThenInclude`, and uses this to perform trimming during creation of the DTO objects.

With an `IncludeTree` in place, our new serialized structure looks like this:

```
Employee#1
    EmployeeProject#1
        Project#1
            EmployeeProject#6
                Employee#2
            EmployeeProject#11
                Employee#3
            ...
    EmployeeProject#2
        Project#2
    ...
```

No more extra data trailing off the end of the projects' employees!

Usage

Custom Data Sources

In most cases, you don't have to worry about creating an `IncludeTree`. When using the *Standard Data Source* (or a derivative), the structure of the `.Include` and `.ThenInclude` calls will be captured automatically and be turned

into an `IncludeTree`.

However, there are sometimes cases where you perform complex loading in these methods that involves loading data into the current `DbContext` outside of the `IQueryable` that is returned from the method. The most common situation for this is needing to conditionally load related data - for example, load all children of an object where the child has a certain value of a Status property.

In these cases, Coalesce provides a pair of extension methods, `.IncludedSeparately` and `.ThenIncluded`, that can be used to merge in the structure of the data that was loaded separately from the main `IQueryable`.

For example:

```
public override IQueryable<Employee> GetQuery()
{
    // Load all projects that are complete, and their members, into the db␣
↪context.
    Db.Projects
        .Include(p => p.EmployeeProjects).ThenInclude(ep => ep.Employee)
        .Where(p => p.Status == ProjectStatus.Complete)
        .Load();

    // Return an employee query, and notify Coalesce that we loaded the␣
↪projects in a different query.
    return Db.Employees.IncludedSeparately(e => e.EmployeeProjects)
                       .ThenIncluded(ep => ep.Project.EmployeeProjects)
                       .ThenIncluded(ep => ep.Employee);
}
```

You can also override the `GetIncludeTree` method of the *Standard Data Source* to achieve the same result:

```
public override IncludeTree GetIncludeTree(IQueryable<T> query,␣
↪IDataSourceParameters parameters)
    => Db.Employees.IncludedSeparately(e => e.EmployeeProjects)
                   .ThenIncluded(ep => ep.Project.EmployeeProjects)
                   .ThenIncluded(ep => ep.Employee)
                   .GetIncludeTree();
```

Model Methods

If you have instance or static methods on your models that return objects, you may also want to control the structure of the returned data when it is serialized. Fortunately, you can also use `IncludeTree` in these situations. Without an `IncludeTree`, the entire object graph is traversed and serialized without limit.

To tell Coalesce about the structure of the data returned from a model method, simply add `out IncludeTree includeTree` to the signature of the method. Inside your method, set `includeTree` to an instance of an `IncludeTree`. Obtaining an `IncludeTree` is easy - take a look at this example:

```
public class Employee
{
    public ICollection<Employee> GetChainOfCommand(AppDbContext db, out␣
↪IncludeTree includeTree)
    {
        var ret = new List<Employee>();
        var current = this;
        while (current.Supervisor != null)
        {
            ret.Push(current);
```

(continues on next page)

(continued from previous page)

```
            current = db.Employees
                .Include(e => e.Supervisor)
                .FirstOrDefault(e => e.EmployeeId == current.SupervisorId);
        }

        includeTree = db.Employees
            .IncludedSeparately(e => e.Supervisor)
            .GetIncludeTree();

        return ret;
    }
}
```

Tip: An `IncludeTree` can be obtained from any `IQueryable` by calling the `GetIncludeTree` extension method (`using IntelliTect.Coalesce.Helpers.IncludeTree`).

In situations where your root object isn't on your `DbContext` (see *External Types*), you can use `Enumerable.Empty<MyNonDbClass>().AsQueryable()` to get an `IQueryable` to start from. When you do this, you **must** use `IncludedSeparately` - the regular EF `Include` method won't work without a `DbSet`.

Without the outputted `IncludeTree` in this scenario, the object graph received by the client would have ended up looking like this:

```
- Steve's manager
    - District Supervisor
        - VP
            - CEO

- District Supervisor
    - VP
        - CEO

- VP
    - CEO

- CEO
```

Instead, with the `IncludeTree`, we get the following, which is only the data we actually wanted:

```
- Steve's manager
    - District Supervisor

- District Supervisor
    - VP

- VP
    - CEO

- CEO
```

If you wanted to get even simpler, you could simply set the `out includeTree` to a `new IncludeTree()`, which would give you only the top-most level of data:

```
- Steve's manager
- District Supervisor
```

(continues on next page)

(continued from previous page)

```
- VP
- CEO
```

External Type Caveats

One important point remains regarding `IncludeTree` - it is not used to control the serialization of objects which are not mapped to the database, known as *External Types*. External Types are always put into the DTOs when encountered (unless otherwise prevented by *[DtoIncludes] & [DtoExcludes]* or *Security Attributes*), with the assumption that because these objects are created by you (as opposed to Entity Framework), you are responsible for preventing any undesired circular references.

