Contents

Introduction

DevForce is a framework for building and operating multi-tier, data-driven enterprise applications.

By "*enterprise application*" we do not mean simply a big application or an application for a big company. Rather, we refer to an application with the following specific characteristics:

- Its users devote many hours to its use, performing tasks essential to conducting the organization's business;
- It requires a rich and responsive graphical user interface, dense with sophisticated controls;
- User interactions are complex; task and context switching is common;
- It presents data that are complex in themselves, and deeply interrelated;
- The data are stored centrally and shared with other users.

Supply chain, customer relationship management (CRM), and asset-tracking applications are typical examples.

User productivity is critical. That puts a premium on the application's ability to provide a highly responsive, richly featured user experience – the kind of experience typical of a desktop application running directly on a client machine.

We expect people to get work done at anytime from anywhere. Those people may be employees or they may be valued partners. In either case, security matters. Accordingly, we often need to deploy and operate enterprise applications over a wide area network – preferably over the internet – with undiminished productivity and security.

DevForce is especially suited to building and running applications that require a rich user experience delivered to remote, Internet-connected clients.

While DevForce contributes at many levels of the enterprise application architecture stack, its Data Services, Object Relational Mapping (ORM) technologies, and object-oriented approach to data management draw most of the attention.

Microsoft has stepped into this arena with the Language Integrated Query (LINQ) and the ADO.NET Entity Framework. The Entity Framework is a robust ORM solution that brings ORM to the mainstream and improves productivity and type safety with its use of LINQ (Language Integrated Query). Developers can retrieve data as "entities" by writing "LINQ to Entities" statements in her preferred .NET programming language.

DevForce delegates to the Entity Framework the mapping between object and relational database schemas, as well as the database persistence operations (queries and saves). These are important and challenging tasks that the Entity Framework handles well.

There is much more to an application than how it handles raw data. There is the business object layer that encapsulates the data and governs those data with business rules. There are higher layers that address the application workflow and user experience. All of this is outside the purview of the Entity Framework.

If we concentrate only on data management, we still find enterprise application requirements untouched by the Entity Framework. Chief among them are:

- N-tier, internet, and Silverlight support.
- Proper support for a business object layer with business rules, validation, and MVVM decomposition.
- Highly responsive client UI's that exploit caching (and compression) to avoid redundant, slow trips across the network.
- Centralized services, server-side events, and performance, scalability, and security.
- Distributed transactions and entity models mapped to multiple data sources.
- POCO support for mapping to non-relational data sources.
- OData support for exposing entities to non-.NET applications.

DevForce satisfies these requirements even as it relies on the Entity Framework for basic object mapping and query facilities. The key components of DevForce include:

- the *EntityManager*, which includes a queryable client-side cache;
- the *EntityServer* for services in a middle tier;

- a provider for the LINQ language that permits LINQ queries to be used with both the client-side cache and remote data sources
- *object* m*apping enhancements* which extend the Entity Framework designer and generate DevForce entity code;
- business object enhancements such as dynamic validation/verification, localization, object lifecycle events, binding support, and other extensions that simplify your code.

This chapter explores the key data management issues for .NET enterprise application developers. It introduces the LINQ and the Entity Framework, explaining what they do and where they leave off. It then describes how DevForce fills in the critical gaps.

The Problem

Every business application is an extended dialogue between a user and the business objects that fulfill the application's purpose. Those business objects are behavioral objects first and foremost. They are the embodiment of the customer stories that describe what the application does and how it does it.

A few behaviors may be stateless; financial calculations come to mind. But there is usually data somewhere in those business objects. An order has a customer and a delivery date and line items describing quantities of goods sold for a price. There is no escaping the data aspect of business objects and all of that data must be managed.

While the application is running, the data are held in session in some form. In an object-oriented system they are held in fields and exposed as properties of a class instance. But because the data are long-lived – longer-lived than any one session – they have to be saved between sessions. And because we share our data with others, we have to save the data in permanent storage accessible over a network. Shuttling data between storage and the application session is one of those necessary but "dirty" jobs, a job completely unrelated to the application's purpose.

Developers long ago discovered three data management problems.

First, the way we store data is not the way we use data in an application. Money, for example, is both an amount and a currency (dollars, euros). The two aspects require separate slots in storage; from the application perspective, it's just one thing: money. An "order" in the context of an application session may be seen as one "thing" with a customer, a shipper, line items, etc. When we store that order in a relational database, the order, customer, shipper and line are five different things. So the best representation of stored data often is not the best representation for session data.

Second, session data are governed by rules. We must know the customer for an order before we can deliver the ordered goods. The date of the order should precede the delivery date. Some other part of the application may need to be alerted when the order is actually delivered. The application is more maintainable and easier to understand when the rules (behavior) and the data are bound together as "business objects" or "entities". Such rules are largely irrelevant when the data are tucked safely away in storage.

Third, there are many mechanical matters surrounding saving and retrieving data that have nothing to do with the application's purpose such as opening and closing connections, composing SQL, detecting concurrency violations, converting raw data into Data Transfer Objects, and managing transaction boundaries. Getting the application dialogue right is hard enough without these distractions. Yes, the application still has to ask for data and stow them away. But there should be a way to express our intent simply and entirely in terms of the application entities. Ordinary operations should make no mention of databases, connections, tables, or columns.

The profound differences between stored data and session data lead developers to expend enormous energy moving and translating between stored and session representations. This is wasted energy from the perspective of the application customer who could not care less about our implementation problems.

It is also wasted energy from the developer's perspective because this problem has been solved by object mapping technology.

Object Mapping Technology

An object mapping technology maintains two views of the data. There is a conceptual model for representing the data within the entities used by the application and there is a storage model that defines how the data are stored in the repository. These two models have completely different characteristics, as we have seen. The conceptual model could include a conceptual order, an order entity, as it is understood by the application. The storage model describes how the order entity's data values are held in the data repository.

If the repository is a relational database, many of the order entity data values – its state – are likely held in columns of a table. The value of a DeliveryDate property of an Order entity might be stored in the [DeliveryDt] column of an [OrderHeader] table row. The correspondence between the conceptual order entity and the table row is obvious and strong in this example. Even so, the correspondence is not literal; there is Order and DeliveryDate on one side; OrderHeader and DeliveryDt on the other.

Therefore, the object mapping technology maintains a "map" of the correspondence between entities of the conceptual model and the table rows in the storage model so that it can transform one representation into the other.

The Order entity has a related Customer entity and related OrderDetail entities. These additional entities might correspond to Company and OrderLineItem tables in a relational database.

Relational database tables don't have relationships. They have foreign key constraints that imply these relationships. Accordingly, the object mapping technology also maintains a map of the associations between entities and the foreign key constraints in the database. The map records the pairing of the relationship between Order and Customer with the foreign key constraint between the OrderHeader and Company tables.

This order example is especially simple. Other mappings could be enormously complex, with values changing shape (type), entities splitting among multiple tables, and relationships weaving through intermediate association tables.

Without an object mapping facility, the application developer would have to be constantly aware of these correspondences as she wrote instructions to retrieve and save application data. Small changes in the actual storage schema or in the application entity model could easily break the code in a hundred places.

Without an object mapping facility, the application would become vulnerable and brittle as it grew and aged. Productivity would fall as developers devoted increasing effort to keeping the conceptual and the storage models aligned.

The Microsoft ADO.NET Entity Framework

The Microsoft ADO.NET Entity Framework is quickly becoming the standard for database access in .NET applications. DevForce builds upon the Entity Framework, so we introduce the Microsoft technology here before explaining DevForce's added value.

Read more about the Entity Framework at http://msdn.microsoft.com/en-us/library/bb399572(VS.100).aspx

Entity Data Model (EDM)

The Entity Framework supports an *Entity Data Model (EDM)* that **describes** data from the application perspective.

The EDM does not include the actual business object classes that contain those data; rather it defines certain of the data and data relationships within those classes in an implementation-agnostic language of its own.

Concretely, the EDM is an XML schema file that defines a conceptual data model. That schema is accompanied by two other XML schema files: one describing how the data are stored (the storage model) and another that maps the conceptual model to the storage model.

The Entity Framework uses this chain of descriptions to move data between the data-laden objects in memory and the actual data repositories. For this to work at runtime, the conceptual schema (the EDM proper) refers to entity classes of the application while the storage model gets matched up, via configuration, with a real database running on a server somewhere.

The Entity Data Model Designer

Most developers prefer to use a tool to work with XML rather than edit XML by hand. EDM XML is dense and forbidding so a tool is a practical necessity.

The Entity Data Model Designer is a Visual Studio design tool that provides the developer with a graphical, drag-and-drop EDM design experience. The designer enables simultaneous development of all three related schemas – the conceptual, storage, and mapping schemas.

Most applications are predicated on a pre-existing database. This database cannot be ignored; the conceptual model must ultimately come to terms with it. Most developers find it convenient to confront this fact early and will prefer to generate the conceptual data model and associated schemas using the Entity Data Model Wizard. The wizard produces the EDM schemas which then can be viewed and edited in the designer.

Entity Object Layer

The Entity Framework business object layer consists of the classes that implement the application business objects.

The Entity Framework includes an entity class generator that uses the EDM to produce class code that defines the business object data fields and their accessor properties. It also generates the navigation properties that enable the application to traverse from one object to its related objects (e.g. from an order to its customer).

The EDM describes only the business object data and their relationships. The Entity Framework knows nothing about the business object *behavior* that applies to the data so there is no business logic in the generated code. The application developer writes business logic separately in a companion class file. The two files – the developer's business logic file and the generated object data management file – combine to form a single definition of the business object, the business object class.

Technically, each file defines a .NET *partial class*. The compiler knits the two together, resulting in the complete business object class.

Entity Persistence

The Entity Framework includes components responsible for moving business object data between the application and the database.

The *ObjectContext* is the most visible of the components. The application uses ObjectContext to retrieve, hold, and save entities. The ObjectContext maintains a cache of all the entities it manages. The developer writes queries and submits them to the ObjectContext, which retrieves the selected entities and adds them to its cache before returning them to the caller. The developer creates new objects and adds them to the ObjectContext. The ObjectContext tracks changes – adds, modifications, deletes – to entities in its cache. A save command tells the ObjectContext to write the changed entities to the database.

The Entity Framework handles all of these relational database persistence operations without troubling the developer with details. The Entity Data Model and a few guiding parameters are all it needs.

LINQ to Entities

Earlier we described three problems for the developer who needs to represent data in the application as business objects. The third problem was how to retrieve and save business objects using a language that hid the underlying mechanisms and stayed true to the entity-oriented paradigm.

While the mechanics of saving business object data are challenging, it has never been difficult for developers to express their intent. It is usually sufficient to tell some service class to "save" and the service knows what to do.

Getting data is a different story. It is not easy to say precisely which data you want, and in what form, using a general purpose programming language. It's harder still to write queries in a strongly-typed manner and stay within an entity-oriented paradigm. Until recently, object mapping vendors offered their own "object query languages" (OQLs) which were, in fact, merely special purpose classes with strangled interfaces. OQL queries were clumsy to write and repugnant to read.

With its release of the .NET 3.5 Framework, Microsoft added new language facilities for finding and accessing data in a general purpose, object-oriented way, without exposing the details of data storage and retrieval. Chief among the new features is LINQ, an abbreviation of *Language Integrated Query*.

A LINQ query looks much like an SQL query. Most programmers have long experience with SQL so, while SQL itself may be tortured, most programmers are accustomed to it and find LINQ expressions familiar:

```
IQueryable<Product> products =
  from prod in anObjectContext.Products
  where prod.ReorderLevel > 100
  select prod;
foreach (Product aProduct in products) {
 //…
}
```

```
Dim products As IQueryable(Of Product) = _
  From prod In anObjectContext.Products _
  Where prod.ReorderLevel > 100 _
 Select prod
For Each aProduct As Product In products
 '…
Next aProduct
```

LINQ defines a set of query operators for interrogating arbitrary sources of data. Anything that can be enumerated can be queried with a LINQ expression. We can use LINQ to select items from a list, nodes from an XML file, file names from a file folder, or records from a database.

LINQ itself does not know how to do any of these things. LINQ defines the *query operators* and *patterns* for writing query expressions. The operators and expressions are meaningless until they are married to an implementation that is specific to a domain. Thus there is a LINQ implementation for querying in-memory objects (LINQ to Objects), an implementation for querying XML structures (LINQ to XML), an implementation for querying relational databases (LINQ to SQL), and so on. Microsoft provides some of these implementations but third parties can develop their own and Microsoft encourages them to do so.

The LINQ facility provides the expressiveness we need for querying entities. What we need is a LINQ implementation that supports an object mapping technology. Microsoft's LINQ to Entities is that implementation for the Entity Framework.

Entity SQL

The Entity Framework supplements *LINQ to Entities* with its own query language called *Entity SQL*. Entity SQL is a storage-independent dialect of SQL that works directly with the conceptual model. An Entity SQL query refers to entities, properties, and associations (e.g. Order and Order_Customer) rather than the database elements in the storage model. The particulars of data storage remain hidden in the object-oriented data design.

Entity SQL queries are strings as seen in this example:

```
string queryString = @"SELECT VALUE Product FROM Products " +
 "AS Product WHERE Product.ReorderLevel > 100";
ObjectQuery<Product> products =
 new ObjectQuery<Product>(queryString, anObjectContext);
foreach (Product result in products) {
 //…
}
```

```
Dim queryString As String = "SELECT VALUE Product FROM Products " _
  & "AS Product WHERE Product.ReorderLevel > 100"
Dim products As New ObjectQuery(Of Product)(queryString, _
  anObjectContext)
For Each result As Product In products
 '…
Next result
```

One significant drawback: Visual Studio will not detect even simple mistakes because the query string won't be evaluated until runtime.