EF Core's Advancements in DDD-Friendly Data Persistence

Entity Framework Core (EF Core) has evolved significantly in its capacity to support Domain-Driven Design (DDD) principles in data persistence. Originally, EF Core addressed two fundamental issues that Entity Framework 6 could not, namely, complete encapsulation of collections and direct mapping to fields rather than properties. Subsequent versions introduced further enhancements:

1. **EF Core 2**: Introduced "owned entities" facilitating the mapping of value objects, as well as value converters for types not inherently supported by the framework.
2. **EF Core 3**: Added interceptors for better pipeline control, and introduced support for Azure CosmosDB, a document-based database by Microsoft.
3. **EF Core 5**: Improved many-to-many relationships and refined owned entity support, resolving issues with null properties or property replacements in value objects.
4. **EF Core 6**: Provided bulk configurations to simplify complex mappings and added several new features for CosmosDB.
5. **EF Core 7 (Future)**: Promises support for persisting JSON objects, better handling of value objects for key properties, and multi-entity mapping to a single table.

Challenges and Solutions in Persisting DDD Models with EF Core

EF Core can automatically recognize and map several DDD constructs, courtesy of its built-in default mappings. However, there are scenarios where manual configurations are necessary. A prominent example involves the use of "backing fields" — a coding pattern that protects properties from unintended use. EF Core recognizes common naming conventions for these backing fields, but if the naming conventions are not adhered to, manual mappings must be specified. EF Core offers flexible mapping APIs to ensure accurate data persistence, even when the domain classes deviate from its default mappings.

By comprehensively understanding these configurations, developers can better leverage EF Core's capabilities for persisting DDD models, thereby avoiding the misconception that the framework's data persistence functionalities are inherently flawed.

### Summary

The implementation of the contract aggregate is specifically designed to enhance ease of interaction for clients. This aggregate encapsulates not just the contract, but also all related versions. One critical method allows the addition of an author to a specific version of the contract, which is the only modification permitted for the author list within that version. Additionally, the aggregate's design incorporates methods to handle pivotal lifecycle events of the contract. These methods mark milestones such as verbal acceptance and final signing by all parties, updating critical properties like `FinalVersionId`. Lastly, it is indicated that future enhancements will include an event-triggering mechanism to notify stakeholders of a new book and its associated authorship. The emphasis is placed on adhering to the aggregate design during the persistence logic to ensure data integrity and business rules are maintained.

# Adaptation of EF Core 6 for Domain-Driven Design with Azure Cosmos DB

In the evolving landscape of Entity Framework Core (EF Core), this paper delineates a specialized module that departs from traditional relational databases, such as SQL Server, to focus on aggregate mapping in Azure Cosmos DB. As document databases rise in popularity due to their ability to manage schema-less, high-volume data, developers with expertise in EF Core seek to extend their skills to this new paradigm. To address this, the EF Core team introduced a Cosmos DB provider.

#### Key Highlights

1. \*\*Unification of Aggregate Structures\*\*: Unlike relational databases, a single document in Cosmos DB can encapsulate an entire DDD aggregate, eliminating the need for EF Core to transform aggregates into a normalized relational schema.

2. \*\*Irrelevance of Some Configurations\*\*: Many configurations pertinent to relational databases become redundant when working with Cosmos DB. These are either flagged by the compiler or recognized through developer expertise.

3. \*\*Natural Data Nesting\*\*: Document databases like Cosmos DB allow for automatic nesting of value objects and related data within parent documents, a feature more naturally aligned with DDD principles.

4. \*\*Ease in Handling Collections\*\*: Storing data in documents solves issues previously encountered in relational databases, such as storing collections and data dictionaries of scalar data.

5. \*\*Quick Problem Identification\*\*: The utility of compiler warnings and integration tests makes it easier to identify and address issues, particularly those related to LINQ queries.

6. \*\*Stability of Aggregate Mappings in Future Versions\*\*: While EF Core 7 promises numerous enhancements to the Cosmos provider, these updates are not expected to substantially impact aggregate mappings.

