OurSpace (Custom List) - Design Doc

Authors: floew@google.com, silviobraendle@google.com

Last updated: 21 Feb 2025

One-page overview

Context

This document outlines the design for the OurSpace application, a social media platform enabling users to create and share lists, managed by administrators. The application utilizes a full-stack architecture with React frontend, Spring Boot backend, and PostgreSQL database. This document encompasses the design of the backend, frontend, and the interaction between them. A core component of this application is the "Custom List" feature, which is described in this document.

Goals

- Functional Requirements:
    - Implement a MyListEntry model storing information about a list entry: title, text, creation date, and importance.
    - Allow each user to create multiple list entries, with each entry associated to one user.
    - Provide CRUD (Create, Read, Update, Delete) API endpoints for managing list entries.
    - Implement an API endpoint to retrieve list entries of a specific user, sorted by importance.
    - Ensure only the creator or an administrator can edit/delete a list entry.
- Data Persistence: Design and implement the database schema necessary for data persistence of the list data.
- API Design: API endpoints should be RESTful and well-documented.
- User Interface: Create a user-friendly React component for creating, viewing, editing, and deleting custom lists.
- Security and Permissions: Implement appropriate authentication and authorization to ensure only authorized users can access and manipulate list entries.

- Code Quality: Maintain clean, well-documented code.
- Multiuser Capability: Ensure all the data operations are multiuser capable.

Outside the Scope

- Advanced UI/UX: While usability is considered, the design will prioritize functionality and correctness over complex visual styling or intricate user experience design.
- Performance Optimization: Initially, optimization for high-traffic scenarios will not be a primary focus. Future iterations may address performance considerations.
- Third-party Integrations (beyond core stack): Integration with external services (beyond the established React, Spring Boot, PostgreSQL stack) is out of scope.
- Comprehensive Testing of the Entire Application: While the Custom List feature will be thoroughly tested, extensive testing of all other OurSpace functionalities is outside the scope of this document.
- Internationalization/Localization: Supporting multiple languages or regional settings is not a goal.

Focus on a Specific UI Component Library: We want the UI to be functional so we do not want to decide on any component library in this design document.

High-Level Design

Overall Architecture

Frontend

The frontend will be developed using React with TypeScript. React components will be responsible for rendering the user interface and handling user interactions. The data will be fetched using the API provided by the backend.

Backend

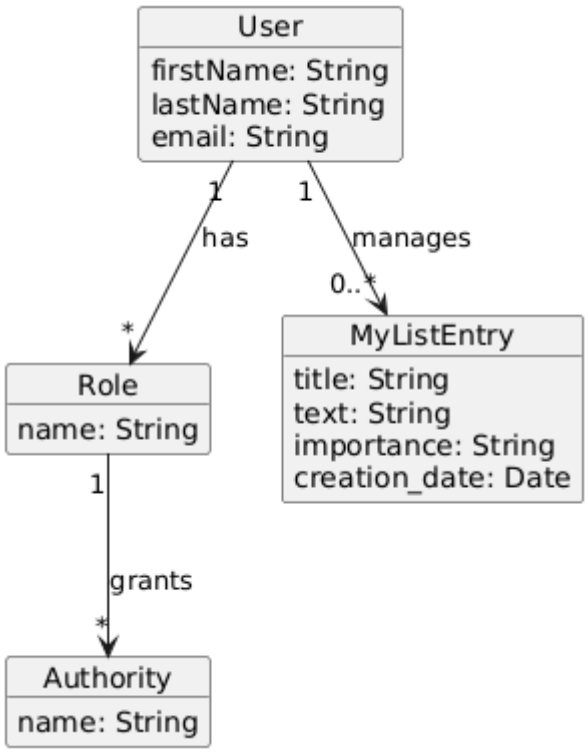
The backend will be implemented using Spring Boot. It will provide RESTful APIs for the frontend to interact with. Spring Data JPA will be used for database access. The backend will handle business logic, data validation, authentication, authorization, and data persistence.

Database

PostgreSQL will be used as the relational database for storing application data. It will host tables that represent users, user profiles, custom lists, and other relevant data. The

database schema will be designed to ensure data integrity, consistency, and efficient querying. Spring Data JPA will be used to map java objects to database tables.

Domain Model Overview



Detailed Design: Custom List Feature

Overview

The "Custom List" feature allows users to create and manage their own lists, with each entry containing a title, text, creation date, and priority. This feature will involve designing a data model for list entries, creating REST endpoints for CRUD operations, and developing a user interface for managing the lists.

User Interface

- **View User Entries:** Users can view a list of entries created by any user.
- **Sort by Importance:** User entries can be sorted by importance.
- **Create Entries:** Authenticated users can create new MyListEntry with title, text, creation date, and importance.
- **Edit Own Entries:** Authenticated users can edit and update their own existing MyListEntry.
- **Delete Own Entries:** Authenticated users can delete their own MyListEntry.

- **Admin Privileges:** Administrators can edit and delete any user's MyListEntry.

API Endpoints

CRUD operations will be performed through REST endpoints defined in Spring Boot.

Method	Endpoint	Description
POST	/my-list-entry	Creates a new MyListEntry.
GET	/my-list-entry/{id}	Retrieves a specific MyListEntry by its ID.
GET	/my-list-entry/user/{userId}	Retrieves all MyListEntry belonging to a specific user.
GET	/my-list-entry/user/{userId}?sort=importance&order=desc	Retrieves all MyListEntry belonging to a specific user and sorted by importance.
PUT	/my-list-entry/{id}	Updates an existing MyListEntry.
DELETE	/my-list-entry/{id}	Deletes a MyListEntry by its ID.

Request Parameters

Pagination & Sorting

Passing the **Pageable** argument allows us to take pagination and sorting data straight out of the QueryParameters in a **Pageable** object that can be passed to a JPA repository

Filtering

The SpringFilter library allows us to use the **@Filter Specification<T>** argument that automatically uses our QueryParameters to generate a **Specification** that can be passed to a JPA repository.
A Spring-Filter-Query-Builder Typescript Library is available for the frontend.

Life of an API Request

The following sequence diagram visualizes the life of an API request through the backend.

Security Considerations

Authentication

The API will use JSON Web Tokens to authenticate the user. After authenticating with a password, the API will send an access token for authentication on the endpoints and a refresh token to refresh when the access token expires.

Security Matrix

Feature	Action	Unauthenticated	Authenticated	Administrator
MyListEntry	**Create**	No	Yes	Yes
	Read	No	Yes (Any)	Yes (Any)
	Update	No	Yes (Own)	Yes (Any)
	Delete	No	Yes (Own)	Yes (Any)
User	**Create**	Yes	Yes	Yes
	Read	No	Yes (Any)	Yes (Any)
	Update	No	Yes (Own)	Yes (Any)

	Delete	No	Yes (Own)	Yes (Any)

Complex Authentication Options Considered

[Chosen] Option 1: Implement Authorization Checks in the Controller with Custom PermissionEvaluator class in PreAuthentication

Example implementation: Spring Security: Authorization with domain logic - Stack Overflow

Pros:

- Abstracts Authorization to a separate place allowing possible reusability and keeping it clear what's Authorization logic and what isn't
- Always calls Authorization checks from PreAuthorization checks - good for consistency.

Cons:

- Annotations don't provide type safe IntelliSense.

Option 2: Implement Authorization Checks in the Controller with Custom Annotation

Pros:

- Abstracts Authorization to a separate place allowing possible reusability and keeping it clear what's Authorization logic and what isn't

Cons:

- Has 2 different Annotations with the same responsibility; checking authorization. We prefer to do all the Authorization checking in one go, thus the chosen option 1.

Option 3: Check Complex Authorization in Service Layer

Check whether the User has write access on MyListEntity in Service Layer

Pros:

- Simplicity in implementation because of making all database calls inside the one function

Cons:

- Sacrificing separation of responsibilities (e.g. Service). Controller should decide if authorization can be granted.
- Checking Authorization at 2 different stages.

Authorization

The application will implement Role-Based Access Control (RBAC) to manage user permissions. The following roles will be defined:

- User: A standard authenticated user.
- Admin: An administrator with elevated privileges.

The roles will be associated with the permitted authorities.

Testing Strategy

The testing strategy for the application will encompass unit, integration, and end-to-end testing to ensure functionality, security, and reliability. Special attention will be given to access control and error handling.

Testing Matrix

Test Type	Target	Tools Used	Focus
Unit Tests	Backend	JUnit, Mockito	Business logic, data manipulation, exception handling.
Integration Tests	Backend	MockMvc	Component interaction, data persistence, REST endpoints, access control.
End-to-End Tests	Both	Cypress	End-to-end workflows, access control, overall application flow.

Endpoints to be Tested by Integration Tests

The integration tests, test for successful requests (eg. Creating a MyListEntry as an admin, then checking for the created MyListEntry), Unauthorized Access requests, Unauthenticated Access requests and invalid requests (eg. Requesting for a resource that does not exist).

The following endpoints will be tested by run through the tests:
AuthController.java:

- POST /auth/authenticate
- POST /auth/refresh

MyListEntryController.java:

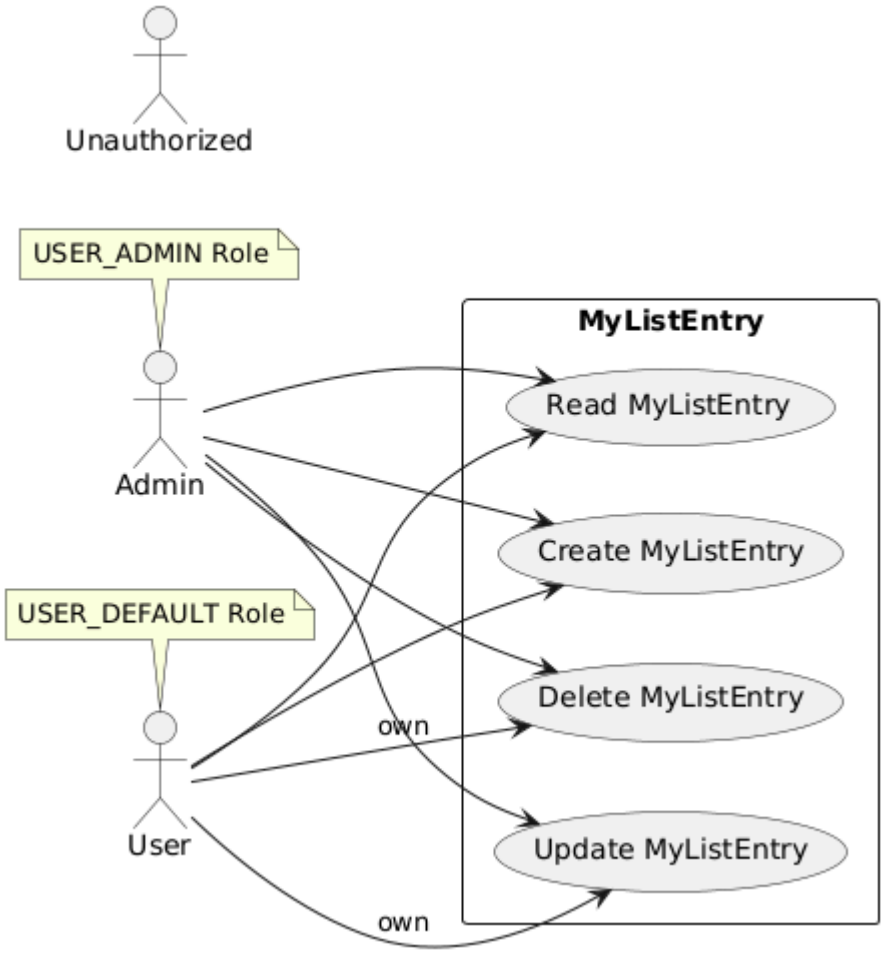
- GET /mylistentry
- GET /mylistentry/{id}
- POST /mylistentry

- PUT /mylistentry/{id}
- DELETE /mylistentry/{id}

UserController.java:

- GET /user/{id}
- GET /user
- POST /user/register
- POST /user/registerUser
- PUT /user/{id}
- DELETE /user/{id}

Use Case Diagram



Use Case: Create MyListEntry

Description	This use case describes the steps a user takes through the OurSpace UI to create a new MyListEntry. This scenario is specifically crafted for automated testing with Cypress.

Actors	Authenticated User
Preconditions	• User has a valid OurSpace account and is already logged in. • The browser is on the OurSpace MyList page.
Postconditions	• A new MyListEntry is created in the database, associated with the logged-in user. • The user is redirected to the list view page to see the newly created list.

Normal Flow:

1. User clicks the create entry button to initiate the process of creating a new MyListEntry. This navigates to the create entry page.
2. The create list entry page is displayed with input fields for the list entry title, text, and importance.
3. The user fills in the title, text, and importance fields.
4. User clicks the create button to submit the new list entry.
5. The system creates the new entry and redirects the user to their list view.
6. The newly created list entry is displayed in the user's list view.

Alternate Scenarios:

Invalid Input	The user enters invalid data in one or more fields. An error message is displayed for the respective fields and the submit action fails.

Appendix

Diagrams

PlantUML for (ERD Diagram):

```
@startuml
!theme plain

entity "MyListEntry" {
 * id : Long (PK)
 --
 title : String
 text : String
 creationDate : Timestamp
 importance : String
 user_id : Long (FK)
}

entity "User" {
 * id : Long (PK)
 --
 // ... other user attributes (e.g., username, etc.) ...
}

User --o{ MyListEntry : creates

note top of MyListEntry
 Represents a custom list entry created by a user.
 - Each entry belongs to exactly one user.
 - Users can have multiple list entries.
end note

note top of User
 Represents a user in the application.
 - Users can create multiple MyListEntry objects.
end note

@enduml
```

PlantUML Use Case Diagram:

```
@startuml
left to right direction

actor Unauthorized
actor User
actor Admin

note top of Admin : USER_ADMIN Role
note top of User : USER_DEFAULT Role

rectangle MyListEntry {
 User --> (Create MyListEntry)
 User --> (Read MyListEntry)
 User --> (Update MyListEntry) : own
 User --> (Delete MyListEntry) : own
 Admin --> (Create MyListEntry)
 Admin --> (Read MyListEntry)
```

```
Admin --> (Update MyListEntry)
Admin --> (Delete MyListEntry)
}

@enduml
```

PlantUML Domain Model:

```
@startuml
object User {
 firstName: String
 lastName: String
 email: String
}

object Role {
 name: String
}

object Authority {
 name: String
}

object MyListEntry {
 title: String
 text: String
 importance: String
 creation_date: Date
}

User "1" --> "0..*" MyListEntry : manages
User "1" --> "*" Role : has
Role "1" --> "*" Authority : grants
@enduml
```

PlantUML for (Life of a Request Diagram):

```
@startuml
skinparam backgroundColor #F3F3F3
skinparam shadowing false

box "Frontend (React)" #DDDDDD
    participant "Browser" as FrontendBrowser
end box

box "Database" #CCCCFF
    box "Controller Layer" #BBBBFF
        participant "MyListController" as Controller
    end box

    box "Service Layer" #AAAAFF
        participant "MyListService" as Service
        ' Business Logic is now integrated within the Service
    end box

    box "Data Access Layer (JPA)" #9999FF
        participant "MyListEntryRepository" as Repository
    end box

    database "PostgreSQL Database" as Database #8888FF
end box

FrontendBrowser -> Controller : PUT /api/my-list-entry/{id}\nJSON Payload (MyListEntryDTO with updates)\n+ JWT (if logged in)
activate Controller #BBBBFF

Controller -> Controller : @PreAuthorize("isAuthenticated()")  // Check if user is logged in -  **Frontend check assumed**

alt user is authenticated
 Controller -> Controller : Extract User details from JWT and populate Roles & Authorities
 Controller -> Controller :  **Security Check: Authorization**\nCheck if User is Author or Admin\n(if nonexsistent item is checked throws an Exception)
  alt user is author or admin
    Controller -> Service : updateMyListEntryById(id, MyListEntryDTO, userDetails)
    activate Service #AAAAFF

    Service -> Service : Map DTO to Entity\nUpdate Entity fields with DTO data
    Service -> Repository : save(updatedMyListEntryEntity)
    activate Repository #9999FF
    Repository -> Database : UPDATE my_list_entry SET ... WHERE id = {id}
    activate Database #8888FF
    Database --> Repository : Success
    deactivate Database
    Repository --> Service : Updated MyListEntry Entity
    deactivate Repository
    Service --> Controller : Updated MyListEntry Entity
    deactivate Service
    Controller --> FrontendBrowser : 200 OK\nJSON Payload (Updated MyListEntryDTO)
    deactivate Controller
  else user is NOT author/admin
    Controller --> FrontendBrowser : 403 Forbidden
    deactivate Controller
 end alt

else user is not logged in
    Controller --> FrontendBrowser : 401 Unauthorized
    deactivate Controller
end alt
FrontendBrowser -> FrontendBrowser : Update UI with updated MyListEntry data

@enduml
```