Accessing Tracked Entities

Article • 09/28/2022

There are four main APIs for accessing entities tracked by a DbContext:

- DbContext.Entry returns an EntityEntry<TEntity> instance for a given entity instance.
- ChangeTracker.Entries returns EntityEntry<TEntity> instances for all tracked entities, or for all tracked entities of a given type.
- DbContext.Find, DbContext.FindAsync, DbSet<TEntity>.Find, and DbSet<TEntity>.FindAsync find a single entity by primary key, first looking in tracked entities, and then querying the database if needed.
- DbSet<TEntity>.Local returns actual entities (not EntityEntry instances) for entities of the entity type represented by the DbSet.

Each of these is described in more detail in the sections below.

> **Tip**
>
> This document assumes that entity states and the basics of EF Core change tracking are understood. See **Change Tracking in EF Core** for more information on these topics.

> **Tip**
>
> You can run and debug into all the code in this document by **downloading the sample code from GitHub** .

Using DbContext.Entry and EntityEntry instances

For each tracked entity, Entity Framework Core (EF Core) keeps track of:

- The overall state of the entity. This is one of `Unchanged`, `Modified`, `Added`, or `Deleted`; see Change Tracking in EF Core for more information.
- The relationships between tracked entities. For example, the blog to which a post belongs.
- The "current values" of properties.

- The "original values" of properties, when this information is available. Original values are the property values that existed when entity was queried from the database.
- Which property values have been modified since they were queried.
- Other information about property values, such as whether or not the value is temporary.

Passing an entity instance to DbContext.Entry results in an EntityEntry<TEntity> providing access to this information for the given entity. For example:

C#

```
using var context = new BlogsContext();

var blog = context.Blogs.Single(e => e.Id == 1);
var entityEntry = context.Entry(blog);
```

The following sections show how to use an EntityEntry to access and manipulate entity state, as well as the state of the entity's properties and navigations.

Working with the entity

The most common use of EntityEntry<TEntity> is to access the current EntityState of an entity. For example:

C#

```
var currentState = context.Entry(blog).State;
if (currentState == EntityState.Unchanged)
{
    context.Entry(blog).State = EntityState.Modified;
}
```

The Entry method can also be used on entities that are not yet tracked. This *does not start tracking the entity*; the state of the entity is still `Detached`. However, the returned EntityEntry can then be used to change the entity state, at which point the entity will become tracked in the given state. For example, the following code will start tracking a Blog instance as `Added`:

C#

```
var newBlog = new Blog();
Debug.Assert(context.Entry(newBlog).State == EntityState.Detached);
```

```
context.Entry(newBlog).State = EntityState.Added;
Debug.Assert(context.Entry(newBlog).State == EntityState.Added);
```

> **Tip**
>
> Unlike in EF6, setting the state of an individual entity will not cause all connected entities to be tracked. This makes setting the state this way a lower-level operation than calling `Add`, `Attach`, or `Update`, which operate on an entire graph of entities.

The following table summarizes ways to use an EntityEntry to work with an entire entity:

Expand table

EntityEntry member	Description
EntityEntry.State	Gets and sets the EntityState of the entity.
EntityEntry.Entity	Gets the entity instance.
EntityEntry.Context	The DbContext that is tracking this entity.
EntityEntry.Metadata	IEntityType metadata for the type of entity.
EntityEntry.IsKeySet	Whether or not the entity has had its key value set.
EntityEntry.Reload()	Overwrites property values with values read from the database.
EntityEntry.DetectChanges()	Forces detection of changes for this entity only; see Change Detection and Notifications.

