UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Software Architecture and Methodologies
&
Quantitative Evaluation of Stochastic Models

Gift List

Edoardo Sarri

7173337

Contents

List of Figures

1 Introduction

This document outlines the concept for a "Gift List" application, designed to streamline the process of gift-giving for various celebrations and events. The core idea is to provide a platform where a celebrant (e.g., a birthday person, a couple getting married) can curate and share a personalized wish list of gifts they would like to receive.

The application serves two primary user roles:

- The Celebrant
  This user creates and manages their gift list. They can add, remove, and describe desired items. Critically, the celebrant will not have visibility into the real-time status of their gift list regarding which items have been claimed. This maintains the element of surprise for them.
- The Guest
  Guests receive a link to the celebrant's gift list. They can browse the available items and, crucially, "claim" a gift they intend to purchase. Once a gift is claimed by one guest, it becomes visibly unavailable to all other guests. This mechanism prevents duplicate gifts and helps guests coordinate their purchases efficiently.

The primary goal of this application is to simplify gift coordination, reduce redundant gifts, and enhance the gift-giving experience by ensuring celebrants receive desired items while preserving the element of surprise.

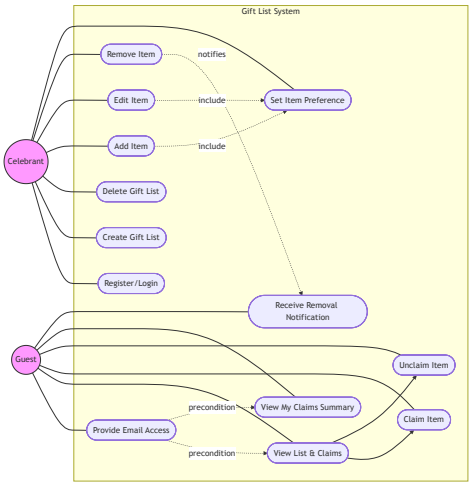

Figure 1.1: Use case diagram of Gift List requirements.

1.1 Requirements

In this Section we will define the requirements for the application. The application must meet the following functional requirements:

- **Celebrant Management**
  - User Authentication (Celebrant Only)
    The application must allow Celebrants to register and log-in to manage their lists securely. Guests should be able to access lists via a shared link without requiring an account, to minimize friction.
  - List Management
    Authenticated Celebrants must be able to create and delete gift lists.
  - Item Management
    Celebrants must be able to add, edit, and remove items from their lists. For each item, they can provide a name, description, an optional external URL, and set a preference level (e.g., Low, Medium, High).
- **Guest Interaction**
  - List Access
    Guests must be able to view a specific gift list using a unique URL provided by the Celebrant. To access the list items, a Guest must provide an email address; this is mandatory to ensure the system can notify the Guest if a claimed item is subsequently removed by the Celebrant.
  - Item Claiming
    Guests must be able to mark an item as "claimed" or "purchased". This action must be visible to other Guests to prevent duplicate purchases but must remain hidden from the Celebrant to preserve the surprise.
  - Item Unclaiming
    Guests must be able to revert a claim (e.g., in case of accidental selection), making the item available again. This action is only permitted for the guest who originally claimed the item.
  - Personal Claim Visibility
    Guests accessing the list should clearly distinguish which items they have personally claimed versus those claimed by others, facilitating easier management of their intended purchases.
  - Guest Claim Summary
    Guests must have access to a dedicated view or section that lists only the items they have personally claimed for a specific list, providing a clear summary of their planned gifts.

- **System Integrity**
  - Claimed Item Removal
    If a Celebrant removes an item that has already been claimed by a Guest, the system must allow the removal but must immediately notify the Guest who claimed it. This prevents the Guest from purchasing an item that is no longer desired, while maintaining the Guest's anonymity.

1.2 Internationalization (i18n)

The application is designed to be accessible to a global audience, with initial support for two languages:

- **English**: The default language for the interface and system communications.
- **Italian**: Full support for the Italian language.

The application will detect the user's browser language setting to provide the most appropriate initial experience, while also allowing users to manually switch between supported languages via a language selector in the UI.

1.3 Accessibility

The "Gift List" application will follow a phased release strategy regarding platform availability:

- Initial Phase: Web Application
  The application will initially be accessible exclusively as a web application through modern web browsers. Official support will be focused on the most widely used browsers (e.g., Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge). Limiting official support to these "evergreen" browsers ensures a consistent user experience, optimal performance, and robust security by leveraging the latest web standards and security protocols.
- Future Phase: Mobile Applications
  Following the stabilization of the web platform, dedicated native or hybrid applications are planned for distribution through the Apple App Store and Google Play Store to provide a more integrated mobile experience.

2 Backend

The backend of the Gift List application is designed to be robust, scalable, and secure. Based on the requirements for real-time interactivity and concurrent access, we have selected a Node.js environment. Node.js is an excellent choice for this application due to its event-driven, non-blocking I/O model, which efficiently handles multiple simultaneous connections—critical for when guests are viewing and claiming gifts at the same time.

