
agap2IT
TRAINING CENTER
Domain Driven Design
Module 5
Strategic Design meets Implementation

Domain Driven Design

Module Overview

Persistence and Event Sourcing

CQRS (Command Query Responsibility Segregation)

Integrating Bounded Contexts

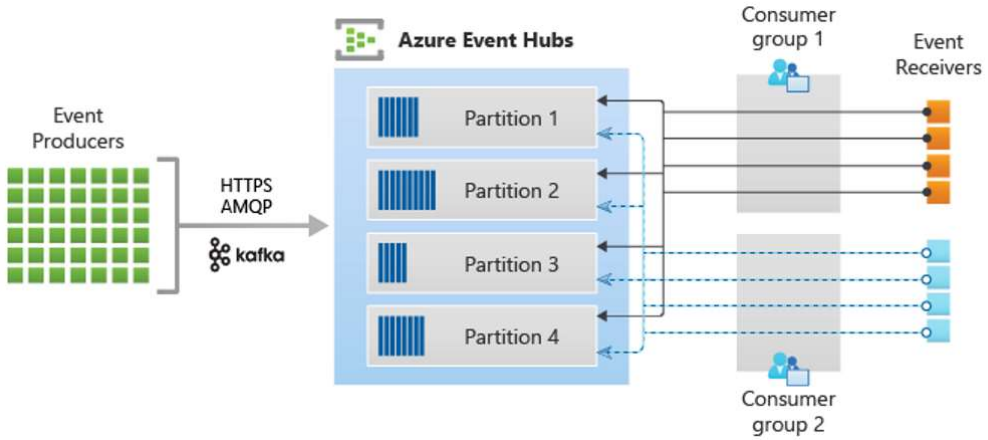
Persistence and Event Sourcing…

“Persistence and event sourcing are two important concepts in the context of domain-driven design (DDD). They provide approaches for designing and implementing robust, scalable, and maintainable software systems.”

“Persistence refers to the mechanism of storing and retrieving domain objects to and from a persistent storage medium, such as a database. In DDD, the goal of persistence is to ensure that the state of domain objects is preserved over time, allowing them to be reloaded and manipulated as needed. To achieve this, persistence is typically implemented using an Object-Relational Mapping (ORM) framework or directly with a database access layer.”

...Persistence and Event Sourcing...

“Event sourcing, on the other hand, is a pattern that focuses on capturing and storing the full history of changes (events) that occur within a system. Instead of persisting only the current state of an object, event sourcing persists a sequence of events that have led to the current state. These events are typically stored in an append-only log, known as the event store. By replaying these events, the system can reconstruct any past state of an object, providing a complete audit trail and enabling temporal querying.”



...Persistence and Event Sourcing...

When applied together, persistence and event sourcing can bring several benefits to a DDD-based system:

Historical transparency: Event sourcing captures the full history of changes, enabling the system to provide a detailed audit trail. This can be useful for debugging, compliance, and forensic analysis.

Temporal querying: By replaying events, it becomes possible to query the system at any point in time. This capability is particularly valuable when dealing with temporal data, as it allows for historical analysis and reporting.

