# Event Processing

Wintersemester 2022/23

Informatik Master (INM)

**Prof. Dr. Bernhard Hollunder**

Department of Computer Science
Furtwangen University of Applied Sciences
Robert-Gerwig-Platz 1, D-78120 Furtwangen
http://www.hs-furtwangen.de/



## Themen

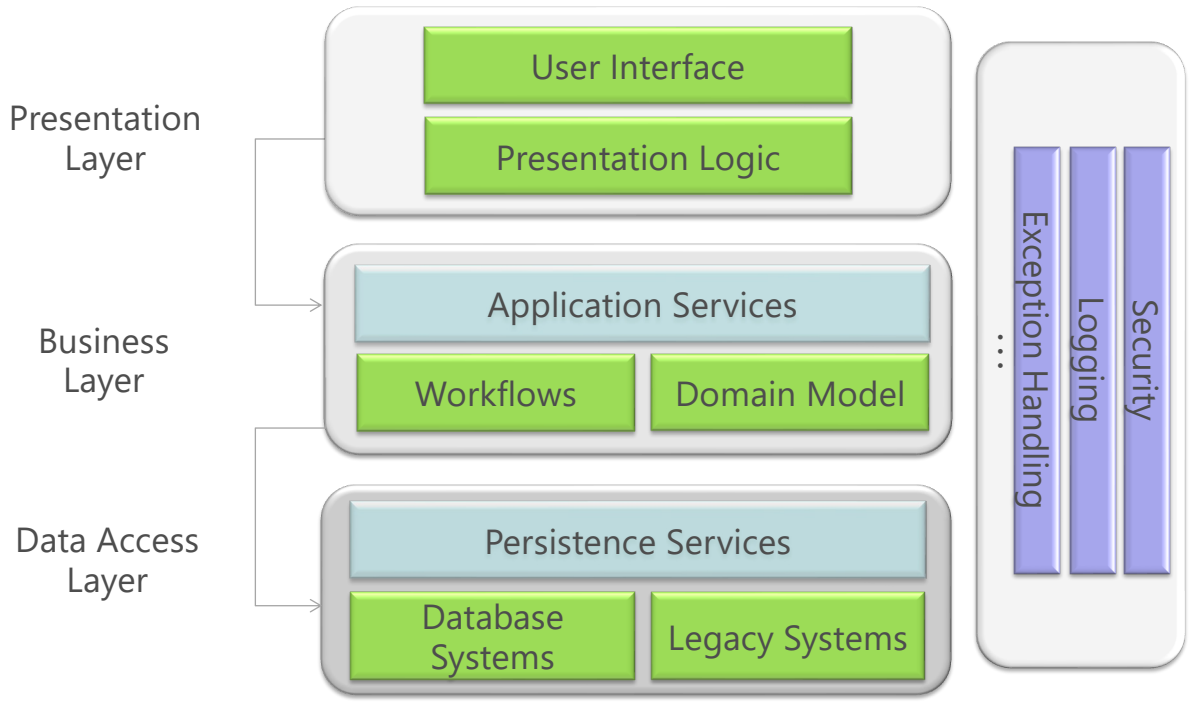
1. Event-driven Computing
2. Event Sourcing and its Application to CQRS
3. Java Messaging Service
4. Messaging and Streaming Platform Apache Kafka
5. Streaming Systems
6. Unified Model for Both Batch and Streaming Processing
7. Complex Event Processing and Event Processing Language
8. Processing Semantics: Read-Process-Write Pattern

# Overview

- Layered Software Architecture
- Command-Query Separation (CQS)
- Command-Query Responsibility Segregation (CQRS)
- Event Sourcing
- Event Store
- CQRS with Event Sourcing
- Example from the Library Domain



