Software Architetures's Project
SAG Documentation

Sports Tournaments and Championship Management System - STCMS

Componenti del gruppo

Claudio Giacomello Matricola: 892126

Tomislav Dzepina Matricola: 883801

Federico Gaudenzi Matricola: 894069

Santino Poli Matricola: 1002776

Sartori Leonardo Matricola: 886069

Anno Accademico
2025 / 2026

Table of Contents

Capitolo 1

Introduction

In this section, we present the architectural features of the Sports Tournaments and Championship Management System (STCMS). This system is designed to facilitate the organization, management, and execution of sports tournaments and championships across various disciplines. The STCMS aims to provide a comprehensive solution that addresses the needs of tournament organizers, teams, players, referees, and fans.

1.1 Project Overview and Goals

The Sports Tournaments and Championship Management System (STCMS) is a web-based platform designed to support the complete management of sports tournaments and championships, regardless of the discipline (team sports or individual sports).

The system allows users to:

- create and configure tournaments and championships;
- automatically generate matches;
- record results and update standings;
- publish news and informational content about events;
- allow fans and users to follow the progress of competitions.

STCMS is conceived as a scalable, secure, and modular solution, accessible via web browsers and mobile devices, capable of supporting a growing number of users and tournaments over time.

The main users of the system are:

- **Administrators**, who manage the platform and configure tournaments;
- **Registered users**, who create tournaments, manage teams, or participate as players;
- **Fans (unregistered users)**, who consult results, standings, and news.

1.1.1 Core System Services

User Management Service (User Service)

Responsible for user management and security.

Main functionalities:

- user registration and authentication;
- role management (Admin, User);
- management of user profiles and preferences;
- authorization through role-based access control (RBAC).

This service is central to ensuring security, privacy, and controlled access to all system functionalities.

Tournament and Championship Management Service (Tournament Service)

Manages the entire lifecycle of tournaments and championships.

Main functionalities:

- creation and configuration of tournaments (sport, format, maximum number of participants);
- management of tournament states (created, open for registration, ongoing, completed);
- support for multiple sports disciplines and competition formats.

It is one of the most critical services, as it defines the structure upon which matches, standings, and results are based.

Enrollment Service

Manages the participation of users, teams, or players in tournaments.

Main functionalities:

- team enrollment;
- validation of entered data (player profiles, team details);
- constraint checks (maximum number of participants, tournament requirements);
- association between users and tournaments.

This service ensures data correctness and consistency of registrations.

Match Management Service (Match Service)

Responsible for the creation and management of individual matches.
Main functionalities:

- automatic generation of pairings (e.g., Round Robin, single elimination);
- management of the match schedule;
- recording of individual match results;
- production of match reports.

It is a high-load service, especially during tournament execution.

Standings Service

Manages the calculation and updating of standings.
Main functionalities:

- automatic calculation of standings based on results;
- score management.

This service is essential for fans and participants who follow tournament progress.

News and Blog Service (News Service)

Manages the informational and social aspects of the platform.
Main functionalities:

- automatic creation of a news item upon tournament creation;
- display of articles and updates;
- management of likes;
- generation of shareable links for social media and messaging platforms.

This service enhances interoperability and user engagement.

Security and Infrastructure Service

A cross-cutting service that supports all others.
Main functionalities:

- management of JWT tokens (access and refresh);
- rate limiting and API protection;
- logging and auditing;
- support for scalability and high availability.

Capitolo 2

Architectural Design

To create the two architectural versions of the Sports Tournaments and Championship Management System (STCMS), a dedicated GitHub repository was created: `https://github.com/STCM`
The work was divided into two branches of the commit tree: one for the monolithic structure and one for the microservices architecture. Each branch implements the same core system functionalities, ensuring functional uniformity. This organization made it possible to directly analyze the differences in terms of scalability, maintainability, and complexity.

2.1 Monolithic Architecture

A monolithic architecture is a software design paradigm in which the entire application is developed, packaged, and deployed as a single cohesive unit. In this architectural model, all functional components—such as the user interface, business logic, and data access layer—are tightly integrated within a single executable process or runtime. This approach simplifies both the development cycle and the deployment process, since all internal components share the same memory space and communicate through direct function calls or dependency injection mechanisms.

For the initial implementation of the Sports Tournaments and Championship Management System (STCMS), a monolithic architecture was deliberately adopted in order to reduce operational complexity, simplify deployment, and ensure rapid development during the early phases of the project.

2.1.1 Modular Monolith Pattern

Although deployed as a single unit, STCMS follows the Modular Monolith pattern. The system is implemented using the NestJS framework, which natively supports a modular architecture and enforces a clear separation of concerns.