Working with a single property

Several overloads of EntityEntry<TEntity>.Property allow access to information about an individual property of an entity. For example, using a strongly-typed, fluent-like API:

C#

```
PropertyEntry<Blog, string> propertyEntry = context.Entry(blog).Property(e =>
e.Name);
```

The property name can instead be passed as a string. For example:

C#

```
PropertyEntry<Blog, string> propertyEntry =
context.Entry(blog).Property<string>("Name");
```

The returned PropertyEntry<TEntity,TProperty> can then be used to access information about the property. For example, it can be used to get and set the current value of the property on this entity:

C#

```
string currentValue = context.Entry(blog).Property(e => e.Name).CurrentValue;
context.Entry(blog).Property(e => e.Name).CurrentValue = "1unicorn2";
```

Both of the Property methods used above return a strongly-typed generic PropertyEntry<TEntity,TProperty> instance. Using this generic type is preferred because it allows access to property values without boxing value types. However, if the type of entity or property is not known at compile-time, then a non-generic PropertyEntry can be obtained instead:

C#

```
PropertyEntry propertyEntry = context.Entry(blog).Property("Name");
```

This allows access to property information for any property regardless of its type, at the expense of boxing value types. For example:

C#

```
object blog = context.Blogs.Single(e => e.Id == 1);

object currentValue = context.Entry(blog).Property("Name").CurrentValue;
context.Entry(blog).Property("Name").CurrentValue = "1unicorn2";
```

The following table summarizes property information exposed by PropertyEntry:

Expand table

PropertyEntry member	Description
PropertyEntry<TEntity,TProperty>.CurrentValue	Gets and sets the current value of the property.

PropertyEntry member	Description
PropertyEntry<TEntity,TProperty>.OriginalValue	Gets and sets the original value of the property, if available.
PropertyEntry<TEntity,TProperty>.EntityEntry	A back reference to the EntityEntry<TEntity> for the entity.
PropertyEntry.Metadata	IProperty metadata for the property.
PropertyEntry.IsModified	Indicates whether this property is marked as modified, and allows this state to be changed.
PropertyEntry.IsTemporary	Indicates whether this property is marked as temporary, and allows this state to be changed.

Notes:

- The original value of a property is the value that the property had when the entity was queried from the database. However, original values are not available if the entity was disconnected and then explicitly attached to another DbContext, for example with `Attach` or `Update`. In this case, the original value returned will be the same as the current value.
- SaveChanges will only update properties marked as modified. Set IsModified to true to force EF Core to update a given property value, or set it to false to prevent EF Core from updating the property value.
- Temporary values are typically generated by EF Core value generators. Setting the current value of a property will replace the temporary value with the given value and mark the property as not temporary. Set IsTemporary to true to force a value to be temporary even after it has been explicitly set.

Working with a single navigation

Several overloads of EntityEntry<TEntity>.Reference, EntityEntry<TEntity>.Collection, and EntityEntry.Navigation allow access to information about an individual navigation.

Reference navigations to a single related entity are accessed through the Reference methods. Reference navigations point to the "one" sides of one-to-many relationships, and both sides of one-to-one relationships. For example:

C#

```
ReferenceEntry<Post, Blog> referenceEntry1 = context.Entry(post).Reference(e =>
e.Blog);
ReferenceEntry<Post, Blog> referenceEntry2 =
context.Entry(post).Reference<Blog>("Blog");
ReferenceEntry referenceEntry3 = context.Entry(post).Reference("Blog");
```

Navigations can also be collections of related entities when used for the "many" sides of one-to-many and many-to-many relationships. The Collection methods are used to access collection navigations. For example:

C#

```
CollectionEntry<Blog, Post> collectionEntry1 = context.Entry(blog).Collection(e
=> e.Posts);
CollectionEntry<Blog, Post> collectionEntry2 =
context.Entry(blog).Collection<Post>("Posts");
CollectionEntry collectionEntry3 = context.Entry(blog).Collection("Posts");
```

Some operations are common for all navigations. These can be accessed for both reference and collection navigations using the EntityEntry.Navigation method. Note that only non-generic access is available when accessing all navigations together. For example:

C#

```
NavigationEntry navigationEntry = context.Entry(blog).Navigation("Posts");
```

The following table summarizes ways to use ReferenceEntry<TEntity,TProperty>, CollectionEntry<TEntity,TRelatedEntity>, and NavigationEntry:

Expand table

NavigationEntry member	Description
MemberEntry.CurrentValue	Gets and sets the current value of the navigation. This is the entire collection for collection navigations.
NavigationEntry.Metadata	INavigationBase metadata for the navigation.
NavigationEntry.IsLoaded	Gets or sets a value indicating whether the related entity or collection has been fully loaded from the database.

