# Event Sourcing for Bike Wear Tracking

## 1. Introduction & Motivation

Modern software systems demand reliability, adaptability, and auditability. Event sourcing is an architectural pattern that addresses these needs by recording all state changes as a sequence of immutable events. This project applies event sourcing to the "Bike Wear Tracker"—a system designed to help cyclists monitor the wear and maintenance of their bicycle components. The goal is to demonstrate how event sourcing, combined with CQRS (Command Query Responsibility Segregation) and Clean Architecture, can create a transparent, evolvable, and robust application.

## 2. Problem Statement

Traditional data management often struggles to balance reliability, scalability, evolvability, and auditability. The Bike Wear Tracker addresses these challenges by:

- **Reliability**: Ensuring consistent, resilient data storage.
- **Evolvability**: Supporting new features and requirements with minimal disruption.
- **Auditability**: Maintaining a complete, immutable history of all operations.
- **Scalability**: Handling growth in users, bikes, and components efficiently.

This project serves as a case study for leveraging event sourcing and CQRS to meet these requirements in a real-world context.

## 3. Methodology

### 3.1 Product Overview

The **Bike Wear Tracker** is a REST-api based tool for cyclists to manage bikes and their components. Key features include:

- Registering and managing multiple bikes.
- Adding, replacing, and tracking the lifecycle of components (e.g., chains, tires).
- Logging rides, with automatic mileage updates for bikes and components.
- Viewing detailed histories and audit trails.
- Accessing a REST API for integration with other tools.

The backend is built with .NET 9.0, following Clean Architecture principles for clear separation of concerns. Marten is used for event storage and projections, Wolverine for mediation, and PostgreSQL as the database.

### 3.2 Service Scope

This project is a proof of concept for event tracking in the context of product maintenance and lifecycle management, using bikes and their components as the domain.

### 3.3 Service Design & Architecture

The application is modular, adhering to single-responsibility principles. The **Domain Layer** contains aggregates (e.g., `Bike`), value objects, and domain events. The **Application Layer** orchestrates use cases via command and query handlers, using Wolverine for mediation. The **Infrastructure Layer** manages persistence with Marten and PostgreSQL. The **Presentation Layer** exposes a REST API, documented with OpenAPI.

### 3.4 Communication: REST and Messaging

- **REST API**: All client interactions use a RESTful API with standard HTTP verbs and resource-oriented URIs. The API is self-documented for ease of use.
- **Internal Messaging**: Wolverine handles command and event dispatching, supporting asynchronous processing and decoupling. This design enables future integration with distributed messaging or external event buses.

### 3.5 API Documentation

The REST API is fully documented using **OpenAPI/Scalar**, providing:

- Interactive documentation for developers.
- Clear contracts for endpoints, request/response models, and error handling.

### 3.6 Deployment with Docker

The application is containerized for:

- Consistent deployment across environments.
- Easy orchestration of dependencies (e.g., PostgreSQL) via Docker Compose.
- Scalability and portability for cloud or on-premises hosting.

A typical deployment includes:

- The .NET application container.
- A PostgreSQL container for event and read model storage.

## 4. Analysis & Results

### 4.1 Implemented Functionalities

The project implements core features to validate the event sourcing approach:

- **Bike Management**:
  - Registering new bikes (`RegisterBikeCommand` → `BikeRegisteredEvent`).
  - Retrieving bike details via projected read models.
- **Component Management**:
  - Adding components (`AddComponentCommand` → `ComponentAddedEvent`).
  - Replacing components (`ReplaceComponentCommand` → `ComponentReplacedEvent`).
- **Ride Logging**:
  - Logging rides (`LogRideCommand` → `RideLoggedEvent`).
  - Updating mileage for bikes and components through event-driven projections.

### 4.2 Architectural Strengths

1. **Feature Evolution and Modularity**
   CQRS separates command (write) and query (read) responsibilities. Commands and events are explicitly defined, with business logic encapsulated in handlers and aggregates. This modularity allows new features—such as additional commands, events, or projections—to be introduced with minimal impact on existing code. For example, adding a new maintenance action involves creating a command and event, leveraging Wolverine's mediation for decoupling.

2. **Efficient Command Processing and Query Execution**
   Commands are processed asynchronously, with events appended directly to the Marten event store. Inline projections update read models in real time, enabling fast queries. PostgreSQL ensures reliable, scalable storage for both events and read models.

3. **Comprehensive Audit Trail**
   Every state change is recorded as an immutable event, providing a complete, tamper-proof audit trail. Marten supports reconstructing entity histories and generating audit reports. The audit trail is accessible via event stream queries or specialized projections.

4. **Reliability and Maintainability**
   The event-driven, transactional architecture ensures atomic persistence of changes. Marten and PostgreSQL provide strong consistency and resilience. The layered structure, clear contracts, and dependency injection support maintainability and testability, isolating business logic from infrastructure.