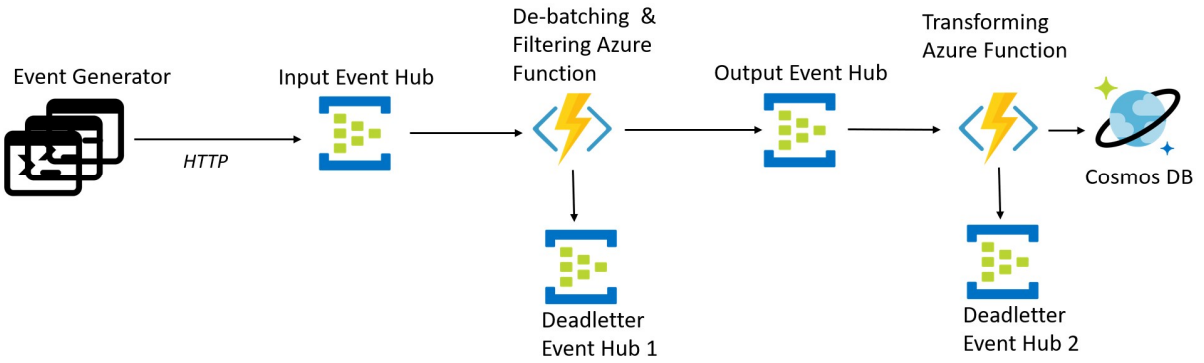
Scalability and concurrency: Since events are append-only, event sourcing can provide excellent write scalability and concurrency, as there are no contention issues when multiple components or users update different parts of the system simultaneously.

Evolving business rules: With event sourcing, it is possible to introduce new business rules or modify existing ones by projecting events into different read models or transforming events into new representations. This flexibility allows for system evolution without breaking existing data structures.

Collaboration and bounded contexts: Event sourcing encourages the exploration of bounded contexts within a domain. Different teams can focus on specific events and project them into their own read models, fostering better collaboration and autonomy.

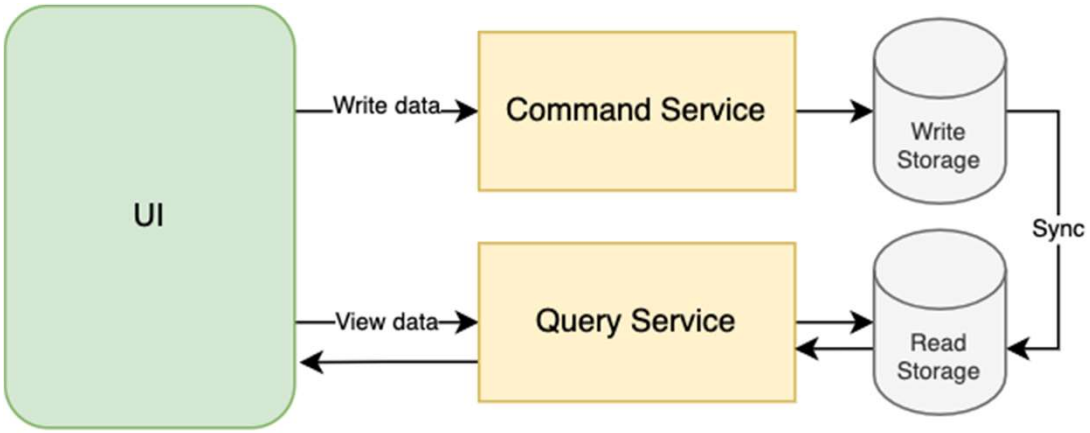
However

“However, it's important to note that implementing persistence and event sourcing requires careful design and consideration. It introduces complexity and additional challenges, such as managing event versioning, snapshotting, and dealing with long-running processes. Therefore, it is crucial to assess the trade-offs and determine if the benefits outweigh the costs for a particular system before adopting these approaches.”



CQRS (Command Query Responsibility Segregation)…

“CQRS (Command Query Responsibility Segregation) is a pattern that aligns well with the principles of domain-driven design (DDD) and addresses the separation of concerns in handling read and write operations in a software system. It introduces a clear distinction between the responsibilities of commands (changes to the system's state) and queries (retrieval of data).”



…CQRS (Command Query Responsibility Segregation)…

“In traditional systems, a single model is often used to handle both commands and queries. However, as systems grow in complexity, the requirements for reading and writing data may diverge. CQRS advocates for splitting the read and write concerns into separate models, providing more flexibility, scalability, and performance optimizations.”

...Persistence and Event Sourcing...

Here are some key aspects of CQRS:

Separate models: CQRS suggests having distinct models for read operations (query model) and write operations (command model). The query model focuses on providing efficient data retrieval and is often denormalized and optimized for specific read use cases. The command model handles the write operations and enforces business rules, ensuring consistency and integrity.

Asynchronous communication: CQRS often relies on asynchronous communication between the command and query models. Commands are sent to the command model and processed asynchronously, typically through a message queue or event bus. The results of write operations are then propagated to the query model, ensuring eventual consistency.

Performance optimizations: since the read and write models are separated, each can be optimized independently. The query model can be tuned for read-intensive operations, leveraging caching, precomputing, and indexing techniques. The command model can focus on enforcing business rules without the performance considerations of data retrieval.

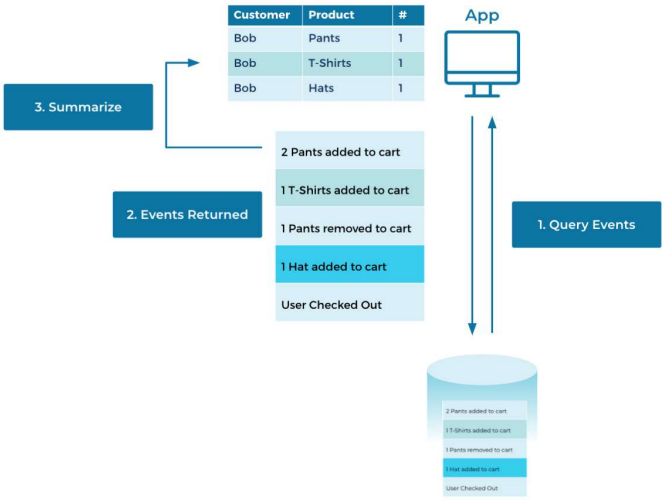
Scalability: CQRS allows for independent scalability of the read and write components. As read operations tend to be more frequent than writes, the query model can be scaled independently to handle increased read traffic without impacting the write operations.

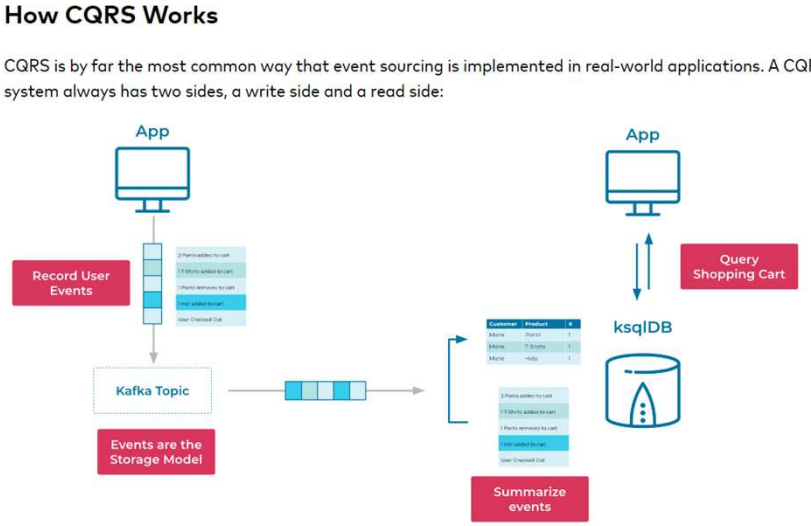
Domain-centric design: CQRS aligns with DDD by emphasizing a domain-centric design approach. It enables developers to model the system based on the distinct command and query requirements of the business domain, leading to a better representation of the domain concepts and their behavior.

Event sourcing integration: CQRS and event sourcing often go hand in hand. Event sourcing can be used to capture the changes made through commands, and the resulting events can be projected into the query model, ensuring that the query model is up-to-date with the latest changes in the system.

