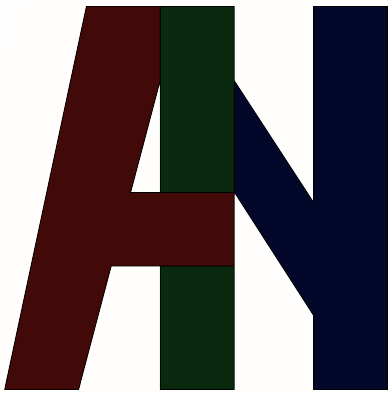
Aion

A distributed documentation and analysis tool



Author: Jannic Hiegert

Date: September 16, 2025

Info

This documentation summarizes the key technical decisions of the project and explains the reasoning behind them. It is intended as a structured guide that highlights implementation choices, trade-offs, and the lessons learned along the way.

Contents

1 Core Architecture

1.1 Clean Architecture (Hexagonal Approach)

The project strictly follows the Dependency Rule, with domain logic at the center. All external modules—such as the Avalonia client and the Blazor web frontend—communicate exclusively via a dedicated Protocol Buffers interface module.

Positives:

- Strong separation of concerns
- Core logic is independent of frameworks and infrastructure
- Easy to test and extend

Negatives:

- Requires discipline to maintain the Dependency Rule
- Initial setup is more complex than in a monolith

This architecture provides a solid foundation for maintainable and scalable software. Database services are not yet split into a separate module; this is planned for future iterations to further improve modularity and adherence to the Dependency Rule.

1.2 Interface-Based Programming

After the first prototype, I introduced interface-based programming wherever it made sense.

Positives:

- Loose coupling between components
- Simplifies unit testing and mocking

Negatives:

- Can lead to a proliferation of classes
- Increases cognitive load

Programming to interfaces rather than concrete implementations proved invaluable, especially when switching persistence technologies: I could implement the new technology in a concrete class without affecting the rest of the system.

1.3 Domain-Driven Design (DDD)

Although the project currently contains a single bounded context, the architecture is designed to be DDD-ready. Domain logic such as Event Sourcing aggregates is separated from infrastructure and application concerns, making it easy to extend the project with additional bounded contexts if needed.

Positives:

- Clear structure within the domain
- Aggregates encapsulate business logic effectively
- Easy to maintain and extend
- Provides a foundation for scaling to multiple bounded contexts

Negatives:

- Familiarity with DDD concepts is required for future maintenance
- Single-context projects may not fully benefit from DDD

While not a full DDD implementation, the structure supports future growth. As a single developer, I benefited from the clear separation of domain logic; in larger teams with complex applications, fully adopting DDD would be even more beneficial.

1.4 Event-Driven Architecture (EDA)

Event-Driven Architecture emerged out of necessity: services react to events as they occur, rather than polling for changes.

Positives:

- Supports asynchronous communication between services
- Reduces tight coupling between components
- Enables eventual consistency across distributed services
- Facilitates extending the system with new services without modifying existing ones

Negatives:

- Defining meaningful events requires careful design and effort

- Mapping events between services can be time-consuming
- Eventual consistency demands idempotent handlers and careful state management

Adopting this approach improved scalability, modularity, and maintainability. Using pub/sub with event records as the medium allows cohesive, asynchronous communication and simplifies testing. It also provides a solid foundation for future services and features.

1.5 Command Query Responsibility Segregation (CQRS)

Initially, the project used a monolithic approach with Entity Framework and change tracking enabled. While this worked for binding single instances to the UI, issues arose when the same entity was displayed in multiple views: simultaneous updates could break the EF pipeline. CQRS mitigated this by separating read and write models.

Positives:

- Integrates seamlessly with Event-Driven Architecture
- Simplifies implementation of pub/sub patterns
- Provides clear structure for data flow
- Improves maintainability and extensibility

Negatives:

- Read and write services must be maintained separately
- Conceptually challenging at first
- Requires careful implementation to avoid inconsistencies

CQRS significantly improved the project structure and clarified data flow. The approach helped me focus on which information to expose or transfer rather than how to transfer it. After overcoming the initial complexity, development speed increased noticeably. Although CQRS adds complexity, it supports scalability, auditability, and long-term maintainability.

1.6 Event Sourcing

Event Sourcing frequently appears alongside CQRS. Since the system already used event-based communication, I experimented with it.

