Final Report Template (DS-PRACTICECOURT)

Final Report for the Distributed Systems Course [2025-2026]

Author
federico.brighi2@studio.unibo.it

September 4, 2026

Abstract

DS-PracticeCourt is a distributed web-based platform designed to manage sports facility reservations (such as soccer, tennis, basketball) and associated services (like lighting, heating, possibility to rent equipment...).

The project focuses on solving the core challenges of distributed environments, such as data consistency, synchronization, high availability, and concurrency in a multi-node environment.

The system uses a distributed client-server architecture consisting of a Field Node, acting as the Two-Phase Commit (2PC) coordinator, and Utility Node serving as participant. A responsive web client provides real-time interaction through WebSockets, ensuring global state transparency for all users.

To ensure a robust and reliable system, several advanced distributed concepts are implemented. Concurrency is managed through distributed locking, exploiting Redis to guarantee that a single time slot cannot be double-booked by simultaneous requests. Atomic transactions are enforced through a Two-Phase Commit protocol that coordinates the Field and Utility Nodes, ensuring that a booking involving both a field and a service is treated as a single atomic unit. Selected slots are protected by a temporary locking mechanism with a timeout, which releases the resource if the user does not confirm the booking in time. Finally, real-time synchronization is achieved through WebSockets, which push instant availability updates to all connected users, maintaining a consistent global state across the system.

Disclaimer

During the preparation of this work, the author used Google Gemini to improve the quality and refinement of the English writing, while also minimizing translation errors. After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the final report/artifact.

1 Concept

1.1 Type Of Product

DS-PracticeCourt is a distributed web-based application supported by a microservices architecture. It provides HTTP-based APIs for core backend operations and exploits WebSockets for real-time client-server updates, all accessible through a responsive HTML/CSS/JS frontend.

1.2 Use Case Description

The software allows users to browse and book sports facilities (football, basketball, tennis) alongside optional utility services such as heating, lighting, or equipment rental. Users interact with the system remotely from any location via standard web browsers on desktop or mobile devices. To make the scenario even more realistic, interaction frequency could become highly bursty and concurrent during peak hours when multiple users compete for the best time slots. To handle this, users experience a visual Temporary Hold mechanism that gives immediate feedback when someone else is evaluating to book a specific time slot. The platform conceptually manages multiple roles: standard Users, who browse availability and book slots, and Administator (only possible acting via terminal or IDE) with the possibility to monitor the system states (like checking if Docker nodes are up or down) and trigger manual recovery procedures for pending transactions. To function correctly the system persistently stores domain entity such as sports field, utilities and transactional booking states. This data is securely stored across MySQL databases associated with each microservice, meanwhile transient data (such as temporary slot hold and distributed locks) is managed in-memory via Redis.

1.3 Why Is Distribution Needed ?

Distribution is a key requirement for this project, driven by the following factors:

1.3.1 Resource Separation And Autonomy:

In a realistic scenario, the management of physical fields and the provisioning of utilities are often handled by distinct subsystems or even different providers. Splitting them into a distributed architecture (Field Node and Utility Node) reflects this domain boundary, allowing independent scaling, maintenance and deployment.

1.3.2 Concurrency And Resource Sharing:

In sports facility booking platforms, especially in the most requested time slots, extremely high contention scenarios may occur, with many users attempting to reserve the same slot simultaneously. To guarantee consistency and prevent double bookings, the system adopts a distributed architecture with Redis-based distributed locking mechanisms, capable of safely handling parallel requests across multiple nodes.

1.3.3 Fault Tolerance And Consistency:

Network failures or temporary node unreachability are inevitable in modern web environments. By distributing the system, fault tolerance is enforced through a Two-Phase Commit (2PC) protocol and a background recovery loop. If the Utility Node crashes or a network partition occurs during a booking, the Field Node coordinator can autonomously roll back or eventually commit the transaction when the connection is restored. This ensures the system never reaches an inconsistent state where a field is booked without the utilities requested by the client.

2 Requirements Elicitation and Analysis

This section provides functional, non-functional and implementation requirements for the implemented system.

2.1 Glossary

To avoid ambiguity, the following domain-specific terms are used throughout this document:

- **Field**: A sport facility (like a soccer or tennis court) that can be reserved by a user.
- **Utility**: An optional extra service associated with a field booking (like lighting, heating or equipment rental).
- **Slot**: A specific time interval on a given date during which a field can be booked.

- **Temporary Hold**: A transient state indicating that a user is currently evaluating a slot, temporarily preventing others from booking it.
- **Booking**: The complete reservation requests, which may involve both a Field and 0 or more Utilities.
- **TTL**: Time To Live of a temporary lock or cached entry, namely the period after which it expires automatically if it is not renewed or removed.

2.2 Functional Requirements

Functional requirements define the specific behaviors and capabilities the system must provide to the users.

- **FR1 - Browse Availability**: Users must be able to view the list of fields and their current availability for specific dates and slots.
  - *Acceptance Criterion*: The system correctly displays available, currently held, and already booked slots for a selected date.
- **FR2 - Temporary Slot Hold**: Users must be able to temporarily reserve a slot while completing the booking process.
  - *Acceptance Criterion*: When a user selects an available slot, it becomes visually locked for other users for a defined timeframe (60 seconds). If the booking is not finalized by the user, the hold expires and the slot becomes available again for all the other users.
- **FR3 - Book Field and Utilities**: Users must be able to finalize a booking that includes a field and optionally 1 or more utilities.
  - *Acceptance Criterion*: The user receives a successful confirmation only if both the field and the selected utilities are successfully booked.
- **FR4 - Real-Time State Updates**: Connected users must receive immediate updates regarding slot availability changes.
  - *Acceptance Criterion*: When a slot is booked, held, or released, all active clients update their UI automatically without requiring a page refresh.
- **FR5 - Manual Recovery Trigger**: Administrators must be able to manually trigger a recovery process for pending or stuck transactions.
  - *Acceptance Criterion*: Calling the dedicated admin endpoint successfully resolves any pending transactions into either a definitively confirmed or failed state.

2.3 Non-Functional Requirements

Non-functional requirements specify the quality attributes and constraints of the system.

- **NFR1 - Atomicity (Consistency)**: A booking involving multiple resources (fields and utilities) must be strictly atomic.
  - *Acceptance Criterion*: The system never leaves a booking in a partially completed state. Either both the field and the selected utilities are confirmed, or neither is.
- **NFR2 - Concurrency Control (Isolation)**: The system must handle parallel booking requests safely, preventing double-bookings.
  - *Acceptance Criterion*: If two users simultaneously attempt to book the exact same slot, only one succeed, while the other receives an availability error.
- **NFR3 - Fault Tolerance**: The system must be able to recover from temporary node failures during distributed bookings.
  - *Acceptance Criterion*: If a participating node becomes unreachable during a transaction, the coordinator eventually restores a consistent global state once the connection is re-established. If the coordinator fails, the system may temporarily stop processing distributed bookings, but no inconsistent booking state is left committed.

2.4 Implementation Requirements

Implementation requirements constrain the technological choices adopted for the realization of the system.

- **IR1 - Python and FastAPI**: The backend was developed using Python 3.12 and FastAPI [4].
  - *Justification*: These technologies were selected because they provide a modern asynchronous framework, good support for API development, and strong compatibility with the distributed and event-driven nature of the project. Also this is the programming language that we used during the practical lessons of the course.
  - *Acceptance Criterion*: The codebase runs on Python 3.12 and successfully exposes endpoints via the FastAPI framework.
- **IR2 - Containerization**: The system was deployed using Docker and Docker Compose. [2].
  - *Justification*: These tools were chosen to easily simulate a multi-node distributed environment and to simplify deployment, reproducibility, and local testing.
  - *Acceptance Criterion*: A single `docker compose up` command builds and starts the entire multi-node infrastructure.
- **IR3 - Redis for Transient State**: Distributed locks and temporary hold are

managed in-memory via Redis [3].

- *Justification*: Redis was selected because it provides fast in-memory operations, native key expiration, and efficient support for distributed locking without overloading the persistent database.
- *Acceptance Criterion*: The system utilizes Redis `SET NX PX` commands for lock acquisition and expiration

2.5 Relevant Distributed System Features

Several distributed-system features are relevant for DS-PracticeCourt, while others play only a marginal role.