NavigationEntry member	Description
NavigationEntry.Load()	Loads the related entity or collection from the database; see Explicit Loading of Related Data.
NavigationEntry.Query()	The query EF Core would use to load this navigation as an `IQueryable` that can be further composed; see Explicit Loading of Related Data.

Working with all properties of an entity

EntityEntry.Properties returns an IEnumerable<T> of PropertyEntry for every property of the entity. This can be used to perform an action for every property of the entity. For example, to set any DateTime property to `DateTime.Now`:

C#

```
foreach (var propertyEntry in context.Entry(blog).Properties)
{
    if (propertyEntry.Metadata.ClrType == typeof(DateTime))
    {
        propertyEntry.CurrentValue = DateTime.Now;
    }
}
```

In addition, EntityEntry contains several methods to get and set all property values at the same time. These methods use the PropertyValues class, which represents a collection of properties and their values. PropertyValues can be obtained for current or original values, or for the values as currently stored in the database. For example:

C#

```
var currentValues = context.Entry(blog).CurrentValues;
var originalValues = context.Entry(blog).OriginalValues;
var databaseValues = context.Entry(blog).GetDatabaseValues();
```

These PropertyValues objects are not very useful on their own. However, they can be combined to perform common operations needed when manipulating entities. This is useful when working with data transfer objects and when resolving optimistic concurrency conflicts. The following sections show some examples.

Setting current or original values from an entity or DTO

The current or original values of an entity can be updated by copying values from another object. For example, consider a `BlogDto` data transfer object (DTO) with the same properties as the entity type:

C#

```
public class BlogDto
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

This can be used to set the current values of a tracked entity using PropertyValues.SetValues:

C#

```
var blogDto = new BlogDto { Id = 1, Name = "1unicorn2" };

context.Entry(blog).CurrentValues.SetValues(blogDto);
```

This technique is sometimes used when updating an entity with values obtained from a service call or a client in an n-tier application. Note that the object used does not have to be of the same type as the entity so long as it has properties whose names match those of the entity. In the example above, an instance of the DTO `BlogDto` is used to set the current values of a tracked `Blog` entity.

Note that properties will only be marked as modified if the value set differs from the current value.

Setting current or original values from a dictionary

The previous example set values from an entity or DTO instance. The same behavior is available when property values are stored as name/value pairs in a dictionary. For example:

C#

```
var blogDictionary = new Dictionary<string, object> { ["Id"] = 1, ["Name"] =
"1unicorn2" };
```

```
context.Entry(blog).CurrentValues.SetValues(blogDictionary);
```

Setting current or original values from the database

The current or original values of an entity can be updated with the latest values from the database by calling GetDatabaseValues() or GetDatabaseValuesAsync and using the returned object to set current or original values, or both. For example:

C#

```
var databaseValues = context.Entry(blog).GetDatabaseValues();
context.Entry(blog).CurrentValues.SetValues(databaseValues);
context.Entry(blog).OriginalValues.SetValues(databaseValues);
```

Creating a cloned object containing current, original, or database values

The PropertyValues object returned from CurrentValues, OriginalValues, or GetDatabaseValues can be used to create a clone of the entity using PropertyValues.ToObject(). For example:

C#