# Layered Software Architecture



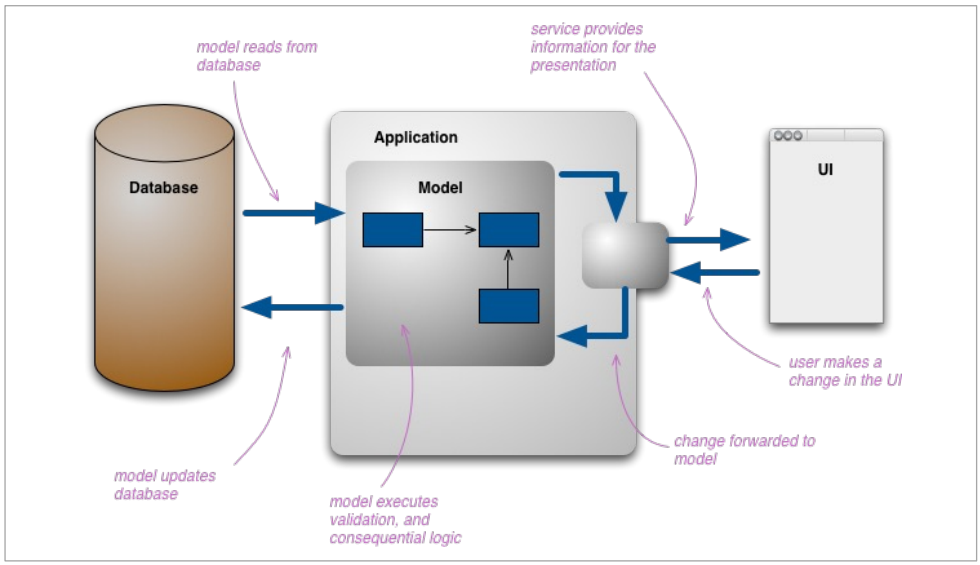
# Business Layer

- The application services
  - define an abstraction of the domain model
  - provide high-level business functions for creating, modifying and retrieving data.
- The domain model
  - is the conceptual representation of the domain and covers the core business entities as well as supporting types such as data transfer objects (DTO) and value objects
  - is often represented with a modeling language such as UML
  - yields as input for an object-relational mapping (O2R), which completely generates the data access layer.
- Workflows
  - group single actions on the domain model to form business processes.



# Persistence – CRUD based

- Typically, the structure of the persistent storage is closely related to the conceptual model, as supported by approaches such as O2R mapper.




[Cf. https://martinfowler.com/bliki/CQRS.html]

# Command-Query Separation (CQS)

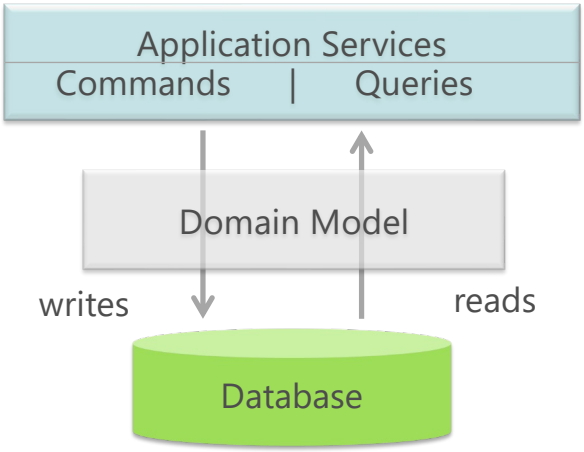
- CQS was introduced by Bertrand Meyer as part of the design by contract (DbC) methodology.
- According to CQS an interface method should either be a command or a query.
  - Commands
    - are operations that change the application's state
    - do not return a value
    - technically speaking are writes / updates.
  - Queries
    - retrieve information
    - are side effect free operations.
- Example

```
public interface BibAPI extends BibCommands, BibQueries {}
```



# Persistence

- Commands result in writes to the database.
- Queries retrieve information from the database.
- Typically, write and read operations use the same data model, e.g., tables in an RDBMS.



