Contents

- When to use it
- Learn more
  - Video

Tips for building applications that will have hundreds, perhaps thousands, of entities.

When to use it

It is impractical to build Entity Framework v.4 models with more than roughly 300 entity types. That's an unfortunate technical limitation but it shouldn't stop you.

A 2009 essay, Large EF Models, describes the Entity Framework "large model problem" and the technical options available to you for breaking them up. It's a bit dated - it talks about EF v.1 - but still useful ... and the problem is still severe in Entity Framework version 4.

We feel the EF technical obstacles should be irrelevant. We think the desire for large models is misplaced and that, even if technically feasible, you should avoid them. We feel that the best approach is to design large applications in modular fashion with several moderately sized models, one for each module "domain".

The modules of your application typically center on a particular application purpose. Domain models in such modules shouldn't be large. The point of modularity and of domain-centered thinking is to limit complexity by focusing on what the domain requires. Everyone working on the domain problem - developers, QA, business analysts - should have a shared understanding of every concept ... and every entity ... in the domain. A domain with thousands of entities is a domain no one person understands ... let alone everyone on the team.

People often worry about what they believe are common entities shared across modules. They ask "how do we make keep from duplicating the business logic for each model?" We offer the perhaps startling answer: such entities don't actually exist.

Yes, several models refer to the same real world thing - a customer for example. But each model looks at "Customer" in its own way. They each have a need to model Customer in their own way. They each need their own Customer entity.

True, they will "share" some data in common. Each Customer entity is likely to have a "Customer Name". All of the models may store the "Customer Name" in the same record in the same database table.

> Important Question: Is this wise? We don't think so but we realize that this is often out of your control. We understand that an organization wants everyone to use the same name for the Customer. We understand that, for legacy reasons, you have a canonical Customer table that supports different applications ... many of them out of your control. So while it may not be wise, Large databases with shared tables is a reality for most of us.

We invite you to consider whether all the domains actually work with "Customer" in the same way. Some domain only need read access. Some need a fraction of the Customer information while others need an entire graph of related objects. Suppose Module A needs to change some aspect of Customer. Is that change important to Module B? Should Module B even know about the aspect being changed? Why make Module B - the code, the developers, QA - vulnerable to a change that has no meaning in Module B's domain?

Of course there will be Customer changes that have a sweeping impact ... and those will have to be negotiated.

We tackle this subject in several places and invite you to explore the topic further.

In sum:

- Build several, modest models that focus on a particular problem domain
- Build applications with multiple modules, each with its own domain
- Use "messages" to communicate across modules about common things (e.g., Customer)
- Don't worry about repeating business logic across modules; in reality there will be little logic in common
- What common logic you have can be delegated to shared libraries.

Learn more

- Large EF Models - an IdeaBlade 2009 essay
- IdeaBlade Forum post on Large Models on large models
- Watch Eric Evans' talk about what he's learned since writing Domain Driven Design.

Video