Coalesce Documentation

IntelliTect

Aug 05, 2020

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
- Vue or 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 TypeScriptViewModels for either Vue or Knockout 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 *Vue Components* or *Knockout bindings* for common controls like dates, selecting objects via drop downs, enums, etc. Dropdowns support searching and paging automatically.
- A complete set of admin pages are provided, allowing you to read, create, edit, and delete data straight away without writing any additional code.

CHAPTER 3

Getting Started

To get started with Coalesce, you first must choose which front-end stack you wish to use - Vue, or Knockout. While the Knockout stack is still fully supported, the Vue stack is the newer, more modern stack and Vue itself has a much bigger worldwide community and ecosystem of libraries, components, plugins, support, and more. The Vue-based stack is the one that will be receiving the bulk of development effort in Coalesce going forward.

If you still need help choosing, check out the overviews for each stack:

- *Vue Overview*
- *Knockout Overview*

Once you've made your decision, check out one of the two links below:

- *Getting Started with Vue*
- *Getting Started with Knockout*

3.1 EF Entity Models

3.1.1 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).

3.1.2 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.

3.1.3 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 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.

3.2.1 Generated Code

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

- A *Generated DTO*
- A TypeScript Model

3.2.2 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);
```

(continues on next page)

(continued from previous page)

```
}

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; }
}
```

3.2.3 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.

3.3 Custom DTOs

In addition to the generated *Generated C# 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 TypeScriptViewModels and TypeScriptListViewModels.

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 mapping data incoming 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 C# DTOs*.

3.3.1 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;
    }

    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;
```

(continues on next page)

(continued from previous page)

```
[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)]
        ...
    }
}
```

3.3.2 Using Custom DataSources and Behaviors

Declaring an IClassDto DataSource

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; }
```

(continues on next page)

(continued from previous page)

```
    ...
}

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

ProjectedDtoDataSource

In addition to creating a *Data Sources* by deriving from *Standard Data Source*, there also exists a class `ProjectedDtoDataSource` that can be used to easily perform projection from EF model types to your custom DTO types using EF query projections. `ProjectedDtoDataSource` inherits from *Standard Data Source*.

```
[Coalesce, DeclaredFor(typeof(CaseDto))]
public class CaseDtoSource : ProjectedDtoDataSource<Case, CaseDto,␣
↪AppDbContext>
{
    public CaseDtoSource(CrudContext<AppDbContext> context) : base(context)
↪{ }

    public override IQueryable<CaseDto> ApplyProjection(IQueryable<Case>␣
↪query, IDataSourceParameters parameters)
    {
        return query.Select(c => new CaseDto
        {
            CaseId = c.CaseKey,
            Title = c.Title,
            AssignedToName = c.AssignedTo == null ? null : c.AssignedTo.Name
        });
    }
}
```

3.4 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.

3.4.1 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.

3.4.2 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.

3.5 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.

3.5.1 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 TypeScriptViewModels typed as TypeScriptExternalViewModel, 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 TypeScriptViewModels, 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.

3.5.2 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 C# 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.

3.6 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`.

3.6.1 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 generated *Metadata* and *ViewModel rules* (for Vue) or Knockout-Validation rules (for Knockout). For both stacks, any failing validation rules prevent saves from going to the server.

> **Warning:** This attribute controls client-side validation only. To perform server-side validation, create a custom *Behaviors* for your types.

Contents

Example Usage

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

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

(continues on next page)

(continued from previous page)

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

Properties

Behavioral Properties

`public bool AllowSave { get; set; }` (Knockout Only) 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

Validation Rule Properties

Knockout

The following attribute 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 attribute 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; }
```

The following two properties may be 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; }
```

Vue

In addition to the following properties, you also customize validation on a per-instance basis of the *ViewModels* using the *Rules/Validation* methods.

```
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 string Pattern { get; set; }
public bool IsEmail { get; set; }
public bool IsPhoneUs { get; set; }
```

[Coalesce]

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

Some types and members will be implicitly 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-overridden 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 suppressing 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 TypeScriptListViewModels.

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:

Vue

```
import { PersonListViewModel } from '@/viewmodels.g'

const personList = new PersonListViewModel();
personList.$includes = "Editor";
await personList.$load();
// Objects in personList.$items will not contain CreatedBy nor Department objects.

const personList2 = new PersonListViewModel();
personList2.$includes = "details";
await personList.$load();
// Objects in personList2.items will be allowed to contain both CreatedBy and␣
↪Department objects.
// Department will be allowed to include its other Person objects.
```

Knockout

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

var personList2 = new ListViewModels.PersonList();
personList2.includes = "details";
personList2.load(() => {
    // objects in personList2.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; } = SecurityPermissionLevels.Al`
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.

[File]

Coalesce supports the uploading and downloading of files and images through the `IntelliTect.Coalesce.DataAnnotations.FileAttribute` annotation.

> **Warning:** This attribute is currently only supported by the Knockout client stack. It is unsupported in Vue.

> **Warning:** Storing large binary objects in relational databases comes with significant performance drawbacks. Use this attribute sparingly, and try to avoid using it for files bigger than a few dozen kilobytes.

Tip: To upload a file in Coalesce, consider using *Custom Methods* instead.

Example Usage

```
public class Product
{
    public int ProductId { get; set; }

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

    [File("text/plain")]
    public byte[] ProductDescription { get; set; }

    [File("image/jpeg")]
    public byte[] ProductThumbnail { get; set; }

    [File]
    public byte[] BinaryDriverFile { get; set; }
}
```

Additionally, Coalesce can store file metadata in optional properties that are specified on the attribute's properties. Security information can also be specified with property-level *Security Attributes*. In the example below, the file can be viewed/downloaded by any user, but only users with the role of `"Admin"` or `"Vendor"` will be able to upload a new file.

```
[Read]
[Edit(Roles = "Admin,Vendor")]
[File("video/mp4", nameof(VideoName), nameof(VideoHash), nameof(VideoSize))]
public byte[] InstallationVideo { get; set; }

public string VideoName
{
    get { return "Product" + ProductId + ".mp4" }
}
public long VideoSize { get; set; }
public string VideoHash { get; set; }
```

The *[InternalUse]* attribute can be used in conjunction with `IntelliTect.Coalesce.DataAnnotations.FileAttribute`. In the example below, Coalesce will store the uploaded filename and provide it again as the name of a file download, but the property itself won't actually get exposed on the DTOs via the API.

```
[File(NameProperty = nameof(InternalUseFileName))]
public byte[] File { get; set; }

[InternalUse]
public string InternalUseFileName { get; set; }
```

Properties

`public string MimeType { get; set; }` The system will identify the nature and format of the file using the type described in the 'MimeType' property. By default it is set to `"application/octet-stream"` to read/write the file as an unknown binary file.

An image preview will be displayed for properties annotated with the `File` attribute whose MIME type contains `"image"`. Other MIME types will result in a download button being displayed.

> **Tip:** If a filename exists on the uploaded file, the MIME type may inferred from the file extension and `MimeType` need not be specified.

`public string NameProperty { get; set; }` A property to store the filename into. If the specified property has no setter, this will return a computed filename and the name of the uploaded file will not be used. If the property does have a setter, the property will be populated by the filename on upload.

`public string HashProperty { get; set; }` The name of the property to store the hash of `Byte[]`. This is set upon file upload.

`public string SizeProperty { get; set; }` A property to store the size of the file into. This is set upon file upload.

[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 rendering of the admin pages. 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("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. This attribute can be considered a *Security Attribute*.

Note that this only needs to be used on members that are public. Non-public members (including `internal`) 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 TypeScriptViewModels.

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]
    [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, 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 TypeScriptViewModels.

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-whitespace 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 C# 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; }

    [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 the generated Knockout HTML views - it does not enforce any relational rules in your data.

This attribute also currently has no effect against the Vue stack.

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]

Note: This attribute only applies to the Knockout front-end stack. It is not applicable to the Vue stack.

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.

3.6.2 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 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 of a property can also be set via the `[DisplayName]` attribute.

[Required]

Properties with `[Required]` will generate client validation rules. See *[ClientValidation]*.

[Range]

Properties with `[Range]` will generate client validation rules. See *[ClientValidation]*.

[MinLength]

Properties with `[MinLength]` will generate client validation rules. See *[ClientValidation]*.

[MaxLength]

Properties with `[MaxLength]` will generate client validation rules. 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]`.

[DatabaseGenerated]

Primary Keys with `[DatabaseGenerated(DatabaseGeneratedOption.None)]` will be settable on the client and will be appropriately handled by the *Standard Behaviors* on the server. Currently unsupported on the *Knockout front-end stack*.

[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*.

3.7 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

3.7.1 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.

Files Methods can accept file uploads by using a parameter of type `IntelliTect.Coalesce.Models.IFile` (or any derived type, like `IntelliTect.Coalesce.Models.File`).

`<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 C# DTOs* and *Include Tree* for more information. If the method returns an `IQueryable`, the out parameter will supersede the include tree obtained from inspecting the query.

3.7.2 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.

Downloading files from custom methods is currently unsupported. Please open a feature request on GitHub *if this would be useful for you.*

3.7.3 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.

3.7.4 Generated TypeScript

See *API Callers* and *ViewModel Layer* (Vue) or *TypeScript Method Objects* (Knockout) for details on the code that is generated for your custom methods.

Tip: Any Task-returning methods with "Async" as a suffix to the C# method's name will have the "Async" suffix stripped from the generated Typescript.

3.7.5 Instance Methods

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.

Instance methods are generated onto the TypeScript ViewModels.

3.7.6 Static Methods

Static methods are generated onto the TypeScript ListViewModels. 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 (Knockout only).

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

3.7.7 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.

3.8 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

3.8.1 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.