2.1 Technology Stack

The proposed technology stack for the backend includes:

- Runtime Environment
  Node.js v24 (last LTS to Febbruary 2026).
- Web Framework
  Express.js. A minimalist and flexible framework that allows for rapid API development.
- Database
  PostgreSQL. Selected for its reliability and support for ACID transactions, which are essential for preventing race conditions (e.g., double-claiming a gift).
- ORM
  Prisma. To provide type-safe database access and simplify schema management.
- Authentication
  JSON Web Tokens (JWT) for stateless authentication of Celebrants, complemented by a Refresh Token strategy to ensure a seamless user experience without compromising security.

2.2 System Architecture

The backend will follow the Model-View-Controller (MVC) architectural pattern to ensure a clean separation of concerns.

- Models
  Represent the data structures and encapsulate the core business logic. In our stack, the Models are defined and managed via Prisma, which handles database interactions and ensures data integrity through its schema.

- Views
  In the context of a RESTful API, the View layer is responsible for formatting the data into standard JSON responses. It ensures that the client receives only the necessary information in a structured format.
- Controllers
  Act as the intermediary between Models and Views. They receive incoming HTTP requests, process user input, interact with the Models to perform operations, and select the appropriate View to return to the client.

2.3 Database Schema (Prisma)

The following Prisma schema defines the source of truth for our data models and their relations.

```
// schema.prisma

enum ItemStatus {
  AVAILABLE
  CLAIMED
}

enum PreferenceLevel {
  LOW
  MEDIUM
  HIGH
}

model User {
  id                 String     @id @default(uuid())
  email              String     @unique
  password           String
  resetToken         String?    @unique
  resetTokenExpires  DateTime?
  refreshToken       String?    @unique // For long-lived sessions
  createdAt          DateTime   @default(now())
  updatedAt          DateTime   @updatedAt
  lists              GiftList[]
}

model GiftList {
```

```
  id           String        @id @default(uuid())
  userId       String
  user         User          @relation(fields: [userId], references: [id])
  slug         String        @unique // Globally unique
  name         String
  items        GiftItem[]
  guestAccess  GuestAccess[]
  deletedAt    DateTime?
  createdAt    DateTime      @default(now())
  updatedAt    DateTime      @updatedAt
}

model GiftItem {
  id          String          @id @default(uuid())
  listId      String
  list        GiftList        @relation(fields: [listId], references: [id])
  name        String
  description String?
  url         String?
  status      ItemStatus      @default(AVAILABLE)
  preference  PreferenceLevel @default(MEDIUM)
  claim       GuestClaim?
  deletedAt   DateTime?
  createdAt   DateTime        @default(now())
  updatedAt   DateTime        @updatedAt
}

model GuestAccess {
  id        String       @id @default(uuid())
  listId    String
  list      GiftList     @relation(fields: [listId], references: [id])
  email     String
  language  String       @default("en")
  claims    GuestClaim[]
  createdAt DateTime     @default(now())

  @@unique([listId, email])
}

model GuestClaim {
```

```
  itemId    String      @id
  item      GiftItem    @relation(fields: [itemId], references: [id])
  guestId   String
  guest     GuestAccess @relation(fields: [guestId], references: [id])
  createdAt DateTime    @default(now())
}
```

2.4 Configuration

The application is configured using environment variables. These variables are essential for both development and production environments. It is recommended to create a `.env.example` file in the project root to serve as a template.

- `DATABASE_URL`
  The connection string for the PostgreSQL database, used by Prisma.
  Example: `postgresql://user:password@localhost:5432/giftlist`
- `JWT_SECRET`
  A long, random, and secret string for signing and verifying JSON Web Tokens. This must be kept private.
- `PORT`
  The port for the Node.js server. Defaults to `3000` if not specified.
- `CORS_ORIGIN`
  The URL of the frontend application allowed to access the API, to enforce Cross-Origin Resource Sharing (CORS) policies.
- `SMTP_HOST`, `SMTP_PORT`, `SMTP_USER`, `SMTP_PASS`
  Credentials for the SMTP server (used by Nodemailer) to send email notifications.

2.4.1 Authentication Strategy: Access & Refresh Tokens

To balance security and usability, the application implements a dual-token system:

- **Access Token (JWT)**: Short-lived (15 minutes). Sent in the `Authorization` header.
- **Refresh Token**: Long-lived (7 days). Stored in a `HttpOnly`, `Secure` cookie.

When the Access Token expires, the frontend automatically calls a `POST /auth/refresh` endpoint, which verifies the Refresh Token and issues a new Access Token. This ensures that the Celebrant stays logged in while they are actively using the dashboard.

2.5 Data Model

The database schema is designed to support the relationships between Celebrants, Lists, and Guests.

- User (Celebrant): Stores celebrant data.
  - `id`: UUID, Primary Key
  - `email`: String, unique, required
  - `password`: String, hashed, required
  - `resetToken`: String, optional, unique (for password recovery)
  - `resetTokenExpires`: Timestamp, optional
  - `createdAt`: Timestamp, default: now
  - `updatedAt`: Timestamp, auto-update
- GiftList: Represents a specific event's wish list.
  - `id`: UUID, Primary Key
  - `userId`: Foreign Key to User
  - `slug`: String, unique. Automatically generated from the name.
  - `name`: String, required
  - `deletedAt`: Timestamp, optional (Soft delete)
  - `createdAt`: Timestamp, default: now
  - `updatedAt`: Timestamp, auto-update
- GiftItem: Represents an individual gift.
  - `id`: UUID, Primary Key
  - `listId`: Foreign Key to GiftList
  - `name`: String, required
  - `description`: Text, optional
  - `url`: String, optional
  - `status`: Enum (`AVAILABLE`, `CLAIMED`), required, default: `AVAILABLE`
  - `preference`: Enum (`LOW`, `MEDIUM`, `HIGH`), required, default: `MEDIUM`
  - `deletedAt`: Timestamp, optional (Soft delete)
  - `createdAt`: Timestamp, default: now

  - `updatedAt`: Timestamp, auto-update

- GuestAccess: Links a guest's email to a specific list for notification purposes.
  - `id`: UUID, Primary Key
  - `listId`: Foreign Key to GiftList
  - `email`: String, required
  - `language`: String (e.g., 'en', 'it'), required. Used for notification emails.
  - `createdAt`: Timestamp, default: now
  - A unique constraint should be on `(listId, email)`.
- GuestClaim: A record linking a `GiftItem` to a `GuestAccess` record.
  - `itemId`: Foreign Key to GiftItem, Primary Key (ensures an item is claimed only once)
  - `guestId`: Foreign Key to GuestAccess
  - `createdAt`: Timestamp, default: now

2.5.1 Deletion Strategy and Data Retention

The application balances the need for data recovery with privacy and resource optimization through a two-tier deletion strategy.

Soft Deletes

To ensure data integrity and preserve history during active usage, the application adopts a soft-delete strategy for records manually removed by users (Celebrants). Instead of physically removing records, a `deletedAt` timestamp is set. API queries exclude these records by default.

Automatic Expiration (Hard Delete)

To prevent the database from growing indefinitely and to comply with data protection principles, an automatic cleanup process is implemented:

- **Inactive Lists**: Any `GiftList` that has not been updated for more than 6 months (based on `updatedAt`) is considered expired. These lists, along with all their associated `GiftItems`, `GuestAccess` records, and `GuestClaims`, are permanently removed from the database.
- **Guest Data**: When a list is deleted (manually or automatically), its associated `GuestAccess` records are removed. The system periodically identifies guest emails

that are no longer associated with any active or soft-deleted list and purges them to ensure no unnecessary personal identifiable information (PII) is retained.

A scheduled background task (cron job) will execute these cleanup operations daily during low-traffic hours.

2.5.2 Data Validation Rules

To ensure data integrity and security, the following validation rules will be enforced at the API level.

Shared Schemas

To guarantee a Single Source of Truth (SSOT) between the Frontend and Backend, validation schemas are defined using **Zod** in a shared workspace package (`packages/shared`). The backend imports these schemas directly to validate incoming request bodies, ensuring that client-side validation logic matches server-side rules exactly.

Entity Rules

- User (Celebrant)
  - `email`: Must be a syntactically valid email address.
  - `password`: Must be at least 8 characters long and contain at least one uppercase letter and at least one number or special character.
- GiftList
  - `name`: Must be between 3 and 50 characters long.
  - `slug`: Must only contain lowercase letters, numbers, and hyphens (`-`).
- GiftItem
  - `name`: Must be between 3 and 50 characters long.
  - `url`: If provided, must be a valid URL format.
  - `description`: Optional. If provided, it must not exceed 200 characters.
- GuestAccess
  - `email`: Must be a syntactically valid email address.

2.5.3 Slug Generation and Collision Strategy

Slugs are URL-friendly strings generated from the list name.

- **Global Uniqueness**: Since lists are accessed via `/api/v1/lists/:slug`, every slug must be unique across the entire system.

- **Generation**: The system uses a standard slugify algorithm (removing special characters and replacing spaces with hyphens).
- **Collision Handling**: If a generated slug already exists, a 5-character random alphanumeric suffix is appended (e.g., `birthday-party` becomes `birthday-party-x7r2e`).

2.6 API Design

The backend will expose a RESTful API versioned under the `/api/v1` prefix.

2.6.1 Architectural Choice

The decision to use a RESTful architecture over alternatives like gRPC or GraphQL is based on several key factors:

- Simplicity
  REST is the native standard of the web. It allows for straightforward integration with any browser-based client without the need for specialized libraries or complex client-side configurations.
- Low Friction for Guests
  Since the application relies on sharing public links, REST ensures that the initial list retrieval is fast and compatible with all web environments.
- Productivity
  RESTful APIs are easy to document, test, and debug using standard tools (e.g., browser dev tools, curl). Given the linear nature of our data (Lists and Items), the overhead of GraphQL's schema management or gRPC's binary serialization (Protobuf) would not provide significant benefits for this specific use case.

