Contents

DevForce client's retrieve and save data by communicating with a middle tier application server in the form of **entities**.

There are different paradigms for storing and communicating data between a client and a data source. The entity-oriented paradigm is the most common and the one favored by DevForce.

In the entity-oriented paradigm, raw data are transformed into objects, entities, that are easy for object-oriented developers to work with.

Developers query for entities, send them across the network, display them, change them, validate them, and save them back. Entities are rich packages of data and functionality, well equipped to participate in back-end processing and support a UI.

RIA application developers favor entity-orientation for good reasons. Recall that the development team is building the application end-to-end. The team is responsible for all code that executes on both the server and the client. That's an opportunity and a burden. The opportunity lies in the full control over everything that happens between the database and the glass. The burden is covering that ground efficiently, without getting hung up in numerous transformations on route. We'd like to worry as little as possible about how the data are transmitted and stored.

Entity-orientation promotes a common representation of data as entities on server and client while hiding the implementation details. It is almost as if the client and server were the same and the entities were capable of performing all tasks required of them in either environment.

Architects are justly suspicious of multi-purpose classes with numerous capabilities. Such classes can be bloated, difficult to test, and hard to maintain. Architects are also sensitive to the fact that server and client are not actually the same: the responsibilities are different, the technologies are often different, and the two environments are separated by a slow network.

The challenge for an entity-oriented framework is to keep the entities light and testable and to handle the mechanics of server/client communication gracefully, affording the developer adequate opportunity to intervene as necessary.

DevForce is up to that challenge.

Conceptual entity

As architects, we should take a moment to look at the entity conceptually. An entity represents a thing that is important in the domain of the application, a thing such as a customer. Our representation of that thing includes its attributes, the facts that we assign to it and call entity data. But it is certainly more than its attributes. In fact, we could change almost every fact about that entity – almost – and it would still be the same entity. We can change where it is headquartered, when the company was founded, who is the chief executive, even the customer name … and it would still be the same customer. The one fact we can't change is its identity. An entity is distinguished by its identity, the something that reliable tells us that we are working with the same customer no matter what else we do to it.

Identity is what makes an instance of an entity unique … across time, across representations. A given customer is the same customer when we create it, when we change it, and when we delete it. It's the same customer whether it is a record in the database or an object in memory. Eric Evans speaks of this lyrically as "the thread of continuity" on page 89 of his book, Domain Driven Design, which we recommend highly.

For our convenience, we select one fact about the entity that does not change, the key to its identity. We call it the key. The best keys are the single values that have no intrinsic meaning. They're integers or GUIDs – some value we'll never be tempted to change because the permanence of that value helps us readily identify the entity wherever it goes, whatever happens to it.

We'll meet the conceptual entity again when we discuss the Entity Data Model (EDM).

Entity classes

To the practical programmer, an entity is a .NET class. Like any class, it has a type (*Customer*) and properties. Because an entity represents something meaningful in the application domain, something whose values are preserved indefinitely in a data store, we can be more specific about some of its characteristics:

Persistent data properties

Most entity properties get and set data that reside in a persistent data store such as a relational database. Such properties could be simple values (e.g., "Name"), complex properties made up of multiple values (e.g., "Address"), or foreign key properties ("RegionID") that "point" to related entities.

Key

The property (or properties in a compound key) that determines its identity; the property whose value distinguishes one instance of an entity type from another. The property of a single-element key is often called the identifier.

Navigation properties

Properties that return a related entity or a list of related entities. An Order entity likely has a reference navigation to its parent Customer and a collection navigation to a list of the order's line items.

Business logic

The object behavior added to a class that implements the domain-specific aspects of the entity such as validation rules, calculations, convenience properties, state transformations, and events.

We almost always talk about an entity as a class because that's the form it takes when we program with it in a .NET language.

Most DevForce developers generate their entity classes from an Entity Data Model prepared with the Entity Framework EDM Designer.

Persistence

That persistence word shows up … well persistently … when we talk about data. It refers to data we retain beyond the lifespan of an individual user session. It refers to data we share with other users of the system, be they human or machine. These are data we'll want to see again later so we save and retrieve them from a "persistent data store", a database, usually a relational database.

The opposite of persistent data is transient data, the kind of data that are accessible only to the current user and only for the life of the current session. The "state of the entity" – whether it is new, modified, or unmodified – is a transient value that changes locally while the user manipulates the entity in memory. You don't save the entity state in the database.

On the other hand, persistent data aren't permanent. They can change; that's why we call them persistent data, not permanent data.

If you catch us saying "persist" the data, we just mean "save to the database".

Navigation, related entities and entity graphs

We wouldn't bother with entities if our application only had one entity type. If we cared about customers and nothing else, we should simply get and save customer data without all the fuss.

Applications that are rich in data have lots of entity types – tens or hundreds of entity types – often grouped into clusters that revolve around a coherent application "facility" such as managing customer relations, taking orders, and shipping products.

These clusters are called domain models because each cluster of entities – the things that matter – constitutes a model of the domain covered by the application facility.

All (or almost all) of the entity types within a domain model are related to at least one other entity. Consider an Order entity. It probably has a parent Customer entity that placed the order. It has child LineItem entities representing the specific items purchased on the order. Each line item is related to a Product entity which in turn has a Manufacturer. The order was sold by a SalesPerson, yet another entity.

In this toy story, Order is already caught in a web of relationships with five other entity types. We should expect many more entities and relationships in real order management system.

The Order entity (viewed from its own perspective) is the root of an entity graph whose nodes are the other entities connected to it along the "edges" defined by their relations to one another.

An Order entity class has navigation properties that lead from it to immediately adjacent entities. It has a Customer property that returns its parent Customer, a SalesPerson property to return the entity of the person who sold the order, and a LineItems property to return the list of line item entities denoting what was sold. A LineItem entity class has a Product property to return a Product entity … which has a Manufacturer property to return the Manufacturer entity.

A program can traverse the Order entity graph by following a succession of navigation properties. That's easiest to illustrate when we traverse from child to parent as when we go from LineItem to Order to Customer.

You might code that traversal as aLineItem.Order.Customer

The relationships between entities are bidirectional. Orders have line items and line items have orders. But Navigation properties are unidirectional, traveling from one entity type to another (or to a list of other entities). The Order's Customer property goes one way: from order to customer. The Customer entity might have its own navigation property, an Orders property following the relationship back to the list of its orders.

We say "might" because it isn't necessary to define a navigation in both directions. An Orders property for Customer seems like a good idea. We can't know for sure until we understand the domain better.
On the other hand, imagine the relationship between Products and Colors. Navigating from Product to Color makes obvious sense. Navigating from a color to all products of that color – from "green" to all green products – seems less useful and could perform poorly if the catalog were very large and the number of colors very small.

Entity business logic

At minimum an entity has an identity (a key property) , data properties returning values stored in a database, and navigation properties returning related entities. You could generate these properties semi-automatically simply by examining the target database schema.

Generated entities are not application ready. They carry data but they don't capture the business rules that constrain data to legitimate values nor do they exhibit behaviors that make them useful in the domain.

Suppose my database has a Customer table. It has a Name column defined as nvarchar(50). We can match that with a Customer entity class that has a Name property returning a string. But in our application domain there are additional constraints. The name is required and only authorized users can change the name after it has been created.

These are validation rules that only we developers could know. They can't be determined from schema. We will add them to the Person entity by hand perhaps by decorating properties with validation attributes or adding validation rules in code … all easy to do in DevForce.

A name change could be a significant event in our domain. Changing the property of a DevForce-generated entity raises the PropertyChanged event which an external object – a user control for instance – might listen to. Perhaps the domain requires a more definitive response such as sending an alert to a supervisor when someone changes the name. That's domain business logic you might add to the Customer entity. You could use DevForce property injection to embed that logic in the Name property setter itself.

We may have other expectations of an entity that are not satisfied by the entity data. Convenience properties can calculate useful values such as the "age" of a person from the birth date or the "invoice total" from the line items of an order. We can add methods to update an entity graph such as an *Order.AddLineItem()* method. We can add methods that delegate to external services such as *Customer.GetCreditRating()*.

Business logic transforms an entity from a mere bag of data into a meaningful application contributor. Learn more about adding logic to your entities in the Modeling topic.