3.8.2 Consuming Data Sources

Vue

The *ViewModels* and *ListViewModels* each have a property called `$dataSource`. This property accepts an instance of a *DataSource* class generated in the *Model Layer*.

```
import { Person } from '@/viewmodels.g'
import { PersonViewModel, PersonListViewModel } from '@/viewmodels.g'
```

(continues on next page)

(continued from previous page)

```
var viewModel = new PersonViewModel();
viewModel.$dataSource = new Person.DataSources.IncludeFamily();
viewModel.$load(1);

var list = new PersonListViewModel();
list.$dataSource = new Person.DataSources.NamesStartingWith();
list.$load(1);
```

Knockout

The *TypeScript ViewModels* and *TypeScript ListViewModels* each 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();
```

3.8.3 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.

3.8.4 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));
}
```

List Auto-loading

You can setup TypeScriptListViewModels to automatically reload from the server when data source parameters change:

Vue

To automatically reload a *ListViewModel* when data source parameters change, simply use the list's `$startAutoLoad` function:

```
import { Person } from '@/models.g';
import { PersonListViewModel } from '@/viewmodels.g';

const list = new PersonListViewModel;
const dataSource = list.$dataSource = new Person.DataSources.NamesStartingWith
list.$startAutoLoad(this);

// Trigger a reload:
dataSource.startsWith = "Jo";
```

Knockout

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");
dataSource.subscribe(list); // Enables automatic reloading.
list.dataSource = dataSource;
list.load();
```

3.8.5 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
        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.

3.8.6 Globally 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.

3.9 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

3.9.1 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)
```

(continues on next page)

(continued from previous page)

```
            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.
        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.

3.9.2 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 C# 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 receiving 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 against 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 =>
    {
```

(continues on next page)

(continued from previous page)

```
        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.

3.10 Code Generation Overview

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

Contents

3.10.1 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" 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`.

3.10.2 Generated Code

Coalesce has the option of two front-end stacks - either Knockout, or Vue. The Vue-based stack is the current focus of all development efforts against Coalesce going forward - the Knockout stack is effectively in maintenance-only mode.

For either stack, Coalesce will generate a variety of different kinds of code for you:

Server-side C#

API Controllers For each of your *EF Entity Models*, *Custom DTOs*, and *Services*, 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.

These controllers can be secured at a high level using *Security Attributes*, and when applicable to the type, with *Data Sources* and *Behaviors*.

C# DTOs For each of your *EF Entity Models*, 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 C# DTOs* for more information.

Vue

An overview of the Vue stack can be found at *Vue Overview*.

Knockout

An overview of the Knockout stack can be found at *Knockout Overview*.

3.11 Generated C# 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*.

3.11.1 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.

3.12 Vue Overview

The Vue stack for Coalesce has been designed from the ground up to be used to build modern web applications using current technologies like Webpack, Vue CLI, ES Modules, and more. It enables you to use all of the features of Vue.js, including building a SPA, and the ability to use modern component frameworks like Vuetify.

Contents

3.12.1 Getting Started

Check out *Getting Started with Vue* if you haven't already to learn how to get a new Coalesce Vue project up and running.

3.12.2 TypeScript Layers

The generated code for the Vue stack all builds on the coalesce-vue NPM package which contains most of the core functionality of the Vue stack. Its version should generally be kept in sync with the IntelliTect.Coalesce NuGet packages in your project.

Both the generated code and coalesce-vue are split into four layers, with each layer building on the layers underneath. From the bottom, these layers are:

Metadata Layer

The metadata layer, generated as *metadata.g.ts*, contains a minimal set of metadata to represent your data model on the front-end. Because Vue applications are typically compiled into a set of static assets, it is necessary for the frontend

code to have a representation of your data model as an analog to the `ReflectionRepository` available at runtime to Knockout apps that utilize *.cshtml* files.

Read more about the Metadata layer

Model Layer

The model layer, generated as *models.g.ts*, contains a set of TypeScript interfaces that represent each client-exposed type in your data model. Each interface declares all the *Properties* of that type, as well as a `$metadata` property that references the metadata object for that type. Enums and *Data Sources* are also represented in the model layer.

Read more about the Model layer

API Client Layer

The API client layer, generated as *api-clients.g.ts*, exports a class for each API controller that was generated for your data model. These classes are stateless and provide one method for each API endpoint. This includes both the standard set of endpoints created for *EF Entity Models* and *Custom DTOs*, as well as any custom *Methods*.

Read more about the API Client layer

ViewModel Layer

The ViewModel layer, generated as *viewmodels.g.ts*, exports a ViewModel class for each API-backed type in your data model (*EF Entity Models*, *Custom DTOs*, and *Services*). It also exports an additional ListViewModel type for *EF Entity Models*, *Custom DTOs*.

These ViewModels contain the majority of functionality that you will use on a day-to-day basis as you build applications with the Coalesce Vue stack. They are all valid implementations of their corresponding model interface, and as such can be used in any place where a model could be used.

Read more about the ViewModel layer

3.12.3 Vue Components

The Vue stack for Coalesce provides *a set of components* based on Vuetify, packaged up in an NPM package coalesce-vue-vuetify. These components are driven primarily by the *Metadata Layer*, and include both low level input and display components like *c-input* and *c-display* that are highly reusable in the custom pages you'll build in your application, as well as high-level components like *c-admin-table-page* and *c-admin-editor-page* that constitute entire pages.

Read more about the Vuetify Components here.

3.12.4 Admin Views

The Vue stack for Coalesce does not generate any admin views for you like the *Knockout stack* does. Instead, it provides some high level components that provide functionality of whole pages like *c-admin-table-page* and *c-admin-editor-page* - these are the analogues of the generated admin Table and CreateEdit views in the *Knockout stack*.

These components are driven off of the generated layers described above rather than being statically generated like the Knockout admin pages - this allows us to keep bundle size to a minimum.

The template described in *Getting Started with Vue* comes with routes already in place for these page-level components. For example, `/admin/Person` for a table, `/admin/Person/edit` to create a new `Person`, and `/admin/Person/edit/:id` to edit a `Person`.

Read more about the Vuetify Components here.

3.13 Getting Started with Vue

3.13.1 Creating a Project

The quickest and easiest way to create a new Coalesce Vue application is to use the `dotnet new` template. In your favorite shell:

```
mkdir MyCompany.MyProject
cd MyCompany.MyProject
dotnet new --install IntelliTect.Coalesce.Vue.Template
dotnet new coalescevue
cd *.Web
npm ci
```

- View on GitHub

3.13.2 Project Structure

Important: The Vue template is based on Vue CLI. You are strongly encouraged to read through at least the first few pages of the Vue CLI Documentation before getting started on any development.

The structure of the Web project follows the conventions of both ASP.NET Core and Vue CLI. The Vue-specific folders are as follows:

- `/src` - Files that should be compiled into your application. CSS/SCSS, TypeScript, Vue SFCs, and so on.
- `/public` - Static assets that should be served as files. Includes index.html, the root document of the application.
- `/tests` - Jest unit tests.
- `/wwwroot` - Target for compiled output.

During development, no special tooling is required to build your frontend code. `WebpackDevMiddleware` in ASP.NET Core will take care of that automatically when the application starts.

Tip: If developing with Visual Studio, you are strongly encouraged to disable Visual Studio's built-in automatic NPM package restore functionality (`Options > Projects and Solutions > Web Package Management > Package Restore`).

This feature of Visual Studio fails to respect your `package.lock.json` file, and the version of NPM that Visual Studio comes with tends to be quite old and will behave differently from the `npm` on your system's $PATH.

You should manually restore your packages with `npm ci` (when you haven't tried to change any versions) or `npm i` (when installing new packages or upgrading versions).

3.13.3 Data Modeling

At this point, you can open up the newly-created 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` npm script (Vue) or gulp task (Knockout) in the Task Runner Explorer

You're now at a point where you can start creating your own pages!

3.13.4 Building Pages & Features

Lets say we've created a *model* called `Person` as follows, 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 Single File Component in `MyApplication.Web/src/views/person-details.vue`:

```
<template>
  <dl>
    <dt>Name</dt>
    <dd>
      <c-display :model="person" for="name" />
    </dd>

    <dt>Date of Birth</dt>
    <dd>
      <c-display :model="person" for="birthDate" format="M/d/yyyy" />
    </dd>
  </dl>
</template>

<script lang="ts">
import { Vue, Component, Watch, Prop } from "vue-property-decorator";
import { PersonViewModel } from "@/viewmodels.g";

@Component({})
export default class extends Vue {
```

(continues on next page)

(continued from previous page)

```
  @Prop({ required: true, type: Number })
  id!: number;

  person = new PersonViewModel();

  created() {
    this.person.$load(this.id);
  }
}
</script>
```

Note: In the code above, *c-display* is a component that comes from the *Vuetify Components* for Coalesce.

For simple property types like `string` and `number` you can always use simple template interpolation syntax, but for more complex properties like dates, *c-display* is handy to use because it includes features like built-in date formatting.

Tip: The code above uses vue-class-component and vue-property-decorator to define the component.

These libraries provide an alternative to the default component declaration syntax in Vue. However, you must be aware of the Caveats if you want to use these tools to build your own class-style components.

We then need to add route to this new view. In `MyApplication.Web/src/router.ts`, add a new item to the `routes` array:

```
// At the top of the file, import the component:
import PersonDetails from '@/views/person-details.vue';
```

```
// In the `routes` array, add the following item:
{
  path: '/person/:id',
  name: 'person-details',
  component: PersonDetails,
  props: route => ({ id: +route.params.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 (or, if it was already running, refresh the page) and navigate to `/person/1` (assuming a person with ID 1 exists - if not, navigate to `/admin/Person` and create one).

From this point, you 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, including the *Vue Overview*.

3.14 Metadata Layer

The metadata layer, generated as *metadata.g.ts*, contains a minimal set of metadata to represent your data model on the front-end. Because Vue applications are typically statically compiled, it is necessary for the frontend code to have a representation of your data model as an analog to the `ReflectionRepository` available at runtime to Knockout apps that utilize *.cshtml* files.

Contents

3.14.1 Concepts

The following is a non-exhaustive list of the general concepts used by the metadata layer. The source code of coalesce-vue provides the most exhaustive set of documentation about the metadata layer:

Metadata

All objects in the metadata layer that represent any kind of metadata have, at the very least, a `name`, the name of the metadata element in code (type names, property names, parameter names, etc). and a `displayName`, the human-readable form of the name that is suitable for presentation when needed. Names follow the casing convention of their corresponding language elements - types are PascalCased, while other things like properties, methods, and parameters are camelCased.

Type

All custom types exposed by your application's data model will have a Type metadata object generated. This includes both C# classes, and C# enums. Class types include `model` (for *EF Entity Models* and *Custom DTOs*) and `object` (for *External Types*).

Value

In the metadata layer, a Value is the usage of a type. This could be any type - strings, numbers, enums, classes, or even void. Values can be found in the collection of an object's properties, a method's parameters or return value, or as a data source's parameters.

All values have a:

`type` Type could be a language primitive like `string` or `number`, a non-primitive JavaScript type (`date`, `file`), or in the case of a custom Type, the type kind of that type (`model`, `enum`, `object`). For custom types, an additional property `typeDef` will refer to the Type metadata for that type.

`role` Role represents what purpose the value serves in a relational model. Either *value* (the default - no relational role), *primaryKey*, *foreignKey*, *referenceNavigation*, or *collectionNavigation*.

Property

A Property is a more refined Value that contains a number of additional fields based on the `role` of the property. k

Domain

The type of the default export of the generated metadata. Serves as a single root from which all other metadata can be accessed. Contains fields `types`, `enums`, and `services` as organizing structures for the different kinds of custom types.

3.15 Model Layer

The model layer, generated as *models.g.ts*, contains a set of TypeScript interfaces that represent each client-exposed type in your data model. Each interface contains all the *Properties* of that type, as well as a `$metadata` property that references the *metadata* object for that type. Enums and *Data Sources* are also represented in the model layer.