The monolithic nature of the system is mainly determined by the deployment strategy: a single Docker container is used to host both the backend and the frontend. In particular, the Angular frontend is compiled into static assets and embedded within the NestJS application through a multi-stage Docker build process. As a result, the entire system—API and user interface—is executed as a single runtime entity.

Despite this unified deployment, the internal structure of the backend is organized into function-oriented modules, each representing a well-defined application domain:

- **UserModule**: Manages user authentication, authorization mechanisms, and profile data.

- **TournamentModule**: Manages the entire lifecycle of tournaments, from creation to the generation of final standings.
- **MatchModule**: Implements match generation logic, pairing systems, result entry, and match state management.
- **NewsModule**: Integrates blog functionalities, tournament news publication, and social interaction components.

This modular organization allows the system to retain many of the advantages typically associated with microservices architectures—such as maintainability and code-level scalability—while preserving the operational simplicity of a monolithic deployment.

2.1.2 Internal Dependencies Between Modules

Within the monolithic architecture, communication between modules occurs through the dependency injection system provided by NestJS. Services are injected directly into dependent modules, enabling low-latency communication and high execution efficiency.

Although this approach offers excellent performance characteristics, it requires careful architectural design to avoid excessive coupling and the formation of circular dependencies. For this reason, module boundaries were defined following principles inspired by Domain-Driven Design, ensuring that the system remains maintainable and ready for future decomposition into independent microservices.

2.1.3 Containerized Monolithic Deployment

The monolithic deployment of STCMS is achieved through Docker-based containerization. The project is structured as a full-stack application composed of a NestJS backend, an Angular frontend, and a MongoDB database, orchestrated using Docker Compose.

The repository follows the structure shown below:

```
.
|-- STCMS_Backend/      # NestJS source code
|-- STCMS_frontend/     # Angular source code
|-- Dockerfile          # Multi-stage build for API and UI
|-- docker-compose.yml # Orchestration for API and MongoDB
|-- .env                # Environment variables
```

A multi-stage Dockerfile is used to compile the Angular frontend and embed the resulting static files within the NestJS application. This approach allows the frontend to be served directly by the backend, eliminating the need for an external web server.

The deployment consists of two main containers:

- **Monolithic Container (app-stcms)**: Hosts the NestJS application and serves the compiled Angular frontend.
- **Database Container (db-stcms)**: Runs a dedicated MongoDB instance for data persistence.

The containers communicate via Docker's internal networking, with the backend connecting to the database using the internal DNS service name. Port mapping exposes a single access point to the system, allowing both frontend and API access through the same host and port.

2.1.4 Unified Execution of Frontend and Backend

In the adopted monolithic architecture, NestJS acts as the primary host application for both frontend and backend components. The Angular application is compiled into static files (HTML, JavaScript, and CSS) and served by NestJS through the `ServeStaticModule`.

Incoming HTTP requests are handled according to the following routing logic:

- Requests targeting paths prefixed with `/api` are routed to the backend REST controllers.
- All other requests are handled by the Angular frontend and return the corresponding static resources.

Since frontend and backend share the same origin (identical host and port), all API calls are performed using relative paths (e.g., `/api/login`). This design choice completely eliminates Cross-Origin Resource Sharing (CORS) issues in production environments, as the browser perceives the entire system as a single application.

Overall, this monolithic deployment strategy provides a solid, simple, and efficient foundation for STCMS, while maintaining the flexibility required for a potential future transition to a distributed microservices architecture.

2.1.5 Steps to Create the Monolithic Architecture

To implement the monolithic architecture of the Sports Tournaments and Championship Management System (STCMS), the following steps were followed:

1. **Angular frontend preparation**: the Angular project (`STCMS_frontend/STCMS_app`) is copied into the Docker context. Dependencies are installed using `npm install`, and the production build is generated with `ng build -configuration production`, producing static files in `dist/STCMS_app/browser`, which are served by the NestJS backend.
2. **NestJS backend preparation**: the NestJS project (`STCMS_Backend`) is copied into the Docker context. After installing dependencies (`npm install`), the backend is compiled (`npm run build`). The frontend static files are copied into the backend directory (`/app/browser`) to allow NestJS to serve the user interface.
3. **Final Docker image creation**: a multi-stage Dockerfile is used to separate build and runtime stages. In the final stage, only the compiled backend, required Node modules, and frontend static files are copied. Port 3000 is exposed and the application is started with `node dist/main.js`.
4. **Orchestration with Docker Compose**: an `api` service is defined to run the monolithic container (backend + frontend) and a `mongo` service for the MongoDB database. Environment variables configure database connection, admin credentials, and JWT settings, while volumes persist data across restarts. The `depends_on` directive ensures that the API starts only after the database.
5. **Execution of the entire stack**: the entire application can be started with a single command:

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

The entire system (backend + frontend + database) is now available as a ready-to-use monolithic unit. This procedure enables an efficient and reproducible monolithic deployment, keeping frontend and backend separate during development and combining them only at build time.