CQRS (Command Query Responsibility Segregation)…

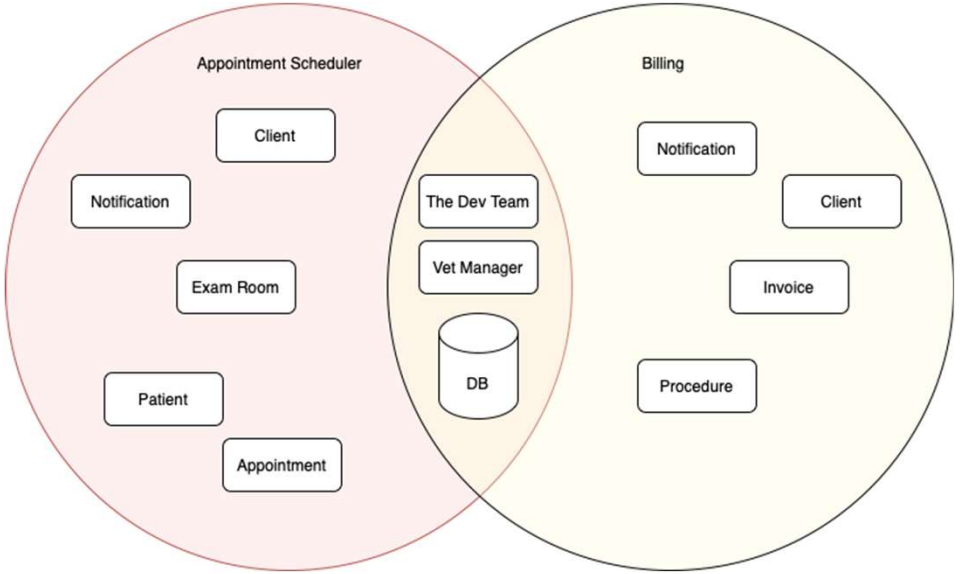
“CQRS can be a powerful pattern for systems with complex and evolving read and write requirements. However, it introduces additional complexity and requires careful design considerations, as it involves managing the synchronization between the command and query models and handling eventual consistency. It is essential to evaluate the trade-offs and ensure that the benefits of CQRS align with the specific needs of the system before adopting it.”

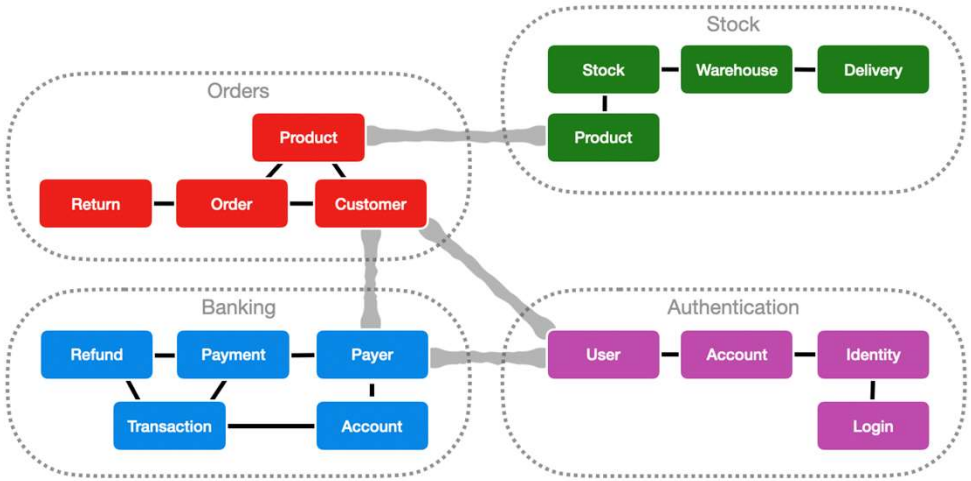




Integrating Bounded Contexts…

“Integrating bounded contexts is a critical aspect of domain-driven design (DDD) that focuses on how different parts of a system, represented as bounded contexts, can interact and collaborate with each other effectively. Bounded contexts are cohesive units within a domain where concepts, language, and rules are consistent.”





