Databases for developers Synopsis

Table of content

Motivation

This project demonstrates a hybrid database, combining a MySql relational database, and a MongoDb document database.

Relational databases are really good at managing relations. There are lots of vendors on the market, so picking one with the right feature set for your exact use case, is doable. With different methods for normalization, redundant data can be avoided, and it is easy to define a very strict datamodel that the database itself will enforce. Letting the database enforce the constraints, takes away the need for developers to enforce them, and they can focus on the interaction alone. It is language agnostic, so two different development teams can work with the same database, and not care about what language other teams choose for their development. On a final note, most relational databases are somewhat easy to begin working with, and should the need come to surface, they can be maintained by DBA's if they need additional tuning later in the lifespan of a project.

Document databases are not new, but have gained a lot of traction in the last decade or two. Usually they do not require a schema, so they can be more flexible than relational databases. Sacrificing the normalization can lead to faster interaction with the database, since all data can often be fetched with a single query. At the time of writing this, memory is cheap (compared to 20 years ago), and storage is even cheaper, so there previous economic bottleneck is not as apparent as earlier.

This project will demonstrate a Bookstore with books and customers in a relational database, and customers orders in a document database.

The argument for splitting the technologies like that is that Customers and Books have their own relations. Books have authors, Customers have addresses and so on. An order is different, although it has books it does not really need to know about the authors of the books, and if a book should be deleted from the bookstore, there can still be customers who bought that book, so it would have to live on the order anyway. Creating a relational datamodel for that is absolutely doable, but it will be more complex than what this project demonstrates.

Description

Starting from the bottom up, the solution consist of a relational database, and a document database.

Every database has a repository that deals with basic CRUD (Create, Remove, Update and Delete) operations. Which gives us a nice abstraction, making maintenance and testing easier.

Controllers talk to the repositories through a service. That gives us a service for each repository, and their responsibility is to deal with any logic there might be involved with shuffling data back and forth. At the time of writing the services deals with transforming database objects to DTO's where needed, it would also be an obvious place to implement resilience strategies.

Finally, there is one controller for the entire solution. Down the line it could be argued if it would make sense to split them up according to their data domain, but for project scope reasons there is only one.

To showcase a hybrid database, as a bit more than just to different database technologies, there been introduces an endpoint that solves a fictive GDPR-issue, and interacts with both databases at the same time

Architecture

The relational database is a MySql database. There are a few other choices that could have been made (SQL Server, PostgresQL i.e). EntityFramework has been used as the data access layer. It is widely adopted within the dotnet community, and fairly easy to work with. The database context is injected as a dependency, making it easy to provide another database provider if needed, or for testing purposes. It is also used to manage database migrations, removing the necessity for writing SQL-scripts

The document database is a MongoDb. As with the relational database there are other options, but during the course this has been used.

Models reside in the Models folder, which also provides housing for data transfer objects (DTO's). A low hanging refactor fruit, would be to separate the DTO's from the other models. At the time of writing this, it is still less than 10 models in total, and already starting to loose the overview

The repositories only depend on a database context, and deals with the same models that are used for the database migrations. There is no logic in them, they only pass data back and forth.

The services get a database context injected, and instantiates a repository. To simplify some of the communication between controllers and services, a limited DTO can be used.

```
public class CustomerModel
{
```

```
    [Key]
    [Column("id")]
    public Guid CustomerId { get; set; }
    [Column("name")]
    public string Name { get; set; }
    [Column("created_date")]
    public DateTime CreatedDate { get; set; }
}
```

Someone dealing with the API, should not care about setting the CreatedDate on a new Customer, so a simplified DTO covers what is needed to create a new Customer

```
public class CustomerModelDto
{
    public Guid CustomerId { get; set; }
    public string Name { get; set; }
}
```

With one exception. Every bit of logic is done in the service layer. The exception is the made up GDPR endpoint that deletes a Customers Orders

The endpoint takes a customer-id as input, looks it up through the Bookstore Service, and deletes the orders submitted by the Customer (with their name, which is the key in the document database)

```
[HttpDelete]
[Route("/gdpr/customers/{customerId}")]
public async  Task<ActionResult> DeleteCustomersOrders(string customerId)
{
    var customer = await _bookstoreService.GetCustomerById(Guid.Parse(customerId));
    _orderService.DeleteCustomersOrders(customer.Name);
    return Ok();
}
```

All communication is done asynchronous

Perspective

Having the orders in the document database, makes it a bit easier to handle things related to actions that pop-up based on your order. A common feature in online stores is “Buy 3 pay for 2”, and having the document as a model gives features like LINQ to deal with exceptions like that. It would be just as possible with the relational database, but the datamodel would become quite complex very fast. Another easy-to-implement-feature would be the sum of the prices in the order. Once again this could be done with SQL, but we would have to manage what document we are dealing with, and in this case we already have the document right by the hand.

If we introduced a key/value database, such as Redis, we could keep count on how many books our inventory consists of. This could also be done in MongoDb and MySql, but this would make it extremly fast to lookup, and the code would be simpler. If this Bookstore becomes the new amazon, we would have to add some functionality to deal with invoicing. Today's relational databases have good support for json-structured objects, so it would be easy to store document data in MySql (a customers historic orders for instance)

Conclusion

We have showcased that a hybrid database is possible to create. We have introduced some more complexity from the beginning (we need to database instances) and up to the controller (we need to inject two different connections) From here on we will reduce complexity because we will not have to mimic document-data in the relational database, so the relational datamodel will be kept simpler.

Having more than one database technology also gives us a few more options regarding resilience strategies. Let us image a disk runs full in the relational database cluster (this happens), Customers could still be created and have their data persisted in MongoDb, and transferred to the relational database when it becomes available again