The model layer also includes a TypeScript class for each type that can be used to easily instantiate a valid implementation of its corresponding interface. However, it is not necessary for the classes to be used, and all parts of Coalesce that interact with the model layer don't perform any *instanceof* checks against models - the `$metadata` property is used to determine type identity.

Contents

3.15.1 Concepts

The model layer is fairly simple - the only main concept it introduces on top of the *Metadata Layer* is the notion of interfaces and enums that mirror the C# types in your data model. As with the *Metadata Layer*, the source code of coalesce-vue is a great documentation supplement to this page.

Model

An interface describing an instance of a class type from your application's data model. All Model interfaces contain members for all the *Properties* of that type, as well as a `$metadata` property that references the metadata object for that type.

DataSource

A class-based representation of a *Data Source* containing properties for any of the *Custom Parameters* of the data source, as well as a `$metadata` property that references the metadata object for the data source.

Data sources are generated as concrete classes in a namespace named `DataSources` that is nested inside a namespace named after their parent model type. For example:

```
import { Person } from '@/models.g'

const dataSource = new Person.DataSources.NamesStartingWith;
dataSource.startsWith = "A";
// Provide the dataSource to an API Client or a ViewModel...
```

3.15.2 Model Functions

The following functions exported from `coalesce-vue` can be used with your models:

`bindToQueryString(vue: Vue, obj: {}, key: string, queryKey: string = key, parse?: (v:`

Binds property `key` of `obj` to query string parameter `queryKey`. When the object's value changes, the query string will be updated using vue-router. When the query string changes, the object's value will be updated.

The query string will be updated using either `router.push` or `router.replace` depending on the value of parameter `mode`.

If the query string contains a value when this is called, the object will be updated with that value immediately.

If the object being bound to has `$metadata`, information from that metadata will be used to serialize and parse values to and from the query string. Otherwise, `String(value)` will be used to serialize the value, and the `parse` parameter (if provided) will be used to parse the value from the query string.

```
import { bindToQueryString } from 'coalesce-vue';

// In the 'created' Vue lifecycle hook on a component:
created() {
    // Bind pagination information to the query string:
    bindToQueryString(this, this.listViewModel.$params, 'pageSize', 'pageSize', v␣
↪=> +v);

    // Assuming the component has an 'activeTab' data member:
    bindToQueryString(this, this, 'activeTab');
}
```

`bindKeyToRouteOnCreate(vue: Vue, model: Model<ModelType>, routeParamName: string = 'id'`

When `model` is created (i.e. its primary key becomes non-null), replace the current URL with one that includes uses primary key for the route parameter named by `routeParamName`.

The query string will not be kept when the route is changed unless `true` is given to `keepQuery`.

```
import { bindKeyToRouteOnCreate } from 'coalesce-vue';

// In the 'created' Vue lifecycle hook on a component:
created() {
    if (this.id) {
        this.viewModel.$load(this.id);
    } else {
        bindKeyToRouteOnCreate(this, this.viewModel);
    }
}
```

Note: The route will be replaced directly via the HTML5 History API such that vue-router will not observe the change as an actual route change, preventing the current view from being recreated if a path-based key is being used on the application's `<router-view>` component.

3.15.3 Advanced Model Functions

The following functions exported from `coalesce-vue` can be used with your models.

Note: These functions are used to implement the *higher-order layers* in the Vue stack.

While you're absolutely free to use them in your own code and can rely on their interface and behavior to remain consistent, you will find that you seldom need to use them directly - that's why we've split them into their own section here in the documentation.

`convertToModel(value: any, metadata: Value | ClassType): any` Given any JavaScript value `value`, convert it into a valid implementation of the value or type described by `metadata`.

For metadata describing a primitive or primitive-like value, the input will be parsed into a valid implementation of the correct JavaScript type. For example, for `metadata` that describes a boolean, a string `"true"` will return a boolean `true`, and ISO 8601 date strings will result in a JavaScript `Date` object.

For metadata describing a type, the input object will be mutated into a valid implementation of the appropriate model interface. Missing properties will be set to null, and any descendent properties of the provided object will be recursively processed with `convertToModel`.

If any values are encountered that are fundamentally incompatible with the requested type described by the metadata, an error will be thrown.

`mapToModel(value: any, metadata: Value | ClassType): any` Performs the same operations as `convertToModel`, except that any objects encountered will not be mutated - instead, a new object or array will always be created.

`mapToDto(value: any, metadata: Value | ClassType): any` Maps the input to a representation suitable for JSON serialization.

Will not serialize child objects or collections whose metadata includes *dontSerialize*. Will only recurse to a maximum depth of 3.

`modelDisplay(model: Model, options?: DisplayOptions): string` Returns a string representing the `model` suitable for display in a user interface.

Uses the `displayProp` defined on the object's metadata. If no `displayProp` is defined, the object will be displayed as JSON. The display prop on a model can be defined in C# with *[ListText]*.

See *DisplayOptions* for available options.

`propDisplay(model: Model, prop: Property | string, options?: DisplayOptions): string` Returns a string representing the specified property of the given object suitable for display in a user interface.

The property can either be a string, representing one of the model's properties, or the actual `Property` metadata object of the property.

See *DisplayOptions* for available options.

`valueDisplay(value: any, metadata: Value, options?: DisplayOptions): string` Returns a string representing the given value (described by the given metadata).

See *DisplayOptions* for available options.

3.15.4 DisplayOptions

The following options are available to functions in coalesce-vue that render a value or object for display:

`format` Options to be used when formatting a date. One of:

string A UTS#35 date format string, to be passed to date-fns's format function.

Defaults to `"M/d/yyyy"` for date-only dates (specified with *[DateType]*), or `"M/d/yyyy h:mm:ss aaa"` otherwise.

{ distance: true; addSuffix?: boolean; includeSeconds?: boolean; }
Options to be passed to date-fns's formatDistanceToNow function.

Note: Values rendered with `formatDistanceToNow` function into a Vue component will not automatically be updated in realtime. If this is needed, you should use a strategy like using a key that you periodically update to force a re-render.

---

collection: { enumeratedItemsMax?: number, enumeratedItemsSeparator?: string }
Options to be used when formatting a collection.

3.16 API Client Layer

The API client layer, generated as *api-clients.g.ts*, exports a class for each API controller that was generated for your data model. These classes are stateless and provide one method for each API endpoint. This includes both the standard set of endpoints created for *EF Entity Models* and *Custom DTOs*, as well as any custom *Methods* on the aforementioned types, as well as any methods on your *Services*.

The API clients provided by Coalesce are based on axios. All API clients used a shared axios instance, exported from *coalesce-vue* as `AxiosClient`. This instance can be used to configure all HTTP requests made by Coalesce, including things like attaching interceptors to modify the requests being made, or configuring defaults.

As with all the layers, the source code of coalesce-vue is also a great supplement to this documentation.

Contents

3.16.1 Concepts

API Client A class, generated for each controller-backed type in your data model as `<ModelName>ApiClient` and exported from *api-clients.g.ts* containing one method for each API endpoint.

Each method on the API client takes in the regular parameters of the method as you would expect, as well as an optional `AxiosRequestConfig` parameter at the end that can be used to provide additional configuration for the single request, if needed.

For the methods that correspond to the standard set of CRUD endpoints that Coalesce provides (`get`, `list`, `count`, `save`, `delete`), an additional parameter `parameters` is available that accepts the set of *Standard Parameters* appropriate for the endpoint.

Each method returns a `Promise<AxiosResponse<TApiResult>>` where `TApiResult` is either `ItemResult`, `ItemResult<T>`, or `ListResult<T>`, depending on the return type of the API endpoint. `AxiosResponse` is the response object from axios, containing the `TApiResult` in its `data` property, as well as other properties like `headers`. The returned type `T` is automatically converted into valid *Model implementations* for you.

API Callers/API States A stateful function for invoking an API endpoint, created with the `$makeCaller` function on an API Client. API Callers provide a wide array of functionality that is useful for working with API endpoints that are utilized by a user interface.

Because they are such an integral part of the overall picture of *coalesce-vue*, they have *their own section below* where they are explained in much greater detail.

3.16.2 API Callers

API Callers (typed with the name `ApiState` in *coalesce-vue*, sometimes also referred to as "loaders" or "invokers") are stateful functions for invoking an API endpoint, created with the `$makeCaller` function on an API Client.

A summary of features:

Endpoint Invocation Each API Caller is itself a function, so it can be invoked to trigger an API request to the server.

State management API Callers contain properties about the last request made, including things like `wasSuccessful`, `isLoading`, `result`, and more.

Concurrency Management Using `setConcurrency(mode)`, you can configure how each individual caller handles what happens when multiple requests are made simultaneously

Argument Binding API Callers can be created so that they have an `args` object that can be bound to, using `.invokeWithArgs()` to make a request using those arguments as the API endpoint's parameters. The API Callers created for the *ViewModel Layer* are all created this way.

Creating and Invoking API Caller

API Callers can be created with the `$makeCaller` method of an API Client. The way in which it was created affects how it is invoked, as the parameters that the caller accepts are defined when it is created.

Tip: During typical development, it is unlikely that you'll need to make a custom API Caller - the ones created for you on the generated *ViewModel Layer* will usually suffice. However, creating your own can allow for some more advanced functionality.

Some examples:

```
// Preamble for all the examples below:
import { PersonApiClient } from '@/api-clients.g';
const client = new PersonApiClient;
```

A caller that takes no additional parameters:

```
const caller = client.$makeCaller(
    "item",
    c => c.namesStartingWith("A")
);

await caller();
console.log(caller.result)
```

A caller that takes custom parameters:

```
const caller = client.$makeCaller(
    methods => methods.namesStartingWith,
    (c, str: string) => c.namesStartingWith(str)
);

await caller("Rob");
console.log(caller.result)
```

A caller that has an args object that can be bound to. This is how the generated API Callers in the *ViewModel Layer* are created:

```
const caller = client.$makeCaller("item",
    // The parameter-based version is always required, even if it won't be used.
    (c, str: string) => c.namesStartingWith(str),
    // A function which creates a blank instance of the args object.
    // All props should be initialized (i.e. not undefined) to work with Vue's␣
↪reactivity.
    () => ({str: null as string | null, }),
    // The function that accepts the args object and uses it:
    (c, args) => c.namesStartingWith(args.str)
);

caller.args.str = "Su";
await caller.invokeWithArgs();
console.log(caller.result)
```

A caller that performs multiple async operations:

```
const deleteFirstNameStartingWith = client.$makeCaller(
    "item",
    async (c, str: string) => {
        const namesResult = await c.namesStartingWith(str)
        return await c.deletePersonByName(namesResult.data.object[0])
    }
);

await caller("Rob");
console.log(caller.result)
```

The first parameter, `resultType`, can either be one of `"item"` or `"list"`, indicating whether the method returns a `ItemResult` or `ListResult` (examples #1 and #3 above). It can also be a function which accepts the set of method metadata for the API Client and which returns the specific method metadata (example #2 above), or it can be a direct reference to a specific method metadata object.