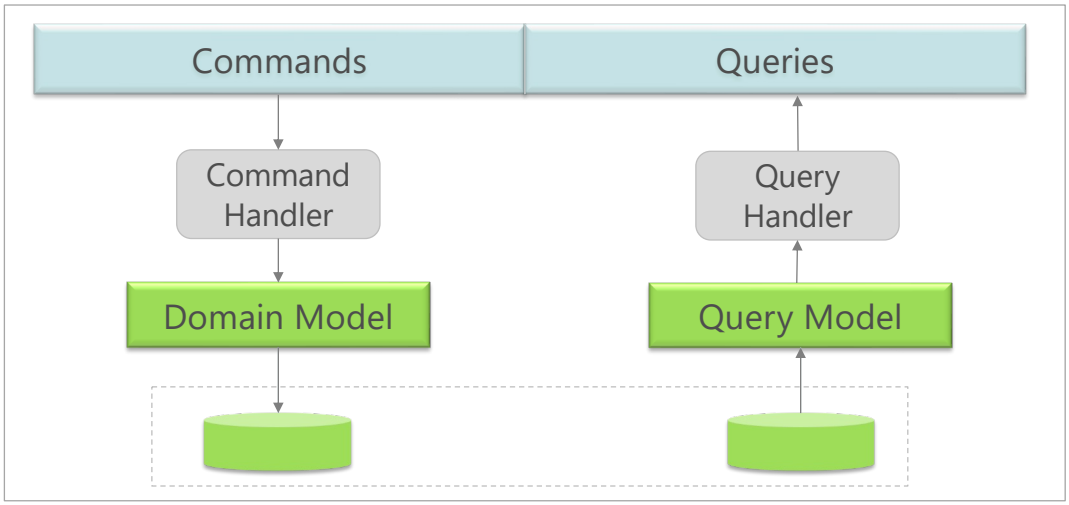
# Observations

- The number of read operations performed by applications is in most cases significantly higher than write operations.
  - ~ 80% reads and 20% writes.
- This asymmetry is typically not reflected by the data access layer.
- Note that …
  - read operations cannot be fully optimized because they are executed on the common read / write data model.
  - queries should not contain any business logic and hence do not require the domain model.
- This motivates CQRS.



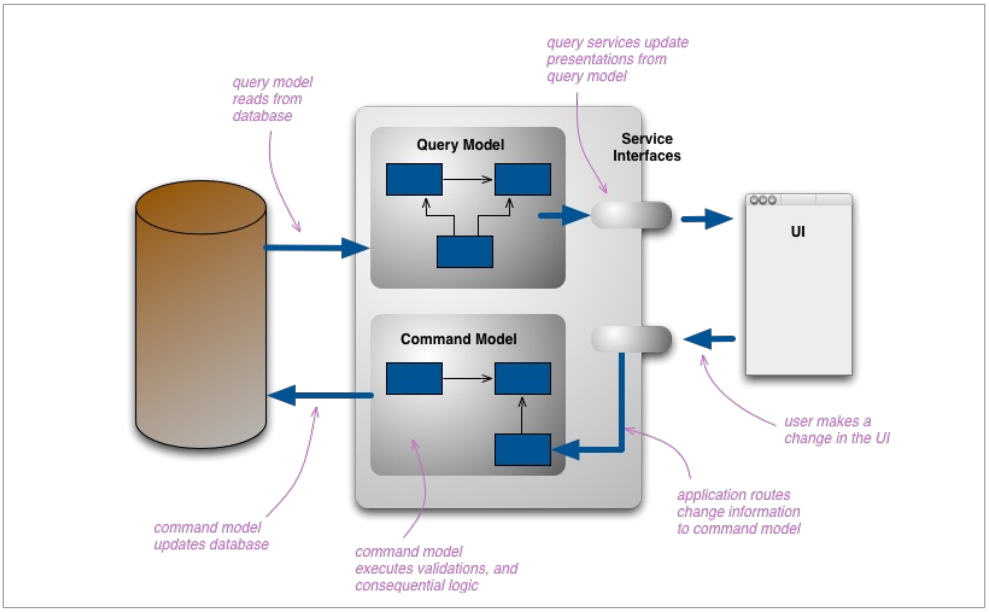
# Command-Query Responsibility Segregation (CQRS)

- According to CQRS, commands and queries
  - are not only separated at interface level
  - but also their processing is isolated.



# Command-Query Responsibility Segregation (CQRS)

- A more detailed view into CQRS:




[Cf. https://martinfowler.com/bliki/CQRS.html]



# Other Topic: Version Control of Data

- Many systems today store only the current state of the domain model entities.
- As a consequence, it is not possible to
  - understand how the system reached a particular state
  - analyze historical behavior.
- Example taken from the library domain:
  - State changes of a book instance.

time

| Title | Author | Availability | Return date |
|---|---|---|---|
| Streaming Systems | Tyler Akidau | true | null |
| Streaming Systems | Tyler Akidau | false | 2022/08/29 |
| Streaming Systems | Tyler Akidau | false | 2022/09/26 |
| Streaming Systems | Tyler Akidau | true | null |
| Streaming Systems | Tyler Akidau | false | 2022/11/14 |

Information no longer available through updates

current state of the object / tuple

# Two Problems to Tackle

1. In the classical database approach, updates overwrite values thus producing loss of historical data.
   - How to execute temporal queries?
   - How to return the state of an object to a given point in time?
   - Which updates are responsible that some object has a particular state?
2. The CQRS approach strictly separates the domain model from the query model. But …
   - How to synchronize both models?
   - After updating the domain model, a query must be aware of the changes.



# Event Sourcing (1)

- … provides a solution for both problems.
- Some definitions:

> "Capture all changes to an application state as a sequence of events." [Martin Fowler]

> "Event Sourcing is just the observation that events (i.e., state changes) are a core element of any system." [Ben Stopford]

> "Event Sourcing speichert statt des Zustandes die Ereignisse, die zum aktuellen Zustand geführt haben. Der Zustand selbst wird nicht gespeichert – lässt sich aber aus den Events rekonstruieren." [Eberhard Wolff]

# Event Sourcing (2)

> "The crucial test of Event Sourcing is that at any time we can blow away the application state and confidently rebuild it from the [event] log." [Martin Fowler]

- In an event sourcing system, every change to a system is stored as an event.
- Events
  - represent facts about things that have already happened
  - are considered immutable.
- Whenever required, the current state as well as former application states can be restored by processing the stream of events.



# Event Store (1)

- „Don't store state, store history."
- All events are persisted in an event store (also referred to as event database).
- Events are immutable and have a temporal order.
- Simplified interface:

```
public interface IEventStore {

  void store(Event event);
  Enumeration<Event> getAllEvents();
  Enumeration<Event> getAllEventsOfScope(EventScope eventScope);

}
```

# Event Store (2)

- … is the infrastructure to manage all changes that happened to an application and its domain objects.
- Events are not only stored, but must be retrieved to restore the state of domain objects.
- To restore a particular object, identifying data must be provided, e.g., order number or credit card number.
- This information can be encoded by `EventScope`.

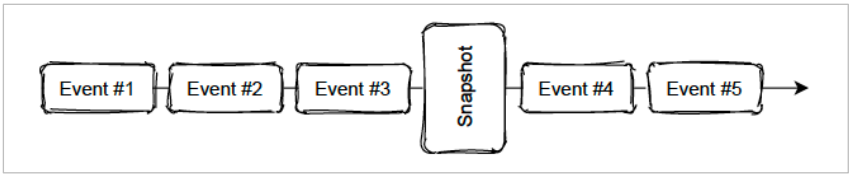
```
public class EventScope {

  public EventScope(Class<? extends Event> eventType, String id) {
    this._eventType = eventType;
    this._id = id;
  }
  ...
}
```