Positives:

- Full audit trail of all changes
- Integrates naturally with Event-Driven Architecture and CQRS
- Enables replaying events to rebuild state or for testing

Negatives:

- Event design and management adds significant development overhead
- Requires careful handling of consistency and idempotency
- Reconstructing state from events can be computationally intensive

This was the most exciting architectural decision in the project. The implementation effort was substantial—especially after establishing a conventional relational schema—and took several weeks to complete. The lessons learned were invaluable. Event Sourcing shines in microservices that require clear audit histories or involve security-relevant operations. For typical applications, however, the overhead can outweigh the benefits.

1.7 gRPC and Protocol Buffers

For service-to-service communication, I chose Protocol Buffers and gRPC.

Positives:

- gRPC is lightweight and efficient
- gRPC provides streaming capabilities
- Protobuf is language-agnostic
- Protobuf automatically generates DTOs

Negatives:

- No message broker functionality
- Retries must be handled manually
- Risk of event loss if services are unavailable

While gRPC lacks conveniences of dedicated message brokers—such as queueing and channel-based subscriptions—it is sufficient for this small setup with Dockerized services. For larger-scale systems, I would consider Kafka or a similar solution.

1.8 Persistence

This project used multiple persistence mechanisms: JSON, SQLite, and PostgreSQL.

Positives:

- JSON and SQLite are perfect for prototyping
- PostgresSQL provides a more solid DBMS

Negatives:

- SQLite has limited functionality
- JSON is not suitable for larger data quantities and references
- PostgreSQL is heavier and requires more setup

Initially, JSON sufficed for simple persistence. SQLite was introduced alongside Event Sourcing, as storage volume increased significantly. PostgreSQL followed when transitioning from a monolith to a distributed system using Docker—by then the previous setup was disposable. The combination of Entity Framework and interface-based programming made switching persistence technologies smooth.

Note: Specialized databases optimized for Event Sourcing (e.g., EventStoreDB) exist. I chose not to experiment with them here but acknowledge they may be better suited for this use case.

2 Avalonia Client

I have programmed a frontend client that emerged iteratively from the monolithic prototype. The client is intended to manage data previously added via the web admin interface, structure documentation, and provide metadata analysis.

2.1 Interface

The interface is implemented in Avalonia with a completely custom design. I am neither a designer nor a UX specialist, but I did my best to provide clear and usable visual representations for the user. The following sections show the UI and briefly describe the functionality.

2.1.1 Time Tracking

This is the main part where time slots for a ticket in the sprint can be created. Each time slot is linked to a ticket. Documentation is shared between all time slots of the same ticket.

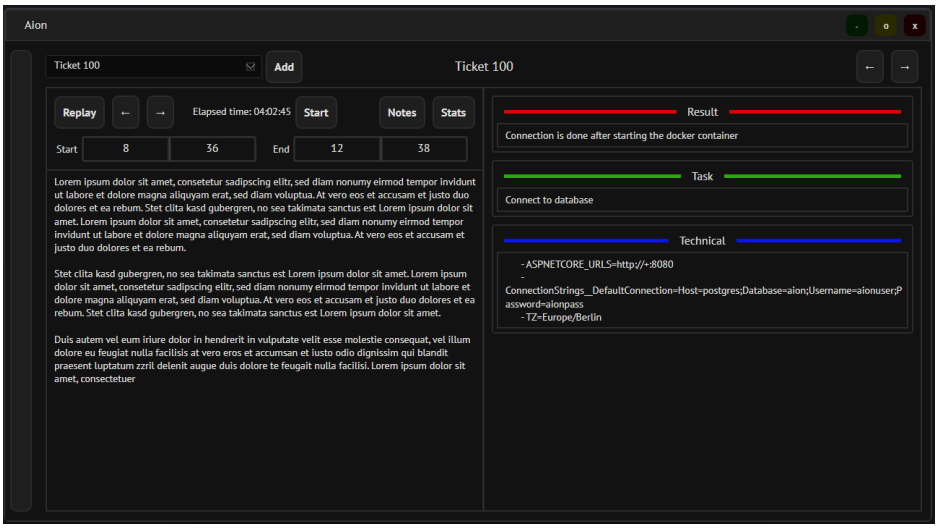


Figure 2.1: Tracking View – Notes

The screenshot above shows the window with activated notes. With the shortcut `Alt+N` you can add new notes. With `Alt + Arrow Keys` you can cycle between the available note types. With `Alt + Enter` you can save the current note.

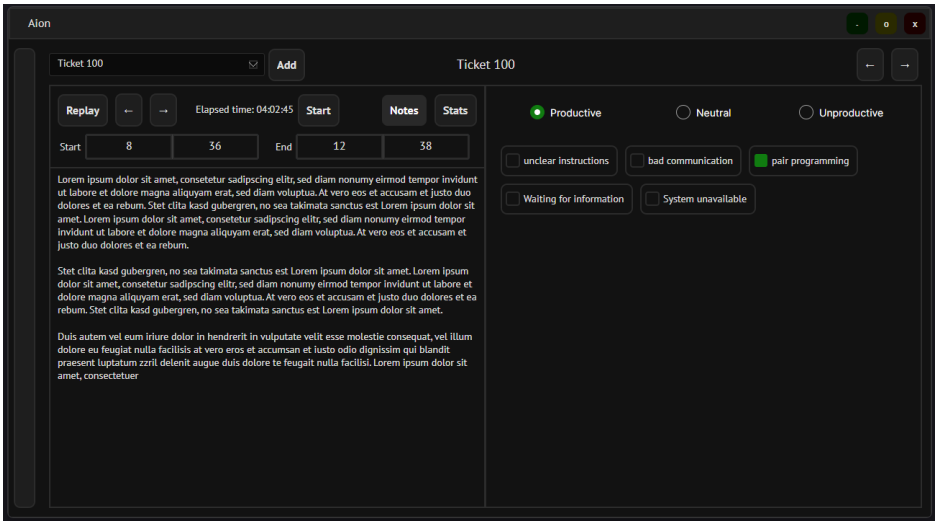


Figure 2.2: Tracking View – Stats

This screenshot shows the stats page. It allows you to categorize the time slot and add metadata, which can later be analyzed.

2.1.2 Export

This view allows the selection of multiple dates and provides a simple Markdown table preview for the time slots created on these dates.

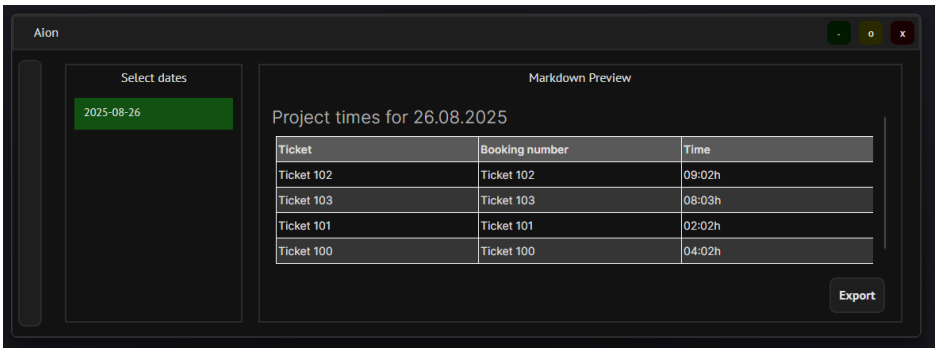


Figure 2.3: Export View

With the export button, you can export the selected dates as a `.md` file and store or share it as needed.

2.1.3 Documentation

This section allows filtering of specific notes for tickets by the provided note ID.

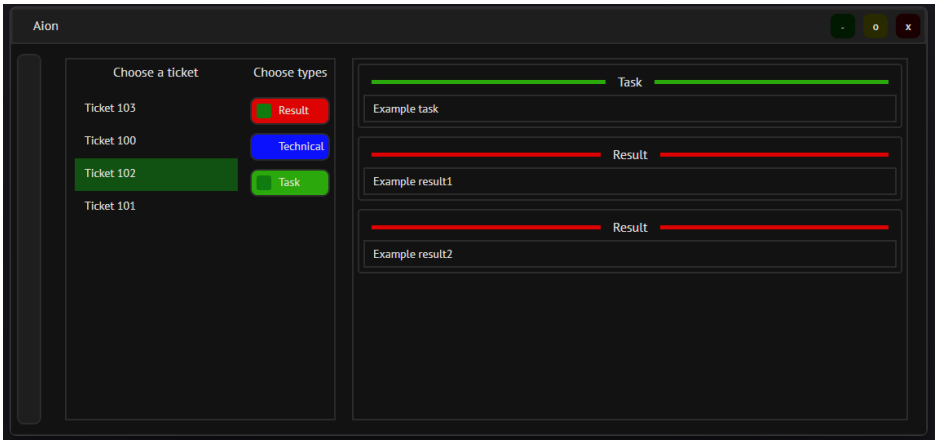


Figure 2.4: Documentation View

This enables easy information retrieval. Since the note types are dynamically generated, you can organize your work in a flexible manner.

2.1.4 Analysis

The analysis view provides long-term data evaluation.

Figure 2.5: Analysis View

Filtering is possible by tags, tickets, or sprints.

2.2 Used Technologies

- Avalonia (Frontend WPF/XAML derivative)
- Microsoft Dependency Injection
- ReactiveUI
- Polly
- Protocol Buffers
- gRPC
- MVVM Community Toolkit

2.2.1 Dependency Injection

DI is implemented via extension methods for simpler startup and service registration. While package scanning was an option, I chose manual registration to keep a clear understanding of the container's wiring.

The extension class can be found in:

`Client.Desktop.ServiceExtensions`

It is used here:

`Client.Desktop.Program.Main`

2.2.2 ReactiveUI

ReactiveUI provides responsive frontend visualization. DTOs are mapped to specific client models to enable MVVM bindings.

An example model:

`Client.Desktop.DataModels.TicketClientModel`

Protobuf DTO mapping is implemented via extension methods:

`Client.Desktop.Communication.Notifications.Ticket.TicketExtensions`

2.2.3 Polly

Polly is used for resilience when sending events with retry strategies. Without a message broker, retries are handled per module.

DI implementation for Polly:

`Client.Desktop.ServiceExtensions.AddPolicyServices`

Usage example:

`Client.Desktop.Communication.Commands.CommandSender`

2.2.4 Protobuf

Together with gRPC, Protobuf defines three message types: Commands, Requests, and Notifications. Commands and Requests follow CQRS patterns. Notifications are delivered via a gRPC stream that the client subscribes to.

Subscription implementation:

`Client.Desktop.Communication.Notifications.Ticket.TicketNotificationReceiver`

This allows the client to receive events (notifications) immediately when the backend updates state.

Commands and requests are sent directly to the server. Examples:

`Client.Desktop.Communication.Commands.TimeSlots.TimeSlotCommandSender`

`Client.Desktop.Communication.Requests.TimeSlots.TimeSlotRequestSender`

2.2.5 MVVM

Model–View–ViewModel is used to bind reactive objects to the AXAML frontends.

Example view:

`Client.Desktop.Presentation.Views.Tracking.TimeSlotControl`

View model (UI logic and reactive bindings):

`Client.Desktop.Presentation.Models.TimeTracking.TimeSlotViewModel`

Model (managing and storing underlying data):

`Client.Desktop.Presentation.Models.TimeTracking.TimeSlotModel`

The MVVM pattern is well documented, so details are omitted here. This example, one of the more complex ones in the application, clearly demonstrates separation of concerns.

2.2.6 Startup initialization and shutdown

A highlight of the frontend is its manual, ordered initialization. I wanted certain services and classes to initialize in a specific sequence: first, register messengers in models so they are available for incoming data; then initialize the streaming connection for backend notifications; next initialize services; and finally send any pending commands that were not processed at shutdown.

Scheduler implementation:

`Client.Desktop.Lifecycle.Startup.Scheduler.StartupScheduler`

To orchestrate the process, I added a task interface for executable steps, allowing fine-grained ordering.

Illustrated here:

`Client.Desktop.Lifecycle.Startup.Tasks.Initialize.AsyncInitializeTask`

The order is Models, Services, then ViewModels. A consistency check and exception for loading issues make problems easy to identify.

3 Blazor Web Admin and Monitoring

The data managed in this Blazor web server is consumed by the client. The streaming capability ensures that the clients are updated in sync with the webfrontend. Which means that changes are published system wide in a (nearly) synchronous manner. This includes creating sprints, setting sprint durations, and managing the active sprint status. Ticket data can also be created, including booking numbers, and assigned to a sprint.

In addition, this web client manages supporting data such as tags for metadata analysis and note types for structured data capturing per ticket. This design ensures that clients cannot directly manage or alter administrative data.

The observability feature is visualized within this module, with inner workings described in the following chapter.

In conclusion, this web module primarily provides administration features. This Blazor Server application is reachable at `http://localhost:5000/`.

3.1 Interface

This section gives a short overview of the interface, including key screenshots.

3.1.1 Sprint Data Page

This page allows management of sprint data. This includes adding sprints with name and timespan. Setting the sprints to an active status. Adding tickets and the corresponding data. The tickets can then be added to the sprint.

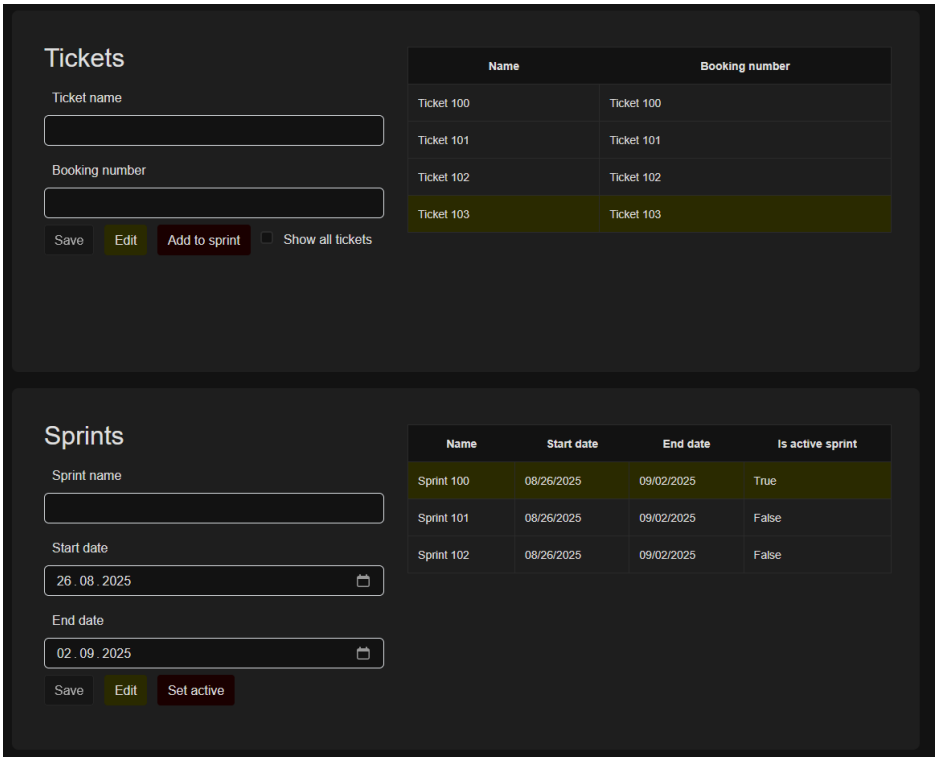


Figure 3.1: Sprint Data Page

The data displayed here can be edited as well. Note: The screenshot shows the current implementation. Theming is not yet fully finished, as focus so far has been on core functionality.

3.1.2 Meta Data Page

This page allows metadata entry for documentation and analysis purposes. These are for once the tags which are used to provide introspection for productivity classification of time slots. The other information which are the note types. These are free text classifiers for strcutured note documentation. These are meant to categorize information data for easier retrieval in the scope of tickets.

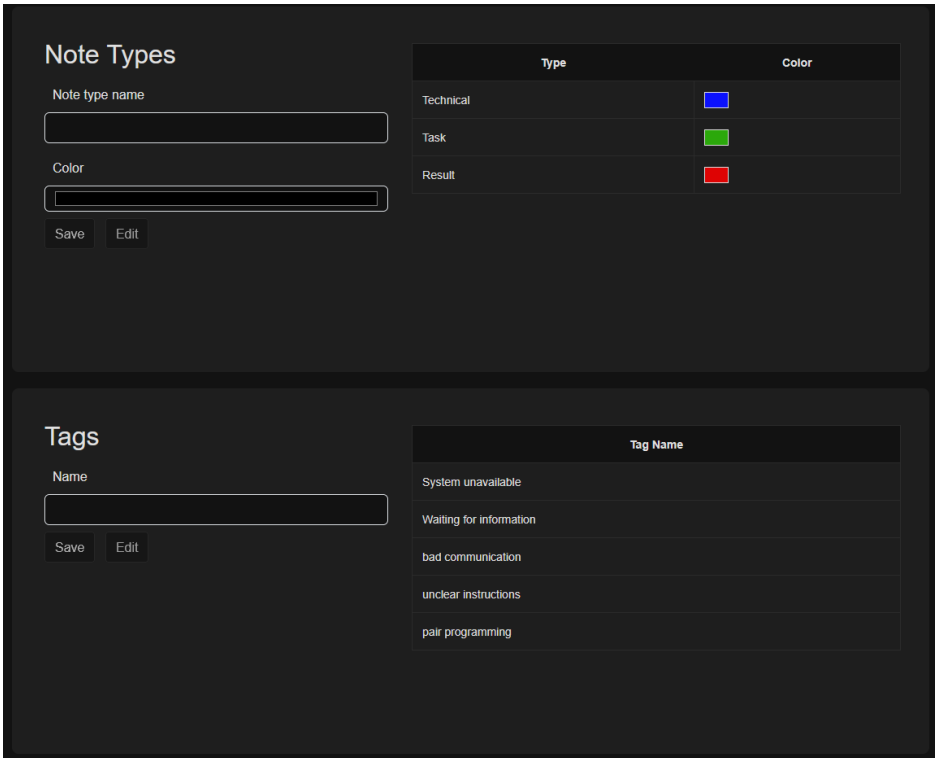


Figure 3.2: Meta Data Page

Existing metadata entries can also be edited and updated here.

3.1.3 Observability Live Feed Page

This is one of the highlights of the project. The live feed provides a chronologically sorted overview of all system actions. Each use case execution is visualized with an evaluated state.

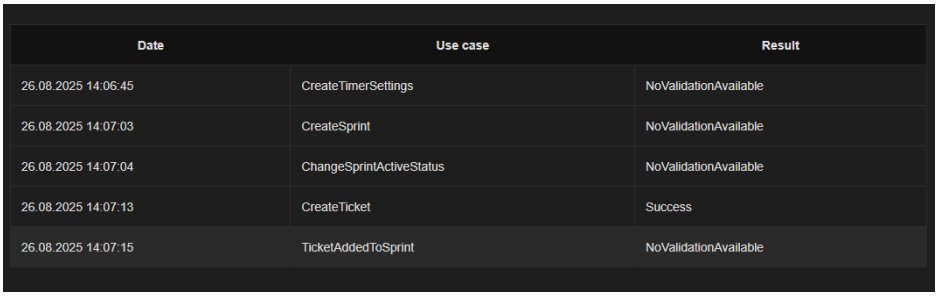


Figure 3.3: Live Feed Page

This feature allows seeing which use cases are successful, aborted, or failed with an exception. At the time of writing, only the *create ticket* use case is fully validated. However, the design allows each use case to define its own steps and validation logic.

3.1.4 Detailed Use Case Execution Report Page

This page provides a detailed stack trace of critical steps for each use case. It enables precise monitoring of data flow and helps identify issues exactly.

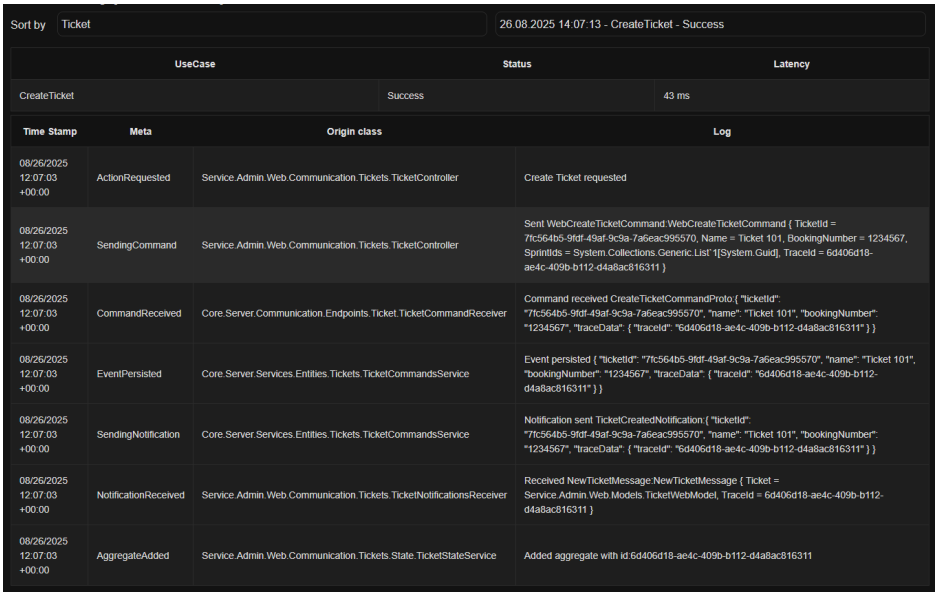


Figure 3.4: Report Page

Dropdown menus allow filtering by use case type. The second dropdown (to the right) allows selecting a specific use case instance and its execution time.

The report shows:

- The executed use case
- Start time
- Latency between start and end
- A custom stack trace with metadata logs and the relevant class

This feature already proved worth the implementation effort. Even small faulty implementation details become visible at a global scale.

Note: The logs column is not yet fully cleaned up, as this was not a priority.

3.2 Used Technologies

- Blazor Server
- Microsoft Dependency Injection
- Polly

- gRPC

No new technologies are introduced beyond those used in the Avalonia client, except for Blazor Server itself. This choice keeps the project fully within the .NET ecosystem.

3.2.1 State Management

One unique aspect of this module is the state management for data displayed in Razor pages. This is achieved using dedicated state services.

An example of such a service:

`Service.Admin.Web.Communication.Tickets.State.TicketStateService`

This service is registered as a singleton in DI, so it exists only once in the application lifecycle. It is bound to the Razor page and reloaded whenever the page is accessed:

`Service.Admin.Web.Pages.Data.Sprint.TicketComponent.razor`

4 Global Tracing

I wanted to challenge myself and created a custom global tracing solution to provide observability in distributed systems.subsection Every part of the application has its own Tracing module with tracing functionality implemented.

These modules are as of now:

- Client.Tracing
- Service.Admin.Tracing
- Core.Tracing

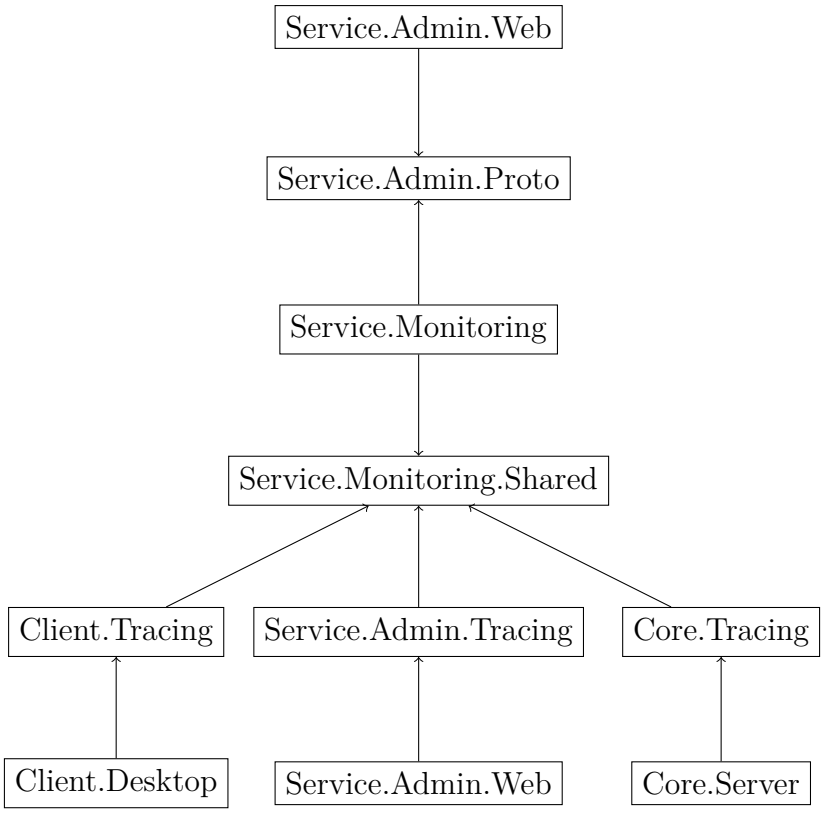


Figure 4.1: Simplified Architecture Diagram showing the Proto interfaces

The idea is to provide an easy maintainable per use case tracing between all participating services in the distributed system. The tracer is to be easily usable and injected via a simple interface.