Properties

The following state properties can be found on API Caller instances. These properties are useful for binding to in a user interface to display errors, results, or indicators of progress.

All Callers

`isLoading: boolean` True if there is currently a request pending for the API Caller.

`wasSuccessful: boolean | null` A boolean indicating if the last request made was successful, or null if either no request has been made yet, or if a request has been made but has not yet completed.

`message: string | null` An error message from the last request, if any. Will be set to null upon successful completion of a request.

`hasResult: boolean` True if `result` is non-null. This prop is useful in performance-critical scenarios where checking `result` directly will cause an overabundance of re-renders in high-churn scenarios.

`args: {}` Holds an object for the arguments of the function, and will be used if the caller is invoked with its `invokeWithArgs()` method. Useful for binding the arguments of a caller to inputs in a user interface.

Only exists if the caller was created with the option of being invoked with an args object as described in the sections above.

ItemResult-based Callers

`result: T | null` The principal data returned by the previous request. Will be set to null if the last response received returned no data (e.g. if the response was an error response)

`validationIssues: ValidationIssue[] | null` Any validation issues returned by the previous request. This is never populated automatically by Coalesce, and is therefore is only used if you have written custom code to populate it in your *Behaviors* or *Methods*.

ListResult-based Callers

`result: Array<T> | null` The principal data returned by the previous request. Will be set to null if the last response received returned no data (e.g. if the response was an error response).

`page, pageSize, pageCount, totalCount: number | null` Properties which contain the pagination information returned by the previous request.

Concurrency Mode

API callers have a `setConcurrency(mode: string)` method that allows you to customize how they behave when additional invocations are performed when there is already a request pending. There are four options available, with `"disallow"` being the default:

`"disallow"` The default behavior - simply throws an error for any secondary invocations.

Note: Having `"disallow"` as the default prevents the unexpected behavior that can happen in a number of ways with the other modes:

- For requests that are performing data-mutating actions on the server, all other concurrency modes could lead to an unexpected end state of the data due to requests either being abandoned, cancelled, or potentially happening out-of-order.
- Throwing errors for multiple concurrent requests quickly surfaces issues during development where concurrent requests are not being correctly guarded against in a user interface - e.g. not disabling a "Save" or "Submit" button while the request is pending, which would otherwise lead to double-posts.

`"debounce"` When a secondary invocation is performed, enqueue it after the current pending invocation completes.

If additional invocations are performed while there is already an invocation enqueued and waiting, the already-enqueued invocation is abandoned and replaced by the most recent invocation attempt. The promise of the abandoned invocation will be resolved with `undefined` (it is NOT rejected).

`"cancel"` When a secondary invocation is performed, cancel the current pending invocation.

This completely aborts the request, propagating all the way back to the server where cancellation can be observed with HttpContext.RequestAborted. The promise of the cancelled invocation will be resolved with `undefined` (it is NOT rejected).

`"allow"` When a secondary invocation is performed, always continue normally, sending the request to the server.

The state of the properties on the caller at any time will reflect the most recent response received from the server, which is never guaranteed to correlate with the most recent request made to the server - that is, requests are not guaranteed to complete in the order they were made. In particular, the `isLoading` property will be `false` after the first response comes back, even if the second response has not yet been received.

> **Warning:** For the reasons outlined above, it is generally not recommended to use `"allow"` unless you fully understand the drawbacks. This mode mirrors the legacy behavior of the Knockout stack for Coalesce.

Other Methods

API Callers have a few other methods available as well:

`cancel()` Manually cancel the current request. The promise of the cancelled invocation will be resolved with `undefined` (it is NOT rejected). If using concurrency mode `"allow"`, only the most recent invocation is cancelled.

`onFulfilled((state: TInvoker) => void | Promise<any>)` Add a callback to the caller to be invoked when a success response is received from the server. If a promise is returned, this promise will be awaited and will delay the setting of the `isLoading` prop to `false` until it completes.

`onRejected((state: TInvoker) => void | Promise<any>)` Add a callback to the caller to be invoked when a failure response is received from the server. If a promise is returned, this promise will be awaited and will delay the setting of the `isLoading` prop to `false` until it completes.

`invoke(...args: TArgs)` The invoke function is a reference from the caller to itself - that is, `caller.invoke === caller`. This mirrors the syntax of the Knockout generated method classes.

`invokeWithArgs(args?: {})` If called a parameter, that parameter will be used as the args object. Otherwise, `caller.args` will be used.

Only exists if the caller was created with the option of being invoked with an args object as described in the sections above.

3.17 ViewModel Layer

The ViewModel layer, generated as *viewmodels.g.ts*, exports for each *EF Entity Models* and *Custom DTOs* in your data model both a ViewModel class representing a single instance of the type, and a ListViewModel class that is used to interact with the list functionality in Coalesce. Additionally, each *Service* also has a ViewModel class generated.

These classes provide a wide array of functionality that is useful when interacting with your data model through a user interface. The generated ViewModels are the primary way that Coalesce is used when developing a Vue application.

Contents

3.17.1 ViewModels

The following members can be found on the generated ViewModels, exported from *viewmodels.g.ts* as `*TypeName*ViewModel`.

Model Data Properties

Each ViewModel class implements the corresponding interface from the *Model Layer*, meaning that the ViewModel has a data property for each *Property* on the model. Object-typed properties will be typed as the corresponding generated ViewModel.

Changing the value of a property will automatically flag that property as dirty. See *Auto-save & Dirty Flags* below for information on how property dirty flags are used.

There are a few special behaviors when assigning to different kinds of data properties on View Models as well:

Model Object Properties

- If the object being assigned to the property is not a ViewModel instance, a new instance will be created automatically and used instead of the incoming object.

- If the model property is a reference navigation, the corresponding foreign key property will automatically be set to the primary key of that object. If the incoming value was null, the foreign key will be set to null.
- If deep auto-saves are enabled on the instance being assigned to, auto-save will be spread to the incoming object, and to all other objects reachable from that object.

Model Collection Properties

- When assigning an entire array, any items in the array that are not a ViewModel instance will have an instance created for them.
- The same rule goes for pushing items into the existing array for a model collection - a new ViewModel instance will be created and be used instead of the object(s) being pushed.

Foreign Key Properties If the corresponding navigation property contains an object, and that object's primary key doesn't match the new foreign key value being assigned, the navigation property will be set to null.

Other Data Properties & Functions

`readonly $metadata: ModelType` The metadata object from the *Metadata Layer* layer for the type represented by the ViewModel.

`readonly $stableId: number` An immutable number that is unique among all ViewModel instances, regardless of type.

Useful for uniquely identifying instances with `:key="vm.$stableId"` in a Vue component, especially for instances that lack a primary key.

`$primaryKey: string | number` A getter/setter property that wraps the primary key of the model. Used to interact with the primary key of any ViewModel in a polymorphic way.

`$display(prop?: string | Property): string` Returns a string representation of the object, or one of its properties if specified, suitable for display.

`$addChild(prop: string | ModelCollectionNavigationProperty)` Creates a new instance of an item for the specified child model collection, adds it to that collection, and returns the item.

API Callers & Parameters

`$load: ItemApiState` An *API Caller* for the `/get` endpoint. Accepts an optional `id` argument - if not provided, the ViewModel's `$primaryKey` is used instead. Uses the instance's `$params` object for the *Standard Parameters*.

`$save: ItemApiState` An *API Caller* for the `/save` endpoint. Uses the instance's `$params` object for the *Standard Parameters*.

This caller is used for both manually-triggered saves in custom code and for auto-saves. If the *Rules/Validation* report any errors when the caller is invoked, an error will be thrown.

When a save creates a new record and a new primary key is returned from the server, any entities attached to the current ViewModel via a collection navigation property will have their foreign keys set to the new primary key.

This behavior, combined with the usage of deep auto-saves, allows for complex object graphs to be constructed even before any model in the graph has been created.

Saving behavior can be further customized with `$loadResponseFromSaves` and `$saveMode`, listed below.

`$loadResponseFromSaves`: `boolean` Default `true` - controls if a ViewModel will be loaded with the data from the model returned by the `/save` endpoint when saved with the `$save` API caller. There is seldom any reason to disable this.

`$saveMode`: `"surgical" | "whole"` Configures which properties of the model are sent to the server during a save.

`"surgical"` (default) By default, only dirty properties (and always the primary key) are sent to the server when performing a save.

This improves the handling of concurrent changes being made by multiple users against different fields of the same entity at the same time - specifically, it prevents a user with a stale value of some field X from overwriting a more recent value of X in the database when the user is only making changes to some other property Y and has no intention of changing X.

Save mode `"surgical"` doesn't help when multiple users are editing field X at the same time - if such a scenario is applicable to your application, you must implement more advanced handling of concurrency conflicts.

> **Warning:** Surgical saves require DTOs on the server that are capable of determining which of their properties have been set by the model binder, as surgical saves are sent from the client by entirely omitting properties from the `x-www-form-urlencoded` body that is sent to the server.
>
> The *Generated C# DTOs* implement the necessary logic for this; however, any *Custom DTOs* you have written are unlikely to be implementing the same behavior. For *Custom DTOs*, either implement the same pattern that can be seen in the *Generated C# DTOs*, or use save mode `"whole"` instead.

`"whole"` All serializable properties of the object are sent back to the server with every save.

`$delete`: `ItemApiState` An *API Caller* for the `/delete` endpoint. Uses the instance's `$params` object for the *Standard Parameters*.

If the object was loaded as a child of a collection, it will be removed from that collection upon being deleted. Note that ViewModels currently only support tracking of a single parent collection, so if an object is programmatically added to additional collections, it will only be removed from one of them upon delete.

`$params`: `DataSourceParameters` An object containing the *Standard Parameters* to be used for the `$load`, `$save`, and `$delete` API callers.

`$dataSource`: `DataSource` Getter/setter wrapper around `$params.dataSource`. Takes an instance of a *Data Source* class *generated in the Model Layer*.

`$includes`: `string | null` Getter/setter wrapper around `$params.includes`. See *Includes String* for more information.

Auto-save & Dirty Flags

`$startAutosave(vue: Vue, options: AutoSaveOptions<this> = {})` Starts auto-saving of the instance when its savable data properties become dirty. Saves are performed with the `$save` *API Caller* (documented below) and will not be performed if the ViewModel has any validation errors - see *Rules/Validation* below.

Requires a reference to a Vue instance in order to manage lifetime (auto-save hooks will be destroyed when the Vue component provided is destroyed). Options are as follows:

```
{
    /** Time, in milliseconds, to debounce saves for.  */
    wait?: number;

    /** If true, auto-saving will also be enabled for all view models that are
        reachable from the navigation properties & collections of the current␣
↪view model. */
    deep?: boolean;

    /** A function that will be called before autosaving that can return false to␣
↪prevent a save.
        Only allowed if not using deep auto-saves.
    */
    predicate?: (viewModel: TThis) => boolean;
}
```

`$stopAutosave()` Turns off auto-saving of the instance. Does not recursively disable auto-saves on related instances if `deep` was used when auto-save was enabled.