2.2 Microservices Architecture

The STCMS project has evolved toward a microservices-inspired architecture to improve system scalability, maintainability, and resilience. Currently, the configuration is **partially containerized**, with some services isolated and others executed manually.

2.2.1 Current State: Hybrid Architecture

- **MongoDB**: fully containerized via Docker, with credentials defined in the `.env` file and ports mapped for backend access.
- **Backend (NestJS)**: executed manually on the host via `npm run start:dev`, communicating with MongoDB through a mapped port.
- **Frontend (Angular)**: executed manually via `ng serve`, communicating with the backend through HTTP APIs.

This configuration allows partial service isolation and enables frontend, backend, and database to be started independently, although only the database is containerized. The backend and frontend behave as separate services but are not yet containerized; therefore, the architecture can be defined as **"microservices-like" or hybrid**.

2.2.2 Decomposition Strategy: Domain-Driven Design

Each module of the monolith follows Domain-Driven Design (DDD) principles and can be treated as an autonomous service in the future:

- **Independent scalability**: modules such as the MatchService can be horizontally scaled during activity peaks.
- **Fault isolation**: a malfunction in one module, such as the NewsService, does not compromise core functionalities like tournament registration.

2.2.3 Component Communication

Communication between components occurs through:

- **REST HTTP**: used by the frontend to communicate with the backend.
- **MongoDB Driver**: used by the backend to communicate with the containerized database.

At this stage, event-driven messaging mechanisms between services have not yet been implemented, but the modular structure allows their introduction in a future evolution.

2.2.4 Data Persistence

Currently, there is a single containerized MongoDB database shared among services. In a complete microservices architecture, each service could have its own dedicated database (Database-per-Service) to ensure low coupling and independent scalability.

2.2.5 Next Steps Toward Pure Microservices

To achieve a true microservices architecture, the following steps will be required:

- containerization of the backend and frontend via Docker;
- separation of modules into independent microservices (Auth, User, Tournament, Match, News, etc.);
- introduction of dedicated databases for each service;
- optional service-to-service communication through events (event-driven) or well-defined APIs.

This evolution will enable a fully distributed architecture with high resilience, horizontal scalability, and modular management of system functionalities. The backend of the *Sports Tournaments and Championship Management System* (STCMS) was developed using the **NestJS** framework, adopting a modular architecture based on functional domains. This design choice is consistent with the scalability, maintainability, and extensibility requirements that emerged from the Kata analysis, and it enables an orderly management of the system's complexity.

The application is structured as a **modular monolith**, meaning a single deployable application that nonetheless features a strong internal separation into independent modules. This approach makes it possible to preserve operational simplicity in the early stages of the project, while keeping open the possibility of a future evolution toward a microservices architecture.

General backend structure

Each module represents a well-defined functional context (*bounded context*) and contains:

- REST controllers, which expose the APIs;
- services, which implement the business logic;
- DTOs, used for data validation and typing;
- Mongoose schemas, for database persistence;
- guards and decorators, for access control.

This organization follows NestJS best practices and promotes a clear separation of responsibilities.

Database and data persistence

The system uses **MongoDB** as the main database, integrated through **Mongoose** as an ODM (*Object Data Modeling*). The choice of MongoDB is motivated by:

- schema flexibility, which is useful for supporting different sports with different data structures;
- ease of data model extension;
- good read and write performance;
- compatibility with scalable architectures.

Each module owns its own Mongoose schema, maintaining low coupling among the various application domains.

2.3 Analysis of the Main Modules

2.3.1 Authentication Module (Auth Module)

The Auth module manages the entire authentication and authorization process of the system. It was developed separately from the User module in order to isolate security-related responsibilities.

It implements a JWT-based authentication system, with a distinction between:

- access tokens, with a short lifespan;
- refresh tokens, used for automatic session renewal.

The module exposes the following main APIs:

HTTP Method	**Endpoint**	**Description**	**Guard / Protection**	**Parameters / Headers**	**Body / DTO**	**Expected Response**
POST	/auth/register	Registers a new user and returns tokens	None	–	CreateUserDto	200: {accessToken, expiresIn}
POST	/auth/login	User login with email and password	None	–	LoginDto	200: {accessToken, expiresIn} 400: Missing email/password 401: Invalid credentials
POST	/auth/refresh	Generates a new access token using a refresh token	None	Cookie: refreshToken	–	200: {accessToken, expiresIn} 400: Missing refresh token 401: Invalid refresh token
POST	/auth/logout	User logout, removes refresh token cookie	None	–	–	200: {message: "Logged out successfully"}

Protected endpoints are managed through JWT and role-based guards. In addition, custom guards and decorators are used to implement *Role-Based Access Control* (RBAC), clearly distinguishing between Admin and User roles.