# Leveraging EF Core for Optimized Aggregate Storage in Cosmos DB's Flexible JSON Documents

### Summary

#### Leveraging EF Core for Optimized Aggregate Storage in Cosmos DB's Flexible JSON Documents

The discourse explores the inherent compatibility of Entity Framework Core (EF Core) with Azure Cosmos DB's JSON document storage. Particularly, it examines the concept of "embedding" as delineated in EF Core documentation—a feature that is adept at exploiting the flexibility of JSON documents for efficient aggregate storage.

#### Salient Features

1. \*\*Automatic Embedding of Value Objects\*\*: In Cosmos DB, EF Core's default behavior is to automatically embed value objects and related data that meet specific criteria into the host object. This is facilitated by the absence of keys in classes defined as value objects, rendering explicit mappings like `OwnsOne` or `OwnsMany` redundant.

2. \*\*Owned Types vs. Value Objects\*\*: It should be noted that EF Core interprets these value objects as "owned types," and the benefit derived from this capability is implicit in the natural nesting within the host objects.

3. \*\*Elimination of Redundant Mappings\*\*: Due to the automatic embedding, existing mappings in the model builder for owned types, such as `OwnsOne` and `OwnsMany`, can be eliminated without affecting functionality.

4. \*\*Embedding Criteria for Related Data\*\*: EF Core embeds related data when certain conditions are met. Specifically, related data is embedded if the entity in question does not have a `DbSet` defined. In the given context, `ContractVersion` does not have a `DbSet` defined and thus, its instances will be embedded within the `Contract` entity.

5. \*\*Caveats and Resolutions\*\*: The study also notes that some previous mappings could interfere with the expected nesting of aggregates. A proposed solution is to temporarily comment out these conflicting mappings, observe the expected behavior, and then make necessary adjustments.

# Query Implications of EF Core's Aggregate Retrieval in Cosmos DB

The document examines the query-level behaviors of Entity Framework Core (EF Core) when used with Azure Cosmos DB for domain-driven design (DDD). Several points about data retrieval and query performance are discussed, underlining the benefits and limitations of using EF Core with Cosmos DB.

#### Key Observations

1. \*\*Automatic Aggregate Retrieval\*\*: EF Core automatically fetches entire aggregates in Cosmos DB, including dependent objects and value objects. This obviates the need for explicit `Include` methods in the query.

2. \*\*Advantages and Trade-offs\*\*: This automatic fetching is advantageous for maintaining the integrity of aggregates. However, it can be restrictive if a developer requires only a subset or specific components of the aggregate.

3. \*\*Ineffectiveness of Filtered Includes\*\*: As of the current state, Cosmos DB does not support filtered includes, which restricts the granularity of data that can be retrieved.

4. \*\*Projections and Their Limitations\*\*: Projections are available but come with restrictions. Scalar properties can be projected easily, but if embedded properties, like contract versions, are to be exclusively retrieved, the query has to be run as a no-tracking query. This is because EF Core's `ChangeTracker` requires the root entity for tracking.

5. \*\*Unsupported Projection Patterns\*\*: Some projection patterns commonly used in relational databases, such as filtering and aggregating embedded dependent objects, are not yet supported in Cosmos DB.

6. \*\*Future Improvements\*\*: The document also provides a link to GitHub issues indicating the enhancements that are planned for Cosmos DB support in EF Core.

# Effortlessly Storing Dictionaries and Lists of Primitives in Cosmos DB with EF Core

#### Introduction

The flexibility of JSON and the schema-less nature of Cosmos DB offer unique advantages in storing collections and dictionaries, particularly when using Entity Framework Core (EF Core) as an ORM tool. This article elucidates the effortless storage of dictionaries and lists of primitive types in Cosmos DB through EF Core, extending the capabilities traditionally seen in relational databases.