`$getPropDirty(propName: string): boolean` Returns true if the given property is flagged as dirty.

`$setPropDirty(propName: string, dirty: boolean = true, triggerAutosave = true)` Manually set the dirty flag of the given property to the desired state. This seldom needs to be done explicitly, as mutating a property will automatically flag it as dirty.

If `dirty` is true and `triggerAutosave` is false, auto-save (if enabled) will not be immediately triggered for this specific flag change. Note that a future change to any other property's dirty flag will still trigger a save of all dirty properties.

`$isDirty: boolean` Getter/setter that summarizes the model's property-level dirty flags. Returns true if any properties are dirty.

When set to false, all property dirty flags are cleared. When set to true, all properties are marked as dirty.

`$loadCleanData(source: {} | TModel)` Loads data from the provided model into the current ViewModel, and then clears all dirty flags.

Data is loaded recursively into all related ViewModel instances, preserving existing instances whose primary keys match the incoming data.

If auto-save is enabled, only non-dirty properties are updated. This prevents user input that is pending a save from being overwritten by the response from an auto-save `/save` request.

`$loadDirtyData(source: {} | TModel)` Same as `$loadCleanData`, but does not clear any existing dirty flags, nor does it clear any dirty flags that will be set while mutating the data properties of any ViewModel instance that gets loaded.

`constructor(initialDirtyData?: {} | TModel | null)` (Constructor) Create a new instance of the ViewModel, loading the given initial data with `$loadDirtyData()` if provided.

Rules/Validation

`$addRule(prop: string | Property, identifier: string, rule: (val: any) => true | string`
Add a custom validation rule to the ViewModel for the specified property. `identifier` should be a short, unique slug that describes the rule; it is not displayed in the UI, but is used if you wish to later remove the rule with `$removeRule()`.

The function you provide should take a single argument that contains the current value of the property, and should either return `true` to indicate that the validation rule has succeeded, or a string that will be displayed as an error message to the user.

Any failing validation rules on a ViewModel will prevent that ViewModel's `$save` caller from being invoked.

`$removeRule(prop: string | Property, identifier: string)` Remove a validation rule from the ViewModel for the specified property with the specified identifier.

This can be used to remove from the ViewModel instance either a rule that was provided by the generated *Metadata Layer*, or a custom rule that was added by `$addRule`. Reference your generated metadata file *metadata.g.ts* to see any generated rules and the identifiers they use.

`$getRules(prop: string | Property)` Returns an array of active rule functions for the specified property, or `undefined` if the property has no active validation rules.

`$getErrors(prop?: string | Property): Generator<string>` Returns a generator that provides all error messages for either a specific property (if provided) or the entire model (if no prop argument is provided).

Tip: You can obtain an array from a generator with `Array.from(vm.$getErrors())` or `[...vm.$getErrors()]`

`readonly $hasError: boolean` Indicates if any properties have validation errors.

Generated Members

Method Callers For each of the instance *Methods* of the type, an *API Caller* will be generated.

`addTo*()` For each *collection navigation property*, a method is generated that will create a new instance of the ViewModel for the collected type, add it to the collection, and then return the new object.

Many-to-many helper collections For each *collection navigation property* annotated with *[ManyToMany]*, a getter-only property is generated that returns a collection of the object on the far side of the many-to-many relationship. Nulls are filtered from this collection.

3.17.2 ListViewModels

The following members can be found on the generated ListViewModels, exported from *viewmodels.g.ts* as `*TypeName*ListViewModel`.

Data Properties

`$items` Collection holding the results of the last successful invocation of the `$load` *API Caller*.

Parameters & API Callers

`$params: DataSourceParameters` An object containing the *Standard Parameters* to be used for the `$load` and `$count` API callers.

`$load: ListApiState` An *API Caller* for the `/list` endpoint. Uses the instance's `$params` object for the *Standard Parameters*.

Results are available in the `$items` property. The `result` property of the `$load` API Caller contains the raw results and is not recommended for use in general development - `$items` should always be preferred.

`$count: ItemApiState` An *API Caller* for the `/count` endpoint. Uses the instance's `$params` object for the *Standard Parameters*.

The result is available in `$count.result` - this API Caller does not interact with other properties on the ListViewModel like `$pageSize` or `$pageCount`.

`readonly $hasPreviousPage: boolean`, `readonly $hasNextPage: boolean` Properties which indicate if `$page` can be decremented or incremented, respectively. `$pageCount` and `$page` are used to make this determination.

`$previousPage()`, `$nextPage()` Methods that will decrement or increment `$page`, respectively. Each does nothing if there is no previous or next page as returned by `$hasPreviousPage` and `$hasNextPage`.

`$page: number` Getter/setter wrapper for `$params.page`. Controls the page that will be requested on the next invocation of `$load`.

`$pageSize: number` Getter/setter wrapper for `$params.pageSize`. Controls the page that will be requested on the next invocation of `$load`.

`readonly $pageCount: number` Shorthand for `$load.pageCount` - returns the page count reported by the last successful invocation of `$load`.

Auto-Load

`$startAutoLoad(vue: Vue, options: AutoLoadOptions<this> = {})` Starts auto-loading of the list as changes to its parameters occur. Loads are performed with the `$load` *API Caller*.

Requires a reference to a Vue instance in order to manage lifetime (auto-load hooks will be destroyed when the Vue component provided is destroyed). Options are as follows:

```
{
    /** Time, in milliseconds, to debounce loads for.  */
    wait?: number;

    /** A function that will be called before loading that can return false to␣
↪prevent a load.
    */
    predicate?: (viewModel: TThis) => boolean;
}
```

`$stopAutoLoad()` Manually turns off auto-loading of the instance.

Generated Members

Method Callers For each of the static *Methods* on the type, an *API Caller* will be created.

3.17.3 Service ViewModels

The following members can be found on the generated Service ViewModels, exported from *viewmodels.g.ts* as `*ServiceName*ViewModel`.

Generated Members

Method Callers For each method of the *Service*, an *API Caller* will be created.

3.18 Vuetify Components

The Vue stack for Coalesce provides *a set of components* based on Vuetify, packaged up in an NPM package coalesce-vue-vuetify. These components are driven primarily by the *Metadata Layer*, and include both low level input and display components like *c-input* and *c-display* that are highly reusable in the custom pages you'll build in your application, as well as high-level components like *c-admin-table-page* and *c-admin-editor-page* that constitute entire pages.

Contents

3.18.1 c-admin-display

Behaves the same as *c-display*, except any collection navigation properties will be rendered as links to an admin list page, and any models will be rendered as a link to an admin item page.

Links for collections are resolved from vue-router with a route name of `coalesce-admin-list`, a `type` route param containing the name of the collection's type, and a query parameter `filter.<foreign key name>` with a value of the primary key of the owner of the collection. This route is expected to resolve to a *c-admin-table-page*, which is setup by default by the template outlined in *Getting Started with Vue*.

Links for single models are resolved from vue-router with a route name of `coalesce-admin-item`, a `type` route param containing the name of the model's type, and a `id` route param containing the object's primary key. This route is expected to resolve to a *c-admin-editor-page*, which is setup by default by the template outlined in *Getting Started with Vue*.

Contents

Examples

```
<!-- Renders regularly as text: -->
<c-admin-display :model="person" for="firstName" />

<!-- Renders as a link to an item: -->
<c-admin-display :model="person" for="company" />

<!-- Renders as a link to a list: -->
<c-admin-display :model="person" for="casesAssigned" />
```

Props

Same as *c-display*.

Slots

Same as *c-display*.

3.18.2 c-admin-editor

An editor for a single *ViewModel* instance. Provides a *c-input* for each property of the model.

Does not automatically enable *auto-save* - if desired, this must be enabled by the implementor of this component.

Contents

Examples

```
<c-admin-editor :model="person" />
```

Props

`model: ViewModel | ListViewModel` The *ViewModel* to render an editor for.

3.18.3 c-admin-editor-page

A page for a creating/editing single *ViewModel* instance. Provides a *c-admin-editor* and a *c-admin-methods* for the instance. Designed to be routed to directly with vue-router.

Contents

Examples

```
// router.ts or main.ts

import { CAdminEditorPage } from 'coalesce-vue-vuetify';

const router = new Router({
    // ...
    routes: [
        // ... other routes
        {
            path: '/admin/:type/edit/:id?',
            name: 'coalesce-admin-item',
            component: CAdminEditorPage,
            props: true,
        },
    ]
})
```

Props

`type:` `string` The PascalCase name of the type to be created/edited.

`id?:` `number | string` The primary key of the item being edited. If null or not provided, the page will be creating a new instance of the provided `type` instead.

3.18.4 c-admin-method

Provides an interface for invoking a *method* and rendering its result, designed to be use in an admin page.

For each parameter of a method, a *c-input* will be rendered to accept the input of that parameter. A button is provided to trigger an invocation of the method, progress and errors are rendered with a *c-loader-status*, and results are rendered with *c-display*.

Contents

Examples

```
<c-admin-method :model="person" for="setTitle" auto-reload-model />
```

Props

for: string | Method A metadata specifier for the method. One of:

- A string with the name of the method belonging to `model`.
- A direct reference to a method's metadata object.
- A string in dot-notation that starts with a type name and ending with a method name.

model: ViewModel | ListViewModel An *ViewModel* or *ListViewModel* owning the method and *API Caller* that was specified by the `for` prop.

autoReloadModel?: boolean = false True if the `model` should have its `$load` invoked after a successful invocation of the method.

3.18.5 c-admin-methods

Renders in a Vuetify v-expansion-panels a *c-admin-method* for each method on a *ViewModel* or *ListViewModel*.

Contents

Examples

```
<c-admin-methods :model="person" auto-reload-model />
```

```
<c-admin-methods :model="personList" auto-reload-model />
```

Props

model: ViewModel | ListViewModel An *ViewModel* or *ListViewModel* whose methods should each render as a *c-admin-method*.

autoReloadModel?: boolean = false True if the `model` should have its `$load` invoked after a successful invocation of any method.

3.18.6 c-admin-table

An full-featured table for a *ListViewModel*, including a *c-admin-table-toolbar*, *c-table*, and *c-list-pagination*.

The table can be in read mode (default), or toggled into edit mode with the button provided by the *c-admin-table-toolbar*. When placed into edit mode, *auto-save* is enabled.

Contents

Examples

```
<c-admin-table :list="personList" />
```

Props

`list:` `ListViewModel` The *ListViewModel* to render a table for.

`pageSizes?:` `number[]` An optional list of available page sizes to offer through the *c-list-pagination*'s *c-list-page-size* component. Defaults to `[10, 25, 100]`.

`queryBind?:` `boolean` If true, the *Data Source Standard Parameters* of the provided *ListViewModel* will be read from and written to the window's query string. The "Editable" state of the table will also be bound to the query string.