2.3.2 User Module

The User module manages the users of the platform. It is responsible for:

- creation and management of user profiles;
- storage of personal data;

- association of users with teams, tournaments, and roles.

Main APIs:

HTTP Method	Endpoint	Description	Guard	Parameters	Body / DTO	Expected Response
GET	/users	Retrieves all users	None	page?, limit?	–	200: List of users
GET	/users/:id	Retrieves a user by ID	JwtAuthGuard, ClientGuard	id (path)	–	200: User 401: Unauthorized
POST	/users	Creates a new user	None	–	CreateUserDto	201: User created
PATCH	/users/:id	Updates a user	JwtAuthGuard, ClientGuard	id (path)	UpdateUserDto	200: User updated 401: Unauthorized
DELETE	/users/:id	Deletes a user	JwtAuthGuard, ClientGuard	id (path)	–	200: {message: "User deleted successfully"} 401: Unauthorized

This module represents the foundation upon which authentication, authorization, and participation in tournaments are built.

2.3.3 Tournament Module

The Tournament module manages the lifecycle of tournaments and championships. It was designed as one of the core modules of the system and allows:

- creation of new tournaments;
- configuration of the sport and competition format;
- management of the tournament status (open, ongoing, completed).

Main APIs:

Method	Endpoint	Authentication	Description
GET	/tournaments	No	Returns the list of all tournaments. Supports optional filtering by sport.
GET	/tournaments/{id}	No	Returns the details of a specific tournament.
GET	/tournaments/{id}/teams	No	Returns all teams registered in a tournament.
GET	/tournaments/{id}/matches	No	Returns the matches of the tournament, filterable by date/time range and status.
POST	/tournaments	Yes (JWT)	Creates a new tournament associated with the authenticated user.
POST	/tournaments/{id}/teams	Yes (JWT)	Creates a new team within a tournament.
POST	/tournaments/{id}/matches	No	Creates a new match for the tournament (requires at least two teams).
PATCH	/tournaments/{id}	Yes (JWT + Creator)	Updates the information of a tournament. Only the creator can modify it.
DELETE	/tournaments/{id}	Yes (JWT + Creator)	Permanently deletes a tournament. Access is restricted to the creator.

Specific guards are implemented to allow only the tournament creator or an Admin to modify its data, ensuring correctness and security.

2.3.4 Team Module

The Team module is dedicated to the management of teams in group sports. It handles:

- team creation;
- member association;
- management of the team captain.

Main APIs:

Method	Endpoint	Authentication	Description
GET	/teams/{id}	No	Returns the details of a team by its ID.
PATCH	/teams/{id}	Yes (JWT + Captain)	Updates the information of a team. The operation is allowed only for the team captain.
DELETE	/teams/{id}	Yes (JWT + Captain)	Deletes a team from the system. Access is restricted to the team captain.

A dedicated guard is implemented to allow only the team captain to perform sensitive operations, accurately reflecting the logic of the sports domain.

2.3.5 Match Module

The Match module is one of the most complex components and represents the functional core of the system. It manages:

- match generation;
- scheduling;
- result submission;
- match status management.

The design of this module is strongly oriented toward extensibility: match and result logic is divided by sport, using specific files and DTOs (football, basketball, volleyball).

Method	**Endpoint**	**Authentication**	**Description**
GET	/matches/{id}	No	Returns the details of a single match identified by its ID.
GET	/matches/{id}/result	No	Returns the match result (if available).
PATCH	/matches/{id}/result	Yes (JWT)	Updates the result. The format is validated based on the sport.
PATCH	/matches/{id}/status	Yes (JWT)	Updates the match status (e.g., scheduled, ongoing, completed).
DELETE	/matches/{id}	Yes (JWT)	Deletes an existing match from the system.

This approach allows new sports to be added without modifying existing logic, simply by extending the already defined structures.

2.3.6 Standings and Results Module

Standings management is integrated into the match workflow: every result update triggers the logic for recalculating scores and ranking positions. This guarantees:

- data consistency;
- immediate updates;
- real-time visibility for fans and users.

2.3.7 Blog and News Module (Blog Module)

The Blog module manages the informational part of the system. It is used to:

- publish news related to tournaments;
- automatically generate an article when a new tournament is created;
- allow likes and content sharing.