A good example for the tracing application is:

`Core.Server.Services.Entities.Tickets.TicketCommandsService`

It shows the semantics of the tracer which is composed of interfaces for tracing each use case.

An example can be seen here:

`Service.Admin.Tracing.Tracing.Ticket.UseCase.CreateTicketTraceCollector`

`Service.Admin.Tracing.Tracing.Ticket.TicketUseCaseSelector`

Note: I know that these trace methods seem to be redundant which vialotes the DRY principle. This is true. But i wanted these traces to be as modifiable as possible while providing OCP principles. If this is a good decision is open for debate, as this was an insane amount of work to implement.

4.1 Monitoring Service

The Service.Monitoring module serves as the central ingestion point for traces emitted by the various tracing modules. Each use case is assigned a unique trace ID, and incoming events are first collected by a trace sink

`Service.Monitoring.Sink.TraceSink`

The sink then forwards the data to a verifier, which aggregates the traces per use case. To reliably determine when a use case has finished, a debounce timer is used

`Service.Monitoring.Verifiers.Common.Verifier`

On each incoming trace, the timer is reset. If no further traces arrive before the configured timeout elapses, the verifier evaluates the collected traces and generates a report. The report is then sent to the admin web frontend for presentation

`Service.Admin.Web.Communication.Reports.ReportReceiver`

5 Server Core

The Core.Server project is the backend of the application. It is used to orchestrate the clients by receiving rquests and commands, persisting the changes and publishing changes to the systems data.

The application is organized into the following modules:

- Core.Boot
- Core.Domain
- Core.Persistence
- Core.Server
- Core.Tracing

Although this could be seperated into multiple services in the future, it is currently implemented as a single modular service. There is no need to split it up as there is no scaling required as of now. But it is certainly designed to be capable of doing so.

The `Core.Boot` module provides the entry point of the server application and handles startup and service configuration.

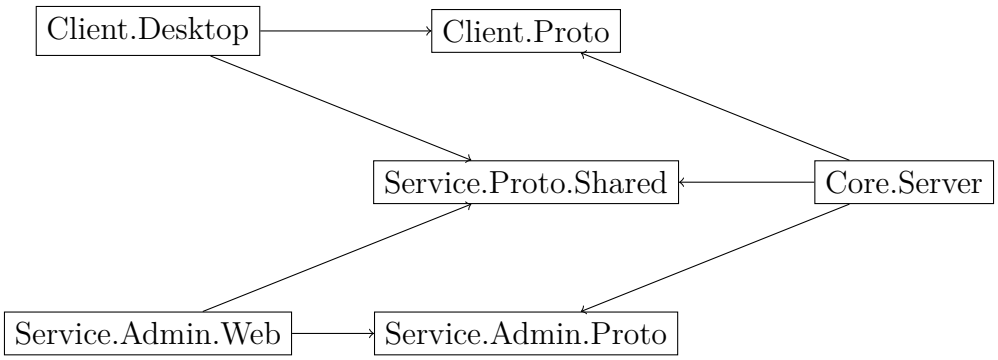


Figure 5.1: Core Proto Communication Interface

The diagram illustrates the communication structure via the Proto interfaces, applying the Interface Segregation Principle. Each Proto DTO is mapped on the client and server sides, fully encapsulating the communication layer logic and ensuring that internal domain logic remains isolated from transport concerns.

5.1 Core Dependencies

Core.Domain contains the core business logic and domain entities, while Core.Persistence is responsible for all database interactions. Core.Server processes incoming events and requests, providing the necessary services to handle them. Finally, Core.Tracing manages logging and observability, sending traces to the monitoring service.

Figure 5.2: Simplified module dependency diagram

This modular structure ensures a clean separation of concerns, keeping the backend maintainable and ready for future extensions or splitting into separate services if needed.