Private / Public

SOLID

1-1

1-m

m-m

ORM

Fordele:

Mindre kode og mindre gentagen kode

Mindre risiko for sql injection

Manuel oversættelse:

"We can just interact only with x objects and x classes"

Kompleksitet: Da du håndterer SQL direkte, skal du forstå databasestrukturer, indexes, fremmednøgler og mange andre detaljer, der vedrører databasens funktion.

Relationship between tables -> this is where ORM becomes very useful:

Uden ORM:

SELECT \* FROM flights JOIN passengers ON flights.id = passengers.flight\_id WHERE passengers.name = 'Alice';

Med ORM: Passenger.query.filer\_by(name="Alice")\first().flight

Når du arbejder uden ORM (Object-Relational Mapping), skal du tage dig af mange detaljer manuelt:

Manuel oversættelse: Du skal konvertere data mellem dine programobjekter og databasens

Man k

The most important ORM features are

• Idiomatic persistence: By enabling to write the persistence classes

using object-oriented classes

• High performance: By enabling fetching and locking techniques

• Reliable: By enabling stability for Jakarta Persistence programmers

Codefirst

Fordele:

Code first in a new project with a new database

An entirely greenfield scenario is an obvious choice for working with a code-first

approach. Even if you don’t want to use migrations for some reason, at some point you

will still need data models that define how to work with the various database objects in

order to use Entity Framework and LINQ to EF (LINQ stands for “Language Integrated

Query” and is covered in more detail later in the book).

Since a greenfield project is new and has a new database to accompany it, using code

first will provide the best flexibility and ease of use from your codebase. In this case, it

only makes sense to use the code-first approach

Negative:

A final reason to avoid using code-first could simply be that there is a high risk of

losing data in a mature database (this is no different than the risk that could exist from

running any database script that drops columns or tables; it’s not that EF code first is

just going to randomly lose data).

With a database that is already mature, you need to be

protected from accidental changes that might truncate data from tables or break critical

performance enhancements (such as a change dropping a view or an index might do).

This is especially important if other legacy line-of-business applications are relying on

these original data structures for normal operation

The way the whole configuration is defined depends on the specific ORM. Entity Framework Core offers three options: • • • Data annotations (property attributes) Name conventions A fluent configuration interface based on configuration objects and methods While the fluent interface can be used to specify any configuration option, the data annotations and name conventions can be used for a smaller subset of them.

A final reason to avoid using code-first could simply be that there is a high risk of

losing data in a mature database (this is no different than the risk that could exist from

running any database script that drops columns or tables; it’s not that EF code first is

just going to randomly lose data).

Databasefirst

Fordele:

Negative:

Relations among tables are represented with object pointers. For instance, in a one-to-many relationship, the class that’s mapped to the one side of the relationship contains a collection that is populated with the related objects on the many side of the relationship. On the other hand, the class mapped to the many side of the relationship has a simple property that is populated with a uniquely related object on the one side of the relationship.

Simpelt: using a class to represent a table

"We can just interact only with x objects and x classes"