ADD

להשביע את הלב – Feeding The Needing

Eden Nahum ***Idan Asis***

Daniel Bril ***Shavit Mor***

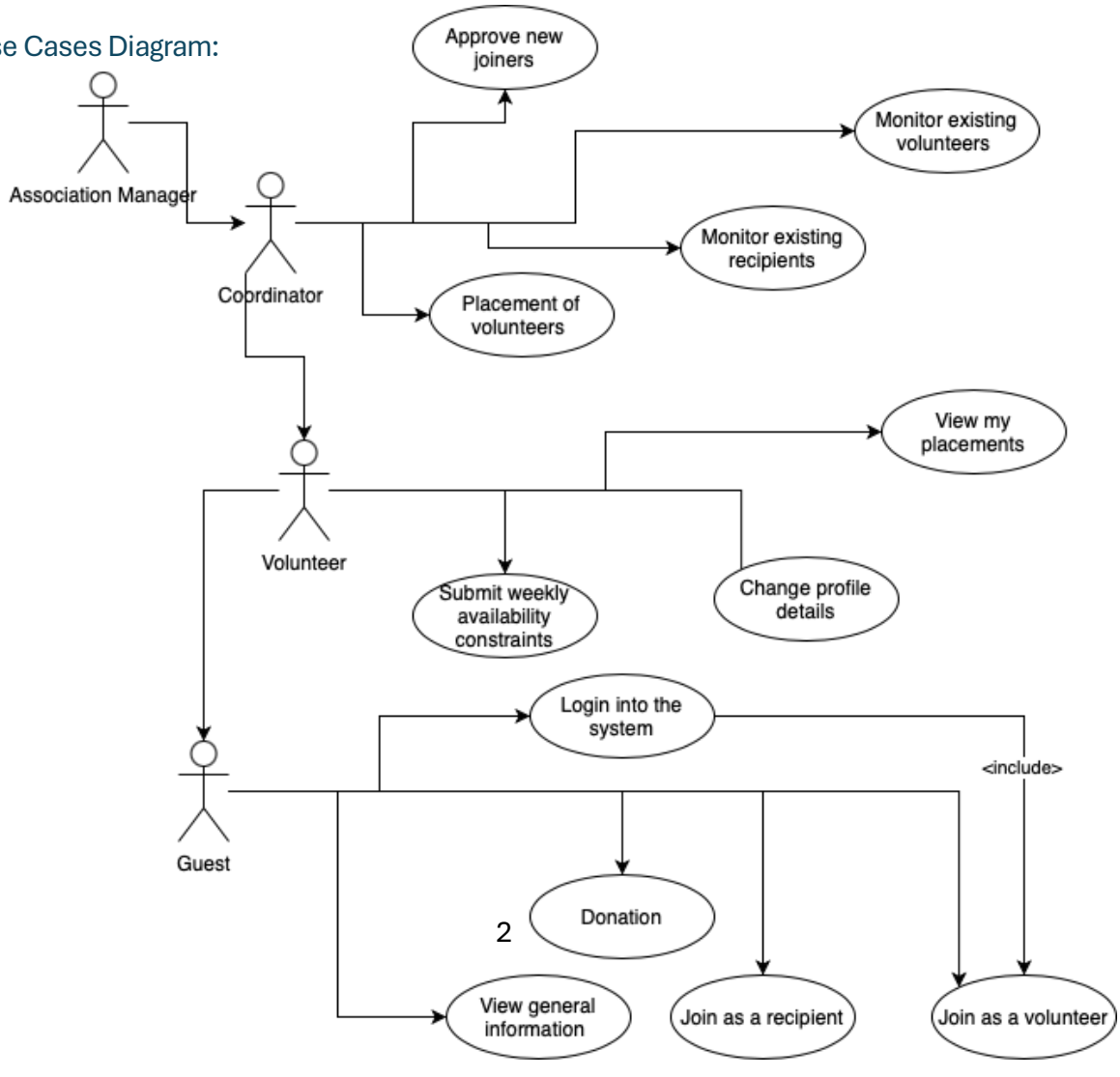
Table Of Contents

Chapter 1 – Use Cases

User Profiles – Roles:

- **Association Manager:**
  This user has access to all system functionalities and can modify any value within the system.
- **Coordinator:**
  There are three types of coordinators – Social Coordinator, Cook Coordinator, and Driver Coordinator. These users manage all logistics for the association and are responsible for approving and assigning various volunteers.
- **Volunteer:**
  A user who has been approved as a new volunteer by the association manager or one of the coordinators. They can contribute to the association as a cook or driver. This user submits their availability constraints each week and can, of course, view their assignments.
- **Guest User:**
  A user who exists in the system but has not yet logged in. This user has the ability to view general information about the association, join the association if they wish, and make a monetary donation.