... Integrating Bounded Contexts...

Here are some considerations for integrating bounded contexts in DDD:

Shared Kernel: When two or more bounded contexts need to collaborate closely, a shared kernel can be established. The shared kernel defines a subset of the domain model and language that is shared between the bounded contexts. This allows for a common understanding and communication between teams working on different contexts.

Context Mapping: Context mapping is a technique for defining the relationships and boundaries between bounded contexts. It helps identify the types of interactions between contexts and provides strategies for handling these interactions. Context mapping techniques include defining explicit boundaries, creating partnership agreements, and establishing translation layers to bridge the semantic gaps between contexts.

Anti-Corruption Layer: An anti-corruption layer is a mechanism used to protect a bounded context from the complexities and constraints of another context with a different model or language. It acts as a translation layer, transforming requests and responses between the two contexts, ensuring that the internal model of each context remains clean and unaffected by the other.

Shared Language: Establishing a shared language between different bounded contexts is crucial for effective collaboration. Each context should have its own ubiquitous language that is well understood within its boundaries. However, when interacting with other contexts, a shared language should be established to bridge the communication gap and align the understanding of concepts and terms.

Integration Events: Integration events, also known as domain events or system events, are notifications that capture meaningful occurrences within a bounded context. These events can be used to communicate changes or important information to other contexts. By publishing integration events, a context can communicate its state changes or decisions to other contexts asynchronously, facilitating loose coupling and decoupled integration.

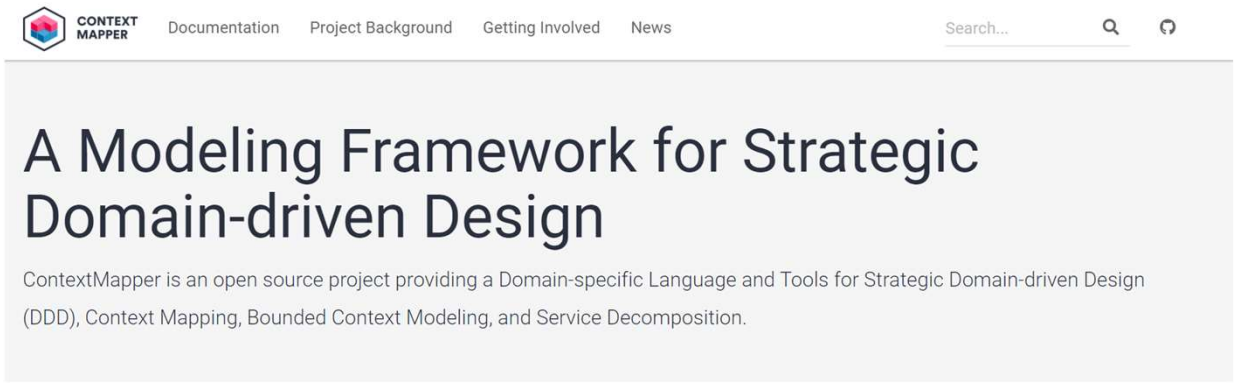
Context-specific APIs: To facilitate the integration between bounded contexts, context-specific APIs can be designed. These APIs provide a well-defined interface that allows external contexts to interact with a bounded context. By exposing only the necessary operations and encapsulating the internal complexities, the APIs enable controlled access and communication between contexts.

Continuous Collaboration: Integrating bounded contexts is an ongoing process that requires continuous collaboration between teams working on different contexts. Regular communication, knowledge sharing, and alignment on the evolving domain model are crucial to ensure a cohesive system design and effective integration.

... Integrating Bounded Contexts

By carefully considering and applying these integration techniques, DDD enables the development of well-structured, modular, and maintainable systems that can evolve and scale over time while preserving the integrity of each bounded context.

CONTEXT MAPPER



Who else needs some coffee ?

Domain Driven Design

Technical Training

Let’s do it