2.6.2 Error Handling

The API uses standard HTTP status codes to indicate the success or failure of a request. In case of an error, the response body will contain a JSON object with a consistent structure:

```
{
  "error": {
    "code": "ERROR_CODE_STRING",
    "message": "A human-readable description of the error."
  }
}
```

Detailed Error Codes

The following specific error codes are used to provide granular feedback to the client:

- `AUTH_INVALID_CREDENTIALS`: The email or password provided is incorrect.
- `AUTH_EMAIL_ALREADY_EXISTS`: Attempted to register with an email that is already in use.
- `AUTH_TOKEN_EXPIRED`: The provided JWT or reset token has expired.
- `LIST_NOT_FOUND`: The requested list slug does not exist or has been deleted.
- `ITEM_NOT_FOUND`: The requested item ID does not exist in the specified list.
- `ITEM_ALREADY_CLAIMED`: Attempted to claim an item that is no longer `AVAILABLE`.
- `ITEM_NOT_CLAIMED_BY_YOU`: Attempted to unclaim an item that was claimed by a different guest.
- `VALIDATION_ERROR`: The request payload failed schema validation (e.g., missing fields, invalid formats).

Common Status Codes

- `400 Bad Request`: The request was malformed or failed validation.
- `401 Unauthorized`: Authentication is required and has failed.
- `403 Forbidden`: Permission denied for the requested action.
- `404 Not Found`: The resource could not be found.
- `409 Conflict`: Conflict with the current state (e.g., item already claimed).
- `500 Internal Server Error`: Unexpected server error.

2.6.3 Authentication and Public Routes

These routes handle celebrant authentication and guest operations. All routes are prefixed with `/api/v1`.

- `POST /auth/register`
  Registers a new Celebrant account.
- `POST /auth/login`
  Authenticates a Celebrant and returns a JWT.
- `POST /auth/refresh`
  Exchanges a valid Refresh Token (from cookie) for a new Access Token.

- `POST /auth/forgot-password`
  Initiates the password reset process. Sends an email with a reset token if the email exists.
- `POST /auth/reset-password`
  Sets a new password using a valid reset token.
- `POST /lists/:slug/access`
  Registers or identifies a guest's email for a specific list. If the email already exists for this list, it refreshes the guest session. **Payload**: `{ "email": "guest@example.com", "language": "it" }`. The backend stores the email and sets a secure, `HttpOnly` session cookie.
- `GET /lists/:slug`
  Retrieves the gift list and all items. Requires a valid guest session. **Note:** For guests, this response includes the `status` of each item to show what is available.
- `GET /lists/:slug/my-claims`
  Retrieves only the items claimed by the current guest.
- `POST /items/:id/claim`
  Allows a guest to claim an item.
- `POST /items/:id/unclaim`
  Allows a guest to unclaim an item they previously claimed.

2.6.4 Protected Routes (Celebrant Dashboard)

These require a valid JWT in the Authorization header.

- `GET /lists`
  Retrieves all lists created by the authenticated Celebrant.
- `POST /lists`
  Creates a new list. Requires a `name`.
- `GET /lists/:slug/manage`
  Retrieves the full details of a specific list for management. **Surprise Protection:** This route *explicitly excludes* any information regarding the `status` (CLAIMED/AVAILABLE) or guest claims.
- `DELETE /lists/:id`
  Soft-removes an entire list. If items were claimed, notifications are triggered.
- `POST /lists/:id/item`
  Adds an item to a list.
- `PATCH /items/:id`
  Updates an item's details.

- `DELETE /items/:id`
  Soft-removes an item. Triggers a notification if the item was claimed.

2.6.5 Slug Generation and Collision Strategy

Slugs are URL-friendly strings generated from the list name. To handle cases where multiple users choose the same name:

1. The system generates the base slug.
2. It checks for existence in the database.
3. If a collision occurs, a short random alphanumeric suffix (e.g., `-x4k2`) is appended.

2.7 API Specification (DTOs)

To ensure consistency between frontend and backend, we define the following Data Transfer Objects.

2.7.1 Authentication

- **Login Request**

```
{
  "email": "user@example.com",
  "password": "SecurePassword123!"
}
```

- **Login Response (200 OK)**

```
{
  "token": "eyJhbG...",
  "user": {
    "id": "uuid-v4",
    "email": "user@example.com"
  }
}
```

2.7.2 Gift Lists

- **Get List Response (200 OK)**

```
{
  "id": "uuid-v4",
  "name": "Birthday 2026",
  "slug": "birthday-2026",
```

```
  "items": [
    {
      "id": "uuid-v4",
      "name": "Mechanical Keyboard",
      "description": "Blue switches preferred",
      "url": "https://amazon.com/...",
      "status": "AVAILABLE",
      "preference": "HIGH",
      "isClaimedByMe": false
    }
  ]
}
```

- **Add Item Request**

```
{
  "name": "Lego Star Wars",
  "description": "The Millennium Falcon set",
  "url": "https://lego.com/...",
  "preference": "MEDIUM"
}
```

- **Update Item Request (PATCH)**

```
{
  "name": "Updated Lego Set",
  "description": "Updated description",
  "url": "https://new-url.com",
  "preference": "HIGH"
}
```

- **Guest Access Request**

```
{
  "email": "guest@example.com",
  "language": "it"
}
```

2.7.3 Error Response Example

```
{
  "error": {
    "code": "ITEM_ALREADY_CLAIMED",
    "message": "This item has already been claimed by another guest."
  }
```

```
}
```

2.8 Security

2.8.1 Rate Limiting

To protect the application from brute-force attacks and DDoS, the following rate limits are applied:

- **Auth Routes**: 5 requests per 15 minutes per IP.
- **Public Guest Access**: 20 requests per minute per IP.
- **General API**: 100 requests per minute per IP.

2.8.2 CORS Policy

The API will strictly enforce CORS, allowing requests only from the verified frontend domain.

2.9 Handling Concurrency

To prevent the race condition when two guests from claiming the same item simultaneously, the backend will utilize database transactions. When a claim request is received:

1. Start Transaction.
2. Lock the specific Item row (e.g., `SELECT ... FOR UPDATE`).
3. Verify the status is `AVAILABLE`.
4. Insert a `GuestClaim` record and update Item status.
5. Commit Transaction.

2.10 Notification Service

The notification service is designed to be resilient and non-blocking.

- **Asynchronous Processing**: Notifications are handled asynchronously to prevent API latency. If an item is deleted, the system schedules the email task and immediately returns a success response to the user.
- **Reliability**: In case of SMTP failure, the system will attempt to retry the delivery up to 3 times with exponential backoff.
- **Privacy**: Guest emails are only used for required system notifications and are never shared or used for marketing.

If a Celebrant deletes an item that is already `CLAIMED`:

1. The backend identifies the associated `GuestClaim`.
2. The system schedules an email notification via the notification service.
3. The item is then soft-deleted (`deletedAt` is set).

2.11 Email Templates

To ensure a consistent and professional communication with users, the following email templates are defined. All emails will be sent in the user's preferred language (English or Italian).

2.11.1 Item Removal Notification (Guest)

Sent when a celebrant removes an item that the guest has already claimed.

- **Subject (EN)**: Important update regarding your gift for [List Name]
- **Subject (IT)**: Aggiornamento importante sul tuo regalo per [List Name]
- **Body Content**:

  Hi,
  We are writing to inform you that the item "[Item Name]" has been removed from the gift list "[List Name]" by the celebrant.
  If you have already purchased this gift, please be aware that it is no longer on their wish list.
  You can visit the list again to see if there are other items you would like to claim: [List URL].

2.11.2 Password Reset (Celebrant)

Sent when a celebrant requests a password recovery.

- **Subject (EN)**: Password Reset Request for Gift List
- **Subject (IT)**: Richiesta di reset password per Gift List
- **Body Content**:

  Hi,
  We received a request to reset the password for your account. Click the link below to set a new password. This link will expire in 15 minutes.

  ResetLinkwithToken

  If you did not request this, you can safely ignore this email.

3 Frontend

This chapter details the architectural decisions and technical specifications for the client-side application of the "Gift List" project. The frontend serves as the primary interface for both Celebrants (to manage lists) and Guests (to view and claim gifts).

3.1 Technology Stack

The frontend will be built as a Single Page Application (SPA) using the following core technologies:

- Framework
  React (v19 or latest stable).
- Language
  TypeScript. To ensure type safety and align with the backend data models.
- Build Tool
  Vite. Chosen for its superior performance and rapid development cycle.
- Routing
  React Router (v7).
- API Client
  Axios. Used for its robust interceptor support, allowing centralized handling of JWT authentication and error responses.
- Form Management & Validation
  React Hook Form combined with Zod for schema-based validation.
- Iconography
  Lucide React. A clean, consistent icon set with a small bundle footprint.
- State Management
  - *Server State:* TanStack Query (React Query). To handle data fetching, caching, and synchronization with the backend.
  - *Client State:* React Context API for global UI state (e.g., theme, user session).
- Styling
  CSS Modules with Vanilla CSS.
- Internationalization
  `react-i18next`. To manage translations and user language preferences.

3.2 Form & Validation Strategy

To ensure a high-quality user experience and data integrity, form management follows a strict pattern:

- Schema Sharing
  Validation schemas (Zod) are defined in a shared package at `packages/shared/validations` to be used by both the frontend forms and backend API controllers, ensuring a single source of truth.
- Error Feedback
  Validation errors are displayed in real-time as the user types, providing immediate corrective feedback.
- Submission Lifecycle
  Submit buttons transition to a "Loading" state during API calls, disabling further clicks to prevent duplicate submissions.

3.3 API Interaction

Axios is configured as a singleton client with the following settings:

- **Base URL**: Configured with the `/api/v1` prefix to ensure compatibility with versioned backend routes.
- `withCredentials: true`
  This is mandatory to allow the browser to send and receive the `guest_session` cookie (for Guests) and the `refresh_token` cookie (for Celebrants).
- Authentication Header
  An interceptor automatically attaches the `Authorization: Bearer <token>` header to all requests if a JWT is present.
- **Token Refresh Interceptor**
  A specialized interceptor catches `401 Unauthorized` errors. If the error is due to an expired Access Token, the client attempts a one-time refresh via the `POST /auth/refresh` endpoint. If successful, the original request is retried with the new token; otherwise, the user is logged out.
- **Global Error Interceptor**
  A response interceptor catches other specific status codes to trigger global notifications (Toasts).
  - **Guest Session Handling**: If a Guest attempts to access a protected list route (e.g., `GET /lists/:slug`) and receives a `401 Unauthorized` or `403 Forbidden` error (indicating an invalid or missing session cookie), the interceptor au-

tomatically redirects the user to the access page (`/lists/:slug/access`) to re-enter their email.

3.4 Internationalization (i18n)

The application supports English and Italian using `react-i18next`.

- **Detection**: The language is detected via the browser settings (`i18next-browser-languagedete`
- **Structure**: Translation strings are stored in JSON files within `/public/locales/en|it/common.`
- **Guest Preference**: When a Guest provides their email to access a list, their current language setting is sent to the backend to ensure notification emails are sent in the correct language.

3.5 Browser Compatibility

To ensure a consistent and secure user experience, the application restricts access based on browser capabilities.

3.5.1 Unsupported Browsers Policy

The frontend application includes a strict check for browser compatibility during the initialization phase.

- Detection Mechanism
  The application detects the user's browser version and capabilities (e.g., support for ES6+ features, CSS Grid, Flexbox) before rendering the main application logic.
- Handling Unsupported Environments
  If an unsupported browser or a legacy version is detected (e.g., Internet Explorer), the application will:
  1. Halt Execution: React will not mount the root component to prevent runtime errors or visual artifacts.
  2. Display Error Message: A full-screen, user-friendly error message will be shown, informing the user that their browser is not supported and recommending an update to a modern browser (Chrome, Firefox, Safari, Edge).

3.6 Architectural Decision Records

A key requirement for this project is the future expansion into a native mobile application. This requirement significantly influenced the choice of the frontend framework.

3.6.1 Why React?

React was selected as the optimal choice for a "Web-First, Mobile-Second" strategy.

- Web Performance
  The primary use case involves Guests accessing lists via shared links. React, combined with Vite, offers a lightweight bundle and fast "Time-to-Interactive", which is crucial for reducing bounce rates on mobile browsers.
- Code Reusability
  By using TypeScript, we can share data types and validation logic directly between the Backend (Node.js) and Frontend. Furthermore, the business logic encapsulated in Custom Hooks (e.g., `useClaimGift`) can be reused almost entirely in a future React Native application.
- Ecosystem
  The React ecosystem is vast, providing robust solutions for every requirement (forms, animations, accessibility) without the need to reinvent the wheel.

3.6.2 Comparison with Alternatives

We evaluated two major alternatives before selecting React.

React vs. Flutter (Dart)

Flutter is a powerful toolkit for building natively compiled applications, but it was deemed less suitable for this project:

- Web "Heaviness"
  Flutter for Web renders the UI on a canvas, resulting in a large initial download size. For a Guest who simply needs to view a list and click a button, this overhead is unjustified.
- Language Barrier
  Choosing Flutter would prevent the sharing of DTOs and validation logic (TypeScript) between backend and frontend.

React vs. Angular

Angular is a comprehensive enterprise framework, but React was preferred for:

- Flexibility vs. Rigidity
  For a relatively simple application, Angular's complexity introduces unnecessary boilerplate.
- Mobile Path
  React Native offers a significantly better performance and "native feel" than Angular's primary mobile path (Ionic/WebView).

3.7 Project Structure

The project will follow a Monorepo structure (using pnpm workspaces) to facilitate code sharing and consistent tooling.

```
/
  /apps
    /backend      # Node.js Express API
    /frontend     # React application
  /packages
    /shared       # Shared Zod schemas, TypeScript types, and utilities
  /docker         # Docker configuration files
  docker-compose.yml
```

3.7.1 Frontend Internal Structure

Within `apps/frontend/src`, the structure follows a feature-based organization:

```
/src
  /assets         # Static files (images, fonts)
  /components     # Shared UI components (Button, Input, Card)
  /features       # Domain-specific logic
    /auth         # Login, Register, Password Reset forms
    /gift-list    # List view, Item card, Claim logic
    /dashboard    # Celebrant admin panel
  /hooks          # Shared custom hooks (useAuth, useTheme)
  /lib            # Configuration (API client, utils)
  App.tsx         # Main entry point with Routing
```

3.8 Future Mobile Strategy

To prepare for the future mobile application, the frontend code will strictly separate **Business Logic** from **UI Components**.