```
var clonedBlog = context.Entry(blog).GetDatabaseValues().ToObject();
```

Note that `ToObject` returns a new instance that is not tracked by the DbContext. The returned object also does not have any relationships set to other entities.

The cloned object can be useful for resolving issues related to concurrent updates to the database, especially when data binding to objects of a certain type. See optimistic concurrency for more information.

Working with all navigations of an entity

EntityEntry.Navigations returns an IEnumerable<T> of NavigationEntry for every navigation of the entity. EntityEntry.References and EntityEntry.Collections do the same thing, but restricted to reference or collection navigations respectively. This can be used to perform

an action for every navigation of the entity. For example, to force loading of all related entities:

C#

```
foreach (var navigationEntry in context.Entry(blog).Navigations)
{
    navigationEntry.Load();
}
```

Working with all members of an entity

Regular properties and navigation properties have different state and behavior. It is therefore common to process navigations and non-navigations separately, as shown in the sections above. However, sometimes it can be useful to do something with any member of the entity, regardless of whether it is a regular property or navigation. EntityEntry.Member and EntityEntry.Members are provided for this purpose. For example:

C#

```
foreach (var memberEntry in context.Entry(blog).Members)
{
    Console.WriteLine(
        $"Member {memberEntry.Metadata.Name} is of type
{memberEntry.Metadata.ClrType.ShortDisplayName()} and has value
{memberEntry.CurrentValue}");
}
```

Running this code on a blog from the sample generates the following output:

Output

```
Member Id is of type int and has value 1
Member Name is of type string and has value .NET Blog
Member Posts is of type IList<Post> and has value
System.Collections.Generic.List`1[Post]
```

Tip

The **change tracker debug view** shows information like this. The debug view for the entire change tracker is generated from the individual **EntityEntry.DebugView** of each

tracked entity.

Find and FindAsync

DbContext.Find, DbContext.FindAsync, DbSet<TEntity>.Find, and DbSet<TEntity>.FindAsync are designed for efficient lookup of a single entity when its primary key is known. Find first checks if the entity is already tracked, and if so returns the entity immediately. A database query is only made if the entity is not tracked locally. For example, consider this code that calls Find twice for the same entity:

C#

```
using var context = new BlogsContext();

Console.WriteLine("First call to Find...");
var blog1 = context.Blogs.Find(1);

Console.WriteLine($"...found blog {blog1.Name}");

Console.WriteLine();
Console.WriteLine("Second call to Find...");
var blog2 = context.Blogs.Find(1);
Debug.Assert(blog1 == blog2);

Console.WriteLine("...returned the same instance without executing a query.");
```

The output from this code (including EF Core logging) when using SQLite is:

Output

```
First call to Find...
info: 12/29/2020 07:45:53.682 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (1ms) [Parameters=[@__p_0='1' (DbType = String)],
CommandType='Text', CommandTimeout='30']
      SELECT "b"."Id", "b"."Name"
      FROM "Blogs" AS "b"
      WHERE "b"."Id" = @__p_0
      LIMIT 1
...found blog .NET Blog

Second call to Find...
...returned the same instance without executing a query.
```

Notice that the first call does not find the entity locally and so executes a database query. Conversely, the second call returns the same instance without querying the database because it is already being tracked.

Find returns null if an entity with the given key is not tracked locally and does not exist in the database.

Composite keys

Find can also be used with composite keys. For example, consider an `OrderLine` entity with a composite key consisting of the order ID and the product ID:

C#

```
public class OrderLine
{
    public int OrderId { get; set; }
    public int ProductId { get; set; }

    //...
}
```

The composite key must be configured in DbContext.OnModelCreating to define the key parts *and their order*. For example:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<OrderLine>()
        .HasKey(e => new { e.OrderId, e.ProductId });
}
```

Notice that `OrderId` is the first part of the key and `ProductId` is the second part of the key. This order must be used when passing key values to Find. For example:

C#

```
var orderline = context.OrderLines.Find(orderId, productId);
```

Using ChangeTracker.Entries to access all tracked entities

So far we have accessed only a single EntityEntry at a time. ChangeTracker.Entries() returns an EntityEntry for every entity currently tracked by the DbContext. For example:

C#

```
using var context = new BlogsContext();
var blogs = context.Blogs.Include(e => e.Posts).ToList();

foreach (var entityEntry in context.ChangeTracker.Entries())
{
    Console.WriteLine($"Found {entityEntry.Metadata.Name} entity with ID
{entityEntry.Property("Id").CurrentValue}");
}
```

This code generates the following output:

Output

```
Found Blog entity with ID 1
Found Post entity with ID 1
Found Post entity with ID 2
```

Notice that entries for both blogs and posts are returned. The results can instead be filtered to a specific entity type using the ChangeTracker.Entries<TEntity>() generic overload:

C#

```
foreach (var entityEntry in context.ChangeTracker.Entries<Post>())
{
    Console.WriteLine(
        $"Found {entityEntry.Metadata.Name} entity with ID
{entityEntry.Property(e => e.Id).CurrentValue}");
}
```

The output from this code shows that only posts are returned:

Output

```
Found Post entity with ID 1
Found Post entity with ID 2
```

Also, using the generic overload returns generic EntityEntry<TEntity> instances. This is what allows that fluent-like access to the `Id` property in this example.

The generic type used for filtering does not have to be a mapped entity type; an unmapped base type or interface can be used instead. For example, if all the entity types in the model implement an interface defining their key property:

C#

```
public interface IEntityWithKey
{
    int Id { get; set; }
}
```

Then this interface can be used to work with the key of any tracked entity in a strongly-typed manner. For example:

C#

```
foreach (var entityEntry in context.ChangeTracker.Entries<IEntityWithKey>())
{
    Console.WriteLine(
        $"Found {entityEntry.Metadata.Name} entity with ID
{entityEntry.Property(e => e.Id).CurrentValue}");
}
```

Using DbSet.Local to query tracked entities

EF Core queries are always executed on the database, and only return entities that have been saved to the database. DbSet<TEntity>.Local provides a mechanism to query the DbContext for local, tracked entities.

Since `DbSet.Local` is used to query tracked entities, it is typical to load entities into the DbContext and then work with those loaded entities. This is especially true for data binding, but can also be useful in other situations. For example, in the following code the database is first queried for all blogs and posts. The Load extension method is used to execute this query with the results tracked by the context without being returned directly

to the application. (Using `ToList` or similar has the same effect but with the overhead of creating the returned list, which is not needed here.) The example then uses `DbSet.Local` to access the locally tracked entities:

C#

```
using var context = new BlogsContext();

context.Blogs.Include(e => e.Posts).Load();

foreach (var blog in context.Blogs.Local)
{
    Console.WriteLine($"Blog: {blog.Name}");
}

foreach (var post in context.Posts.Local)
{
    Console.WriteLine($"Post: {post.Title}");
}
```

Notice that, unlike ChangeTracker.Entries(), `DbSet.Local` returns entity instances directly. An EntityEntry can, of course, always be obtained for the returned entity by calling DbContext.Entry.

The local view

DbSet<TEntity>.Local returns a view of locally tracked entities that reflects the current EntityState of those entities. Specifically, this means that:

- `Added` entities are included. Note that this is not the case for normal EF Core queries, since `Added` entities do not yet exist in the database and so are therefore never returned by a database query.
- `Deleted` entities are excluded. Note that this is again not the case for normal EF Core queries, since `Deleted` entities still exist in the database and so *are* returned by database queries.

All of this means that `DbSet.Local` is view over the data that reflects the current conceptual state of the entity graph, with `Added` entities included and `Deleted` entities excluded. This matches what database state is expected to be after SaveChanges is called.

This is typically the ideal view for data binding, since it presents to the user the data as they understand it based on the changes made by the application.