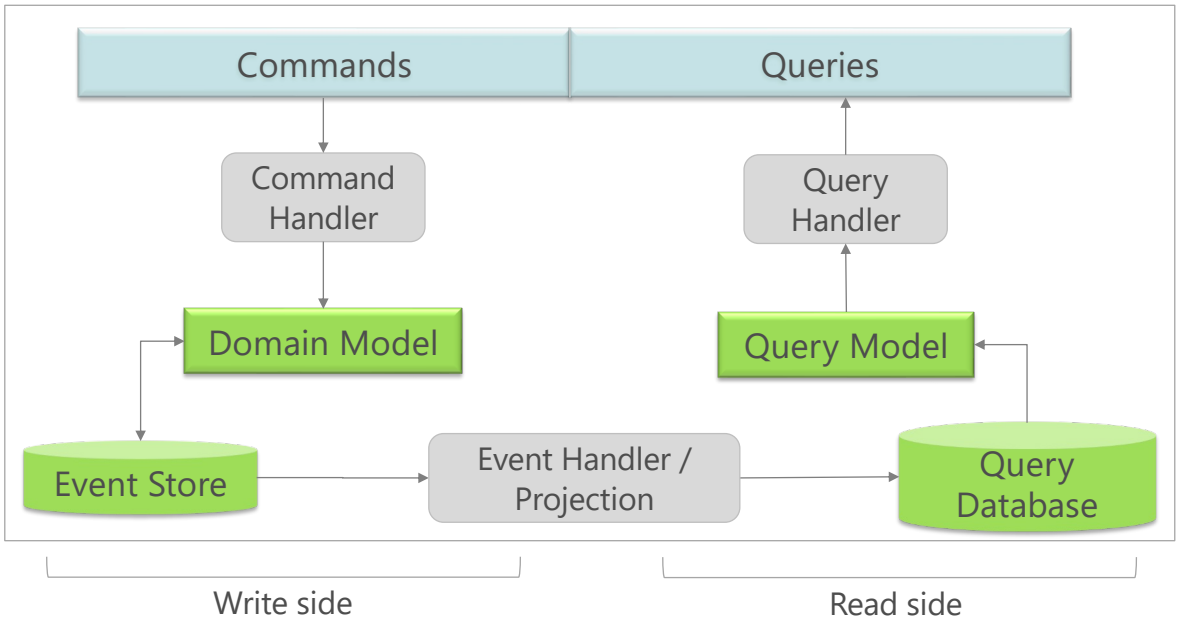
# Snapshots

- The number of events that can be managed by an event store can be large and is not limited theoretically.
- Performance slows down as more events have to be processed.
- Snapshots
  - store a domain object's state at a certain point in time
  - are a means of optimization that can be used instead of replaying the events prior to the snapshot
  - can be created after every *n* number of events or a selected period
  - can be stored, e.g., in-memory or in a separate database.



# CQRS with Event Sourcing

- Core components of CQRS based on event souring:






# Write Side

- A command issued by a user or client application is received by the command handler.
- Depending on the command the required domain object (e.g., aggregate known from domain-driven design) is loaded from the event store:
  - Fetch the events for a given type and identifier, e.g., order number.
  - Reconstruct the current state of the domain object.
- Validation
  - Check whether to accept or reject the command.
- Handle the command
  - During the processing of the command events are created and appended to the event store (“source of truth”).
  - Transactional context would guarantee data integrity.
- Handle the newly appended events
  - Projections are applied to update the query model.

# Read Side

- Event handler
  - Reacts on incoming events.
  - Plays the role of the so-called projector that maps updates performed on the write model to the query model.
  - This is the way to synchronize changes between domain model and query model.
- Query handler
  - Responsible for processing a query by retrieving data from the query model.
- Query database
  - Contains the last known state of the application.
  - Note: the state of the query database can be completely rebuilt from the event store.



# Example: Library Domain – Big Picture

Command API

```
public interface BibCommands {...}
```

Query API

```
public interface BibQueries {...}
```

Command Classes, e.g.,

```
public class AddItemCommand {...}
public class BorrowItemCommand {...}
public class ReturnItemCommand {...}
```

Query Model, e.g.,

```
public class AvailableItems {...}
public class TopItems {...}
public class Loans {...}
```

Command Handler

```
public class CommandHandler {...}
```

Event Handler / Projection

```
public class EventHandler {...}
```

Domain Model, e.g.,

```
public class ItemAggregate {...}
public class UserAggregate {...}
public class LoanAggregate {...}
```

Events, e.g.,

```
public class ItemBorrowed {...}
public class ItemReturned {...}
public class LoanTerminated {...}
```

Event Store

```
public interface EventStore {...}
```

# Library Domain – Command API

- The application services provide methods such as

```
public interface BibCommands {
  void addItemToLibarary(String title, String author, String ean, String signature);
  void addUser(String name) throws Exception;
  void borrowItem(String userName, String itemSignature) throws Exception;
  void removeItem(String itemsignature) throws Exception;
  void extendLoan(String itemSignature) throws Exception;
  void returnItem(String itemSignature) throws Exception;
}
```

- The implementation of these methods is rather straight-forward:
  - Each method creates a corresponding command object.
  - The methods' arguments are passed to that object (see next slide).



# Library Domain – Commands

- Commands
  - are represented by classes, e.g., `ReturnItemCommand`.
  - are processed by a specific `handle(…)` method of the command handler.
- Examples of commands:
  - `AddItemCommand`: creates a new item offered by the library
  - `BorrowItemCommand`: represents a new loan including the return date
  - `ExtendLoanCommand`: changes the return date of a loan
  - `ReturnItemCommand`: terminates a loan
- Command objects are created in the API implementation, e.g.:

```
public class BibAPIImpl implements BibAPI {

  @Override
  public void returnItem(String itemSignature) {
    CommandHandler.theInstance().handle(new ReturnItemCommand(itemSignature));
  }
  ...
}
```

# Library Domain – Command Handler

- The command handler
  - receives commands
  - retrieves and updates domain objects
  - creates required events that reflect the changes applied to the domain objects.
- Structure of the command handler:

```
public class CommandHandler {

  public void handle(AddItemCommand addItemCommand) {...}
  public void handle(BorrowItemCommand borrowItemCommand) {...}
  public void handle(ExtendLoanCommand extendLoanCommand) {...}
  public void handle(ReturnItemCommand returnItemCommand) {...}
  ...

}
```



