Università
Ca'Foscari
Venezia

Book Shop

Tecnologie e Applicazioni Web project

Diego Caspi, Marco Mihai Condrache, Alberto Campagnolo

Diego Caspi
Ca' Foscari University
2024/25

I. Introduction

This project implements a modern academic book auction platform using a sophisticated technology stack and following industry best practices. The system is built as a monorepo using Nx for efficient workspace management and better code organization.

II. Project Architecture

The project has been built using Nx as a monorepo organization tool, providing several key advantages for development and maintenance:

- **Shared Code**: Common utilities and components can be easily reused across applications
- **Consistent Tooling**: Development experience remains uniform across the entire project
- **Atomic Changes**: Updates affecting multiple packages can be tested and deployed together
- **Simplified Dependencies**: Better management of internal dependencies between packages

II.1. Monorepo structure

The project follows a clean architecture pattern with clear separation of concerns:

Listing 1 — Folder structure

```
- apps/
  - frontend/     // Angular 18 client application
  - backend/      // Express JS server application
- libs/
  - api/          // Shared API interfaces and API related types
  - core/         // Common utilities and patterns
  - domain/       // Business logic and models
```

II.2. Libraries

The project follows a modular architecture using a libraries-based approach within the Nx monorepo structure. The libraries are organized in the `libs/` directory and serve specific purposes in the application architecture.

Core Library (`@shared/core`)

The Core library serves as the foundation for shared utilities and patterns across the entire application. It provides essential functionality that supports the entire system's operation through several specialized modules:

JSON Patch Operations (`json-patch`)

The JSON Patch module implements RFC 6902, providing a standardized way to describe changes to JSON documents. This is particularly useful for handling partial updates in our REST API endpoints.

Listing 2 — JSON Patch operation example

```
[
  { "op": "replace", "path": "/book/title", "value": "New Title" },
  { "op": "remove", "path": "/book/oldField" }
]
```

These operations allow for precise modifications to resources without requiring full object replacement, reducing network payload and preventing race conditions during updates.

Left-hand Side Operations (`lhs`)

The LHS module enables flexible API filtering through a standardized query parameter syntax. It allows clients to construct complex filters using dot notation and operators:

Listing 3 — LHS filtering examples

```
// Example filters:
book.price[gt]=50            // Books with price greater than 50
book.university[eq]=Ca Foscari // Books from Ca' Foscari
auction.endDate[lt]=2024-04-01 // Auctions ending before April 1st
```

Error Handling (`errors`)

The centralized error system implements a comprehensive approach to error management across the application. It provides domain-specific error classes that accurately represent different failure scenarios, while maintaining a standardized error response structure for consistent client-side handling. The following example represents the structure adopter for the error response:

Listing 4 — Error response content example

```
{
  "message": "string",
  "status": 401,
  "traceId": "5cbd8075-48e6-4801-bced-f8fd16f7b357"
}
```

Pagination (`pagination`)

The system implements a robust pagination mechanism that utilizes a straightforward `page` / `pageSize` approach, making it ideal for common use cases where simplicity and efficiency are paramount. Each paginated response includes comprehensive metadata alongside the results, providing essential information about the total number of items, current page position, and the overall page count. Here an exampe:

Listing 5 — Paginated response example

```json
{
  "list": [
    ...
  ],
  "metadata": {
    "totalItems": 100,
    "page": 4,
    "totalPages": 10
  }
}
```

API Sorting (`sorting`)

The sorting module provides a simple way to sort the results of a paginated request using a `sort` parameter. The `sort` parameter is a string that contains the field to sort by prefixed by the direction of the sort, represented by a `+` or `-` sign. The following example shows how to sort the results by price in ascending order:

Listing 6 — Sorting example

```
// Example sort:
sort=+book.price
```

This sorting approach provides as the flexibility to sort the results by multiple fields, for example:

Listing 7 — Sorting multiple fields example

```
// Example sort:
sort=+book.price,-book.title
```

Domain Library (`@shared/domain`)

The Domain library serves as the cornerstone of our application's data architecture, encapsulating all core business models and schemas while ensuring type safety throughout the entire system. At its heart, the library is structured around two fundamental components that work in harmony to maintain data integrity across the application stack.