The following code demonstrates this by marking one post as `Deleted` and then adding a new post, marking it as `Added`:

C#

```
using var context = new BlogsContext();

var posts = context.Posts.Include(e => e.Blog).ToList();

Console.WriteLine("Local view after loading posts:");

foreach (var post in context.Posts.Local)
{
    Console.WriteLine($"  Post: {post.Title}");
}

context.Remove(posts[1]);

context.Add(
    new Post
    {
        Title = "What's next for System.Text.Json?",
        Content = ".NET 5.0 was released recently and has come with many...",
        Blog = posts[0].Blog
    });

Console.WriteLine("Local view after adding and deleting posts:");

foreach (var post in context.Posts.Local)
{
    Console.WriteLine($"  Post: {post.Title}");
}
```

The output from this code is:

Output

```
Local view after loading posts:
  Post: Announcing the Release of EF Core 5.0
  Post: Announcing F# 5
  Post: Announcing .NET 5.0
Local view after adding and deleting posts:
  Post: What's next for System.Text.Json?
  Post: Announcing the Release of EF Core 5.0
  Post: Announcing .NET 5.0
```

Notice that the deleted post is removed from the local view, and the added post is included.

Using Local to add and remove entities

DbSet<TEntity>.Local returns an instance of LocalView<TEntity>. This is an implementation of ICollection<T> that generates and responds to notifications when entities are added and removed from the collection. (This is the same concept as ObservableCollection<T>, but implemented as a projection over existing EF Core change tracking entries, rather than as an independent collection.)

The local view's notifications are hooked into DbContext change tracking such that the local view stays in sync with the DbContext. Specifically:

- Adding a new entity to `DbSet.Local` causes it to be tracked by the DbContext, typically in the `Added` state. (If the entity already has a generated key value, then it is tracked as `Unchanged` instead.)
- Removing an entity from `DbSet.Local` causes it to be marked as `Deleted`.
- An entity that becomes tracked by the DbContext will automatically appear in the `DbSet.Local` collection. For example, executing a query to bring in more entities automatically causes the local view to be updated.
- An entity that is marked as `Deleted` will be removed from the local collection automatically.

This means the local view can be used to manipulate tracked entities simply by adding and removing from the collection. For example, let's modify the previous example code to add and remove posts from the local collection:

C#

```
using var context = new BlogsContext();

var posts = context.Posts.Include(e => e.Blog).ToList();

Console.WriteLine("Local view after loading posts:");

foreach (var post in context.Posts.Local)
{
    Console.WriteLine($"  Post: {post.Title}");
}

context.Posts.Local.Remove(posts[1]);
```

```
context.Posts.Local.Add(
    new Post
    {
        Title = "What's next for System.Text.Json?",
        Content = ".NET 5.0 was released recently and has come with many...",
        Blog = posts[0].Blog
    });

Console.WriteLine("Local view after adding and deleting posts:");

foreach (var post in context.Posts.Local)
{
    Console.WriteLine($"  Post: {post.Title}");
}
```

The output remains unchanged from the previous example because changes made to the local view are synced with the DbContext.

Using the local view for Windows Forms or WPF data binding

DbSet<TEntity>.Local forms the basis for data binding to EF Core entities. However, both Windows Forms and WPF work best when used with the specific type of notifying collection that they expect. The local view supports creating these specific collection types:

- LocalView<TEntity>.ToObservableCollection() returns an ObservableCollection<T> for WPF data binding.
- LocalView<TEntity>.ToBindingList() returns a BindingList<T> for Windows Forms data binding.

For example:

C#

```
ObservableCollection<Post> observableCollection =
context.Posts.Local.ToObservableCollection();
BindingList<Post> bindingList = context.Posts.Local.ToBindingList();
```

See Get Started with WPF for more information on WPF data binding with EF Core, and Get Started with Windows Forms for more information on Windows Forms data binding with EF Core.

Tip

The local view for a given DbSet instance is created lazily when first accessed and then cached. LocalView creation itself is fast and it does not use significant memory. However, it does call **DetectChanges**, which can be slow for large numbers of entities. The collections created by `ToObservableCollection` and `ToBindingList` are also created lazily and then cached. Both of these methods create new collections, which can be slow and use a lot of memory when thousands of entities are involved.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback