

Contents

- [Why Code First?](#)
- [Mapping friction](#)
- [When you can't use Code First](#)
- [Why Database First?](#)

A [2011 survey](#) showed application developers about evenly split between those who would develop their entity models in Code First style and those who would use Database First. Apparently there is no “right way” to build your entity model.

We see advantages and disadvantages to both styles.

Why Code First?

Code First is especially popular in the early stages of a brand new project (the so-called “green field” scenario).

The primary focus is getting the data and logic right. You're usually just trying to find your way, trying to understand the application requirements and model those requirements in working code. You're not ready to lock in the data structures. You're not ready to settle upon a database schema and it's too soon to worry about database optimization.

You care most about getting something working to show your “client”. You want rapid feedback and you want to iterate quickly to show the “client” that you're being responsive.

Ideally you don't have to be concerned with existing data and schemas; you are working with test data and are free to create and destroy databases at will.

You're also working with only a small number of entity types, perhaps fewer than twenty.

You favor fast, short iterations. You want to minimize the productivity drag from sources that seem irrelevant to the process ... such as maintaining a relational database and mapping entities to tables.

Code First is ideal in these circumstances. You don't create or update database schemas in the early phase of the project. You let the Entity Framework create (and recreate) the database for you.

You don't run the Entity Framework visual design tool, neither to design entities (Model First) nor to map entities to an existing database (Database First). There is no XML file – no EDMX file – to represent your model and storage schemas. You aren't generating entity classes from an EDMX.

You just work with the small, simple entity classes that you wrote yourself, changing them as you go, unencumbered by external tools, XML files, or databases. It's fast, it's clean, and it's transparent.

Entity class simplicity is another advantage of Code First. In Code First you always write the class yourself. Your class has only the code that you decide is necessary and appropriate. You write the class members in the style you prefer and in the order that makes sense to you.

Best of all, you typically put all of your class code in one file. You don't need partial classes to separate generated code from custom code – as you would when generating entity classes from an EDMX. You don't need that hacky [metadata "buddy" class](#) to define attributes that actually belong on the generated properties.

Mapping friction

The day comes when you must commit to a real database that holds production data, perhaps a database optimized for performance and structured accordingly with some degree of difference from your idealized model. You won't be able to throw that database away or change it at will any more.

Fortunately, you can continue development in CF style, especially when adding new entities backed by new tables.

You will have to accommodate your model to the schema as it actually is. You'll have to map your in-code entity model to database objects manually. With luck, your model and the database schema are well aligned so that most of the mappings can be inferred by [convention](#); you needn't move a muscle when there's a “Name” column in the “Customer” table to match the Name property of your Customer entity. Where model and database disagree, you'll have to specify their correspondences explicitly with [attribute annotations](#) in your entity class code or by configuration with the [Entity Framework Fluent API](#).

The application grows as it matures. You'll add more tables and more columns to those tables. You'll find it increasingly difficult to manage the mapping between your .NET entity model and the database as it actually is.

The more entities and entity properties you support, the more mappings you must maintain. Convention-based mapping can help cope with the increased volume. But you won't always control the evolution of the schema. Other developers will add their tables and modify “your” tables with the new columns. You'll need to specify more mappings by configuration using attributes or the fluent interface. That code becomes unwieldy and obscure.

Unfortunately, the database can easily grow out-of-sync with your .NET class model. You may detect mapping mismatches with automated integration tests ... if you have them ... and you really should have them. In practice, mismatches usually reveal themselves in runtime failures: obscure exceptions and unexpected behavior that is frustratingly difficult to diagnose.

Code-based mapping of entities to database is the Achilles heel of Code First. The Entity Data Mapping (EDM) tool in Visual Studio does a better job of detecting and analyzing the correspondence between entities and databases. It's easy to keep the database schema and entity classes in sync when the classes are generated from the EDMX.

When you can't use Code First

The current release of Entity Framework Code First lacks several of the features included in its EDMX-driven approaches. Two of the most important omissions are support of [Stored Procedures](#) and [Defining Queries](#). There are workarounds (see comments in [this StackOverflow post](#)) but if you depend heavily on either feature, you may want to stick with the EDMX-based workflow.

Why Database First?

Database First is popular in so-called "brown field" scenarios with developers who build applications that access existing production databases holding large numbers of tables and columns.

The entity model must conform to the existing database; you can't bend the database to suit your preferred class model. You can't on a whim create, destroy, and morph the database with its precious cargo of production data. You'll have to consider the implications of schema changes on other database users and on the IT professionals who will have to propagate your changes across the many instances of your database both within the organization and among product customers. The database – as it actually exists – is an unforgiving, constant presence in your thinking about the model.

Existing enterprise databases tend to have many tables, each with many columns. They are rarely shaped as you would prefer. The tables and columns often have names that defy the Code First conventions. If you choose the Code First approach, you'll have to write and maintain mappings by configuration; there may be hundreds or thousands of such mappings.

The Entity Framework EDM tool does a fine job of reading the database schema and suggesting the corresponding entity definitions. It's easy to revise those definitions visually in the designer. The EDM tool can detect when the database has changed and tell you which entity definitions and relationships no longer correspond to the revised schema. In many cases, the tool can correctly update your entity definitions to reflect schema changes. If you generate entity class code from your EDMX – as most Database First developers do – you can expect to keep your .NET entity model and database in sync with comparative ease.

You pay a price for this convenience. You are dependent upon an XML representation of your entire Entity Data Model. Every EDM change, no matter how small – a property rename, a new entity addition, a new relationship, the removal of an entity - requires editing that XML file.

The entire XML file is at risk. Only one person can edit it at a time. If anything goes wrong, you'll have to revert all session changes and restore from a backup (you do have a backup, right?).

The entire process is burdensome relative to the complexity of most update tasks.

Suppose you're adding a new "DbA" ("Doing business as") string property to the "Customer entity", one of 300 entities in your model. You launch the EDM Designer, launch the "Update from Database" wizard within the designer, wait for the wizard to analyze the differences between your model and the database schema, click "Finish" and wait for the wizard to update the design canvas with the new Customer property, find the Customer entity on the canvas or in the "Model Browser", confirm that the new "DbA" column was discovered and a corresponding property was added, revalidate the entire model, save it ... which regenerates every one of your entity classes.

A thousand things could go wrong. Suppose someone renamed the "Color" column in the "Product" table to "Colour". That will be a problem for someone. It shouldn't be your problem right now as no part of your application module concerns products or colors. Unfortunately, the EDM tool just made it your problem because the Product entity (about which you know nothing) now has both the old "Color" property and a new "Colour" property, only one of which is mapped to the database.

Compare that workflow to the process for adding "DbA" to "Customer" in Code First: open the Customer class file, add the "DbA" property, close, save, done.

I can work on the "Customer" class while you repair the "Product" class. We're both working at the same time, checking in our changes at our individual convenience, unconcerned that the other party will hinder or ruin our own progress.

Database First shines when managing a large, complex database schema with numerous unconventional mapping requirements ... but you're a prisoner of the EDMX and the EDM Designer. Code First shines when you're making lots of small changes iteratively to a model that aligns well with your database.