The first component, housed in the `/api` directory, leverages the `zod` library to define our API schemas. These schemas act as a contract between frontend and backend services, providing runtime type validation while simultaneously generating TypeScript types. This approach not only ensures type safety during development but also automatically generates OpenAPI documentation, keeping our API documentation perpetually synchronized with the actual implementation. The schema-first approach allows us to implement complex business validation rules directly within our type definitions, creating a single source of truth for data validation.

Listing 8 — Advanced schema validation example

```
export const ApiAuctionCreationSchema = ApiAuctionSchema.omit({
  seller: true,
  winningBid: true,
})
  .refine(data => data.startingPrice < data.reservePrice, {
    message: "Starting price must be less than reserve price",
    path: ["startingPrice"],
  });
```

Complementing the API models, the `/db` directory contains our database schemas implemented using `typegoose` . This sophisticated ODM (Object-Document Mapper) brings the power of TypeScript to MongoDB operations, offering type-safe database interactions through decorated class definitions. These models handle not only the basic schema structure but also incorporate sophisticated features such as middleware hooks for pre and post database operations, index definitions, and relationship mappings between collections.

The synergy between these two components creates a seamless, type-safe data pipeline from the database through to the client application. When data flows through our system, it's validated at every step, ensuring consistency and reliability. This architectural decision significantly reduces the likelihood of runtime errors while simultaneously improving the developer experience through comprehensive IDE support and compile-time type checking.

API Library (`@shared/api`)

The API Library establishes a robust contract between the frontend and backend services, ensuring type safety and consistency across the entire application. Through its thoughtfully organized structure, it provides a comprehensive framework for defining and managing API interactions.

At its foundation, the Backbone component forms the core infrastructure by defining essential types and utilities. The cornerstone of this system is the `Endpoint` type, which serves as a blueprint for all API endpoints, ensuring they contain necessary information for both runtime execution and development-time type checking.

Building upon this foundation, the Endpoints section implements concrete API definitions using a declarative approach. Each endpoint is structured as a complete specification, encompassing all aspects needed for both implementation and documentation. As shown in the following example:

Listing 9 — API endpoint definition example

```
export const getListingsEndpoint = endpoint({
  path: "/v1/listings",
  method: "get",
  lhs: ["book.university", "book.course", "auction.startingPrice"],
  paramsSchema: PaginationRequestSchema.innerType(),
  responseSchema: PaginatedResponseSchemaOf(ApiListingSchema),
  config: {
    authentication: false,
    tags: ["Listing"],
    summary: "Get all listings",
    description: "Get all listings with pagination",
  },
});
```

This structured approach provides several key benefits:

- Type-safe parameter validation through schema definitions
- Built-in support for left-hand side (LHS) operations enabling flexible data filtering
- Automatic OpenAPI documentation generation from endpoint configurations
- Clear visibility of authentication requirements through the `authentication` flag
- Organized API grouping using tags for better documentation structure

The Schema component complements these elements by providing shared API schemas that extend beyond domain models. These schemas handle cross-cutting concerns such as authentication payloads, statistical data structures, and other utility types that don't directly map to domain entities but are essential for the API's operation.

