Contents

This is an advanced topic and is not recommended for developers new to DevForce.

The ability to create *Persistent Database Connections* was developed because of the need for some applications to be able to assign a dedicated database connection to each user session and have that connection stay open for the duration of the user session. This ability is turned off by default, because it introduces a number of constraints on the performance and scalability of the DevForce server.

In particular, load balancing and connection thread pooling do not operate in this environment. Moreover, all asynchonous calls from the client to the server for each session become serialized on the server in this model.

Primary use case

The primary use case for this feature is where a developer wants to make use of temporary database tables that are expensive to create and may be reused over the duration of a user session. Typically, these temp tables are created via an InvokeServerMethod call early in a user session and are then available to all future stored procedures and other InvokeServerMethod calls later in the session.

Enabling "one database connection per session"

Set the usePersistentDbConnectionPerSession attribute to true in the serverSettings element of the web.config file:

```
<objectServer useDCS="true" remoteBaseURL="http://localhost" serverPort="9009" serviceName="EntityService" >
     <serverSettings usePersistentDbConnectionPerSession="true"/>
</objectServer>
```

Accessing the connection on the server

Once the 'usePersistentDbConnectionPerSession' has been turned on, the server will insure that each session has its own dedicated DbConnection.

To access this DbConnection, or its corresponding EntityConnection. on a session from server side code you will need to make use of the GetServerSessionInfo() extension method on the SessionBundle. This extension method can be found in the IdeaBlade.EntityModel.Edm assembly. From the ServerSessionInfo instance returned you can the access a map of all of the 'persistent connections' available to this session keyed by the datasource key name. i.e. the name of the connection in the app.config or web.config file.

```
var persistentConnection = SessionBundle.CurrentSessionBundle.GetServerSessionInfo().PersistentConnectionMap[dataSourceKeyName];
var dbConnection = persistentConnection.DbConnection;
var entityConnection = persistentConnection.EntityConnection;
```

Ensuring serialized access

In order to ensure serialized access to a single PersistentConnection, DevForce provides a SessionScope class that may be used as shown below. The SessionScope class is in the IdeaBlade.EntityModel.Edm assembly and namespace.

```
using (new SessionScope(SessionType.Exclusive)) {
 using (new TransactionScope(....)) {
    var pcMap = SessionBundle.CurrentSessionBundle.GetServerSessionInfo().PersistentConnectionMap;
    var connection1 = pcMap[connection1Name];
    -- DO SOMETHING WITH THE CONNECTION HERE --
  }
}
```

This pattern is important because it ensures that all of the connections within the session are locked until the SessionScope exits. Locking is critical because unlike in regular DevForce where each call to the server will get its own connection, in this case there is only one connection that needs to be shared by all server side calls within this session. This means that any queries, saves or other server operations that talk to the database, **within this session**, are locked until the SessionScope completes. Note that all of DevForce's internal database operations are already protected by such a SessionScope, so as a developer you only need to insure that any 'custom' access to these persistent connections follow the same rules.

The pattern above should be the default where only a single database is being accessed within a application. However, if multiple databases are going to be accessed then there is a second pattern that may be used that will allow for locking individual database connections within a session without have to lock down the entire session. Note that the parameter passed to the SessionScope constructor above was SessionScope.Exclusive whereas below it is SessionScope.Shared.

```
using (new SessionScope(SessionType.Shared)) {
 using (new TransactionScope(....)) {
    var pcMap = SessionBundle.CurrentSessionBundle.GetServerSessionInfo().PersistentConnectionMap;
    var connection1 = pcMap[connection1Name];
    connection1.Lock()  // you do not need an unlock, the SessionScope will unlock when complete
   // connection1 work here
   var connection2 = pcMap[ connection2Name];
    connection2.Lock()  // you do not need an unlock, the SessionScope will unlock when complete
   // connection2 work here
  // connection3, if it existed, could still be accessed by another method at the same time that this method is executing.
 }
}
// exiting the SessionScope will automatically unlock any connections locked within it.
```

When using SessionType.Shared, you must call 'Lock()' on any connections before you attempt to use them to perform any other database operation. By calling 'Lock()' you are informing the system that this specific connection is unavailable for use until its surrounding SessionScope is exited. The idea here is that you only lock those connections that you actually plan to use and any other connections within the session are still available for use by other processes operating within the same session. Unlocking occurs automatically when the SessionScope ends.

The 'SessionType.Shared' scope means that any other database operations running within their own 'SessionType.Shared' scope will be able to access any 'unlocked' persistent connections. An 'unlocked' connection is one where the 'Lock()' method has not be called within a active SessionScope.

The 'SessionType.Exclusive' scope means that all database operations, within the current session, are blocked until the session scope completes regardless of whether they involve active executing database connections.

Note that DevForce wraps its 'Save' and 'IdFixup' operations within an Exclusive scope and its 'Query' operations within a 'Shared' scope. The reason for this is that a distributed save transaction may cross databases but a query operation will not.

Closing connections

Connections can be closed on either the client or server.

On the client:

```
entityManager.Logout();
```

This will immediately close all connections within the current session.

On the server, you can either close just individual connections or all of them.

```
var persistentConnectionMap = SessionBundle.CurrentSessionBundle.GetServerSessionInfo().PersistentConnectionMap;
persistentConnectionMap.Remove(dataSourceKeyName);
//      or
persistentConnectionMap.RemoveAll()
```

Special notes

In order for a PersistentConnection corresponding to a given datasource key to be available via the PersistentConnectionMap property, there must have been at least one prior access to this datasource key on the server. This is usually accomplished by creating an 'empty' query for some type associated with this datasource.

Because there is only a single connection being shared across an entire session and because this connection is never closed, it is important for any 'custom' server side code to leave the connection in the same 'condition' that it finds it. In particular,

1) Do NOT close the connection
2) Do NOT create a data reader on the connection and start reading and then forget to either close the reader or read to its end.

Unknown macro: IBNote

```
The "IBNote" macro is not in the list of registered macros. Verify the spelling or contact your administrator.
```