#### Key Features

1. \*\*Storing Lists of Primitive Types\*\*: Unlike EF Core's limitations with relational databases, where storing a list of scalar values is complex, Cosmos DB allows you to easily store lists of certain primitive types such as `char` and `int`.

2. \*\*Storing Dictionaries\*\*: Cosmos DB and EF Core facilitate the storage of dictionaries, although there's a caveat: only dictionaries with `string,string` type pairs are supported.

#### Case Study: Author and Book Management

A new feature was introduced in the domain to track social media accounts of potential authors. Two key changes were made to the domain model:

- An `enum` was introduced to specify different types of social media accounts.

- A `Dictionary<string, string>` property, named `SocialMediaAccounts`, was added to the `Author` value object to store the account type and account ID.

Note: The `enum` exists for semantic clarity, but the dictionary stores strings.

The absence of explicit mapping for these new features in the `DbContext` class highlights the simplicity with which EF Core and Cosmos DB can handle such structures.

#### Discussion and Conclusion

This feature-rich flexibility around collections and dictionaries, which often requires cumbersome workarounds in traditional relational databases, comes as a built-in feature with Cosmos DB when used in conjunction with EF Core. This makes Cosmos DB a compelling choice for certain domain-driven design (DDD) scenarios where the ability to naturally persist complex structures can significantly simplify the architecture and coding effort.

The robustness of the feature has been confirmed through a battery of tests, instilling confidence in the ability of Cosmos DB and EF Core to seamlessly handle collections and dictionaries, thereby streamlining the development process.

# Implementing Contract Search: A Design and Development Approach

#### Abstract

Contract search functionality is an intricate problem that can be cumbersome to implement when only using Entity Framework Core (EF Core) and domain aggregates. This article presents a multi-layered solution for contract search. By leveraging database capabilities, we enhance the performance and maintainability of the application. Furthermore, we incorporate Domain-Driven Design (DDD) principles to structure the implementation.

#### Introduction

Contract search is a non-trivial operation due to the complexity of the contract aggregate and its associated DbContext. Given the limitations of querying with EF Core alone, the optimal approach is to create a separate model, DbContext, and service that can manage search queries.

#### Solution Architecture

##### 1. The Search Model

The system uses a specialized model named `SearchResults` that encapsulates the result set from various stored procedures.

##### 2. DbContext - `SearchContext`

This DbContext is read-only and configured not to track any changes, thus optimizing resource utilization. It also defines a DbSet of type `GuidKeyAndDescription`, which is used to populate the drop-down lists in the user interface.

##### 3. Stored Procedures and Views

Each search filter has an associated stored procedure in the database. A database view exposes the necessary columns for search queries, enhancing modularity and maintainability.

#### Implementation Details

1. \*\*Search Services\*\*

- `GetContractPickListForAuthorLastName`: Accepts an author's last name or its initial characters and retrieves corresponding contracts.

- `GetContractPickListForInitiatedDateRange`: Filters contracts based on a given date range.

2. \*\*Reusable Components\*\*

- A generic class named `GuidKeyAndDescription` captures the results and lives in the shared kernel.

3. \*\*Database\*\*

- Stored procedures are defined for each filter, encapsulating the SQL logic for search queries.

- A scalar function generates the contract description, ensuring consistency across queries.

4. \*\*Testing\*\*

- Unit tests and integration tests validate the search functionality.

#### Synchronization between DbContexts

Both the `ContractContext` and `SearchContext` share the same underlying database. While `ContractContext` is responsible for database migrations and CRUD operations, `SearchContext` serves purely for read-only operations.

#### Evaluation and Conclusion

This multi-layered architecture achieves a separation of concerns by decoupling search functionality from the primary contract model. The database-centric approach optimizes query performance, allowing for highly extensible and maintainable code. Through comprehensive tests, the solution's robustness is validated.

By integrating this search mechanism into an API, one can seamlessly combine it with the existing contract service, thus offering a holistic contract management solution. This approach reflects the application of DDD principles to solve a complex problem efficiently and effectively.