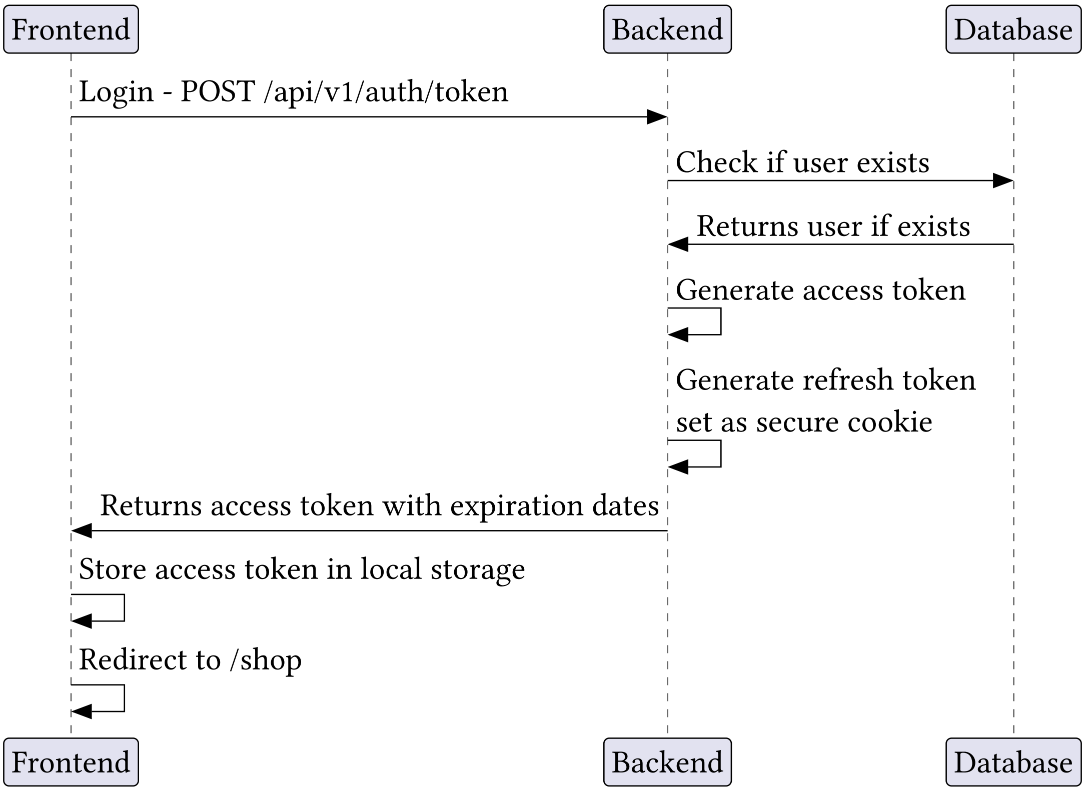
II.3. Backend application (`apps/backend`)

The application's structure revolves around feature-based organization, where each domain concept is encapsulated within its own feature module. These modules, located in the src/features directory, contain controllers, services, and related business logic. This approach ensures that related functionality remains cohesive while maintaining loose coupling between different parts of the system.

Authentication

Central to the architecture is the authentication system, implemented through JWT tokens with a dual-token approach - access and refresh tokens - providing secure user sessions while maintaining a smooth user experience. The AuthService manages token generation, validation, and refresh mechanisms, working in tandem with the UserService to handle different user roles (students and moderators) with appropriate access controls.

Figure 1 — Authentication flow sequence diagram



All the generated tokens are JWTs (JSON Web Tokens), they are signed using a secret key stored in the environment variables. Each token contains the following information in its payload:

- `sub` : The public ID of the user's profile
- `username` : User's username
- `email` : User's email address
- `type` : Set to `access` for access tokens and `refresh` for refresh tokens
- `scope` : Either `user` or `admin` indicating the user's role
- `exp` : Expiration timestamp in milliseconds

Real-time

In order to provide a real-time experience to the users, we've implemented a websocket connection between the frontend and the backend using the `socket.io` library.

The websocket connection is used to share messages between users, listening for real-time updates of the auction status and the current price on the auction details page.

The following diagrams show how the system handles the websocket on different usecases:

Figure 2 — Websocket authentication / connection flow

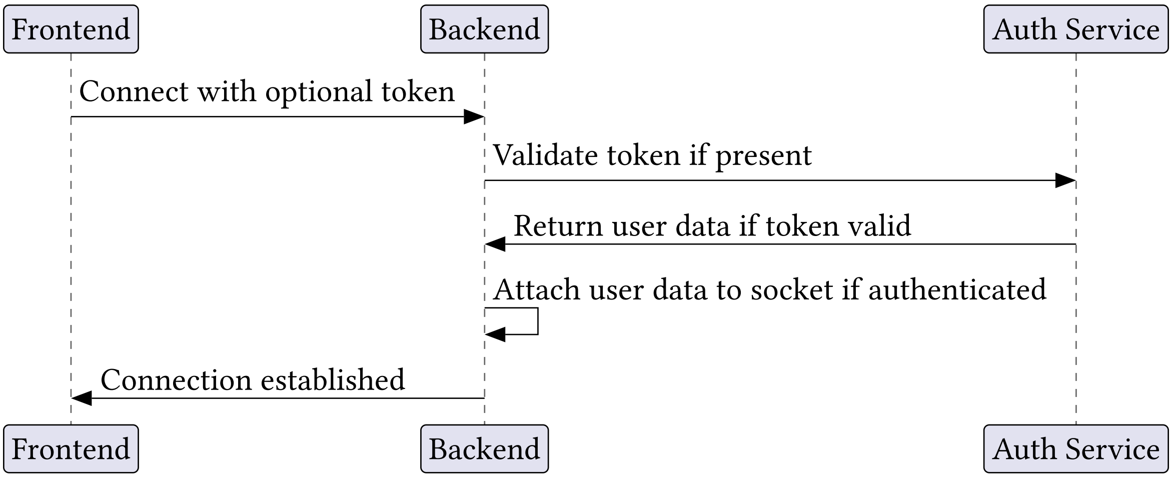

Frontend
Backend
Auth Service
Connect with optional token
Validate token if present
Return user data if token valid
Attach user data to socket if authenticated
Connection established
Frontend
Backend
Auth Service

Figure 3 — Listing Watch Flow

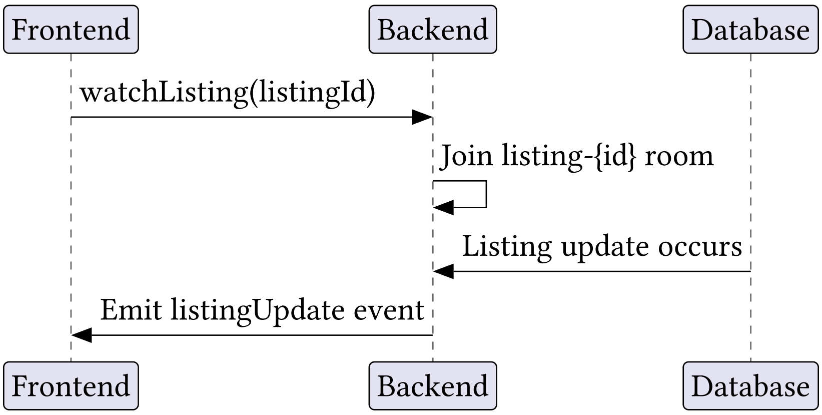


Figure 4 — Chat Message Flow

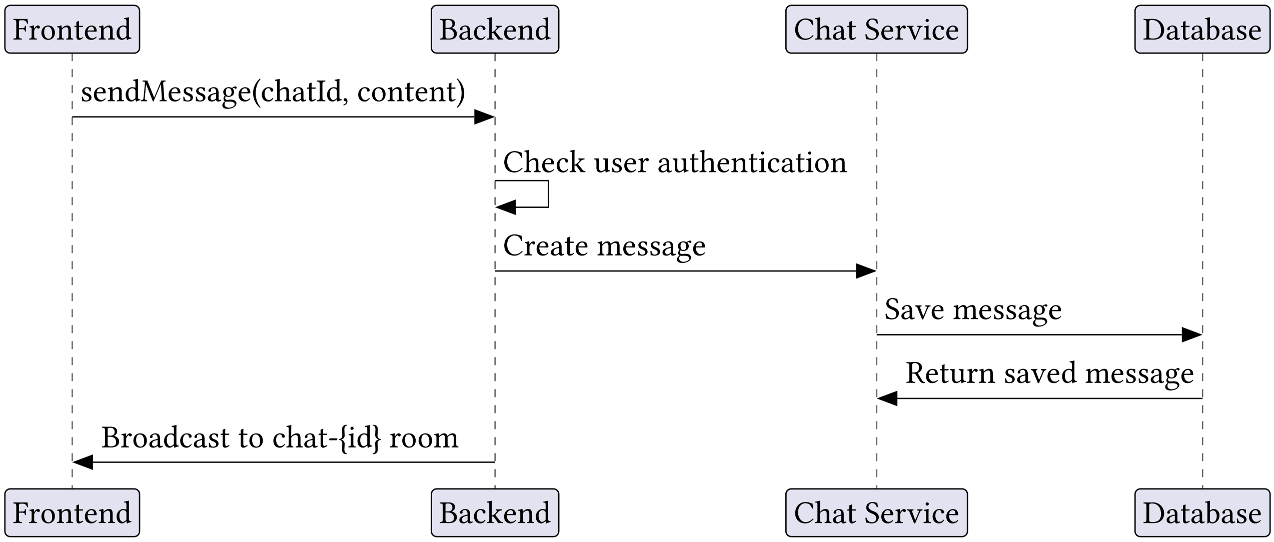
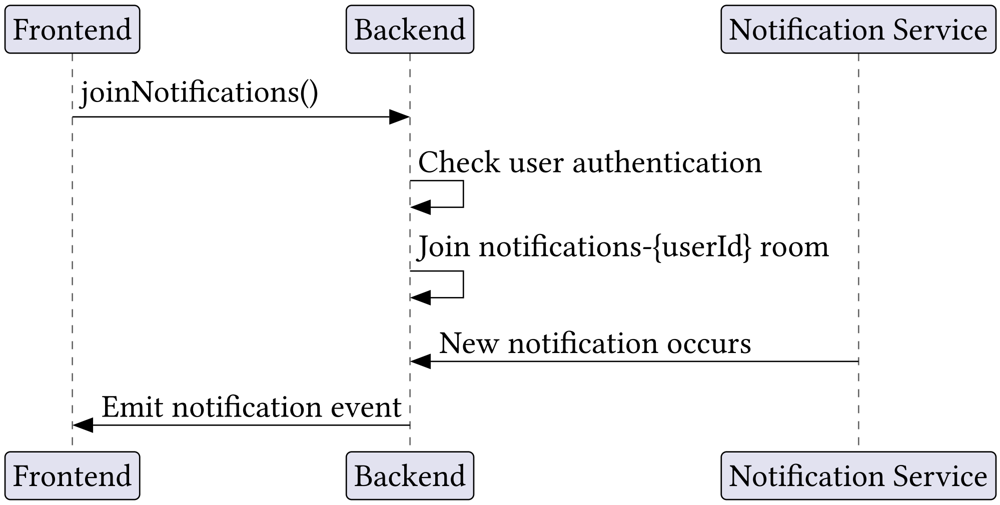


Figure 5 — Notifications Flow



II.4. Frontend application (`apps/frontend`)

The frontend application is built using Angular 18 and follows a feature-first architecture pattern, where the application is organized into distinct feature modules that encapsulate related functionality. The application is structured into two main sections: the public shop interface and the administrative dashboard, each with its own layout and routing configuration.

The UI is built using Taiga UI v3 components, enhanced with Tailwind CSS for styling, creating a consistent and modern look throughout the application.

State management

The application uses `rxjs` to handle the state management, providing a reactive approach to data flow and state management.

This reactive pattern allows to implement a more streamlined flow management when dealing with state changes, for example, when a user logs in, the application will emit a login event, which will be listened by the `UserService` that will update the user's state and redirect the user to the shop page.

All the state is managed as `Observable` / `Subscription` converted to Angular `signals` to provide a more intuitive and easy to use interface.

Routing

The routing system is handled via the `app.routes.ts` file, which is the main entry point for the routing configuration. Here all the routes are hiearchically organized into sections, each with its own layout and routing configuration.

Server communication

In order to provide an easy and adapted interface for the server communication, we've implemented the `ApiService` class, which is a wrapper around the `HttpClient` service. This service takes care of all the necessary operations defined in the endpoints declared in the `@shared/api` library, providing an easy to use interface for the frontend components.

From a developer perspective, when integrating a new endpoint, it's just necessary to call the `ApiService` method with the endpoint we want to use and the body/params we want to send to the server, the `ApiService` will take care of the rest.

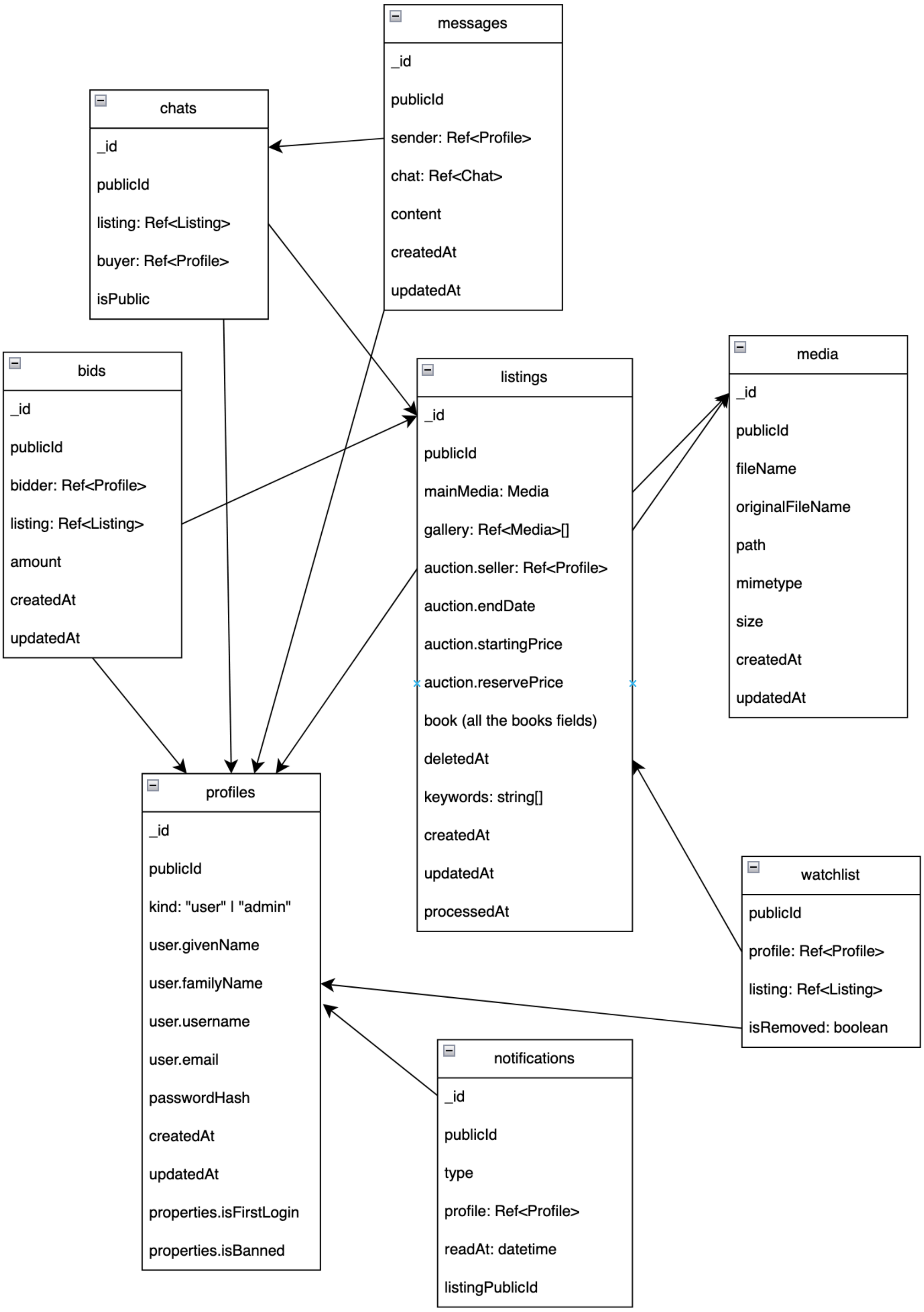
Listing 10 — Frontend API service example

```
manageUserBan(publicIds: string[], action: "ban" | "unban") {
  const body = { publicIds, action };
  return this.apiService.request(userBanEndpoint, { body }).subscribe();
}
```

III. Database schema

The database schema is defined using `typegoose` , a library that allows to define the schema of the database using TypeScript classes.

Figure 6 — Database schema



IV. Deployment

The project leverages `nx` and `docker` to manage the build and deployment of the application.

Each application has its own `Dockerfile` where the build configuration is defined, in particular the frontend application is built using the `nginx` image, while the backend application is built using the `node:lts-alpine` image.

In order to build the app image, a build through `nx` is required, the following command will build the applications and then the docker image:

Listing 11 — Build command

```
nx run-many --target=docker-build --all
```

This command will build the applications using `esbuild` for both the frontend and the backend, then it will build the docker images including the built artifacts.

In order to improve the security of the containers, a multi-stage build is used, where the final image includes just the crucial files for the application to run, reducing the attack surface of the container.

V. Screenshots

VI. Conclusion

The project is a modern academic book auction platform using a sophisticated technology stack and following industry best practices. The system is built as a monorepo using Nx for efficient workspace management and better code organization.