By not filtering unmapped properties, you as the developer don't need to account for them in every place throughout your application where they appear - instead, they 'just work' and show up on the client as expected.

Note also that this statement does not apply to database-mapped objects that hang off of unmapped objects - any time a database-mapped object appears, it will be controlled by your include tree. If no include tree is present (because nothing was specified for the unmapped property), these mapped objects hanging off of unmapped objects will be serialized freely and with all circular references, unless you include some calls to `.IncludedSeparately(m => m.MyUnmappedProperty.MyMappedProperty)` to limit those objects down.

3.2.18 Includes String

Coalesce provides a number of extension points for loading & serialization which make use of a concept called an "includes string" (also referred to as "include string" or just "includes").

Contents

Includes String

The includes string is simply a string which can be set to any arbitrary value. It is passed from the client to the server in order to control data loading and serialization. It can be set on both the TypeScript ViewModels and the ListViewModels.

```
var person = new ViewModels.Person();
person.includes = "details";

var personList = new ListViewModels.PersonList();
personList.includes = "details";
```

The default value (i.e. no action) is the empty string.

Special Values

There are a few values of `includes` that are either set by default in the auto-generated views, or otherwise have special meaning:

`none` Setting `includes` to `none` suppresses the *Default Loading Behavior* provided by the *Standard Data Source* - The resulting data will be the requested object (or list of objects) and nothing more.

`Editor` Used when loading an object in the generated CreateEdit views.

`<ModelName>ListGen` Used when loading a list of objects in the generated Table and Cards views. For example, `PersonListGen`

DtoIncludes & DtoExcludes

Main document: *[DtoIncludes] & [DtoExcludes]*.

There are two C# attributes, `DtoIncludes` and `DtoExcludes`, that can be used to annotate your data model in order to control what data gets put into the DTOs and ultimately serialized to JSON and sent out to the client.

When the database entries are returned to the client they will be trimmed based on the requested includes string and the values in `DtoExcludes` and `DtoIncludes`.

> **Caution:** These attributes are **not security attributes** - consumers of your application's API can set the includes string to any value when making a request.
>
> Do not use them to keep certain data private - use the *Security Attributes* family of attributes for that.

It is important to note that the value of the includes string will match against these attributes on *any* of your models that appears in the object graph being mapped to DTOs - it is not limited only to the model type of the root object.

Example Usage

```
public class Person
{
    // Don't include CreatedBy when editing - will be included for all other␣
↪views
    [DtoExcludes("Editor")]
    public AppUser CreatedBy { get; set; }

    // Only include the Person's Department when :ts:`includes == "details"`␣
↪on the TypeScript ViewModel.
    [DtoIncludes("details")]
    public Department Department { get; set; }

    // LastName will be included in all views
    public string LastName { get; set; }
}
```

(continues on next page)

(continued from previous page)

```
public class Department
{
    [DtoIncludes("details")]
    public ICollection<Person> People { get; set; }
}
```

In TypeScript:

```
var personList = new ListViewModels.PersonList();
personList.includes = "Editor";
personList.load(() => {
    // objects in personList.items will not contain CreatedBy nor Department␣
↪objects.
});
```

```
var personList = new ListViewModels.PersonList();
personList.includes = "details";
personList.load(() => {
    // objects in personList.items will be allowed to contain both CreatedBy␣
↪and Department objects. Department will be allowed to include its other␣
↪Person objects.
});
```

Properties

`public string ContentViews { get; set; }` **1** A comma-delimited list of values of `includes` on which to operate.

For `DtoIncludes`, this will be the values of `includes` for which this property will be allowed to be serialized and sent to the client.

Important: `DtoIncludes` does not ensure that specific data will be loaded from the database. Only data loaded into current EF DbContext can possibly be returned from the API. See *Data Sources* for more information.

For `DtoExcludes`, this will be the values of `includes` for which this property will **never** be serialized and sent to the client.

3.2.19 Application Configuration

In order for Coalesce to work in your application, you must register the needed services in your `Startup.cs` file. Doing so is simple:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCoalesce<AppDbContext>();
    ...
}
```

This registers all the basic services that Coalesce needs in order to work with your EF DbContext. However, there are many more options available. Here's a more complete invocation of `AddCoalesce` that takes advantage of many of the options available:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCoalesce(builder => builder
        .AddContext<AppDbContext>()
        .UseDefaultDataSource(typeof(MyDataSource<,>))
        .UseDefaultBehaviors(typeof(MyBehaviors<,>))
        .UseTimeZone(TimeZoneInfo.FindSystemTimeZoneById("Pacific Standard
↪Time"))
    );
}
```

A summary is as follows:

`.AddContext<AppDbContext>()` Register services needed by Coalesce to use the specified context. This is done automatically when calling the `services.AddCoalesce<AppDbContext>();` overload.