# Library Domain – Domain Objects / Events

- Possible domain objects in the library context are:
  - `ItemAggregate`
  - `UserAggregate`
  - `LoanAggregate`

  } domain objects
- Events are represented as types such as
  - `ItemBorrowed`
  - `ItemReturned`
  - `ItemChanged`
  - `UserAdded`
  - `LoanExtended`
  - `LoanTerminated`

  } events

# Commands as Micro-Processes

- The realization of a command can be viewed as a micro-process:
  - Retrieve the required domain object(s) from the event store.
  - Check whether to accept or reject the command.
  - Invoke business methods to the domain object(s) as required by the command.
  - Create events that reflect the applied changes.
- Example: `ReturnItemCommand`
  1. Load required loan, user and item objects from the event store.
  2. Terminate loan (change 'open' → 'closed').
  3. Update user account (→ decrement number of current loans by one).
  4. Check if item is reserved (→ inform new borrower, change 'not available' → 'reserved').
  5. Otherwise modify status of item (change 'not available' → 'available').



# Command Handler – Example

```
public class CommandHandler {

  ...

  public void handle(ReturnItemCommand returnItemCommand) {

    // Get the respective domain objects
    ItemAggregate itemAggregate = (ItemAggregate) _eventStore.loadAggregate
           (ItemAggregate.class, new EventScope(returnItemCommand.getItem()));
    LoanAggregate loanAggregate = (LoanAggregate) _eventStore.loadAggregate
           (LoanAggregate.class, new EventScope(returnItemCommand.getItem()));

    // Update domain objects.
    // Note: Some actions are omitted for sake of simplicity, e.g., check for reservation.

    loanAggregate.close();
    itemAggregate.setAvailable(true);

    // Create and persist the required events
    _eventStore.store(new ItemReturned(itemAggregate));
    _eventStore.store(new LoanTerminated(loanAggregate));

}
```

# Library Domain – Event Handling

- The structure of the query database is optimized to quickly answer API queries.
- The changes applied to the domain objects (i.e., aggregates) are mapped into this structure. To achieve this, each event has a `handle()` method to specifically update the query database.
- The query database may be realized as in-memory storage.
- In case of failure, the content of the query database can be rebuilt from the (persistent) event store.




# Library Domain – Query API

- API example:

```
public interface BibQueries {
  Collection<Item> getItems();
  Collection<Item> getBorrowedItems();
  Collection<Loan> getLoans();
  Collection<Loan> getOpenLoans();
  Collection<User> getUsers();
  Item getItemBySignature(String itemSignature);
  Loan getBorrowedItemBySignature(String itemSignature);
  boolean isItemAvailable(String itemSignature);
  ...
}
```

- `Item`, `Loan`, and `User` defined in the query model play the role of data transfer objects (DTOs) and are independent of the domain objects!
- The query database may manage collections of items, loans, currently loaned items, etc. as needed by the query API.