3.18.7 c-admin-table-page

A full-featured page for interacting with a *ListViewModel*. Provides a *c-admin-table* and a *c-admin-methods* for the list. Designed to be routed to directly with vue-router.

Contents

Examples

```
// router.ts or main.ts

import { CAdminTablePage } from 'coalesce-vue-vuetify';

const router = new Router({
    // ...
    routes: [
        // ... other routes
        {
            path: '/admin/:type',
            name: 'coalesce-admin-list',
            component: CAdminTablePage,
            props: true,
        },
    ]
})
```

Props

type: **string** The PascalCase name of the type to be listed.

list?: **ListViewModel** An optional *ListViewModel* that will be used if provided instead of the one the component will otherwise create automatically.

3.18.8 c-admin-table-toolbar

A full-featured toolbar for a *ListViewModel* designed to be used on an admin page, including "Create" and "Reload" buttons, a *c-list-range-display*, a *c-list-page*, a search field, *c-list-filters*, and a *c-list-page-size*.

Contents

Examples

```
<c-admin-table-toolbar :list="personList" />
```

```
<c-admin-table-toolbar :list="personList" color="pink" :editable.sync="isEditable" />
```

Props

list: **ListViewModel** The *ListViewModel* to render the toolbar for.

color: **string = "primary"** The color of the toolbar.

editable?: **boolean** If provided, adds a button to toggle the value of this prop. Should be bound with the `.sync` modifier.

3.18.9 c-datetime-picker

A general, all-purpose date/time input component that can be used either with *models* and *metadata* or as a standalone component using only `v-model`.

Contents

Examples

```
<c-datetime-picker :model="person" for="birthDate" />

<c-datetime-picker v-model="standaloneDate" />

<c-datetime-picker
    v-model="standaloneTime"
    date-kind="time"
    date-format="h:mm a"
/>
```

Props

for?: **string | DateProperty | DateValue** A metadata specifier for the value being bound. One of:

- A string with the name of the value belonging to `model`.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name.

model?: **Model | DataSource** An object owning the value that was specified by the `for` prop. If provided, the input will be bound to the corresponding property on the `model` object.

value?: **Date** If binding the component with `v-model`, accepts the `value` part of `v-model`.

dateKind?: **"date" | "time" | "datetime" = "datetime"** Whether the date is only a date, only a time, or contains significant date *and* time information.

If the component was bound with metadata using the `for` prop, this will default to the kind specified by *[DateType]*.

dateFormat?: **string** The format of the date that will be rendered in the component's text field, and the format that will be attempted first when parsing user input in the text field.

Defaults to:

- `M/d/yyyy h:mm a` if `dateKind == 'datetime'`,
- `M/d/yyyy` if `dateKind == 'date'`, or
- `h:mm a` if `dateKind == 'time'`.

Important: When parsing a user's text input into the text field, c-datetime-picker will first attempt to parse it with the format specified by `dateFormat`, or the default as described above if not explicitly specified.

If this fails, the date will be parsed with the Date constructor, but only if the `dateKind` is `datetime` or `date`. This works fairly well on all modern browsers, but can still occasionally have issues. c-datetime-picker tries its best to filter out bad parses from the Date constructor, like dates with a year earlier than 1000.

---

readonly?: **boolean** True if the component should be read-only.

disabled?: **boolean** True if the component should be disabled.

Slots

3.18.10 c-display

A general-purpose component for displaying any *Value* by rendering the value to a string with the *display functions from the Models Layer*. For string and number *values*, usage of this component is largely superfluous. For all other value types including dates, booleans, enums, objects, and collections, it is very handy.

Contents

Examples

Typical usage, providing an object and a property on that object:

```
<c-display :model="person" for="gender" />
```

Customizing date formatting:

```
<c-display :model="person" for="birthDate" format="M/d/yyyy" />
```

A contrived example of using c-display to render the result of an *API Caller*:

```
<c-display
    :value="person.setFirstName.result"
    :for="person.$metadata.methods.setFirstName.return"
    element="div"
/>
```

Props

for: string | Property | Value A metadata specifier for the value being bound. Either a direct reference to the metadata object, or a string with the name of the value belonging to `model`, or a string in dot-notation that starts with a type name.

model?: Model | DataSource An object owning the value that was specified by the `for` prop.

format: DisplayOptions["format"] Shorthand for `:options="{ format: format }"`, allowing for specification of the format to be used when displaying dates.

See *DisplayOptions* for details on the options available for `format`.

format: DisplayOptions Specify options for formatting some kinds of values, including dates. See *DisplayOptions* for details.

value: any Can be provided the value to be displayed in conjunction with the `for` prop, as an alternative to the `model` prop.

This is an uncommon scenario - it is generally easier to use the `for`/`model` props together.

Slots

`default` Used to display fallback content if the value being displayed is either `null` or `""` (empty string).

3.18.11 c-input

A general-purpose input component for most *Values*. c-input does not have much functionality of its own - instead, it delegates to the right kind of component based on the type of value to which it is bound. This includes both other Coalesce Vuetify Components as well as direct usages of some Vuetify components.

All attributes are passed through to the delegated-to component, allowing for full customization of the underlying Vuetify component.

A summary of the components delegated to, by type:

- string, number: v-text-field, or v-textarea if flag attribute `textarea` is provided to `c-input`.
- boolean: v-switch, or v-checkbox if flag attribute `checkbox` is provided to `c-input`.
- enum: v-select
- file: v-file-input
- date: *c-datetime-picker*
- model: *c-select*
- *[ManyToMany]* collection: *c-select-many-to-many*
- Non-object collection: *c-select-values*

Any other unsupported type will simply be displayed with *c-display*, unless a default slot is provided - in that case, the default slot will be rendered instead.

When bound to a *ViewModel*, the *validation rules* for the bound property will be obtained from the *ViewModel* and passed to Vuetify's `rules` prop.

Contents

Examples

Typical usage, providing an object and a property on that object:

```
<c-input :model="person" for="firstName" />
```

Customizing the Vuetify component used:

```
<c-input :model="comment" for="content" textarea solo />
```

Binding to *API Caller* args objects:

```
<c-input
    :model="person.setFirstName"
    for="newName" />
```

Or, using a more verbose syntax:

```
<c-input
    :model="person.setFirstName.args"
    for="Person.methods.setFirstName.newName" />
```

Binding to *Data Source Parameters*:

```
<c-input :model="personList.$dataSource" for="startsWith" />
```

Usage with `v-model` (this scenario is atypical - the model/for pair of props are used in almost all scenarios):

```
<c-input v-model="person.firstName" for="Person.firstName" />
```

Props

for?: string | Property | Value A metadata specifier for the value being bound. One of:

- A string with the name of the value belonging to `model`.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name.

model?: Model | DataSource An object owning the value that was specified by the `for` prop. If provided, the input will be bound to the corresponding property on the `model` object.

value?: any If binding the component with `v-model`, accepts the `value` part of `v-model`.

Slots

default Used to display fallback content if c-input does not support the type of the value being bound. Generally this does not need to be used, as you should avoid creating c-input components for unsupported types in the first place.

3.18.12 c-list-filters

A component that provides an interface for modifying the `filters` prop of a *ListViewModel*'s *parameters*.

Contents

Examples

```
<c-list-filters :list="list" />
```

Props

list: **ListViewModel** The *ListViewModel* whose filters will be editable.

3.18.13 c-list-page

A component that provides previous/next buttons and a text field for modifying the `page` *parameter* prop of a *ListViewModel*.

Contents

- *Examples*
- *Props*

Examples

```
<c-list-page :list="list" />
```

Props

list: **ListViewModel** The *ListViewModel* whose current page will be changeable with the component.

3.18.14 c-list-page-size

A component that provides an dropdown for modifying the `pageSize` *parameter* prop of a *ListViewModel*.

Contents

- *Examples*
- *Props*

Examples

```
<c-list-page-size :list="list" />
```

Props

list: **ListViewModel** The *ListViewModel* whose pagination will be editable.

pageSizes?: **number[]** An optional list of available page sizes to offer through *c-list-page-size*. Defaults to `[10, 25, 100]`.

3.18.15 c-list-pagination

A component that provides an interface for modifying the pagination *parameters* of a *ListViewModel*.

This is a composite of *c-list-page-size*, *c-list-range-display*, and *c-list-page*, arranged horizontally. It is designed to be used above or below a table (e.g. *c-table*).

Contents

Examples

```
<c-list-pagination :list="list" />
```

Props

`list:` `ListViewModel` The *ListViewModel* whose pagination will be editable.

`pageSizes?:` `number[]` An optional list of available page sizes to offer through *c-list-page-size*. Defaults to `[10, 25, 100]`.

3.18.16 c-list-range-display

Displays pagination information about the current `$items` of a *ListViewModel* in the format `<start index> - <end index> of <total count>`.

Uses the pagination information returned from the last successful `$load` call, not the current `$params` of the *ListViewModel*.

Contents

Examples

```
<c-list-range-display :list="list" />
```

Props

`list:` `ListViewModel` The *ListViewModel* to display pagination information for.

3.18.17 c-loader-status

A component for displaying progress and error information for one or more *API Callers*.

Tip: It is highly recommended that all *API Callers* utilized by your application that don't have any other kind of error handling should be represented by a *c-loader-status* so that users can be aware of any errors that occur.

Progress is indicated with a Vuetify v-progress-linear component, and errors are displayed in a v-alert. Transitions are applied to smoothly fade between the different states the the caller can be in.

Note: This component uses the legacy term "loader" to refer to *API Callers*. A new `c-caller-status` component may be coming in the future with a few usability improvements - if that happens, *c-loader-status* will be preserved for backwards compatibility.

Contents

Examples

Wrap contents of a details/edit page:

```
<h1>Person Details</h1>
<c-loader-status
    :loaders="{
        'no-initial-content no-error-content': [person.$load],
        '': [person.$save, person.$delete],
    }"
    #default
>
    First Name: {{ person.firstName }}
    Last Name: {{ person.lastName }}
    Employer: {{ person.company.name }}
</c-loader-status>
```

Use `c-loader-status` to render a progress bar and any error messages, but don't use it to control content:

```
<c-loader-status :loaders="{'': [list.$load]}" />
```

Wrap a save/submit button:

```
<c-loader-status
    :loaders="{ 'no-loading-content': [person.$save] }"
>
    <button> Save </button>
</c-loader-status>
```

Hides the table before the first load has completed, or if loading the list encountered an error. Don't show the progress bar after we've already loaded the list for the first time (useful for loads that occur without user interaction, e.g. `setInterval`).:

```
<c-loader-status
    :loaders="{
        'no-secondary-progress no-initial-content no-error-content': [list.$load]
    }"
    #default
>
    <table>
        <tr v-for="item in list.$items"> ... </tr>
    </table>
</c-loader-status>
```

Props

`loaders: { [flags: string]: ApiCaller | ApiCaller[] }` A dictionary object with entries mapping zero or more flags to one or more *API Callers*. Multiple entries of flags/caller pairs may be specified in the dictionary to give different behavior to different API callers.