Use Cases Diagram:



1. Join as a Volunteer

Primary Actors: Guest User

Description:
This use case allows guest users to join the association as a new volunteer (cook, driver, or both).

Preconditions:
The system is up and running as required.

Postconditions:
The user's membership request is sent to the system and awaits approval.

Process:

- A guest user logs into the system and opens the registration form.
- They fill in their personal details in the form, including email, full name, phone number, address, password, and confirmation that they have no criminal record, then submit the request.

Alternative Scenario:
An error occurs while submitting the registration form, preventing the request from being approved.

2. Join as a Recipient

Primary Actors: Guest User

Description:
This use case allows guest users to request assistance from the association as a recipient.

Preconditions:
The system is up and running as required.

Postconditions:
The user's assistance request is sent to the system and awaits approval.

Process:

- A guest user logs into the system and opens the assistance request form.
- They fill in their personal details in the form, including full name, phone number, address, number of household members, and additional notes (if any), then submit the request.

Alternative Scenario:
An error occurs while submitting the assistance request form, preventing the request from being approved.

3. View General Information

Primary Actors: All Existing Users

Description:
This use case allows various users within the system to view general information about the association and get acquainted with its activities.

Preconditions:
The system is up and running as required.

Postconditions:
None.

Process:
A user logs into the system and views all information about the association.

Alternative Scenario:
The system fails to load or does not operate as required, preventing the user from viewing the information.

4. Donation

Primary Actors: All Existing Users

Description:
This use case allows various users within the system to navigate to the donations page and make monetary contributions to the association conveniently.

Preconditions:
The system is up and running as required.

Postconditions:
The donation is transferred to the association through the selected applications (e.g., Bit, PayBox, etc.).

Process:

- A user logs into the system and navigates to the donations page.
- The user selects their preferred method of donation.

- The user makes the monetary contribution using their chosen method.
- The payment is transferred to the association through the selected application.

Alternative Scenarios:

- An error occurs while opening the donations page.
- After opening the donations page, the information about the various applications through which the user can donate to the association is unavailable.

5. Login into the System

Primary Actors: Guest User

Description:
This use case allows a guest user who has already been approved in the system as a volunteer, coordinator, or association manager to log into the system and access all its functionalities.

Preconditions:

- The system is up and running as required.
- The user attempting to log in has already been approved by the coordinators or the association manager.

Postconditions:
The user is logged into the system.

Process:

1. The user accesses the system and navigates to the login screen.
2. The user enters their login details—phone number and password—and clicks the login button.
3. The user is successfully logged into the system.

Alternative Scenarios:

- The user enters incorrect login details and receives an error message.
- The user has not yet been approved and receives an error message when attempting to log in.

6. Change Profile Details

Primary Actors: Volunteer, Coordinator, Association Manager

Description:
This use case allows various users within the system to edit their personal details, including name, email, phone number, and address.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a volunteer, coordinator, or association manager.

Postconditions:
The user's updated personal details are reflected in the system.

Process:

1. The user is logged into the system as a volunteer, coordinator, or association manager.
2. The user navigates to the personal details editing screen.
3. The user edits their personal information and clicks the confirmation button.
4. The new personal details are updated in the system.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The new personal details entered exceed the system's constraints (e.g., invalid phone number), resulting in an appropriate error message being displayed to the user.

7. Submit Weekly Availability Constraints

Primary Actors: Volunteer, Coordinator, Association Manager

Description:
This use case allows various users within the system to submit their weekly availability constraints related to cooking and driving for the upcoming week.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a volunteer, coordinator, or association manager.
- For cooking constrains only: Recipient requests for the current week must already exist in the system - constraints are based on actual recipient requests that have already been submitted. This ensures that cooking

volunteers select specific meals that match real demands and avoid unnecessary or mismatched preparation.

- For driving constraints: Volunteers can submit their availability at any time, regardless of recipient requests.

Postconditions:
The user's new constraints are entered into the system and successfully updated.