- Logic (Hooks)
  All API calls, state transformations, and validation rules will be encapsulated in Custom Hooks. These are UI-agnostic and can be reused in React Native.
- UI (Components)
  Visual components will be the only part requiring a rewrite (replacing HTML with React Native primitives).

3.9 Implementation Details

This section outlines the specific implementation strategies for key frontend components, routing, and user feedback mechanisms.

3.9.1 Routing Strategy

The application uses React Router (v7) to manage navigation. The routes are divided into Public (Guest) and Protected (Celebrant) areas.

- Public Routes
  - `/`: Landing page with Login/Register options for Celebrants.
  - `/auth/forgot-password`: Form to request a password reset.
  - `/auth/reset-password`: Form to set a new password using the token from the email.
  - `/lists/:slug`: The main view for a Guest to see the gift list.
  - `/lists/:slug/access`: A simple form to capture the Guest's email.
  - `/lists/:slug/my-claims`: A filtered view showing only the items claimed by the current Guest.
- Protected Routes (Celebrant Dashboard)
  Requires a valid JWT. Unauthenticated access redirects to `/`.
  - `/dashboard`: Overview of all created lists.
  - `/dashboard/new`: Form to create a new list.
  - `/dashboard/:slug`: Management view for a specific list.

3.9.2 Component Architecture

Key components will encapsulate specific business logic states to ensure a consistent user experience.

Error Boundaries

The application will implement React Error Boundaries at the feature level. This ensures that a failure in a specific component (e.g., a broken GiftCard) does not crash the entire application, allowing the user to continue using other parts of the system or simply refresh the affected section.

GiftCard Component

This is the core component for displaying a gift item. It has distinct visual states based on the item's status and the user's role.

- Available State
  Displays the item details and a prominent "Claim" button.
- Claimed by Current User
  Visually distinct. Displays an "Unclaim" button.
- Claimed by Other
  The item appears "disabled" or transparent.

ClaimButton Component

Handles the interaction with the backend API.

- Optimistic UI
  When clicked, the button immediately reflects the new state before the API response is received.
- Error Handling
  If the API call fails, the button reverts to its original state and triggers a global notification.

3.9.3 Global Styling Strategy

To maintain consistency, a set of CSS variables will be defined at the `:root` level.

- Colors (The "Elegant Indigo" Palette)
  - `-color-primary`: `#6366f1`, `-color-secondary`: `#f43f5e`, `-color-bg`: `#f8fafc`.
- Spacing
  Standardized scale: 4px, 8px, 16px, 24px, 32px.
- Typography
  `'Inter', system-ui, sans-serif`.

3.9.4 Feedback Mechanisms

The application will use the following mechanisms to ensure a smooth and informative user experience:

- **Toasts**: Non-intrusive notifications for success (e.g., "Item claimed!") and errors.
- **Skeleton Loading**: Used during initial data fetching to prevent layout shifts and provide a sense of progress.

- **Empty States**: Custom informative messages or "Call-to-Action" (CTA) buttons when a view has no data.
  - *Dashboard*: If a Celebrant has no lists, a prominent "Create your first list" CTA is displayed.
  - *Gift List*: If a list has no items, Guests see a friendly message indicating that the celebrant is still working on it.
  - *My Claims*: If a Guest has not claimed anything, they are encouraged to browse the list again.

3.10 Testing Strategy

To ensure reliability, a multi-layered testing approach is adopted.

- **Unit Testing (Vitest + React Testing Library)**
  Used for testing individual components and utility functions.
- **Integration Testing (Vitest)**
  Focuses on the interaction between multiple components and hooks (TanStack Query).
- **End-to-End (E2E) Testing (Playwright)**
  Simulates real user workflows for both Celebrants and Guests.

3.11 Guest Session Management

Since guests do not have a traditional account, their identity is tracked via a secure session mechanism.

- **Cookie-Based Identity**
  Upon providing an email, the backend issues a `guest_session` cookie (`HttpOnly`, `Secure`, `SameSite=Lax`).
- **Persistence**
  The session is tied to the browser. If a guest returns to the same list from the same browser, they will automatically see their claimed items highlighted. If the cookie is expired, the guest simply needs to re-enter their email on the access page to restore their session.

4 Deployment

This chapter provides an implementation-ready guide for deploying the "Gift List" application in a self-hosted environment using a Raspberry Pi 4 (4GB RAM). The strategy focuses on high reliability, security, and automated maintenance.

4.1 Infrastructure Overview

The application is deployed as a suite of Docker containers managed by Docker Compose. This ensures environment consistency and simplifies updates.

- **Host OS**: Raspberry Pi OS (64-bit) or any Debian-based distribution.
- **Runtime**: Docker Engine + Docker Compose V2.
- **Storage**: Persistent data is stored in Docker Volumes, which are mapped to the host's filesystem for easier backups.