`.UseDefaultDataSource(typeof(MyDataSource<,>))` Overrides the default data source used, replacing the *Standard Data Source*. See *Data Sources* for more details.

`.UseDefaultBehaviors(typeof(MyBehaviors<,>))` Overrides the default behaviors used, replacing the *Standard Behaviors*. See *Behaviors* for more details.

`.UseTimeZone(TimeZoneInfo.FindSystemTimeZoneById("Pacific Standard Time"))` Specify a static time zone that should be used when Coalesce is performing operations on dates/times that lack timezone information. For example, when a user inputs a search term that contains only a date, Coalesce needs to know what timezone's midnight to use when performing the search.

`.UseTimeZone<ITimeZoneResolver>()` Specify a service implementation to use to resolve the current timezone. This should be a scoped service, and will be automatically registered if it is not already. This allows retrieving timezone information on a per-request basis from HTTP headers, Cookies, or any other source.

3.2.20 Code Generation Configuration

In Coalesce, all configuration of the code generation is done in a JSON file. This file is typically named `coalesce.json` and is typically placed in the solution root.

File Resolution

When the code generation is ran by invoking `dotnet coalesce`, Coalesce will try to find a configuration file via the following means:

1. If an argument is specified on the command line, it will be used as the location of the file. E.g. `dotnet coalesce C:/Projects/MyProject/config.json`
2. If no argument is given, Coalesce will try to use a file in the working directory named `coalesce.json`
3. If no file is found in the working directory, Coalesce will crawl up the directory tree from the working directory until a file named `coalesce.json` is found. If such a file is never found, an error will be thrown.

Contents

A full example of a `coalesce.json` file, along with an explanation of each property, is as follows:

```
{
    "webProject": {
        // Required: Path to the csproj of the web project. Path is relative to␣
↪location of this coalesce.json file.
        "projectFile": "src/Coalesce.Web/Coalesce.Web.csproj",

        // Optional: Framework to use when evaluating & building dependencies.
        // Not needed if your project only specifies a single framework - only␣
↪required for multi-targeting projects.
        "framework": "netcoreapp2.0",

        // Optional: Build configuration to use when evaluating & building␣
↪dependencies.
        // Defaults to "Debug".
        "configuration": "Debug",

        // Optional: Override the namespace prefix for generated C# code.
        // Defaults to MSBuild's `$(RootNamespace)` for the project.
        "rootNamespace": "MyCompany.Coalesce.Web",
    },

    "dataProject": {
        // Required: Path to the csproj of the data project. Path is relative to␣
↪location of this coalesce.json file.
        "projectFile": "src/Coalesce.Domain/Coalesce.Domain.csproj",

        // Optional: Framework to use when evaluating & building dependencies.
        // Not needed if your project only specifies a single framework - only␣
↪required for multi-targeting projects.
        "framework": "netstandard2.0",

        // Optional: Build configuration to use when evaluating & building␣
↪dependencies.
        // Defaults to "Release".
        "configuration": "Debug",
    }

    "generatorConfig": {
        // A set of objects keyed by generator name.
        // Generator names may optionally be qualified by their full namespace.
        // All generators are listed when running 'dotnet coalesce' with '--verbosity␣
↪debug'.
        // For example, "Views" or "IntelliTect.Coalesce.CodeGeneration.Knockout.
↪Generators.Views".
        "GeneratorName": {
            // Optional: true if the generator should be disabled.
            "disabled": true
        },
        // Indentation for generated C# is configurable by type (API controllers, DTO␣
↪classes and regular View controllers)
        // It defaults to 4 spaces
        "ApiController": {
            "indentationSize": 2
        },
        "ClassDto": {
            "indentationSize": 2
        },
```

(continues on next page)

(continued from previous page)

```
        "ViewController" : {
            "indentationSize": 2
        }
    }
}
```

Additional CLI Options

There are a couple of extra options which are only available as CLI parameters to `dotnet coalesce`. These options do not affect the behavior of the code generation - only the behavior of the CLI itself.

`--debug` When this flag is specified when running `dotnet coalesce`, Coalesce will wait up to 60 seconds for a debugger to be attached to its process before starting code generation.

`-v|--verbosity <level>` Set the verbosity of the output. Options are `trace`, `debug`, `information`, `warning`, `error`, `critical`, and `none`.