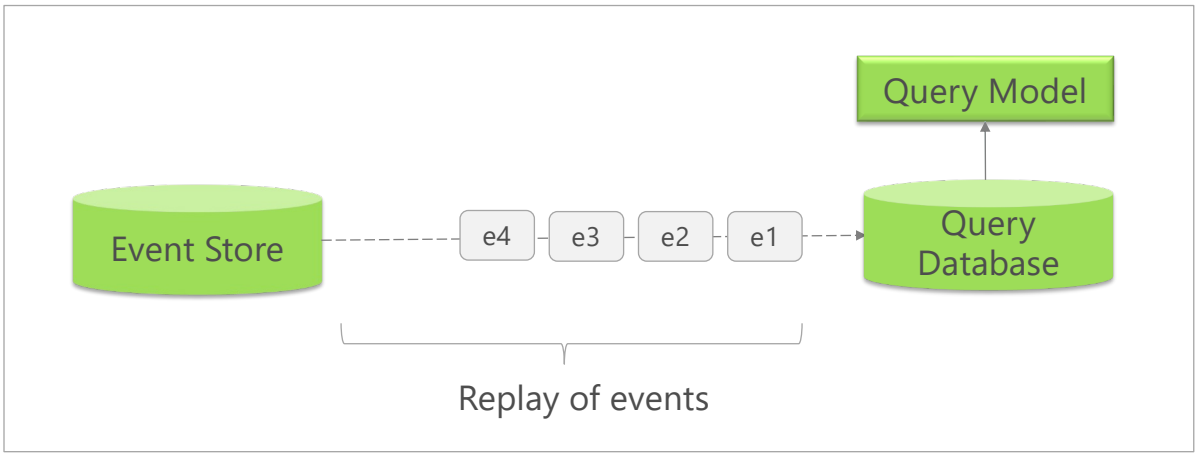
# Library Domain – Event Handling

- Examples for the projection logic encoded in the `handle()` methods to update the query database:
- `ItemReturned.handle()`
  - Extract the item's signature from the domain object contained in the event.
  - Remove the corresponding item from the collection of borrowed items in the query database.
- `LoanTerminated.handle()`
  - Remove the loan from the collection of open loans.
- `ItemChanged.handle()`
  - Retrieve the new item state from the item object contained in the `ItemChanged` event.
  - Create a new item object in the query model with the retrieved state.
  - Replace the old item object by the newly created item object in the collection of items.



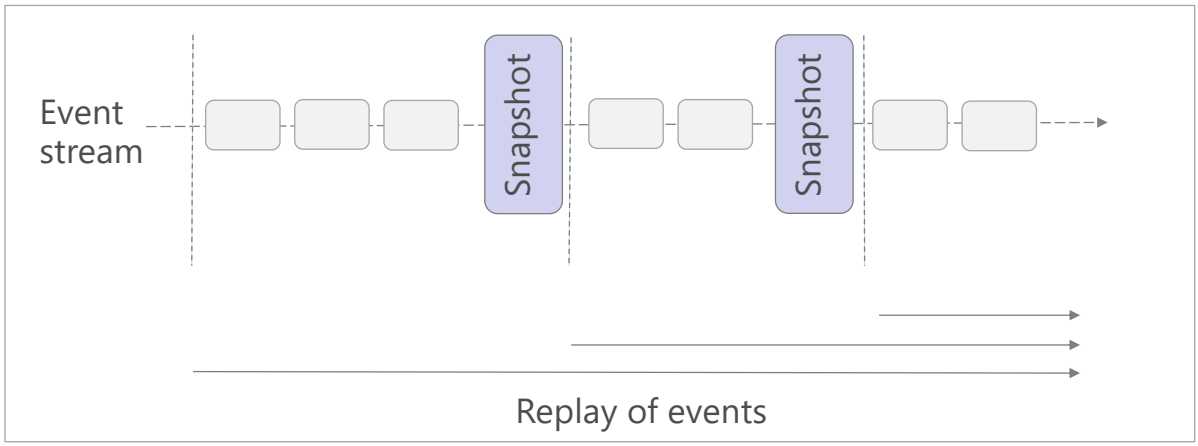
# Query Model

- The query model can be stored with different technologies such as in-memory storage.
- In case of failure, the query model can be reconstructed from the event store.



# Snapshots

- Snapshots are a performance enhancement to avoid loading the entire event history.






# The Power of Replaying Events

- Complete rebuild
  - The current state of the application can be discarded and rebuilt by replaying the events on an "empty application state".
  - Alternative technologies to manage the query model can easily be integrated.
  - Basis for high scalability of the read side.
- Temporal query
  - The application state can be rewound to a previous point in history.
- Simulation
  - Determine the consequences if a modified sequence of events is applied to the application.

# Discussion

- The CRUD approach
  - Feasible and appropriate for a large class of systems.
  - Technically driven approach.
  - Has broad tool support.
- CQRS
  - Adds complexity in application design: definition of domain model, aggregates, commands, events, projections, and query model.
  - Business driven approach.
  - Often applied in the context of microservices.
  - Increases performance and scalability due to the separation of reads und writes.
  - Can be implemented based on event sourcing.
  - Access to data history.



# Check Points

- Discuss the advantages / disadvantages of separating an application API into queries and commands.
- Explain the main difference between CQS and CQRS.
- What is the relationship between commands and events in the CQRS approach?
- Explain the responsibilities of the command handler and event handler, respectively.
- What is meant by event sourcing?
- Explain the core components of CQRS based on event sourcing.
- Describe important features of event stores.
- Snapshots: Nice to have or a necessary feature of an event store?
- How to achieve scalability of the read side?
- What is meant by: „Don‘t store state, store history.“?
- Apply the CQRS approach to a simple application domain.