- **Transparency**: Partial transparency is desirable. Users should not need to know how bookings are coordinated internally, nor how the temporary holds and distributed locks are implemented. However, full location transparency is not required, since the system is intended to expose a web-based interface and the user does not directly interact with the individual nodes. On the other hand, some implementation details, such as the presence of the Field Node and the Utility Node, remain relevant to developers and are reflected in the architecture.
- **Fault Tolerance & Dependability**: This is a key feature of the project. Since booking operations involve multiple distributed components, failures must not lead to inconsistent reservations. In particular, the system must preserve integrity even when a participant becomes temporarily unreachable during a transaction. Uninterrupted service is not always guaranteed, especially if the coordinator (Field Node) fails, but the system must recover to a consistent state as soon as possible and avoid partial commits or data corruption. From a theoretical perspective, DS-PracticeCourt deliberately prioritizes consistency over availability, following the CP trade-off described by the CAP Theorem. In the presence of a network partition betweemm the FieldNode and the UtilityNode, the system refuses to comple a distributed booking rather than risk an inconsistent reservation stte. I found this choice appropriate for the domain of my project, where an incorrect booking confirmation is worse than a temporary refusal of service.
- **Scalability**: The system should support an increasing number of users attempting to reserve fields and services concurrently, (in a realistic case during peak-hour time slots). Although the project is not designed for massive scale. it should remain functional and responsive as the number of bookings, users, and available resources grows.
- **Security & Trust**: Security is relevant but not the main focus of the project. The system does not process sensitive personal data or require strong access-control policies, and authentication is not a primary requirement. Yet, trust in the booking process is important, since users must rely on the system to reserve resources correctly and consistently

- **Resource Sharing**: This is one of the central concerns of the application. Multiple users compete for the same finite sport facilities and utility services, so the system must coordinate concurrent access to shared resources. Synchronization is therefore essential to avoid double bookings and to ensure that booking states remain coherent across the nodes.
- **Openness, Interoperability & Heterogeneity of components**: These aspects are relevant only to a limited measure. The system is built as a controlled academic project with a predefined technology stack, so interoperability with external systems is not a major requirement. Yet, the use of standard web protocols and a clear separation between components makes the architecture reasonably open to future integration or replacement of individual modules.
- **Evolvability & Mantainability**: These features are important because the system is modular and could be extended with additional services, booking rules, or UI features in the future. The separation between Field Node, the Utility Node, shared models and the frontend improves maintainability and makes it easier to evolve the system without redesigning it from scratch.
- **Performance, Concurrency, Computation/Communication, Efficiency & Bandwidth**: Performance is highly relevant, because the system must handle concurrent reservation requests and provide fast feedback to users. Booking operations should remain responsive even under contention, and real-time updates should propagate with low latency. Network usage should also remain controlled, since the application exchanges frequent but relatively small messages between clients, the coordinator, and the participant node.
- **Economy & Costs**: This aspect is secondary: the system is developed as an academic project and deployed locally through containerization, so there are no significant operational costs. Minimizing resource usage is useful but it is not a primary design driver.

3 Design

3.1 Architecture

The system adopts a distributed client-server architecture organized as a small microservice-based system. In particular, the application is split into two main backend nodes: the *Field Node*, which acts as the coordinator of distributed booking operations, and the *Utility Node*, which participates in the reservation of additional services. A web-based frontend acts as the client-side entry point for users. This architectural style was chosen because it naturally matches the domain structure of the project: sports fields and additional utilities are logically distinct resources, but must be reserved atomically when part of the same booking request. The client-server perspective is also important from the user's point of view: users interact only with the web frontend, while the backend

handles coordination, persistence, and real-time synchronization transparently. This reduces complexity for the user and keeps the distributed structure hidden behind a simple browser-based interface.

3.2 Infrastructure

The system infrastructure is composed of a web client, two backend nodes, a shared support layer, and two infrastructural services, namely Redis and MySQL. As shown in the following component diagram fig. 1, the *Field Node* acts as the coordinator of distributed booking operations, while the *Utility Node* acts as the participant responsible for managing additional services associated with a reservation. The *Web Client* is the only component directly accessed by end users: through the booking user interface, clients interact with the system by sending HTTP requests to the public APIs exposed by the Field Node. In addition, a WebSocket listener allows the client to receive real-time updates about booking availability, thus keeping the frontend synchronized with the current global state of the system.

The *Field Node* contains the main entry points of the application: it exposes the public booking APIs, handles field-related operations through the *Field Repository* and the *Field Booking Repository*, and includes the logic for distributed transaction coordination and recovery. The *Utility Node*, instead, exposes only internal APIs, which are used by the coordinator during the execution of the Two-Phase Commit protocol. Its repositories manage the persistence and update of utility-related data.

Both backend nodes rely on a set of *Shared Resources*, including configuration utilities, database-session management, shared domain models, validation schemas, and Redis-based locking support. This shared module ensures consistency between the two services and avoids duplicating common coordination logic. From an infrastructural point of view, MySQL is used as the persistent storage layer of the application, while Redis is used to support transient distributed state. In particular, Redis is exploited for distributed locks, temporary slot holds, and coordination-related state, whereas MySQL stores the persistent entities of the booking domain. All components are deployed within the same controlled environment and communicate through explicit service-level interactions.

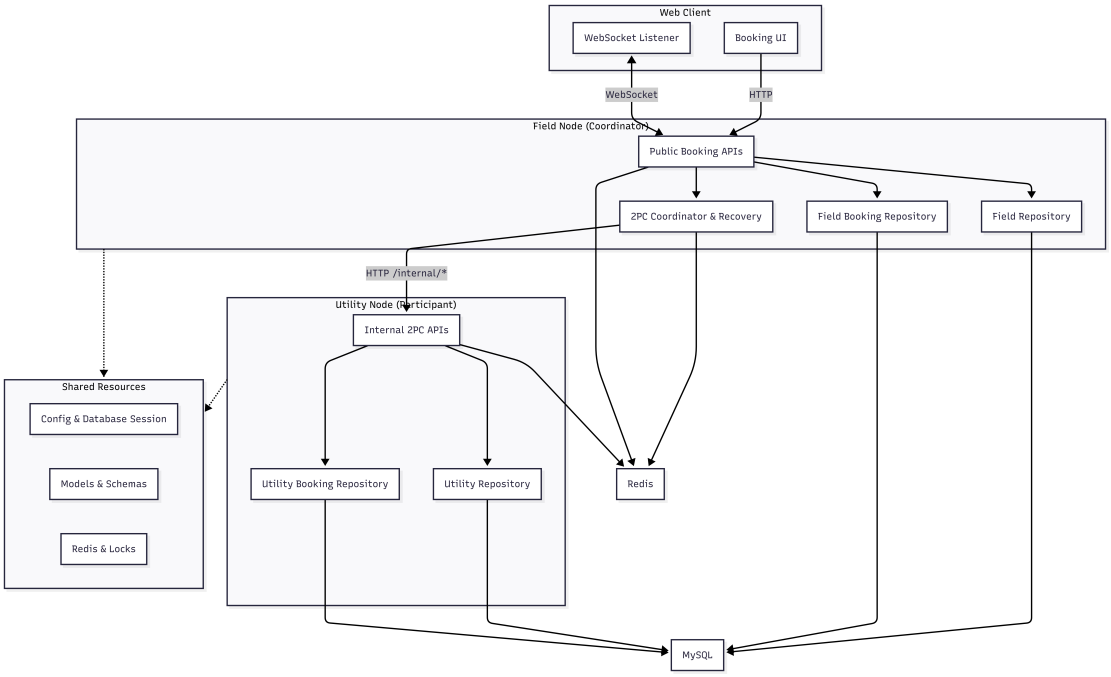

Figure 1: Component Diagram of the infrastructure

3.3 Modelling

The domain model of my project revolves around a small set of entities that represent the booking logic of the application. The main domain entities are *Field*, *Utility*, *FieldBooking*, and *UtilityBooking*. A *Field* represents a sports facility that can be reserved by a user, while a *Utility* represents an optional extra service associated with a booking. A *FieldBooking* stores the reservation of a field for a specific time interval, while a *UtilityBooking* stores the reservation state of the utilities associated with the same transaction.

The domain model is shown in the class diagram which highlights the main relationships between fields, bookings, utilities, and utility bookings as well as the *BookingStatus* enumeration used to represent the lifecycle of a reservation.

These entities are mapped to the infrastructural components in a distributed way: the persistent state of fields, bookings, utilities and utility bookings is stored in MySQL, while the application logic that manipulates them is split across the Field Node and the Utility Node. The Field Node is responsible for field-related entities and acts as the coordinator of distributed bookings. The Utility Node is responsible for utility-related entities and participates in the distributed transaction through internal endpoints. Shared domain models and schemas are reused by both nodes to guarantee a consistent repre-

sentation of the same entities across the system.

The most relevant domain events are *slot selected*, *booking created*, *booking confirmed*, *booking failed*, *utility reservation prepared*, and *transaction recovered*. These events reflect the main transitions in the lifecycle of a reservation and are used to keep the system state coherent across components.

The system exchanges several types of message: User-facing requests are expressed as HTTP commands from the web client to the Field Node, Internal coordination messages are exchanged between the Field Node and the Utility Node during the Two-Phase Commit protocol, while real-time availability updates are published as events through Redis and delivered to connected clients through WebSockets. Queries are used to retrieve availability, field, utilities, and booking information, while commands are used to create, confirm, or recover reservations. The overall system state includes the set of sports fields, the available utilities, the bookings in progress, the confirmed reservations, the failed or cancelled transactions, and the transient coordination data stored in Redis. In particular, Redis keeps temporary information such as distributed locks, temporary holds, and 2PC transaction state, while MySQL stores the persistent booking domain data.

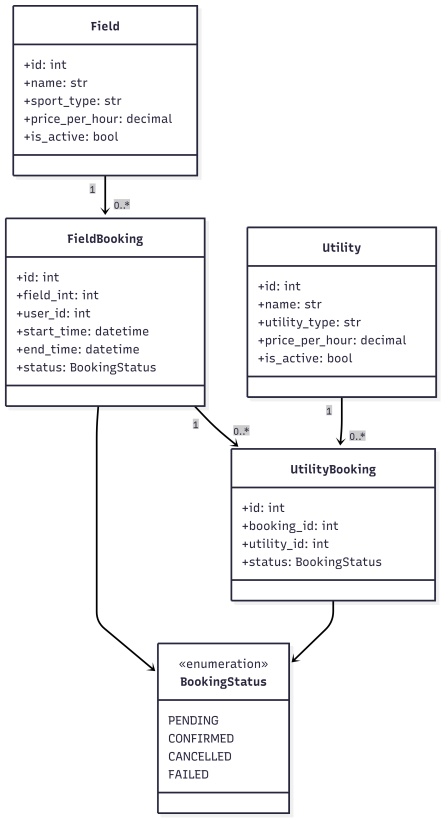

Figure 2: Domain Model of the project (Class Diagram).

3.4 Interaction

The components of the project interact through a combination of synchronous HTTP requests, asynchronous event notifications, and real-time WebSocket communication. The most important interaction pattern is the distributed booking workflow, in which the Field Node coordinates the reservation of a field and the associated utility services. In this workflow, the web client sends a booking request to the Field Node, which in turn contacts the Utility Node through internal HTTP calls during the execution of the Two-Phase Commit protocol.

The main interaction pattern is therefore a coordinator-participant scheme: the Field Node acts as the coordinator, while the Utility Node acts as a participant that evaluates prepare, commit, and rollback requests. This pattern is used to guarantee atomicity across distributed resources and to prevent inconsistent reservation states. If all participants vote positively, the transaction is committed; otherwise, it is aborted and the system returns to a consistent state.

A second relevant interaction pattern is real-time state propagation. After a booking is confirmed, failed, or released, the Field Node publishes an availability event through Redis Pub/Sub. The Web Client listens to these updates through a WebSocket connection and updates the user interface immediately, without requiring a manual refresh. This keeps all connected users synchronized with the current availability of fields and services.

A further interaction pattern appears during recovery: if a distributed transaction remains in an intermediate state, the Field Node scans the stored transaction metadata and attempts to complete or abort the operation by contacting the Utility Node again. This recovery interaction is periodic and best-effort, and it is used to restore consistency after temporary failures or incomplete 2PC executions. The following Sequence Diagram shows the booking workflow based on the Two-Phase Commit protocol.

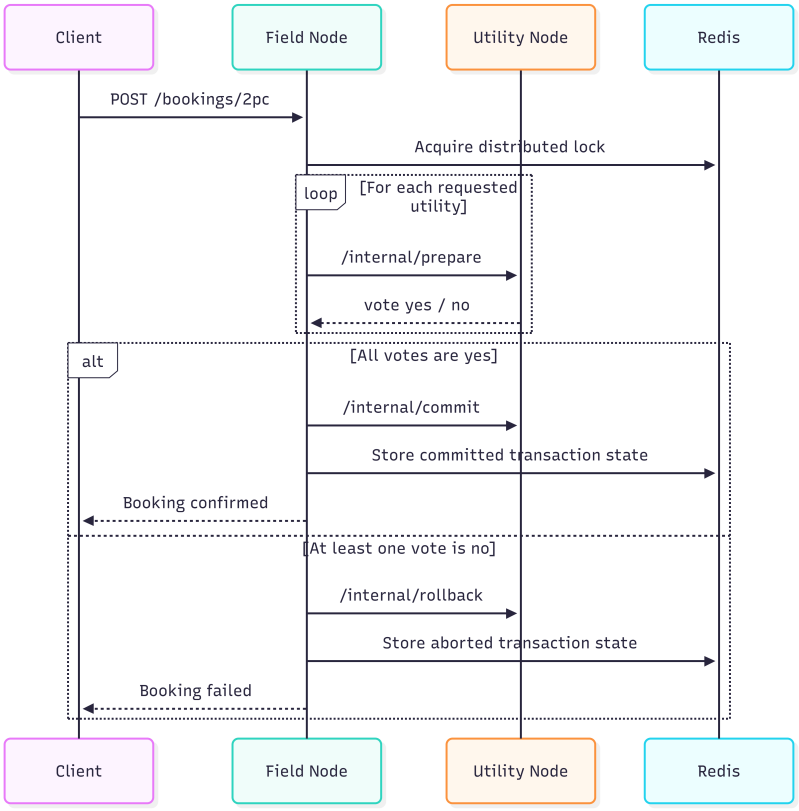

Figure 3: Sequence diagram of the Two-Phase Commit booking workflow.

3.5 Behaviour

The components of this project show different behavioral roles depending on their role in the booking workflow. The Web Client is mostly stateful from the user's perspective, because it maintains the current view of the booking interface and reacts to real-time updates. The Field Node is the main stateful backend component: it updates the global booking state, coordinates distributed transactions, and publishes availability changes. The Utility Node is also stateful with respect to utility reservations, but its behavior is more limited since it only reacts to the internal requests coming from the coordinator. Redis behaves as a transient coordination component, storing temporary locks, temporary holds, and transaction metadata, while MySQL stores the persistent booking data.

The system state is updated mainly by the Field Node during the booking process: when a user selects a slot the Field Node records the temporary hold and prevents other users from reserving the same slot. If the booking is confirmed, the Field Node updates the booking status and coordinates the final state of the related utility reservations through

the Two-Phase Commit protocol. If the booking is aborted or the temporary hold expires, the slot is released and becomes available again. In this way, the Field Node is the main component in charge of updating the state of the system, while Redis is used to support transient state transitions and MySQL is used to persist final outcomes.

The lifecycle of a *FieldBooking* is shown in fig. 4: a booking is created in the `PENDING` state when the user finalizes the reservation request. If the 2PC protocol completes successfully, or the recovery procedure confirms the transaction, the booking transitions to `CONFIRMED`. If at least one participant votes NO, or the recovery fails, the booking is marked as `FAILED`. Finally, if the user explicitly cancels before confirmation, the booking moves to `CANCELLED`.

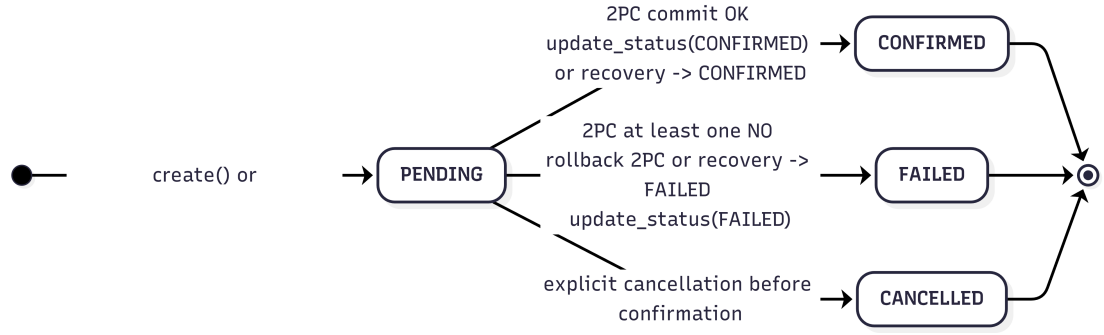

Figure 4: State Diagram of the Field Booking

Instead, the lifecycle of the distributed transaction is shown in fig. 5: the transaction starts in *INIT*, moves to *PREPARED* after all participants vote positively, and then reaches either *COMMITTED* or *ABORTED*, depending on the outcome of the second phase or the recovering procedure. This state model captures the coordination logic behind atomic reservations and highlights the role of the Field Node as the component responsible for driving the system towards a consistent final state.

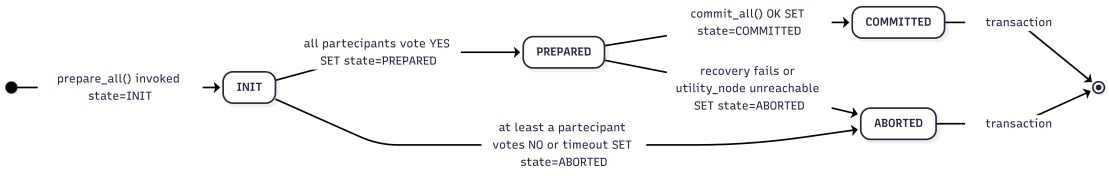

Figure 5: State Diagram of the 2PC Transaction

3.6 Data and Consistency Issues

DS-PracticeCourt stores both persistent and transient data: persistent data includes sports fields, utilities, field booking, and utility bookings, while transient data includes distributed locks, temporary hold, and transaction metadata used during booking coordination. Persistent data is stored in MySQL because the application requires relational storage with referential integrity, structured querying, and durable persistence across sessions. Transient coordination data is stored in Redis because it must be updated quickly, expire automatically when need, and support lightweight distributed synchronization.

The persistent state is organized as a relational model: the system stores fields, utilities, field bookings, and utility bookings in separate tables, with foreign-key relationships connecting the booking entities to their associated resources. I found this choice appropriate because the booking domain is naturally structured and the system needs to query and update related records consistently. The use of a relational database also simplifies the management of booking states and makes it easier to enforce integrity constraints on the stored data.

Database queries are performed mainly by the Field Node and the Utility Node: the Field Node queries the database when users browse fields, create bookings, check availability, update booking status, or execute recovery actions, while the Utility Node queries the database when it evaluates internal prepare, commit, and rollback requests for utilities. The queries are both read and write operations, and concurrent writes are carefully controlled through distributed locking and transactional coordination. Concurrent reads are common when multiple users browse availability, while concurrent writes occur when several users attempt to reserve the same slot at the same time.

Some data must be shared between the two backend nodes, like shared configuration, domain models, validation schemas, and database-session utilities are reused by both nodes to guarantee a consistent representation of the same entities. In addition, the Redis-based temporary coordination state is shared during distributed booking operations, so that the Field Node can coordinate the transaction and the Utility Node can participate in it using the same transaction metadata. This shared state is essential to keep the distributed workflow coherent and to ensure that final booking outcomes remain consistent across components.

3.7 Fault-Tolerance

My project does not implement full data replication across nodes. Persistent domain data is stored in MySQL, while transient coordination data is shared through Redis. This design does not duplicate the persistent state, but it does provide a controlled form of shared state that allows the backend nodes to coordinate distributed booking operations. In particular, Redis is used to store distributed locks, temporary holds, and transaction metadata so that the Field Node and the Utility Node can observe and update the same coordination state during a booking workflow.

Timeouts and retry-like mechanisms are used in several parts of the system: HTTP calls between the Field Node and the Utility Node are performed with explicit timeouts, so that the coordinator does not wait indefinitely for a participant that is temporarily unavailable. The same principle applies to booking-related operations exposed to the web client, which must fail fast if the system cannot guarantee a consistent outcome. In addition, temporary holds are implemented with a TTL, so that a slot is automatically released if the user does not complete the booking within the configured time window.

The handling of errors is centered on preserving consistency. If a participant fails during the prepare or commit phase, the Field Node aborts the distributed transaction and restores the booking state to a consistent outcome. If a transaction remains in an intermediate state, the recovery procedure can later inspect the stored transaction metadata and attempt to complete or abort it. When the coordinator itself fails, the system may temporarily stop processing distributed bookings, but no invalid final state is committed. This behavior is intentional because the project prioritizes consistency and integrity over uninterrupted availability.

The recovery behavior is illustrated in fig. 6: the recovery loop periodically scans Redis for transactions in the *PREPARED* state and, for each one, attempts to complete the commit phase. If the commit succeeds, the related field booking is marked as *CONFIRMED* and the transaction metadata is removed. If the commit fails or the Utility Node remains unreachable, the recovery logic rolls back the utilities, marks the booking as *FAILED*, and deletes the corresponding coordination data. In both cases, the system converges to a consistent final state.

Overall, fault tolerance in DS-PracticeCourt is achieved through a combination of distributed locks, timeouts, transaction recovery and explicit error handling. These mechanisms do not eliminate failures, but they ensure that failures are detected, contained, and resolved without leaving the booking domain in an inconsistent state.

From the CAP theorem perspective, the system is a **CP System**: it guarantees consistency and partition tolerance, but sacrifices availability when the UtilityNode is unreachable. This is a deliberate desgin decision, because accepting an inconsistent reservation state (like a field booked without its requested utilities) whould be semantically incorrect and unaccettable for the booking domain. It is also worth noticing that the 2PC transaction state persisted in Redis carries a TTL of 300 seconds, so if the FieldNode crashes and remains unreachable for longer than this window, the PREPARED state will expire before the recovery loop can act on it, leaving the corresponding FieldBooking in a permanent PENDING state on MySQL.

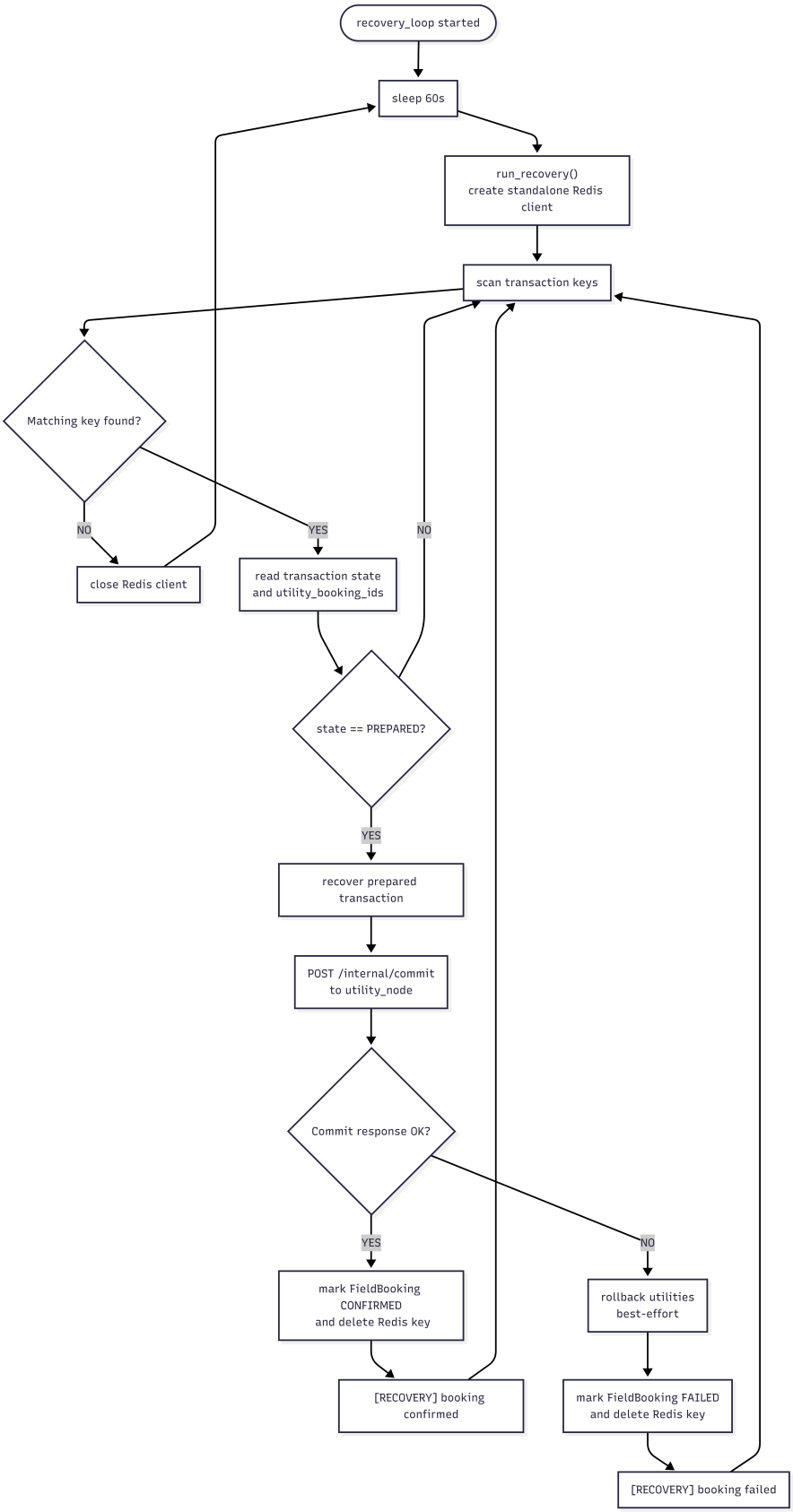

Figure 6: Activity Diagram of the recovery loop for prepared 2PC transactions.

3.8 Availability

DS-PracticeCourt does not rely on a traditional caching layer for persistent data. However, the system uses transient in-memory state to improve responsiveness and keep the booking workflow usable under contention. Especially, Redis stores temporary holds, distributed locks, and transaction metadata, which act as a lightweight coordination cache for short-lived booking operations. This improves responsiveness because the system can quickly determinate whether a slot is already being evaluated or reserved by another user.

No explicit load balancing mechanism is introduced in the current design. The system is intentionally small and composed of a limited number of components deployed in a controlled Docker Compose environment, so requests are routed directly to the relevant service without the need for additional balancing infrastructure. The simplicity of this deployment model is sufficient for the expected workload of the project idea and avoids unnecesary complexity.

In the presence of network partitioning, the system prioritizes consistency over uninterrupted availability: if the Field Node cannot reach the Utility Node, distributed bookings cannot be safely completed and the affected transaction is aborted or left in a recoverable intermediate state. Temporary slot holds expire automatically, so resources are not permanently blocked, and the recovery mechanism can later restore consistency once communication is re-established. This behaviour is appropriate for the domain, because accepting inconsistent reservations would be worse than temporarily refusing a booking request.

3.9 Security

The system does not implement any authentication or authorization mechanisms. It is designed as an academic prototype focused on distributed coordination, consistency, and real-time synchronization, rather than on user account management or access control. For this reason, users are not required to log-in on the application, and no role-based access policy is enforced beyond the functional distinction bewteen the web client and the backend services.

4 Implementation

This section describes the main technology-dependant choices that I adopted while implementing DS-PracticeCourt. The system was developed as a distributed web application composed of two backend services, a browser-based frontend, and 2 infrastructural components dedicated to persistence and coordination. In particular, the implementation follows the design introduced in the previous chapter: the *Field Node* acts as the coordinator of the distributed booking operations, while the *Utility Node* acts as the

participant responsible for additional services.

The whole system is deployed through Docker Compose, which starts the two application nodes together with MySQL and Redis in the same controlled environment. This approach simplifies reproducibility and local testing, while still allowing the project to simulate a realistic distributed setting with multiple cooperating services.

At the communication level, the system mainly relies on HTTP over TCP. Public operations exposed to the frontend, such as browsing fields, creating bookings, and cancelling reservations are implemented as HTTP endpoints in the Field Node. Internal coordination between the Field Node and the Utility Node is also implemented through HTTP calls, especially for the *prepare*, *commit*, and *rollback* phases of the Two-Phase Commit protocol.

For a real-time synchronization, the system uses WebSockets between the Field Node and the web client. This mechanism allows the frontend to receive immediate notifications about booking confirmations, failures, cancellations, and temporaru slot holds or releases. As a consequence, connected clients remain synchronized with the current availability state without requiring continuous polling.

Data exchanged between components is represented in JSON, using shared schemas for serialization and validation. Persistent data is stored in MySQL and accessed through SQLAlchemy, with asynchronous sessions, while Redis is used for transient coordination state such as distributed locsk, temporary holds, Pub/Sub events, and 2PC transaction metadata. This separation between durable data and short-lived coordination data reflects the different consistency and lifetime requirements of the two kinds of information.

Concurrency control is enforced through Redis-based distributed locks built on the `SET NX PX` primitive. Locks are acquired both for ordinary booking requests and for distributed bookings coordinated through 2PC, preventing concurrent requests from reserving the same slot independently and preserving consistency under contention. The acquisition is atomic by design: `nx=True` ensures the key is set only if it does not already exist, while `px=ttl_ms` attaches an automatic expiry in milliseconds, so that a lock is released even if the holding node crashes before completing the operation. Each lock acquisition returns a unique token generated via `uuid4`, which acts as a ownership proof and is required to release the lock.

The release path is equally critical ... a plain `GET` followed by `DEL` would not be sufficient: even though Redis is single-threaded, a sequence of two separate commands is not atomic from the client perspective, because another client could acquire the same lock in the interval between the two calls. To solve this, the release is implemented as a Lua script sent to Redis and executed server-side as a single indivisible operation. The script checks that the stored token matches the caller's token before issuing the `DEL`: if the lock has already expired and been acquired by another node, the delete is skipped, preventing an accidental release of a lock the caller no longer owns. The full acquire and release logic is shown in listing 1.

Listing 1: Distributed Lock: Atomic Acquire (SET NX PX) and Lua-based atomic release

```
async def acquire(self, key, ttl_ms) -> Optional[str]:
    token = str(uuid.uuid4())
    result = await self._client.set(
        self._build_key(key),
        token,
        nx=True,
        px=ttl_ms,
    )
    return token if result else None

_LUA_RELEASE_SCRIPT = """
if redis.call('get', KEYS[1]) == ARGV[1] then
    return redis.call('del', KEYS[1])
else
    return 0
end
"""
```

The Two-Phase Commit protocol is implemented explicitly in the Field Node, which acts as the coordinator. During the prepare phase, the coordinator first records the transaction in Redis with an `INIT` state, then contacts the Utility Node for each requested utility via an HTTP `POST` to the `/internal/prepare` endpoint. Each call is subject to a fixed timeout: if the Utility Node is unreachable, or returns an error, the exception is caught and treated as an implicit *NO* vote, immediately aborting the transaction without waiting for the remaining participants. Only if all participants respond with a *YES* vote the coordinator advance the Redis state to `PREPARED` and proceed to the commit phase; otherwise, a rollback is triggered and the field booking is marked as `FAILED`. This design ensures that the system never reaches a state in which a field is confirmed while the associated utility reservations are not, preserving atomicity across nodes. The prepare phase is shown in listing 2.

Listing 2: Two-Phase Commit coordinator: prepare phase with timeout and Redis state tracking

```
async def prepare_all(utility_node_url, redis, field_booking_id,
   utility_ids):
    confirmed_ids = []
    await _set_txn_state(redis, field_booking_id,
       TwoPCTransactionState.INIT, [])

    async with httpx.AsyncClient(timeout=_HTTP_TIMEOUT) as client:
        for utility_id in utility_ids:
            req = PrepareRequest(field_booking_id=field_booking_id
               , utility_id=utility_id)
            try:
                resp = await client.post(
```

```
                    f"{utility_node_url}/internal/prepare",
                    json=req.model_dump()
                )
                resp.raise_for_status()
                vote = PrepareResponse.model_validate(resp.json())
                if vote.vote == TwoPCVote.YES:
                    confirmed_ids.append(vote.utility_booking_id)
                else:
                    return False, confirmed_ids
            except (httpx.HTTPStatusError, httpx.TimeoutException,
                httpx.RequestError):
                return False, confirmed_ids

    await _set_txn_state(redis, field_booking_id,
        TwoPCTransactionState.PREPARED, confirmed_ids)
    return True, confirmed_ids
```

To handle temporary failures, the system includes a background recovery mechanism for transactions left in the `PREPARED` state after a node crash or a network partition. A dedicated coroutine, started as an `asyncio` task at Field Node startup, wakes up every `_RECOVERY_INTERVAL_S` seconds and scans Redis for incomplete transactions. The scan uses the incremental `SCAN` command, so it does not block the Redis server during execution. For each transaction found in the `PREPARED` state, the loop attempts to re-contact the Utility Node and complete the commit; if the node is still unreachable, the transaction is rolled back and the corresponding field booking is marked as `FAILED`, ensuring that no booking is left in a permanently inconsistent intermediate state. A manual recovery endpoint is also exposed for testing and demonstration purposes. The implementation is shown in listing 3.

Listing 3: Background recovery loop: incremental Redis scan for PREPARED transactions

```
async def recovery_loop(utility_node_url: str):
    while True:
        await asyncio.sleep(_RECOVERY_INTERVAL_S)
        await run_recovery(utility_node_url)

async def run_recovery(utility_node_url: str) -> None:
    redis = redis_manager.get_client()
    cursor = 0
    try:
        while True:
            cursor, keys = await redis.scan(
                cursor, match=f"{_TXN_KEY_PREFIX}*", count=100
            )
            for key in keys:
                data = json.loads(await redis.get(key))
                if data.get("state") != TwoPCTransactionState.
                    PREPARED.value:
```

```
                continue
            field_booking_id = int(key.removeprefix(
                _TXN_KEY_PREFIX))
            utility_booking_ids = data.get("
                utility_booking_ids", [])
            await _recover_one(
                redis, field_booking_id, utility_booking_ids,
                    utility_node_url
            )
        if cursor == 0:
            break
finally:
    await redis.aclose()
```

The frontend is implemented as a static web interface based on HTML, CSS and JavaScript. It communicates with the backend through HTTP for standard operations and WebSockets for live updates.

The system does not currently implement authentication or authorization. This is a delibarate simplification consistent whith the academic nature of the prototype and with the main goal of the project, which is distributed cooridnation rahter than access controlled, as disscussed in the Security subsection. Logging is instead configured centrally in both nodes, so that execution traces include timestamps, severity levels, and node identifiers, making debugging and observation of distributed interactions easier.

4.1 Technological details

The backend services are implemented in Python 3.12 using FastAPI. This framework was chosen because it supports asynchronous request handling, dependency injection, HTTP APIs, and WebSockets in a clean and compact way. Dependency management is handled through Poetry, which helps keep the project reproducible and well organized.

At the persistence layer, the system uses MySQL 8.0 together with SQLAlchemy ORM and the `aiomysql` driver. Database access is managed through a shared asynchronous session manager, while repository classes separate endpoint logic from persistence logic. This improves modularity and keeps the implementation easier to maintain.

For transient coordinatiom the system uses Redis 7: Redis supports distributed locking, temporary holds with TTL, Pub/Sub notifications, and transaction-state storage for the 2PC protocol. These features make it particularly suitable for short-lived coordination data that must be updated quickly and expire automatically.

Deployment is managed through Docker and Docker Compose. The Compose configuration defines 4 main services: MySQL, Redis, Field Node, and Utility Node. Environment variables are used to configure datebase access, Redis communication, node identitity, and the address of the Utility Node used by the coordinator.

5 Validation

The validation strategy was designed to verify both the correctness of the individual components and the correctness of their distributed interaction. To this end, the system was tested in a Docker Compose environment that mirrors the production-like deployment used for development and execution. This choice ensured that the test results reflected the behavior of the real multi-node setup, including the Field Node, the Utility Node, Redis, and MySQL.

5.1 Automatic Testing

The automated test suite was organized into separate files, each targeting a specific aspect of the system. The following list summarizes the main test files and their corresponding testing category.

- `test_2pc_coordinator.py` – Unit Tests for the Two-Phase Commit coordinator.
- `test_2pc_internal_endpoints.py` – Integration Tests for the internal Utility Node 2PC endpoints.
- `test_2pc_end2end.py` – End-to-End Tests for complete distributed booking workflows.
- `test_concurrency.py` – Concurrency Tests for simultaneous booking requests on the same slot.
- `test_fault_tolerance.py` – Fault-Tolerance and Recovery Tests.
- `test_temporary_hold.py` – Temporary Hold and Redis Coordination Tests.
- `test_websocket.py` – WebSocket and Real-Time Notification Tests.

All automated tests are located in the top-level `tests/` directory and were executed with `pytest` inside the `field_node` container via `docker compose exec`, using the same services and configuration adopted by the running application. This allows to validate the system in a production-like environment rather than in an isolated unit-test setup.

Unit tests were used to validate the internal logic of the Two-Phase Commit coordinator: the tests covered the `prepare, commit, rollback` phases under nominal and corner-case conditions. The test cases included scenarios in which all utilities voted `YES`, a utility voted `NO`, the utility node was unreachable due a timeout, and no utilities were involved in the transaction. These tests primarly target the atomicity requirment, esnuring that a booking is either fully committed or fully aborted.

Integration tests were used to verify communication between the Field Node and the Utility Node through the internal 2PC endpoints. The test suite checked that active utilities vote positively, nonexistent utilities vote negatively, pending utility bookings are created during the prepare phase, and booking states are updated correctly during commit and that a rollback with an empty utility list completes without errorrs. These

tests validate the interaction logic between distributed components and support the requirements related to atomic transactions and consistent state transitions.

End-to-end tests exercised complete booking flows from the public API of the Field Node. The tested scenarios included successful bookings with and without utilities, rollback on missing utilities, and double booking prevention on the same slot. These tests verify the behavior of the whole system as observed by a client and directly cover the requirements concerning booking correctness, resource sharing and distributed coordination.

Concurrency tests were used to validate isoletion under contention: multiple clients attempted to book the same time slot simultaneously, both in the simple booking flow and in the 2PC workflow. The expected outcome was that onlu one request succeded while the others were rejected (in the simple booking flow the tests accepts at most one winner, while in the 2PC flow exaclty one success is required). these tests specifically address the requirement that concurrent users must not be able to double-book the same resource.

Fault Tolerance tests were used to validate recovery behaviour in the presence of partial failures, with the system tested with prepared transactions left in an intermediate state, with an empty Redis state, and with an a prepared transaction referencing a non-existent utility booking, simulatung an unrecoverable partial failure. The recovery procedure was exprected to either complete the transaction or abort it without leaving inconsistent data behind. These tests directly support the fault tolerance requirement and verify that the recovery loop restores consistency after failures.

Temporary-hold tests were used to verify the Redis-based slot reservation mechanism implemented. The tests checked that selected slots are written to Redis with the correct TTL, that releasing a hold removes the corresponding keys, that the endpoint returining active holds works correctly, and the expired holds make the slot available again. These tests cover the temporary-slot-locking requirement and the real-time availability behavior of the system.

WebSocket tests validated the real-time notification mechanism: the system was checked to ensure that booking confirmations events, booking failure events, and slot updates are broadcast to all connected clients simultaneously, verifying that multiple WebSocket connections receive the same envent. These tests support the requirementthat all users receive immediate availability updates without refreshing the page.

The automated suite was run in the containerized environment using these commands (**Please check the user guide before doing this**):

```
docker compose up -d --build
docker compose exec field_node poetry run pytest tests -v
```

5.2 Acceptance Tests

Manual acceptance testing was performed on the web interface to verify the usability of the booking worflow and the clarity of the real-time feedback. The tested actions included opening the booking page, selecting a field, inserting a username, choosing with a starting and ending time slots, optionally adding utilities, confirming the booking, and observing the live updates on connected clients, with also the possibility of cancelling the booking. Manual testing was particularly useful for checking the visual behaviour of temporary holds and WebSocket-driven updates, which are easier to evaluate interactively than through purely automated assertions.

6 Deployment

The system is deployed through `Docker` and `Docker Compose`. No manual installation of Python, MySQL, or Redis is required on the host machine, because all services run inside containers. **The only prerequisite is a recent Docker installation, such as Docker Desktop or the Docker Engine with the Compose plugin.**

6.1 Installation and Startup

To run the project from scratch, it is enough to clone the repository and execute this command (**Please check the user guide before doing this**):

```
docker compose up -d --build
```

This command builds the Field Node and the Utility Node images locally, and starts the four services in the correct order. MySQL and Redis are started first, followed by the two FastAPI services. Once the system is up, the application is reachable through the Field Node at **http://localhost:8001/static/frontend/index.html**

6.2 Required Software

The host machine only needs Docker: the backend services are based on Python 3.12, FastAPI, SQLAlchemy, Redis 7, and MySQL 8.0, but these dependencies are already packaged inside the containers. The application is therefore self-contained and can be executed without installing the runtime libraries separately.

6.3 Configuration

All required environment variables are defined in `docker-compose.yml`: these include the MySQL credentials, the Redis host and port, the node identifier, and the URL used

by the Field Node to contact the Utility Node during the Two-Phase Commit protocol. The database connection string and the Redis URL are computed automatically from these values by the shared configuration module.

6.4 Deployment Structure

The deployment architecture is shown in the following deployment diagram fig. 7: it includes the browser client, the Field Node, the Utility Node, Redis. The two application containers are attached to the same bridge network as the infrastructure services, so they can communicate through Docker service names. The deployment also uses container health checks and startup dependencies conditions to ensure that application nodes are launched only after Redis and MySQL are ready.The MySQL data are stored in a persistent Docker volume, ensuring persistence across container restarts. The shared directory is moundted into both application nodes, providing common code and configuration without duplication.

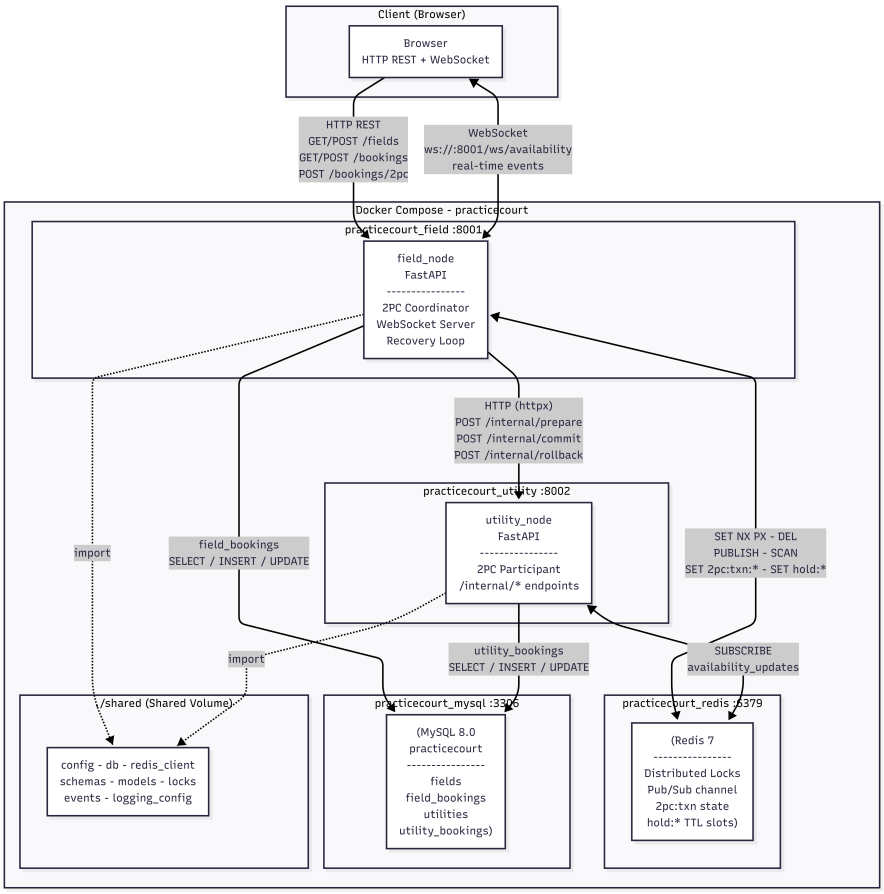

Figure 7: Deployment Architecture of DS-PracticeCourt.

7 User Guide

The following instructions describe how to interact with the DS-PracticeCourt web interface to browse sport facilites and perfom a booking.

7.1 Starting the Application

Clone the repository, if you have the Docker application open it, and start all services with:

```
git clone https://github.com/fedebrighi/DS-PracticeCourt.git
cd DS-PracticeCourt
docker compose up -d --build
```

WARNING: If an error regarding port `3306` already in use appears during/after the docker compose command, it is sufficient to open the Task Manager, locate the `mysqld.exe` process and terminate it, then re-run the command in the terminal.

Once all containers are runninng and healthy, open a browser and navigate to:

```
http://localhost:8001/static/frontend/index.html
```

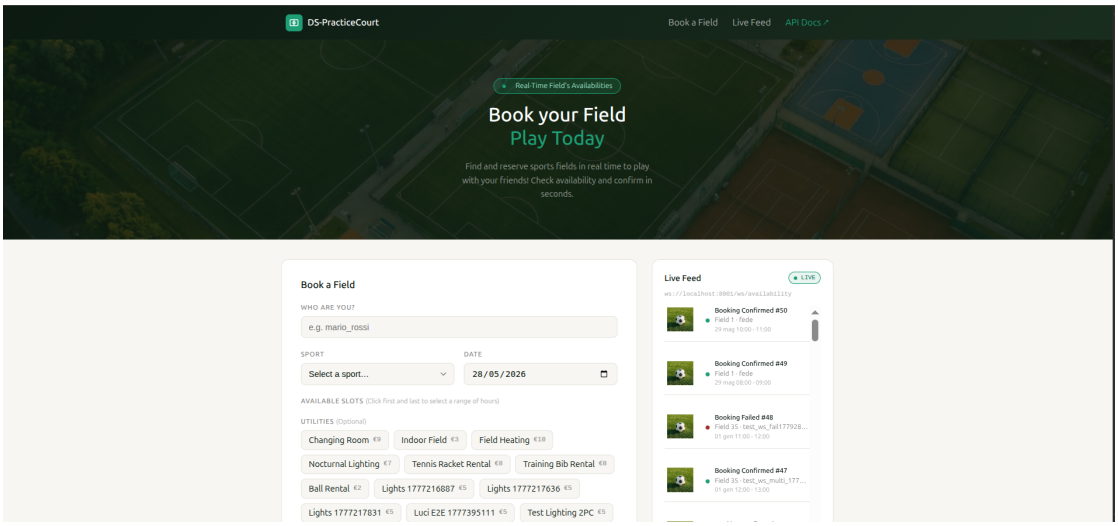

Figure 8: Overview of the DS-PracticeCourt web interface.

7.2 Setting a Username

Before any interaction with the booking system is possible, the user must entrer a username in the **Who Are You?** section of the interface. This identifier is used to associate bookings with a specific user and to control which cancellation actions are visible. Without a username, the booking flow cannot be initiated.

Exepcted outcome: once a username is entered, the rest of the interface becomes fully interactive, allowing the user to browse fields, select time slots, and perform bookings.

7.3 Selecting a Field and a Date

The user must select a sport facility from the **Sport** section and a date from the **Date** section. Both selections are required before the availability grid is displayed.

Expected outcome: once both a field and a date are selected, a time slot grid appears showing the availability for that specific field on that specific day.

7.4 Reading the Availability Grid

The grid displays all half-hours time slots for the selected field and date. Each slot is rendered in one of three states:

- **Available** - the slot is free and can be selected.
- **Booked** - the slot is already reserved by another user and cannot be selected. It appears disabled with a strikethrough label and cannot be selected.
- **Held** - the slot is temporally locked by another user who is currently completing a booking; it appersa highlighted in yellow and cannot be selected.