The available flags are as follows. All flags may be prefixed with `no-` to set the flag to `false` instead of `true`. Multiple flags may be specified at once by delimiting them with spaces.

- `loading-content` - default `true` — Controls whether the default slot is rendered while any API caller is loading (i.e. when `caller.isLoading === true`).
- `error-content` - default `true` — Controls whether the default slot is rendered while any API Caller is in an error state (i.e. when `caller.wasSuccessful === false`).
- `initial-content` - default `true` — Controls whether the default slot is rendered while any API Caller has yet to receive a response for the first time (i.e. when `caller.wasSuccessful === null`).
- `initial-progress` - default `true` — Controls whether the progress indicator is shown when an API Caller is loading for the very first time (i.e. when `caller.wasSuccessful === null`).
- `secondary-progress` - default `true` — Controls whether the progress indicator is shown when an API Caller is loading any time after its first invocation (i.e. when `caller.wasSuccessful !== null`).

`progressPlaceholder: boolean = true` Specify if space should be reserved for the progress indicator. If set to false, the content in the default slot may jump up and down slightly as the progress indicator shows and hides.

`height: number = 10` Specifies the height in pixels of the v-progress-linear used to indicate progress.

Slots

default Accepts the content whose visibility is controlled by the state of the supplied *API Callers*. It will be shown or hidden according to the flags defined for each caller.

Important: Define the default slot as a scoped slot (e.g. with `#default` or `v-slot:default` on the `c-loader-status`) to prevent the VNode tree from being created when the content should be hidden. This improves performances and helps avoid null reference errors that can be caused when trying to render objects that haven't been loaded yet.

3.18.18 c-select

A dropdown component that allows for selecting values fetched from the generated `/list` API endpoints.

Used both for selecting values for foreign key and navigation properties, and for selecting arbitrary objects or primary keys independent of a parent or owning object.

Contents

Examples

Binding to a navigation property or foreign key of a model:

```
<c-select :model="person" for="company" />
<!-- OR: -->
<c-select :model="person" for="companyId" />
```

Binding an arbitrary primary key value or an arbitrary object:

```
<!-- Binding a key: -->
<c-select for="Person" :key-value.sync="selectedPersonId" />

<!-- Binding an object: -->
<c-select for="Person" :object-value.sync="selectedPerson" />
<c-select for="Person" v-model="selectedPerson" />
```

Examples of other props:

```
<c-select
  for="Person"
  v-model="selectedPerson"
  :clearable="false"
  preselect-first
  :params="{ pageSize: 42, filter: { isActive: true } }"
  :create="createMethods"
  dense
```

(continues on next page)

(continued from previous page)

```
  outlined
  color="pink"
/>
<!-- `createMethods` is defined in the docs of `create` below -->
```

Props

`for: string | ForeignKeyProperty | ModelReferenceNavigationProperty | ModelType`

A metadata specifier for the value being bound. One of:

- The name of a foreign key or reference navigation property belonging to `model`.
- The name of a model type.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name that resolves to a foreign key or reference navigation property.

Tip: When binding by a key value, if the corresponding object cannot be found (e.g. there is no navigation property, or the navigation property is null), c-select will automatically attempt to load the object from the server so it can be displayed in the UI.

`model?: Model` An object owning the value that was specified by the `for` prop. If provided, the input will be bound to the corresponding property on the `model` object.

If `for` specifies a foreign key or reference navigation property, both the foreign key and the navigation property of the `model` will be updated when the selected value is changed.

`value?: any` When binding the component with `v-model`, accepts the `value` part of `v-model`. If `for` was specified as a foreign key, this will expect a key; likewise, if `for` was specified as a type or as a navigation property, this will expect an object.

`keyValue?: any` When bound with `:key-value.sync="keyValue"`, allows binding the primary key of the selected object explicitly.

`objectValue?: any` When bound with `:object-value.sync="objectValue"`, allows binding the selected object explicitly.

`clearable?: boolean` Whether the selection can be cleared or not, emitting `null` as the input value.

If not specified and the component is bound to a foreign key or reference navigation property, defaults to whether or not the foreign key has a `required` validation rule defined in its *Metadata*.

`preselectFirst?: boolean = false` If true, then when the first list results for the component are received by the client just after the component is created, c-select will emit the first item in the list as the selected value.

`preselectSingle?: boolean = false` If true, then when the first list results for the component are received by the client just after the component is created, if the results contained exactly one item, c-select will emit that only item as the selected value.

`params?: ListParameters` An optional set of *Data Source Standard Parameters* to pass to API calls made to the server.

`create?`

A object containing a pair of methods that allowing users to create new items from directly within the c-select if a matching object is not found.

The object must contain the following two methods. You should define these in your component's `script` section - don't try to define them inline in your component.

`getLabel: (search: string, items: TModel[]) => string | false,`

A function that will be called with the user's current search term, as well as the collection of currently loaded items being presented to the user as valid selection options.

It should return either a `string` that will be presented to the user as an option in the dropdown that can be clicked to invoke the `getItem` function below, or it should return `false` to prevent such an option from being shown to the user.

`getItem: (search: string, label: string) => Promise<TModel>`

A function that will be invoked when the user clicks the option in the dropdown list described by `getLabel`. It will be given the user's current search term as well as the value of the label returned from `getLabel` as parameters. It must perform the necessary operations to create the new object on the server and then return a reference to that object.

For example:

```
createMethods = {
  getLabel(search: string, items: Person[]) {
    const searchLower = search.toLowerCase();
    if (items.some(a => a.name?.toLowerCase().indexOf(searchLower) == 0)) {
      return false;
    }
    return search;
  },
  async getItem(search: string, label: string) {
    const client = new PersonApiClient();
    return (await client.addPersonByName(label)).data.object!;
  }
}
```

Slots

`#item="{ item }"` - Slot used to customize the text of both items inside the list, as well as the text of selected items. By default, items are rendered with *c-display*. Slot is passed a single parameter `item` containing a *model instance*.

3.18.19 c-select-many-to-many

A multi-select dropdown component that allows for selecting values fetched from the generated `/list` API endpoints for collection navigation properties that were annotated with *[ManyToMany]*.

Tip: It is unlikely that you'll ever need to use this component directly - it is highly recommended that you use *c-input* instead and let it delegate to *c-select-many-to-many* for you.

Contents

Examples

```
<c-select-many-to-many :model="case" for="caseProducts" />
```

```
<c-select-many-to-many
    :model="case"
    for="caseProducts"
    dense
    outlined

/>
```

```
<c-select-many-to-many
    v-model="case.caseProducts"
    for="Case.caseProducts"
/>
```

Props

`for:` **`string | Property | Value`** A metadata specifier for the value being bound. One of:

- A string with the name of the value belonging to `model`.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name.

> **Important:** c-select-many-to-many expects metadata for the "real" collection collection navigation property on a model. If you provide it the string you passed to *[ManyToMany]*, an error wil be thrown.

`model?:` **`Model`** An object owning the value that was specified by the `for` prop. If provided, the input will be bound to the corresponding property on the `model` object.

`value:` **`any`** If binding the component with `v-model`, accepts the `value` part of `v-model`.

`params?:` **`ListParameters`** An optional set of *Data Source Standard Parameters* to pass to API calls made to the server.

3.18.20 c-select-string-value

A dropdown component that will present a list of suggested string values from a custom API endpoint. Allows users to input values that aren't provided by the endpoint.

Effectively, this is a server-driven autocomplete list.

Contents

Examples

```
<c-select-string-value
    :model="person"
    for="jobTitle"
    method="getSuggestedJobTitles"
/>
```

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

    public string JobTitle { get; set; }

    [Coalesce]
    public static Task<ICollection<string>> GetSuggestedJobTitles(AppDbContext db,␣
↪string search)
    {
        return db.People
            .Select(p => p.JobTitle)
            .Distinct()
            .Where(t => t.StartsWith(search))
            .OrderBy(t => t)
            .Take(100)
            .ToListAsync()
    }
}
```

Props

for: string | Property | Value A metadata specifier for the value being bound. One of:

- A string with the name of the value belonging to `model`.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name.

model: Model An object owning the value that was specified by the `for` prop. If provided, the input will be bound to the corresponding property on the `model` object.

method: string The camel-cased name of the *Custom Method* to invoke to get the list of valid values. Will be passed a single string parameter `search`. Must be a static method on the type of the provided `model` object that returns a collection of strings.

params?: DataSourceParameters An optional set of *Data Source Standard Parameters* to pass to API calls made to the server.

listWhenEmpty?: boolean = false True if the method should be invoked and the list displayed when the entered search term is blank.

3.18.21 c-select-values

A multi-select input component for collections of non-object values (primarily strings and numbers).

Tip: It is unlikely that you'll ever need to use this component directly - it is highly recommended that you use *c-input* instead and let it delegate to *c-select-values* for you.

Contents

Examples

```
<c-select-values
    :model="post.setTags.args"
    for="Post.methods.setTags.params.tagNames"
/>
```

Props

`for:` `string | CollectionProperty | CollectionValue` A metadata specifier for the value being bound. One of:

- A string with the name of the value belonging to `model`.
- A direct reference to a metadata object.
- A string in dot-notation that starts with a type name.

`model?:` `Model` An object owning the value that was specified by the `for` prop.

`value:` `any` If binding the component with `v-model`, accepts the `value` part of `v-model`.

3.18.22 c-table

A table component for displaying the contents of a *ListViewModel*. Also supports modifying the list's *sort parameters* by clicking on column headers. Pairs well with a *c-list-pagination*.

Contents

Examples

A simple table, rendering the items of a *ListViewModel*:

```
<c-table :list="list" />
```

A more complex example using more of the available options:

```
<c-table
  :list="list"
  :props="['firstName', 'lastName']"
  :extra-headers="['Actions']"
>
  <template #item.append="{item}">
    <td>
      <v-btn
        title="Edit"
        text icon
        :to="{name: 'edit-person', params: { id: item.$primaryKey }}"
      >
        <i class="fa fa-edit"></i>
      </v-btn>
    </td>
  </template>
</c-table>
```

Props

list: ListViewModel The *ListViewModel* to display pagination information for.

props?: string[] If provided, specifies which properties, and their ordering, should be given a column in the table.

If not provided, all non-key columns that aren't annotated with *[Hidden(HiddenAttribute.Areas.List)]* are given a column.

extraHeaders?: string[] The text contents of one or more extra `th` elements to render in the table. Should be used in conjunction with the `item.append` slot.

editable: boolean = false If true, properties in each table cell will be rendered with *c-input*. Non-editable properties will be rendered in accordance with the value of the `admin` prop.

admin: boolean = false If true, properties in each table cell will be rendered with *c-admin-display* instead of *c-display*.

Slots

item.append A slot rendered after the `td` elements on each row that render the properties of each item in the table. Should be provided zero or more additional `td` elements. The number should match the number of additional headers provided to the `extraHeaders` prop.

3.18.23 Setup

Tip: The template described in *Getting Started with Vue* already includes all the necessary setup. You can skip this section if you started from the template.

First, ensure that NPM package coalesce-vue-vuetify is installed in your project.

Then, in your Vue application's `main.ts` file, you need to add the `coalesce-vue-vuetify` plugin to your application, like so:

```
import $metadata from '@/metadata.g';
// viewmodels.g has side-effects - it populates the global lookup on ViewModel and␣
↪ListViewModel.
// It must be imported for c-admin-editor-page and c-admin-table-page to work␣
↪correctly.
import '@/viewmodels.g';

import CoalesceVuetify from 'coalesce-vue-vuetify';
Vue.use(CoalesceVuetify, { metadata: $metadata, });
```

Also ensure that you have setup Vuetify correctly in your application as described in Vuetify's documentation.

Note: An important note if you're using Vuetify's A-la-carte builds:

coalesce-vue-vuetify expects that the Vuetify components that *c-input* can delegate directly to have been registered globally. Currently, vuetify-loader is not capable of picking up these particular references.

To make things work correctly, do the following when you `Vue.use(Vuetify)`:

```
import Vuetify, { VTextField, VTextarea, VCheckbox, VSwitch, VSelect, VFileInput }␣
↪from 'vuetify/lib';

Vue.use(Vuetify, {
    components: { VTextField, VTextarea, VCheckbox, VSwitch, VSelect, VFileInput },
});
```

You're now ready to start using the components that coalesce-vue-vuetify provides! See the list below for a summary of each component and links to dive deeper into each component.

3.18.24 Display Components

c-display

A general-purpose component for displaying any *Value* by rendering the value to a string with the *display functions from the Models Layer*. For string and number *values*, usage of this component is largely superfluous. For all other value types including dates, booleans, enums, objects, and collections, it is very handy.

Full Documentation: *c-display*.

c-loader-status

A component for displaying progress and error information for one or more *API Callers*.

Tip: It is highly recommended that all *API Callers* utilized by your application that don't have any other kind of error handling should be represented by a *c-loader-status* so that users can be aware of any errors that occur.

Full Documentation: *c-loader-status*.

c-list-range-display

Displays pagination information about the current `$items` of a *ListViewModel* in the format `<start index> - <end index> of <total count>`.

Full Documentation: *c-list-range-display*.

c-table

A table component for displaying the contents of a *ListViewModel*. Also supports modifying the list's *sort parameters* by clicking on column headers. Pairs well with a *c-list-pagination*.

Full Documentation: *c-table*.

3.18.25 Input Components

c-input

A general-purpose input component for most *Values*. c-input does not have much functionality of its own - instead, it delegates to the right kind of component based on the type of value to which it is bound. This includes both other Coalesce Vuetify Components as well as direct usages of some Vuetify components.

Full Documentation: *c-input*.

c-select

A dropdown component that allows for selecting values fetched from the generated `/list` API endpoints.

Used both for selecting values for foreign key and navigation properties, and for selecting arbitrary objects or primary keys independent of a parent or owning object.

Full Documentation: *c-select*.

c-datetime-picker

A general, all-purpose date/time input component that can be used either with *models* and *metadata* or as a standalone component using only `v-model`.

Full Documentation: *c-datetime-picker*.

c-select-many-to-many

A multi-select dropdown component that allows for selecting values fetched from the generated `/list` API endpoints for collection navigation properties that were annotated with *[ManyToMany]*.

Full Documentation: *c-select-many-to-many*.

c-select-string-value

A dropdown component that will present a list of suggested string values from a custom API endpoint. Allows users to input values that aren't provided by the endpoint.

Effectively, this is a server-driven autocomplete list.

Full Documentation: *c-select-string-value*.

c-select-values

A multi-select input component for collections of non-object values (primarily strings and numbers).

Full Documentation: *c-select-values*.

c-list-filters

A component that provides an interface for modifying the `filters` prop of a *ListViewModel*'s *parameters*.

Full Documentation: *c-list-filters*.

c-list-pagination

A component that provides an interface for modifying the pagination *parameters* of a *ListViewModel*.

This is a composite of *c-list-page-size*, *c-list-range-display*, and *c-list-page*, arranged horizontally. It is designed to be used above or below a table (e.g. *c-table*).

Full Documentation: *c-list-pagination*.

c-list-page-size

A component that provides an dropdown for modifying the `pageSize` *parameter* prop of a *ListViewModel*.

Full Documentation: *c-list-page-size*.

c-list-page

A component that provides previous/next buttons and a text field for modifying the `page` *parameter* prop of a *ListViewModel*.

Full Documentation: *c-list-page*.

3.18.26 Admin Components

c-admin-method

Provides an interface for invoking a *method* and rendering its result, designed to be use in an admin page.

Full Documentation: *c-admin-method*.

c-admin-methods

Renders in a Vuetify v-expansion-panels a *c-admin-method* for each method on a *ViewModel* or *ListViewModel*.

Full Documentation: *c-admin-methods*.

c-admin-display

Behaves the same as *c-display*, except any collection navigation properties will be rendered as links to an admin list page, and any models will be rendered as a link to an admin item page.

Full Documentation: *c-admin-display*.

c-admin-editor

An editor for a single *ViewModel* instance. Provides a *c-input* for each property of the model.

Full Documentation: *c-admin-editor*.

c-admin-editor-page

A page for a creating/editing single *ViewModel* instance. Provides a *c-admin-editor* and a *c-admin-methods* for the instance. Designed to be routed to directly with vue-router.

Full Documentation: *c-admin-editor-page*.

c-admin-table

An full-featured table for a *ListViewModel*, including a *c-admin-table-toolbar*, *c-table*, and *c-list-pagination*.

Full Documentation: *c-admin-table*.

c-admin-table-toolbar

A full-featured toolbar for a *ListViewModel* designed to be used on an admin page, including "Create" and "Reload" buttons, a *c-list-range-display*, a *c-list-page*, a search field, *c-list-filters*, and a *c-list-page-size*.

Full Documentation: *c-admin-table-toolbar*.

c-admin-table-page

A full-featured page for interacting with a *ListViewModel*. Provides a *c-admin-table* and a *c-admin-methods* for the list. Designed to be routed to directly with vue-router.

Full Documentation: *c-admin-table-page*.

3.19 Knockout Overview

The Knockout stack for Coalesce offers the ability to build pages with the time-tested Knockout JavaScript library using all of the features of the Coalesce generated APIs and ViewModels. It can be used for anything between adding simple interactive augmentations of MVC pages to building a full MPA-SPA hybrid application.

Contents

3.19.1 Getting Started

Check out *Getting Started with Knockout* if you haven't already to learn how to get a new Coalesce Knockout project up and running.

3.19.2 Generated Code

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

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 Entity Models* and *Custom DTOs*. 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 Entity Models* and *Custom DTOs*. 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 non-primitive types which are not themselves a *EF Entity Models* or *Custom DTOs* which are accessible through the aforementioned types, either through one of its *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.

View Controllers

For each of your *EF Entity Models* and *Custom DTOs*, 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 Entity Models* and *Custom DTOs*, 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.

3.20 Getting Started with Knockout

3.20.1 Creating a Project

The quickest and easiest way to create a new Coalesce Knockout application is to use the `dotnet new` template. In your favorite shell:

```
dotnet new --install IntelliTect.Coalesce.KnockoutJS.Template
dotnet new coalesceko
```

- View on GitHub

3.20.2 Data Modeling

At this point, you can open up the newly-created 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` npm script (Vue) or gulp task (Knockout) in the Task Runner Explorer

You're now at a point where you can start creating your own pages!

3.20.3 Building Pages & Features

Lets say we've created a *model* called `Person` as follows, 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, including the *Knockout Overview*.

3.21 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.

3.21.1 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): JQueryPromise<any 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.

3.21.2 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,
    };
}
```

3.22 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.

3.22.1 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.

3.22.2 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();
    }
}
```

3.23 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.

3.23.1 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.

3.23.2 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;
```

3.24 TypeScript Method Objects

For each *Custom 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.
}
```

3.24.1 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.

3.24.2 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.

3.24.3 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.

3.25 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.

3.25.1 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.

3.25.2 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()`.

3.25.3 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.

3.26 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

3.26.1 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.

3.26.2 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.

3.26.3 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>
```

3.26.4 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.

3.27 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

3.27.1 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; }
```

(continues on next page)

(continued from previous page)

```
}

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:

Vue

```
import { Employee } from '@/viewmodels.g'
import { EmployeeViewModel } from '@/viewmodels.g'

var employee = new EmployeeViewModel();
employee.$dataSource = new Employee.DataSources.WithProjectsAndMembers();
employee.$load(1);
```

Knockout

```
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!

3.27.2 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)
```

(continues on next page)

(continued from previous page)

```
                        .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);
            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
- 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.

3.28 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

3.28.1 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.

Vue

```
import { PersonViewModel, PersonListViewModel } from '@/viewmodels.g'

var person = new PersonViewModel();
person.$includes = "details";

var personList = new PersonListViewModel();
personList.$includes = "details";
```

Knockout

```
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 Knockout CreateEdit views.

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

3.28.2 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; }
}

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

In TypeScript:

Vue

```
import { PersonListViewModel } from '@/viewmodels.g'

const personList = new PersonListViewModel();
personList.$includes = "Editor";
await personList.$load();
// Objects in personList.$items will not contain CreatedBy nor Department objects.
```

(continues on next page)

(continued from previous page)

```
const personList2 = new PersonListViewModel();
personList2.$includes = "details";
await personList.$load();
// Objects in personList2.items will be allowed to contain both CreatedBy and␣
↪Department objects.
// Department will be allowed to include its other Person objects.
```

Knockout

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

var personList2 = new ListViewModels.PersonList();
personList2.includes = "details";
personList2.load(() => {
    // objects in personList2.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.

3.29 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.

3.30 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.

3.30.1 File Resolution

When the code generation is run 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.

3.30.2 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",
    },

    // The name of the root generator to use. Defaults to "Knockout".
    // Available values are "Vue" and "Knockout".
    "rootGenerator": "Vue",

    // If set, specifies a list of whitelisted root type names that will restrict
    // which types Coalesce will use for code generation.
    // Root types are those that must be annotated with [Coalesce].
    // Useful if want to segment a single data project into multiple web projects,
    // or into different areas/directories within a single web project.
    "rootTypesWhitelist": [
        "MyDbContext", "MyCustomDto"
    ],

    "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".
```

(continues on next page)

(continued from previous page)

```
        "GeneratorName": {
            // Optional: true if the generator should be disabled.
            "disabled": true,
            // Optional: Configures a path relative to the default output path for␣
↪the generator
            // where that generator's output should be placed instead.
            "targetDirectory": "../DifferentFolder"
        },
        // 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
        },
        "ViewController" : {
            "indentationSize": 2
        }
    }
}
```

3.30.3 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`.