DRAFT - IT Project Guidance

On Event Datastores

Version:

0.1

## Purpose

The purpose of this document is to introduce and clarify the differences between event datastores—such as those implemented with Martin.db—and more traditional state management approaches, for example using Entity Framework (EF). This guidance is specifically designed for non-technical decision makers, aiming to shed light on the contrasting methodologies of capturing events over time versus maintaining current state. By explaining these distinctions, readers will be equipped with a foundational understanding to make better-informed decisions regarding IT systems and data strategies. Whether assessing new projects or reviewing existing solutions, this document will provide practical perspectives on how event-based models and state-based models influence data storage, reporting capabilities, and the agility of business processes.

## Synopsis

TODO

## Contents

[Purpose 1](#_Toc195791704)

[Synopsis 1](#_Toc195791705)

[Contents 2](#_Toc195791706)

[Introduction 3](#_Toc195791707)

[Background of this work 3](#_Toc195791708)

[Context 3](#_Toc195791709)

[Background of issue 3](#_Toc195791710)

[Problem Statement 3](#_Toc195791711)

[H3 4](#_Toc195791712)

[H4 4](#_Toc195791713)

[Appendices 5](#_Toc195791714)

[Appendix A - Document Information 5](#_Toc195791715)

[Versions 5](#_Toc195791716)

[Images 5](#_Toc195791717)

[Tables 5](#_Toc195791718)

[References 5](#_Toc195791719)

[Review Distribution 5](#_Toc195791720)

[Audience 5](#_Toc195791721)

[Structure 5](#_Toc195791722)

[Diagrams 6](#_Toc195791723)

[Acronyms 6](#_Toc195791724)

[Terms 6](#_Toc195791725)

# Purpose and Audience

Introduction

## Traditional ORMs

Object Relational Modellers (ORMs) are used to persist aggregates of entities, value-objects.

In the .NET.Core development stacks, the most well known and used ORM is Microsoft’s Entity Framework (EF).

ORMs maps classes to tables, properties to columns, and tracks changes to entities to generate SQL for Create, Update Delete statements.

It assumes a **mutable state model**—you load an entity, change it, and save it.

While EF is most used to manage SQL Server databases, it has many adapters, permitting it to be run over other industry leading databases, such as Oracle, PostgresSQL, MySQL, etc.

## MartenDB

Like EF, Marten is also an adapter or abstraction layer that in teracts with the underlying database using its native JSONB capabilities.

It leverages PostgreSQL’s support for storing and querying JSON documents to provide a document-oriented and event-sourced programming model.

Like EF, installed into the code base as one or more libraries.

Unlike EF, it only works on PostgresSQL. It cannot run over other database types such as SQL Server, MySQL, or Oracle. This is a deliberate design choice as PostGRes’s JSONB is mature, performant, and well integrated with indexing and querying.

It is best viewed as a **PostgreSQL-native document and event store for .NET**, not a general-purpose ORM.

that is In contrast, Marten is a document and event store built on top of PostgreSQL, designed for .NET applications.

It supports both document based persistence and event sourcing.

It can be superficially described as an event-focused ORM, however it deliberately diverges from traditional ORMs like Entity Framework (EF) in both philosophy and implementation.

Marten allows you to persist .NET objects Plain Old Class Objects (POCOs) as **JSON documents** inside PostgreSQL.

In contrast to EF, which maps object models to relational tables and columns, Marten maps object models to JSON documents stored in a single column (data) within a PostgreSQL table. It also manages event streams by writing to dedicated tables (mt\_events, mt\_streams) that it creates and maintains.

It also provides **first-class support for event sourcing**, where state is derived from a stream of events rather than being stored as mutable rows in relational tables. It’s not just an ORM alternative—it’s a different model of thinking about data persistence.

Considerations

Using MartenDB imposes some constraints.

As PostgresSQL as a hard dependency, **Use Marten where PostgreSQL is already accepted as the standard**, and where event sourcing or document storage is a sufficiently better fit than relational modelling and it is worth the necessary training and changes to management processes.

# Example work flows

## Prerequisites

### Server

You’ll need access to a existing or new PostgresSQL server, 9.4+ as it requires a database with JSONB support.

PostgresSQL uses a different port than the 1433 used by sql server. 5432 is the default.

### Database

If a database doesn’t exist, consider using psql to:

createdb marten\_test

or via SQL:

CREATE DATABASE marten\_test

### Database User

You’ll need to create a user who has permission to create tables and indexes.

### Connection String

You’ll need a connection string to the server that includes the target device, the name of the database, and credentials. For example:

Host=localhost;Database=marten\_test;Username=postgres;Password=yourpassword

### Connectivity and Permissions

**Test connectivity and permissions before initializing Marten**, to avoid runtime surprises.

## Installation

You can install it from NuGet into your code base:

dotnet add package Marten

It will install sub libraries as well, including Npgsql (the .NET PostgreSQL driver)

## Configuration

At startup, you’ll want to registers Marten’s IDocumentStore in the DI container. You can also configure schema names, serialization, and document mappings here.  
  
using Marten;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddMarten(options =>

{ options.Connection("Host=localhost;Database=marten\_test;Username=postgres;Password=yourpassword");

// define schema of objects to save:  
options.Schema.For<UserProfile>().Identity(x => x.Id);

// Whether to make tables automatically or not:  
options.AutoCreateSchemaObjects = AutoCreate.All;

});

## Initialization

Marten does **not** create the database itself if it doesn’t exist. It assumes the database already exists and is reachable via the connection string you provide.

What Marten *does* create automatically (if configured to do so) are the **tables, indexes, and schema objects** required for storing documents and events *within* that database.

This will create a table for each document type you store, with columns for id, data (JSONB), and metadata.

* One table per document type (e.g., user\_profiles)
* with columns for id, data (JSONB), and metadata.
* Indexes and metadata tables as needed
* Event stream tables (mt\_events, mt\_streams) if you use event sourcing

## Upsert

Marten performs an **upsert** operation when you call Store() followed by SaveChanges(). If the document with the given Id already exists, it is **replaced entirely** with the new version. If it doesn’t exist, it is **inserted**.

This behaviour is consistent and predictable:

* Marten does not track changes like EF does.
* You must explicitly call Store() even if you’ve loaded and modified an object.
* The entire JSON document is replaced in the database.

## Performance Considerations

EF excels at complex relational queries, joins, and projections. Marten is **not designed for relational querying**. If your application relies heavily on joins or relational integrity, EF will outperform Marten.

Marten is faster and simpler for:

* Flat document retrieval
* Event stream reads
* Snapshot-based models

But slower or less suitable for:

* Multi-table joins
* Relational constraints
* Partial updates

### Writing

Marten’s upsert model—replacing the entire JSON document—is generally **faster** than EF for simple inserts and updates, because:

* It avoids change tracking.
* It issues a single INSERT ... ON CONFLICT DO UPDATE statement.
* It doesn’t need to compute a diff between object state and database state.

However, if you're updating only a few fields in a large document, Marten may be **less efficient**, since it rewrites the entire JSON payload.

### Reading

Reading a document by ID is fast and comparable to EF, especially if the id column is indexed (which it is by default). Querying by fields inside the JSON is **slower than querying traditional columns**, unless those fields are explicitly indexed.

PostgreSQL’s JSONB indexing is powerful, but not as fast as native column indexing for complex queries or joins.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).