4.2 Containerization Strategy

The system consists of several specialized services orchestrated via `docker-compose.yml`.

4.2.1 Service Definition

1. **Frontend**: A production-ready Nginx container serving the React static files.
2. **API (Backend)**: The Node.js Express application.
3. **Database (PostgreSQL)**: The primary data store.
4. **Cloudflare Tunnel**: Handles secure external access without port forwarding.
5. **Backup Agent**: A lightweight container that performs scheduled database dumps.

4.2.2 Database Migrations

To ensure the database schema is up-to-date before the application starts, the API service is configured to run migrations on startup.

- **Entrypoint Command**: The Docker definition for the `api` service overrides the default command to execute `npx prisma migrate deploy` before starting the server. This guarantees that all pending migrations are applied to the production database automatically.

4.2.3 Nginx Configuration for SPA

Since the Frontend is a Single Page Application (SPA) using React Router, the Nginx server must be configured to handle client-side routing correctly.

- **Fallback Strategy**: The Nginx configuration file (`nginx.conf`) must include a `try_files` directive. This ensures that any request for a non-existent file (e.g., `/lists/my-list`) is internally redirected to `index.html`, allowing React Router to handle the URL.

```
location / {
    root   /usr/share/nginx/html;
    index  index.html index.htm;
    try_files $uri $uri/ /index.html;
}
```

4.2.4 Docker Compose Configuration (Implementation Ready)

```
services:
  frontend:
    build:
      context: ./apps/frontend
      dockerfile: Dockerfile
    restart: always
    environment:
      - VITE_API_BASE_URL=https://giftlist.yourdomain.com/api/v1
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"

  api:
    build: ./apps/backend
    restart: always
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/giftlist
      - PORT=3000
      - JWT_SECRET=${JWT_SECRET}
      - CORS_ORIGIN=https://giftlist.yourdomain.com
      - SMTP_HOST=${SMTP_HOST}
      - SMTP_PORT=${SMTP_PORT}
```

```
    - SMTP_USER=${SMTP_USER}
    - SMTP_PASS=${SMTP_PASS}
  depends_on:
    - db
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"

db:
  image: postgres:17-alpine
  restart: always
  volumes:
    - pgdata:/var/lib/postgresql/data
  environment:
    - POSTGRES_DB=giftlist
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=pass
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"

tunnel:
  image: cloudflare/cloudflared:latest
  restart: always
  command: tunnel run --token ${CLOUDFLARE_TOKEN}
  depends_on:
    - frontend
    - api

db-backup:
  image: prodrigestivill/postgres-backup-local
  restart: always
  volumes:
    - pgdata:/var/lib/postgresql/data:ro
    - ./backups:/backups
  environment:
```

```
      - POSTGRES_HOST=db
      - SCHEDULE=@daily
      - BACKUP_KEEP_DAYS=7

volumes:
  pgdata:
```

4.3 Networking and Security

To expose the application securely without opening ports on the home router, we utilize **Cloudflare Tunnels**.

4.3.1 Cloudflare Tunnel Integration

A Cloudflare Tunnel (`cloudflared`) creates a secure, encrypted connection between the Raspberry Pi and the Cloudflare edge.

- **Traffic Routing**: The tunnel is configured via the Cloudflare Dashboard to route traffic:
  - `giftlist.yourdomain.com` → `http://frontend:80`
  - `giftlist.yourdomain.com/api` → `http://api:3000/api`
- **No Port Forwarding**: The router remains closed to incoming traffic.
- **Automatic HTTPS**: Cloudflare manages SSL/TLS certificates automatically.

4.3.2 Security Hardening

- **Internal Networking**: Only the Frontend and API services are indirectly exposed via the tunnel. The Database is only accessible within the internal Docker network.
- **Log Rotation**: Configured globally in the Compose file to prevent the SD card from filling up (max 3 files of 10MB each per service).
- **Environment Secrets**: Sensitive data (DB passwords, Cloudflare tokens) are managed via a `.env` file.

4.4 Reliability and Backup Strategy

Reliability is achieved through automated recovery and off-site data preservation.

4.4.1 Auto-Restart

Docker's `restart: always` policy ensures that if a service crashes or the Raspberry Pi reboots, the entire stack restarts automatically.

4.4.2 Automated Backups

The `db-backup` service performs a daily compressed dump at 00:00. The host machine is configured with a cron job to synchronize the `./backups` folder to an off-site location (e.g., encrypted S3 bucket).

4.5 Architectural Decision Records

4.5.1 Docker Compose vs. Kubernetes (K3s)

Docker Compose was selected for its minimal resource footprint (approx. 50MB RAM overhead), ensuring that the majority of the Raspberry Pi's 4GB RAM is available for PostgreSQL and the Node.js API, unlike K3s which requires significant resources for its control plane.

4.6 Monitoring and Logs

- **Docker Logs**: `docker compose logs -f [service_name]` for real-time debugging.
- **Health Checks**: A simple heartbeat monitoring service (e.g., UptimeRobot) targets the public URL.