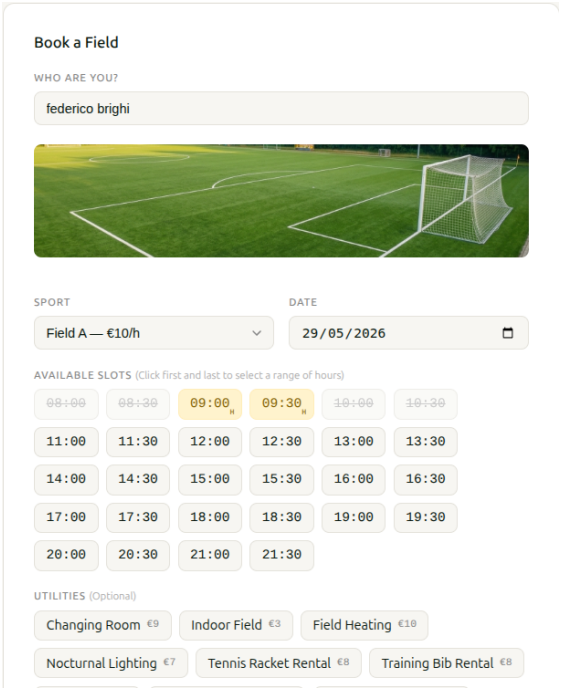

Figure 9: Availability grid showing available, booked, and held time slots.

7.5 Selecting a Time Range

To book a field, the user must select a start slot and an end slot from the gird. The selection order does not matter: the system automatically determines which slot comes first. For example, to book from 10.00 to 11.00, the user selects the `10.00` and the `10.30` slots.

Expected outcome: once two slots are selected, a **60-seconds countdown** begings. During this period the selected slots are temporarily held and appear yellow with the `H` indicator for all other connected users, preventing them from selecting the same slots. If the booking is not confirmed within the timeout, the hold is released and the slots become available again to all connected users.

7.6 Selecting Utility Services

After selecting a time range, the user may optionally choose one of more additional services from the **Utilities** section. The total price displayed at the bottom of the form updates in real time based on the selected utilities and the duration of the booking. Note that some utilities are priced per hour, so their cost is multiplied by the number of hours covered by the selected time range.

Expected outcome: the price summary reflects the combined cost of the selected field

slot and any chosen utility services.

7.7 Confirm the Booking

Once the time range and any desired utilites have been selected, the user presses the **Confirm** button to submit the booking request.

Expected outcome: if the booking is successful, the live feed visible to all connected users is updated with the new reservation, and the corrisponding time slots are immediately disabled in the availability grid for every connected client. If the booking fails an error message is displayed and the slots are relased.

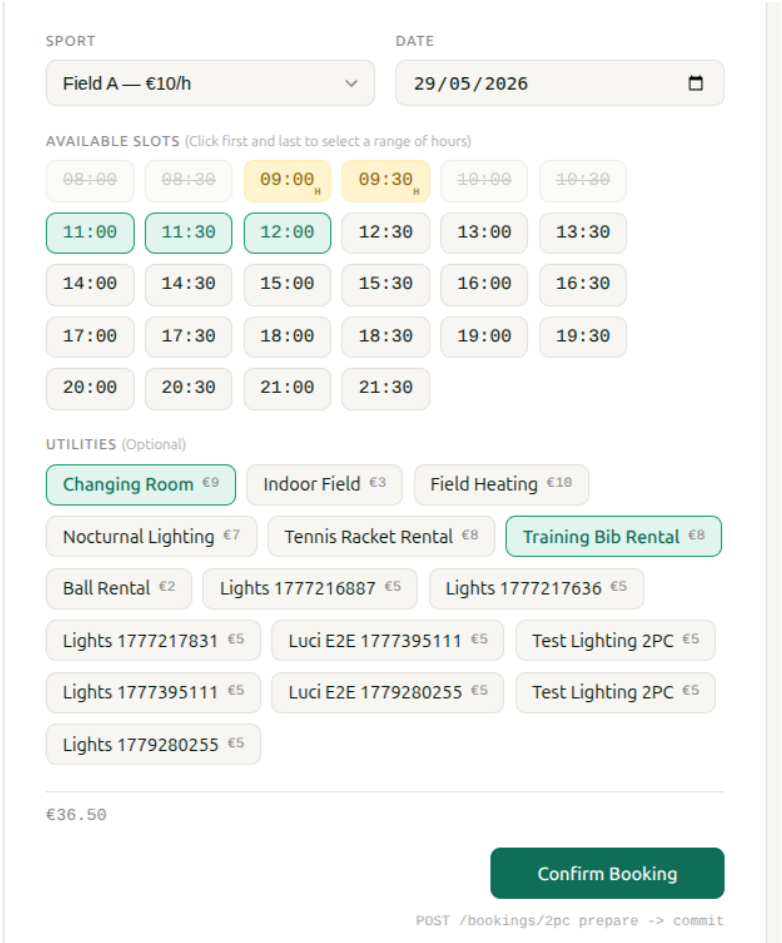

Figure 10: Booking form with time range and utility services selected (in this image are present also utilities created by the running of tests).

7.8 Optional: Cancelling a Booking

A user can possibly cancel on of their own bookings directly from the live feed by clicking the `red cancel button X` displayed next to their reservatio. This button is only visible to the user who made that specific booking; other users see the same live feed entry without the cancellation option.

Expected outcome: upon cancellation, the booking is removed from the live feed and the corresponding time slots become available again for all connected users.

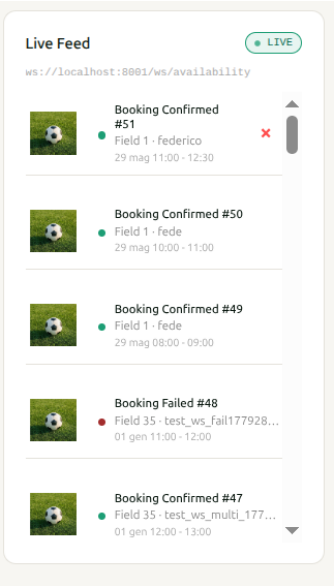

Figure 11: Live feed showing active bookings with the cancellation option for the current user.

8 Self-evaluation

This project represented my first real experience with distributed systems, both from a design and from an implementation perspective. In particular, it was also my first practical use of Docker for building and running a multi-service application in a reproducible environment. From a techincal point of view, I tried to use knowledges acquired from other courses: WebSocket-based communication was developed by reusing concpets learned in the "Programmazione di Reti" course, the development process also benefited from the Test-Driven-Development approach introduced in the "PPS" course and, in addition, the frontend part allowed me to apply the skills acquired in the "Tecnologie Web" course.

Among the main strengths of my work, I would mention the clear separation of responsibilities among the different components, the use of a distributed architecture suitable

for the problem domain, and the support for concurrency and real-time updates. The system was designed to manage bookings consistently, preventing conflicts on the same slot and keeping the state synchronized across the involved services. Among the main weaknesses,I would mention the initial complexity of coordinating the different parts of the system, especially in the presence of failures and concurrent requests. A more important limitation of this project from my point of view is the absence of complete fault tolerance in all possible failure scenarios, which I will talk about in the Future Works section. Overall, this project was a valuable learning experience that helped me consolidate existing knowledge while also developing new practical skills in distributed programming and deployment.

9 Future works

Several improvements could be introduced to make the application more robust, complete and user-oriented. A first imporatant extension that I found during the final testing of the system concerns **complete fault tolerance**: at the moment, the system already includes some recovery-oriented mechanism, but it does not yet fully coverall possible faiulre scenarios. In a future version, a more comprehensive fault tolerance strategy could be implemented by introducing stronger recovery procedures for partially completed transaction, more systematic state reconciliation after crashers and additional monitoring of the distrivuted components. This would make the system more resilient to node failures, especially for the Field Node, that right now, if fails, cant proceed with the booking, and network disruptions. Another improvement could be the introduction of an authenitcation mechanism: users could log in with personal credentials and access a dedicated profile area; in this way, instead of relying only on the live feed to observe bookings, each user could view a personal page showing their own reservations, together with their status and possible cancellation options.

The full source code is available in the project repository on Github at [1].

References

[1] Federico Brighi. Ds-practicecourt. `https://github.com/fedebrighi/DS-PracticeCourt`, 2026. Repository of the DS-PracticeCourt project.

[2] Inc. Docker. Docker. `https://docs.docker.com/manuals/`, 2026. Documentation for Docker containerization platform.

[3] Redis Labs. Redis. `https://redis.io/docs/latest/develop/`, 2026. Documentation for the Redis in-memory data structure store.

[4] Sebastián Ramírez. Fastapi. `https://fastapi.tiangolo.com/`, 2026. Documentation for the FastAPI framework.