Method	**Endpoint**	**Authentication**	**Description**
POST	/blog/{userId}	No	Creates a new post (supports image upload).
GET	/blog	No	Returns the complete list of all posts.
PATCH	/blog/{blogId}	Owner	Updates a post (title, tags, image). Owner only.
DELETE	/blog/{blogId}	Owner	Permanently deletes a post.
PATCH	/blog/{blogId}/like	Yes (JWT)	Adds a like to the post (only once per user).
PATCH	/blog/{blogId}/unlike	Yes (JWT)	Removes a previously added like.
PATCH	/blog/{blogId}/toggle-like	Yes (JWT)	Adds or removes a like based on the current state.
GET	/blog/by-tags	No	Returns posts filtered by specified tags.

The module supports interoperability, allowing content to be shared through unique links.

2.4 Summary of Development Objectives

The backend was developed in this way in order to:

- reduce coupling between functionalities;
- facilitate code maintenance;
- support multiple sports and tournament formats;
- ensure security and access control;
- prepare the system for a future decomposition into microservices.

Functional Logic of the System (with References to Backend Services)

The application logic of the system is implemented in the NestJS backend through a modular structure based on dedicated services, each responsible for a specific functional area. In particular, the management of users, tournaments, teams, and matches is distributed across the `TournamentService`, `TeamService`, and `MatchService`, ensuring a clear separation of responsibilities.

User Management

User creation and management are handled by the authentication module and its related service. During registration, the system validates the incoming data, applies security policies (password hashing), and assigns the appropriate role.

This information is then used by the application services to verify permissions and associate operations (such as tournament creation) with authorized users.

Tournament Creation and Management

The core tournament logic is implemented within the `TournamentService`, which manages the entire lifecycle of the competition.

File: `tournament/tournament.service.ts`

```
@Injectable()
export class TournamentService {
  constructor(
    @InjectModel(Tournament.name)
    private tournamentModel: Model<Tournament>,
  ) {}

  async createTournament(dto: CreateTournamentDto, adminId:
     string) {
    const tournament = new this.tournamentModel({
      ...dto,
      createdBy: adminId,
    });
    return tournament.save();
  }
}
```

Creation

Administrators define the fundamental parameters of the tournament, such as the sport discipline, maximum number of teams, and format. The service validates the data received through DTOs and persists the tournament in the MongoDB database.

This design choice makes it possible to:

- centralize the creation logic;

- prevent the creation of invalid tournaments;
- keep the controller lightweight and focused on API exposure.

Team Registration and Management

Team management is handled by the `TeamService`, which coordinates team creation and their registration to tournaments, while enforcing capacity constraints.

File: `team/team.service.ts`

```
@Injectable()
export class TeamService {
  constructor(
    @InjectModel(Team.name)
    private teamModel: Model<Team>,
    private tournamentService: TournamentService,
  ) {}

  async registerTeamToTournament(teamId: string, tournamentId:
     string) {
    const tournament = await this.tournamentService.findById(
       tournamentId);

    if (tournament.teams.length >= tournament.maxTeams) {
      throw new BadRequestException('Tournament is full');
    }

    tournament.teams.push(teamId);
    await tournament.save();

    return tournament;
  }
}
```

Registration

Before persisting the registration:

- the system verifies that the tournament exists;
- it checks that the maximum number of teams has not been exceeded;
- it updates the tournament status in a consistent manner.

This mechanism guarantees data integrity and prevents logical error conditions (e.g., over-registration).

Match Management and Automatic Generation

Match generation and management are the responsibility of the `MatchService`, which implements the logic for pairing teams based on the tournament format.

File: `match/match.service.ts`

```
@Injectable()
export class MatchService {
  constructor(
    @InjectModel(Match.name)
    private matchModel: Model<Match>,
  ) {}

  async generateRoundRobinMatches(teams: string[], tournamentId:
     string) {
    const matches = [];

    for (let i = 0; i < teams.length; i++) {
      for (let j = i + 1; j < teams.length; j++) {
        matches.push({
          tournament: tournamentId,
          homeTeam: teams[i],
          awayTeam: teams[j],
        });
      }
    }

    return this.matchModel.insertMany(matches);
  }
}
```

Pairing Algorithms

Depending on the tournament format, the system applies automatic match generation algorithms (e.g., Round Robin). This approach:

- eliminates the need for manual scheduling;
- ensures fairness in match distribution;
- reduces the likelihood of human errors.

Result Updates and Dynamic Standings

When a match result is submitted, the `MatchService` updates the persisted data and triggers the logic for recalculating tournament statistics.

```
async updateResult(matchId: string, homeScore: number, awayScore:
    number) {
  const match = await this.matchModel.findById(matchId);

  match.homeScore = homeScore;
  match.awayScore = awayScore;
  match.isCompleted = true;

  return match.save();
}
```

Following this update, an application service recalculates:

- points earned by teams;
- goals or baskets scored;
- ranking positions.

This ensures consistency and real-time updates of the information displayed to users.

