Contents

The *EntityManager* is **not thread-safe** and **neither are entities**. An EntityManager throws an exception if you try to use it on multiple threads.

This topic touches upon the challenges and risks of cross-threading and describes how you use the *AuthorizedThreadId* to control which thread an EntityManager calls home.

The EntityManager is not thread safe

The *EntityManager* is **not thread-safe**.

The RIA Services *DomainContext*, the Entity Framework *ObjectContext*, and the NHibernate *Session* classes aren't thread-safe either.

Internally the EntityManager maintains mutable collections of mutable objects. That is in the very nature of entities and of the components with which you manage them. They cannot be made thread safe.

You may think you need to write background tasks to improve performance. You generally do not. The EntityManager can perform many operations asynchronously for you.

Perhaps you want to retrieve several large entity collections in background. You can launch multiple asynchronous queries from a single EntityManager running on the main thread; DevForce will handle the background threading and marshal the results back to the main thread safely. See the topic on asynchronous queries.

AuthorizedThreadId and the EntityManager's home thread

An EntityManager remembers the id of the thread on which it was created. This identifies its home thread.

The EntityManager's *AuthorizedThreadId* property tells you what thread id that is.

You won't care about this id as long as you stay clear of multi-threaded scenarios. Unfortunately, some times you can't. Two scenarios come to mind:

1. Automated MS Tests of asynchronous queries
2. ASP.NET clients that maintain an EntityManager across requests.

In both cases, the EntityManager can be called on a thread other than the thread on which it was created. That is usually a big, red "danger" flag. It turns out to be safe in these "free threading" scenarios because the EntityManager will never be called on its original thread again.

Of course the EntityManager doesn't know that. It will throw an exception (see below) when called on the new thread, because it assumes the worst and fears that concurrent multi-threaded access may occur. We have to tell it that its home thread has changed which we do by setting its *AuthorizedThreadId* property to the id of the new thread. Here's how to set the *AuthorizedThreadId* to the currently executing thread:

```
manager.AuthorizedThreadId = System.Threading.Thread.CurrentThread.ManagedThreadId;
```

```
manager.AuthorizedThreadId = System.Threading.Thread.CurrentThread.ManagedThreadId
```

A moment ago you read: "*the EntityManager will never be called on its original thread again*."

We recommend that you be deeply suspicious of any such claim. You may trust ... but you should verify.

Fortunately, you get that verification for free when you change the *AuthorizedThreadId*. If something calls upon the EntityManager back on the original thread, the EntityManager will throw an exception ... because the original thread is no longer the home thread. Let's try it.

```
// Now executing on original thread with Id=18
// Change the home ThreadId to a phoney
manager.AuthorizedThreadId = 1234;
// Call the EntityManager on thread Id=18
Manager.Customers.ExecuteAsync(); // Throws exception
```

```
' Now executing on original thread with Id=18
' Change the home ThreadId to a phoney
manager.AuthorizedThreadId = 1234
' Call the EntityManager on thread Id=18
Manager.Customers.ExecuteAsync() ' Throws exception
```

ExecuteAsync throws an exception:

> Customers query failed with exception: System.InvalidOperationException: An EntityManager can only execute on a single thread. This EntityManager is authorized to execute on the thread with id='1234'; the requested operation came from the thread with Id='18'. ...

EntityManagers on ASP Clients

In most cases, an ASP.NET web application developers should create a new EntityManager for each request.

However, when writing a "wizard" that carries user data forward from request to request, some developers choose to hold an EntityManager in an in-memory *Session* variable. That way, they can reuse cached entities across requests rather than have to struggle with managing temporary storage for changed entities.

There are other, perhaps safer and more scalable ways, to address this scenario.

In essence, you store the EntityManager in *Session* just before the current request ends; when the follow-up request begins, you pull the EntityManager instance out of *Session* and assign it to your *manager* variable.

This will fail. The follow-on-request is on a different thread than the prior request. The EntityManager you put into *Session* is pinned to the thread of the prior request. It will throw the *System.InvalidOperationException* the moment you use it on the new request.

The solution is as described above. Set the *AuthorizedThreadId* property to the new request's thread immediately after restoring the EntityManager from *Session*, well before calling any other of its members.

Disabling thread id checking

You can disable thread id checking by setting *AuthorizedThreadId* to *null*.

Unless you are an expert at writing multi-threaded code and can design an architecture that avoids livelocks and deadlocks, we strongly recommend that you leave thread checking enabled. Turning off thread checking is especially dangerous in server methods which may be called by multiple threads.

Please only disable thread id checking for a very good reason and with full awareness of the risks.

Customize the thread id check

If you don't want to disable thread id checking altogether but instead implement your own logic, use the *SafeThreadingCheck* property on the *EntityManager*. Set the *SafeThreadingCheck* to a method of your choosing to perform custom logic, and be sure to throw an exception if the threading check fails.

For example:

```
entityManager.SafeThreadingCheck = (em) => {
    var id = em.AuthorizedThreadId;
    Assert.IsTrue(id == AppEnv.Current.GetCurrentThreadId());
};
```

Entities are not thread safe either

The EntityManager throws an exception when exercised on multiple threads. Unfortunately, there is no similar guard logic for entities nor for collections of entities. You have to ensure that they are used in a thread-safe manner.

As mentioned, it is best to avoid multiple threads in the first place. You rarely need background threads in a client application. Regard claims to the contrary with deep suspicion.

Server programming is another matter. Strive to make custom server logic single-threaded. Avoid stateful static classes. When multiple threads are unavoidable, make sure you use the necessary synchronization logic.