Contents

The DevForce **EntityServer** is a controller class that mediates between the client application and server-side services. This topic describes its purpose and principal functions.

Introduction

The *EntityServer* is a DevForce controller class that operates in the background. There is usually one instance per Application Server but you can have multiple instances running simultaneously on the server.

The developer never works with the *EntityServer* directly; it's an internal class that the the developer can't even see. But the developer knows it is there, responding to client requests and distributing tasks to other components including components the developer can see and often write.

The *EntityServer* is primarily responsible for fielding and routing client requests of which there are four basic types:

1. Authentication
2. Query
3. Save
4. Custom service method invocation

Login and logout are the two authentication requests. The *EntityServer* can authenticate from an ambient security context in lieu of explicit login. You can run with no security at all - a risky proposition potentially acceptable on a secure internal network. See the introduction to security below.

Query and save requests are routed to query and save pipelines as described below.

The *EntityServer* forwards calls to your custom service methods; your methods are responsible for authorizing the caller (the caller's *IPrincipal* is one of the parameters) and performing the service.

Stateless

The *EntityServer* is "stateless", meaning that it handles each client request on its own thread, independent from every other request, and the *EntityServer* does not remember anything from one client request to the next.

We can't dictate how you write your server-side customizations. We can't - and shouldn't - stop you from writing a "stateful" service that the *EntityServer* executes on your behalf. That is an extremely risky thing to do; cross-thread concurrency bugs are easy to create and hard to remove. Please be careful.

Self-sufficient

The EntityServer can perform its tasks with very little help from the developer. You have to provide an Entity Model (aka Domain Model) – the assembly of entity classes that collectively define the data and behavior of entities in the application domain.

That Entity Model may be the *only* thing the developer provides. The developer does not write a “domain service” to tell DevForce how to perform data-oriented operations. You neither write nor maintain an enormous class file with query, insert, update, and delete service methods for every entity. You don't write that same old code, over and over, for each entity.

DevForce knows how to query, insert, update, and delete on its own. You can intervene as necessary - where necessary - to prescribe, influence or block these operations. You do that, as we'll see, by modifying the pertinent *EntityServer* pipelines rather than by adding entity-specific operation methods.

Extensible

Most developers will customize the *EntityServer* behavior by extending it in five principle ways:

1. Declaring security constraints in the entity model, in the configuration, and with custom authentication logic.

2. Injecting custom behaviors in the *EntityServer* query and save pipelines to add or change processing as when specifying operation authorizations or pre-processing entities to be saved.
3. Enriching the entity classes with custom business logic such as validation rules that DevForce recognizes and executes on the server.
4. Adding custom service methods that clients can call to perform arbitrary actions on the server.
5. Replacing DevForce components with alternative implementations.

Again, you don't have to do any of this. DevForce works out-of-the-box with the entity model you define. However, if client machines are distributed across a public network, you should add logic to secure your server (#1) at the very least.

Secure

The "Secure" topic details the numerous security gates distributed throughout DevForce. At this level we'll merely inventory the opportunities available to the developer:

- ASP.NET Authentication is turned on by default. The *EntityService* picks up the *IPrincipal* from IIS and presents that principal to all methods you see and write.
- You can substitute your own authentication scheme and specify your own *IPrincipal* that is extended with application-specific roles and claims.
- Every client request (except a login request) must be accompanied by the security token that the *EntityServer* created and sent to the client when that client initiated its current authenticated session.
- Entity class security attributes you specify during model definition can block query and save operations outright. You can add security-oriented property interceptors and class level logic to ensure the fine-grained integrity of entity data.
- Every query and save passes through a security check in the query and save pipelines. You can closely examine these requests before letting them through, rejecting or modifying them as appropriate. You can log every request and every affected entity for audit purposes if you choose, in the manner you see fit.
- You must mark explicitly the service method that clients may invoke and you can easily restrict them to authorized users.
- You can scrub errors returned to the client to ensure that no sensitive information is disclosed in an exception.

Query and Save pipelines

The *EntityServer* routes queries and saves to their respective pipelines. The outward manifestation of the default pipelines are the *EntityServerQueryInterceptor* and the *EntityServerSaveInterceptor* respectively.

Each pipeline consists of "segments" represented by a virtual method that performs some function on the path from request to response. You can supplement or even replace a segment by deriving from the DevForce interceptor class and overriding methods and properties.

Query pipeline

The "Query" topic covers the *EntityServerQueryInterceptor* in detail. Here is a summary of the four methods you can override.

Method	Typical Usage
AuthorizeQuery	Determine if the user is authorized to perform this particular query.
FilterQuery	Modify the query, perhaps adding filters and limiting the size of the potential results.
ExecuteQuery	Control what happens immediately before and after the query is executed. Call the base method to perform the query.
AuthorizeQueryResults	Inspect the entities and query data by-products that are about to be sent to the client. This is your chance to reject a query that escaped earlier filtering and managed to produce data that should not be sent to the client.

Outside of the query pipeline you can define a catalog of query filters, organized by entity type. DevForce inspects the incoming query and applies these filter automatically to constrain the types mentioned in the query.

Save pipeline

The entities-to-be-saved arrive at the front of the save pipeline as entities in the cache of a server-side *EntityManager*. You are free to use this *EntityManager* to query for additional entities. You can add, modify, or remove entities. The changed entities that are still in cache when *ExecuteSave* is called will be the entities actually saved.

The "Save" topic covers the default pipeline *EntityServerSaveInterceptor* class in detail. Here is a brief summary of the methods you can override.

Method	**Typical Usage**
AuthorizeSave	Determine if the user is authorized to perform this save. The base implementation walks all of the types involved in the save and calls the *ClientCanSave* method to determine if the user is trying to save any unauthorized types.
ClientCanSave	Determine which entity types this user is authorized to save.
ValidateSave	Add or remove server-side instance validation rules; then let the base implementation apply the rules to each entity-to-be-saved.
ExecuteSave	Control what happens immediately before and after performing the database save. Call the base method to perform the save.
OnError	Could log the errors, something the base implementation does not do.