2.5 Security Implementation

Security represents one of the fundamental pillars of the entire backend system and has been evaluated with the highest level of importance (10/10). This design choice is motivated by the need to protect users' personal data, ensure the correctness of the competitions managed by the system, and prevent unauthorized access or data tampering.

In accordance with the architectural requirements defined in the *Kata reference* (*"The system must ensure data security and privacy for all users"*), the backend implements a **multi-layer** security strategy that covers:

- user authentication;
- authorization and access control;
- protection and integrity of persisted data;
- security of the exposed REST APIs.

2.5.1 Cryptographic Standards and Credential Protection

To ensure maximum protection of user credentials, the system adopts an approach based on one-way cryptographic hashing. In particular, passwords are processed using the **SHA-512** algorithm, a robust and widely recognized cryptographic standard.

Before being stored in the MongoDB database:

- each password is *salted*, meaning it is combined with a unique random value;
- the result is then hashed using SHA-512;
- only the final hash is persisted in the database.

This approach ensures that:

- passwords are never stored in plain text;
- even in the event of a database compromise, the original credentials cannot be recovered through decryption;
- *rainbow table* and offline *brute-force* attacks are mitigated.

The encryption management of persisted data is supported by middleware provided by the **Mongoose** library, which allows protection mechanisms to be integrated directly at the schema level.

In the NestJS backend, credential verification never occurs by comparing plain-text passwords. This behavior is evident in the `validateUser` method of the *AuthService*:

File: `auth/auth.service.ts`

```
async validateUser(email: string, password: string) {
  const user = await this.userService.findByEmail(email);
  if (user && user.matchPassword(password) && user.isActive)
     return user;
  throw new UnauthorizedException('Invalid credentials');
}
```

The `matchPassword` method is implemented directly within the user's Mongoose schema (and not in the controller), ensuring that:

- password hashing and comparison are encapsulated within the model;
- cryptographic logic is neither duplicated nor exposed at higher layers.

This approach adheres to the principle of *separation of concerns* and significantly reduces the risk of security-related errors.

2.5.2 Authentication: Dual-Token JWT System

The authentication system was designed by adopting a *JWT-based* strategy with a *dual-token* architecture, capable of balancing security and usability.

Access Token

The *Access Token* is a short-lived *JSON Web Token* used to authenticate requests to the protected backend APIs. It is sent by the client within the `Authorization` header and contains essential information about the user's identity and related privileges.

The short validity period significantly reduces the impact of a potential token compromise. Validation of *access tokens* is handled through a custom *Passport* strategy:

File: `auth/jwt.strategy.ts`

```
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(configService: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: configService.get<string>('JWT_SECRET'),
    });
  }

  validate(payload: any) {
    if (payload.token_type && payload.token_type !== 'access') {
      throw new UnauthorizedException('Invalid token type');
    }
    return payload;
  }
}
```

This implementation introduces several security measures:

- the token is extracted exclusively from the `Authorization` header;
- token expiration is always enforced (`ignoreExpiration: false`);
- the token type is checked, preventing the misuse of *refresh tokens* as *access tokens.*

Refresh Token

The *Refresh Token* has a longer lifespan and is used exclusively to obtain a new *Access Token* upon its expiration. This token is:

- stored inside an `HttpOnly` cookie;
- made inaccessible to client-side JavaScript.

This design choice provides effective protection against *Cross-Site Scripting* (XSS) attacks, as it prevents token exfiltration through malicious code executed in the browser.

Automatic renewal of the *Access Token* occurs transparently for the user, improving the user experience without compromising the overall security level of the system.

2.5.3 Authorization: Role-Based Access Control (RBAC)

In addition to authentication, the system implements a role-based authorization mechanism (*RBAC – Role-Based Access Control*). Users are classified into distinct roles (e.g., *Admin* and *User*), each with well-defined permissions.

In the NestJS backend, access control is implemented through the use of *Guards*, components that intercept incoming requests and verify:

- the validity of the JWT token;
- the role associated with the authenticated user;
- the compatibility of the role with the requested operation.

This approach ensures:

- a clear separation of responsibilities;
- protection of sensitive operations (such as tournament management and user administration);
- improved maintainability and scalability of the system.

Role Definition

File: `auth/roles.ts`

```
export enum ROLES {
  ADMIN = 'ADMIN',
  USER = 'USER',
}
```

Decorator for Role Assignment

The decorator allows authorized roles to be specified directly at the controller level or on individual endpoints.