### 4.3 Event Sourcing Tools in .NET

The **Critter Stack**—comprising Marten and Wolverine—streamlines the development of event-driven .NET applications:

- **Marten**: Acts as an event store and document database on PostgreSQL. It stores domain events as immutable streams, supports projections for read models, and enables efficient querying and state reconstruction.
- **Wolverine**: Provides mediation and messaging, handling command/query dispatching (CQRS), asynchronous processing, and internal messaging. Its design encourages decoupling and modularity, simplifying extension and integration.

Together, Marten and Wolverine enable clean, maintainable, and scalable applications that leverage event sourcing and CQRS. This stack is ideal for systems requiring strong auditability, modularity, and adaptability.

## 5. Conclusion

This project demonstrates the effectiveness of event sourcing, CQRS, and Clean Architecture—supported by Marten and Wolverine—in building a robust, auditable, and maintainable application for bike wear tracking. The system is modular, scoped around a clear bounded context, and exposes a well-documented REST API. Internal communication uses REST and messaging for extensibility and decoupling. Docker streamlines deployment, ensuring consistency and scalability. The result is a system that meets the requirements for reliability, evolvability, auditability, and scalability, serving as a practical reference for event-sourced solutions in real-world domains.

---

## References

- Building a Critter Stack Application
- Marten Documentation
- Wolverine Documentation
- EventSourcing.NetCore (Sample Project)
- Kurrent: A Beginner’s Guide to Event Sourcing

## Appendix

### System Architecture Diagram

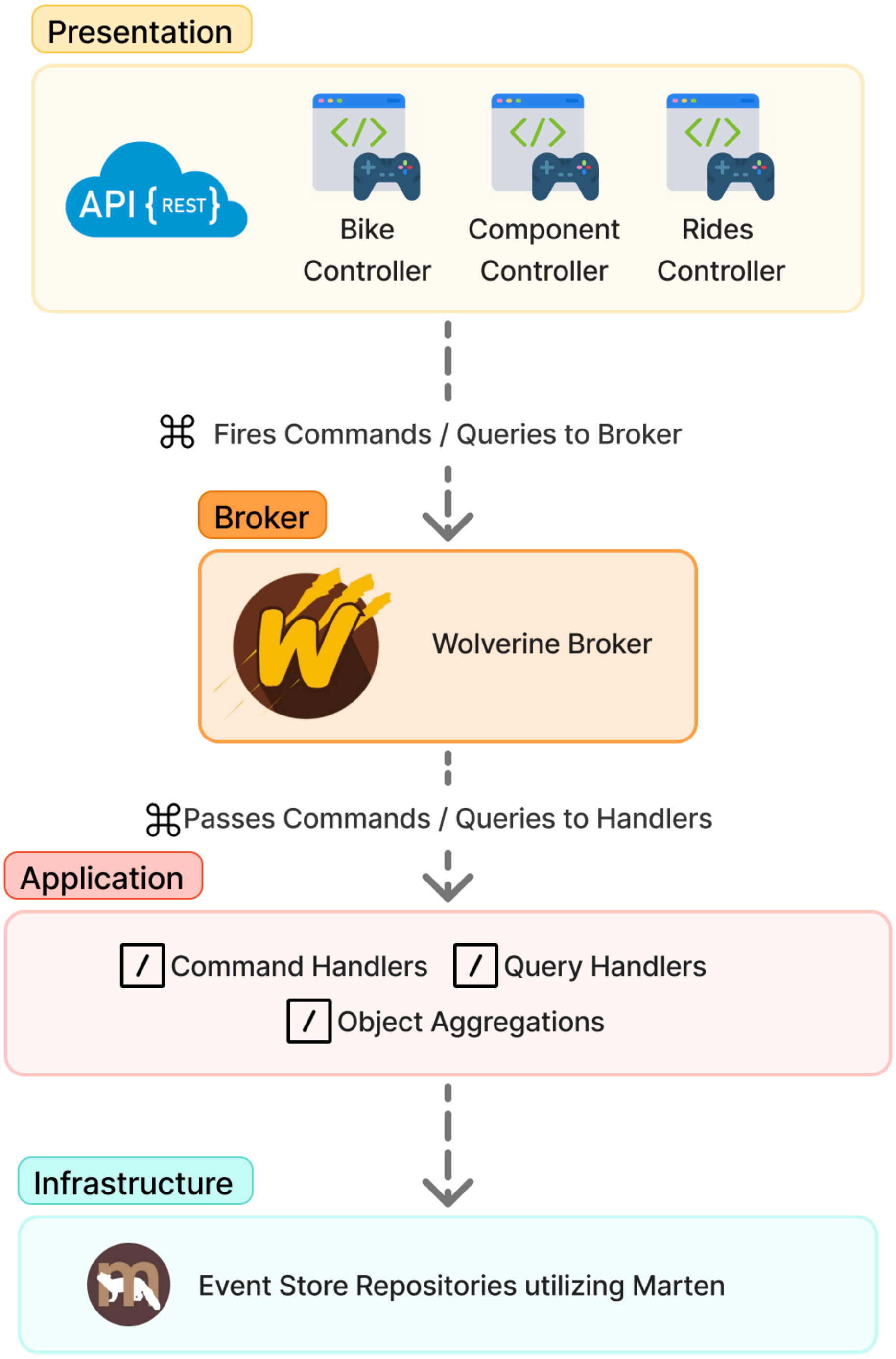

*Figure: High-level architecture showing Clean Architecture layers, CQRS, event sourcing with Marten, Wolverine for mediation, PostgreSQL for persistence, and REST API exposure.*

## A. Implementation Plan & Event Stream Concept

The Bike Wear Tracker leverages event sourcing: every state change is captured as an immutable event. The current state of a bike or component is reconstructed by replaying its event stream. CQRS separates write (commands) and read (queries) operations, optimizing for scalability and maintainability.

**Event Stream Example:**

```
[
  {
    "eventType": "BikeRegisteredEvent",
    "id": "bike-123",
    "brand": "Trek",
    "model": "Domane",
    "serialNumber": "SN-001",
    "year": 2022,
    "bikeType": "road"
  },
  {
    "eventType": "ComponentAddedEvent",
    "componentId": "chain-abc",
    "bikeId": "bike-123",
    "componentType": "Chain",
    "brand": "Shimano",
    "model": "Ultegra",
    "purchaseDate": "2024-06-01",
    "position": null,
    "addedAt": "2024-06-01"
  },
  {
    "eventType": "RideLoggedEvent",
    "bikeId": "bike-123",
    "rideId": "ride-1",
    "distance": 50,
    "rideDate": "2024-06-02",
    "loggedAt": "2024-06-02"
  },
  {
    "eventType": "RideLoggedEvent",
    "bikeId": "bike-123",
    "rideId": "ride-2",
    "distance": 60,
    "rideDate": "2024-06-05",
    "loggedAt": "2024-06-05"
  },
  {
    "eventType": "ComponentReplacedEvent",
    "bikeId": "bike-123",
    "oldComponentId": "chain-abc",
    "newComponentId": "chain-def",
    "componentType": "Chain",
    "brand": "Shimano",
    "model": "Ultegra",
    "purchaseDate": "2024-07-01",
    "addedAt": "2024-07-01",
    "position": null
  }
]
```

## B. Core Entities

**Bike**

```
{
  "id": "guid",
  "brand": "string",
  "model": "string",
  "serialNumber": "string",
  "year": "number",
  "bikeType": "string",
  "totalDistance": "number",
  "components": [
    {
      "componentId": "guid",
      "bikeId": "guid",
      "componentType": "string",
      "brand": "string",
      "model": "string",
      "purchaseDate": "date",
      "position": "string",
      "addedAt": "date",
      "mileage": "number"
    }
  ]
}
```

**Ride**

```
{
  "id": "guid",
  "bikeId": "guid",
  "distance": "number",
  "rideDate": "date",
  "addedAt": "date"
}
```

## C. Commands & Queries

**Commands**

- **RegisterBikeCommand**

```
{
  "brand": "string",
  "model": "string",
  "serialNumber": "string",
  "year": 2022,
  "bikeType": "road"
}
```

- **AddComponentCommand**

```
{
  "bikeId": "guid",
  "componentType": "Chain",
  "brand": "Shimano",
  "model": "Ultegra",
  "purchaseDate": "2024-06-01",
  "position": null
}
```

- **ReplaceComponentCommand**

```
{
  "bikeId": "guid",
  "oldComponentId": "guid",
  "componentType": "Chain",
  "brand": "Shimano",
  "model": "Ultegra",
  "purchaseDate": "2024-07-01",
  "position": null
}
```

- **LogRideCommand**

```
{
  "bikeId": "guid",
  "distance": 50,
  "rideDate": "2024-06-02"
}
```

**Queries**

- **GetBikeQuery**

```
{
  "bikeId": "guid"
}
```

D. Events

- **BikeRegisteredEvent**

```
{
  "id": "guid",
  "brand": "string",
  "model": "string",
  "serialNumber": "string",
  "year": 2022,
  "bikeType": "road"
}
```

- **ComponentAddedEvent**

```
{
  "componentId": "guid",
  "bikeId": "guid",
  "componentType": "Chain",
  "brand": "Shimano",
  "model": "Ultegra",
  "purchaseDate": "2024-06-01",
  "position": null,
  "addedAt": "2024-06-01"
}
```

- **ComponentReplacedEvent**

```
{
  "bikeId": "guid",
  "oldComponentId": "guid",
  "newComponentId": "guid",
  "componentType": "Chain",
  "brand": "Shimano",
  "model": "Ultegra",
  "purchaseDate": "2024-07-01",
  "addedAt": "2024-07-01",
  "position": null
}
```

- **RideLoggedEvent**

```json
{
  "bikeId": "guid",
  "rideId": "guid",
  "distance": 50,
  "rideDate": "2024-06-02",
  "loggedAt": "2024-06-02"
}
```

---