RolesGuard – Permission Verification

File: `auth/roles.guard.ts`

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler())
      ;
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    const hasRole = roles.some((role) => user.type_user === role)
      ;
    if (!hasRole) {
      throw new ForbiddenException('Access denied');
    }
    return true;
  }
}
```

This *guard*:

- intercepts the request before controller execution;
- verifies the role associated with the authenticated user;
- blocks access to unauthorized resources.

The use of `RolesGuard` ensures centralized permission management that is easy to extend and integrate across all application modules.

This approach enables clear, extensible, and centralized permission handling.

2.5.4 API Security and Input Validation

The security of the REST APIs exposed by the backend is further strengthened through additional preventive measures:

- **Input validation**: every request is validated to prevent malformed or potentially malicious data from being processed;
- **Rate limiting**: request-limiting mechanisms reduce the risk of brute-force or denial-of-service attacks;
- **CSRF protection**: the combined use of `HttpOnly` cookies and JWT tokens helps mitigate Cross-Site Request Forgery attacks.

2.5.5 Data-in-Transit Security and Frontend Integration

Communication between the frontend and backend occurs exclusively over secure connections (HTTPS), ensuring encryption of data in transit. On the frontend side, secure session management and XSS prevention strategies complete the overall security architecture of the system. The use of Mongoose Schemas makes it possible to apply structural and security constraints directly at the model level.
File: `blog/blog.schema.ts`

```typescript
@Schema({ timestamps: true })
export class Blog extends Document {
  @Prop({ type: Types.ObjectId, ref: 'User', required: true })
  createdBy: Types.ObjectId;

  @Prop({ type: String, required: true, maxlength: 50 })
  subject: string;

  @Prop({ type: String, required: true, maxlength: 1000 })
  description: string;
}
```

This definition:

- enforces length and type constraints on data;
- reduces the risk of inserting invalid or malicious content;
- ensures database integrity and consistency.

Capitolo 3

Frontend Implementation

3.1 Frontend: Architecture, Navigation, and User Experience

The STCMS frontend was developed as a Single Page Application (SPA) using Angular, with the goal of providing a modern, high-performance interface accessible from any device and modern browser. The adoption of Angular allows the application to initially load only the main structure, while content is dynamically updated through components and services, significantly improving perceived performance and user experience.

3.1.1 Modular and Component-Based Structure

The application is organized according to a domain-driven structure, visible in the `src/app/domains` folder, where each functional domain is logically separated:

- **Auth**: authentication management (login, signup);
- **Home**: homepage, hero section, and tutorial;
- **Results**: tournaments, categories, results, and matches;
- **Sports**: team creation and management;
- **Admin**: administrative dashboard;
- **About**: informational page.

Each functionality is encapsulated in independent Angular components, for example:

- Homepage (`homepage.ts`);
- AdminDashboard (`admin-dashboard.ts`);
- CreateTournament (`create-tournament.ts`);
- Match (`match.ts`);
- Result (`result.ts`).

This approach promotes maintainability, code reuse, and visual consistency, allowing the system to be extended without introducing rigid coupling between components.

3.1.2 UI/UX, Bootstrap, and Responsiveness

To ensure optimal usability on desktop, tablet, and mobile devices, the frontend integrates Bootstrap as a CSS framework. The grid system and responsive components allow the layout to automatically adapt to different screen resolutions.

Each view (e.g., `homepage.html`, `tournaments.html`, `admin-dashboard.html`) uses flexible layouts and standardized components, ensuring:

- intuitive navigation;
- graphical consistency;
- compatibility with major modern browsers;
- support for older versions (backward compatibility).

3.1.3 Navigation and Routing

Navigation is centrally managed through the `app.routes.ts` file, which clearly and declaratively defines all application routes. Some significant examples include:

```
{ path: 'home', component: Homepage }
{ path: 'login', component: Login }
{ path: 'admin', component: AdminDashboard }
{ path: 'create-tournament', component: CreateTournament }
{ path: 'result', component: Result }
```

This configuration enables smooth navigation without page reloads, improving user experience and reducing browser load.

3.1.4 State Management and Backend Communication

Communication with the backend REST APIs is managed through dedicated HTTP services, which fully encapsulate API access logic:

- AuthService (`core/auth/services/auth.ts`);
- ResultsService (`domains/results/services/results.ts`);
- AdminService (`domains/admin/services/admin.ts`).

For example, the ResultsService handles:

- tournament creation (`createTournament`);
- retrieval of tournaments by sport (`getTournamentsBySport`);
- team creation (`createTeam`);
- retrieval of tournament matches.

The service uses `HttpClient`, dynamic parameters (`HttpParams`), and Angular Signals to maintain a reactive data state, such as selected tournaments and associated matches, ensuring immediate interface updates.

3.1.5 Client-Side Authentication and Security

Frontend-side security is managed by the AuthService, which implements the dual-token model designed in the backend:

- the Access Token is stored in `sessionStorage`;
- the Refresh Token is managed by the backend through HttpOnly cookies.

The service is responsible for:

- login (`login`);
- registration (`register`);
- logout (`logout`);
- automatic token persistence;
- authentication state updates via signals and computed properties.

This solution reduces the risk of XSS attacks and fully centralizes session management, avoiding duplication across UI components.

3.1.6 Performance and Efficient Loading

Frontend performance is optimized through:

- SPA architecture;
- logical separation of domains;
- loading components only when required by navigation;
- use of centralized services to avoid redundant requests.

Interaction with the backend is designed to be fast and minimal, enabling rapid loading of standings, tournaments, and results even under heavy usage scenarios.

3.2 Single Page Application (SPA) Design

The frontend of the STCMS is built as a Single Page Application (SPA) using the Angular framework. This architecture allows for a highly dynamic user experience by loading a single HTML page and dynamically updating content as users interact with the application. This approach minimizes server load and provides a native-app-like experience, which is essential for real-time visualization of tournament brackets and match results.

3.3 UI/UX and Responsiveness

3.3.1 Bootstrap Integration and Cross-Platform Compatibility

To meet the Portability requirement (Importance 6/10), the application integrates the Bootstrap CSS framework. This choice ensures that STCMS is fully responsive and compatible with all major web browsers and mobile devices.

Key aspects include:

- **Grid System**: used to create flexible layouts that adapt to different screen resolutions;
- **Backward Compatibility**: ensures that the application remains functional even on older browser versions, maintaining broad accessibility.

3.3.2 Component-Driven Development

The user interface is decomposed into independent, reusable components. Each functional area (e.g., tournament list, match scoring form, news feed) is encapsulated within its own Angular component. This modularity simplifies maintenance and ensures consistent styling and behavior across the entire platform.

3.4 Performance Optimization

3.4.1 Lazy Loading and Module Separation

Performance is a high-priority requirement (Importance 10/10). To optimize initial load time, the application implements Lazy Loading:

- **Feature Modules**: the application is split into distinct modules (e.g., AuthModule, AdminModule, TournamentModule);
- **On-Demand Loading**: modules are downloaded by the browser only when the user navigates to the corresponding route.

This strategy significantly reduces the size of the initial JavaScript bundle, ensuring a fast Time to Interactive (TTI).

3.5 Client-Side Security and Interoperability

3.5.1 Security and Interceptors

On the client side, security is managed through Angular Interceptors. These services automatically inject the JWT Access Token into the headers of every outgoing HTTP request, ensuring secure communication with the backend without requiring manual token handling in each component.

Capitolo 4

Quality Assurance and Performance

4.1 Testing Strategy

To ensure the reliability, functional correctness, and security of the STCMS, a testing strategy was adopted that primarily focuses on the validation of the REST APIs exposed by the NestJS backend. Given the nature of the system, which manages sensitive user data and critical information related to tournaments and competitions, testing the application interfaces represents a central element of the development process.

4.1.1 API Validation via Swagger

The main tool used for backend testing was Swagger (OpenAPI), which is directly integrated into the NestJS application. Swagger enables automatic documentation of all REST endpoints and allows interactive testing, facilitating both the development phase and functional verification.

Through the Swagger interface, all application controllers were systematically tested, with particular attention to the following aspects:

- **Functional Testing**: verification of the correct creation and management of users, tournaments, teams, and matches. In particular, tournament creation workflows, team enrollment, match generation, and result updates were tested to ensure consistency of the data persisted in the MongoDB database.
- **Authentication and Authorization Testing**: validation of the JWT-based authentication system with dual tokens (Access Token and Refresh Token). Tests verified that:
  - unauthenticated requests are correctly rejected with HTTP status code 401;
  - authenticated users without the required permissions receive HTTP status code 403;
  - roles (Admin and User) are correctly enforced through NestJS Guards.
- **Security Testing**: verification of correct handling of authorization headers, JWT token validity, and protection of sensitive endpoints. Scenarios involving expired or malformed tokens were also tested to ensure robust error handling.

- **Data Validation**: verification of input validation mechanisms (DTOs and NestJS Validation Pipes), ensuring that invalid or incomplete data is rejected before reaching the business logic.

The use of Swagger made it possible to promptly identify inconsistencies between frontend and backend, facilitating debugging and improving alignment between API documentation and actual behavior.

4.1.2 Support for Regression Testing

Swagger was also used as a support tool for regression testing. After each significant modification to application logic or the introduction of new functionalities, existing endpoints were re-tested to verify that previous behavior had not been compromised. This approach helped maintain high system stability throughout the evolution of the project.

4.2 Non-Functional Requirements Assessment

The adopted testing approach supported the verification of the main non-functional requirements defined during the architectural analysis phase, particularly security, reliability, and correctness of communication between frontend and backend. Continuous API validation ensured that the system was ready for real-world usage and aligned with the project objectives.