Process:

1. The user logs into the system as a volunteer, coordinator, or association manager.
2. The user navigates to the Driving Assistance screen to submit driving constraints or to the Cooking Assistance screen to submit cooking constraints.
3. The user submits their various constraints.
4. The new constraints are successfully updated in the system.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The new constraints entered exceed the system's conditions, resulting in an appropriate error message being displayed to the user.
- **For cooks:** If no recipient requests are available, the system disables the cooking constraint submission.
- Constraints are rejected if they violate system conditions, such as time overlaps with existing assignments, invalid time ranges, or submission past deadline.

8. View My Assignments

Primary Actors: Volunteer, Coordinator, Association Manager

Description:
This use case allows various users within the system to view their assignments for both driving and cooking for the upcoming week.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a volunteer, coordinator, or association manager.
- The user has already submitted their various constraints (otherwise, they will see an empty assignments screen).

Postconditions:
The user's assignments are displayed in an organized and easily understandable manner.

Process:

1. The user logs into the system as a volunteer, coordinator, or association manager.
2. The user navigates to the "My Assignments" screen.
3. The user views their various assignments that match the constraints they submitted.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The user views their assignments, but they do not match their constraints, prompting them to contact the association manager for clarification.
- The page fails to load properly, preventing the user from viewing their assignments.

9. Placement Of Volunteers

Primary Actors: Coordinator, Association Manager

Description:
This use case allows coordinators and the association manager to view the available constraints of various volunteers and assign them according to the association's needs based on their assignments. Alternatively, they can send updates to volunteers if their assistance is required at different times.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a coordinator or association manager.
- Volunteers have already submitted their various constraints for the current week in the system.
- Requests from the current week's recipients have been recorded in the system.

Postconditions:
The assignments are sent to the driving volunteers and cooking volunteers.

Process:

1. The user logs into the system as a coordinator or association manager.
2. The user navigates to the Driver Management screen for assigning drivers or to the Cook Management screen for assigning cooks.
3. The user assigns an appropriate cook and driver to each recipient's request to prepare and deliver the meals to their destinations.
4. The assignments are updated in the system and displayed to the assigned volunteers.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The user is unable to perform a basic assignment due to missing information or lack of available assignments, resulting in appropriate notifications being sent to the relevant users.
- The page fails to load properly, preventing the user from making the desired assignments.

10. Approve New Joiners

Primary Actors: Coordinator, Association Manager

Description:
This use case allows coordinators and the association manager to review new volunteers and recipients, and either approve or reject their membership requests to join the association.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a coordinator or association manager.
- There is at least one volunteer or recipient who has submitted a membership request to the association.

Postconditions:
The various volunteers and recipients are either approved or rejected and receive an appropriate notification accordingly.

Process:

1. The user logs into the system as a coordinator or association manager.
2. The user navigates to the "Pending Volunteers" or "Pending Recipients" screen and reviews the details of each membership request.
3. The user contacts each applicant privately to conduct a personal introduction.
4. The user approves or rejects the membership requests based on the association's criteria and the personal conversations held.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The personal details of a specific candidate are invalid, preventing approval or contact.

11. Monitor Existing Volunteers

Primary Actors: Coordinator, Association Manager

Description:
This use case allows coordinators and the association manager to view all available volunteers within the association and monitor their various activities.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a coordinator or association manager.

Postconditions:
The details of all volunteers within the association are displayed on the screen in a clear and understandable manner.

Process:

1. The user logs into the system as a coordinator or association manager.
2. The user navigates to the "Volunteer Monitoring" screen.
3. The details of all volunteers are displayed in an organized and easily readable format for review and management.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The page fails to load properly or does not display all volunteers in the system due to a communication issue.

12. Monitor Existing Recipients

Primary Actors: Coordinator, Association Manager

Description:
This use case allows coordinators and the association manager to view all available recipients within the association and monitor their various activities.

Preconditions:

- The system is up and running as required.
- The user is logged into the system as a coordinator or association manager.

Postconditions:
The details of all recipients within the association are displayed on the screen in a clear and understandable manner.

Process:

1. The user logs into the system as a coordinator or association manager.
2. The user navigates to the "Recipient Monitoring" screen.
3. The details of all recipients are displayed in an organized and easily readable format for review and management.

Alternative Scenarios:

- The user is not authorized and cannot perform this action.
- The page fails to load properly or does not display all recipients in the system due to a communication issue.

Chapter 2 – System Architecture

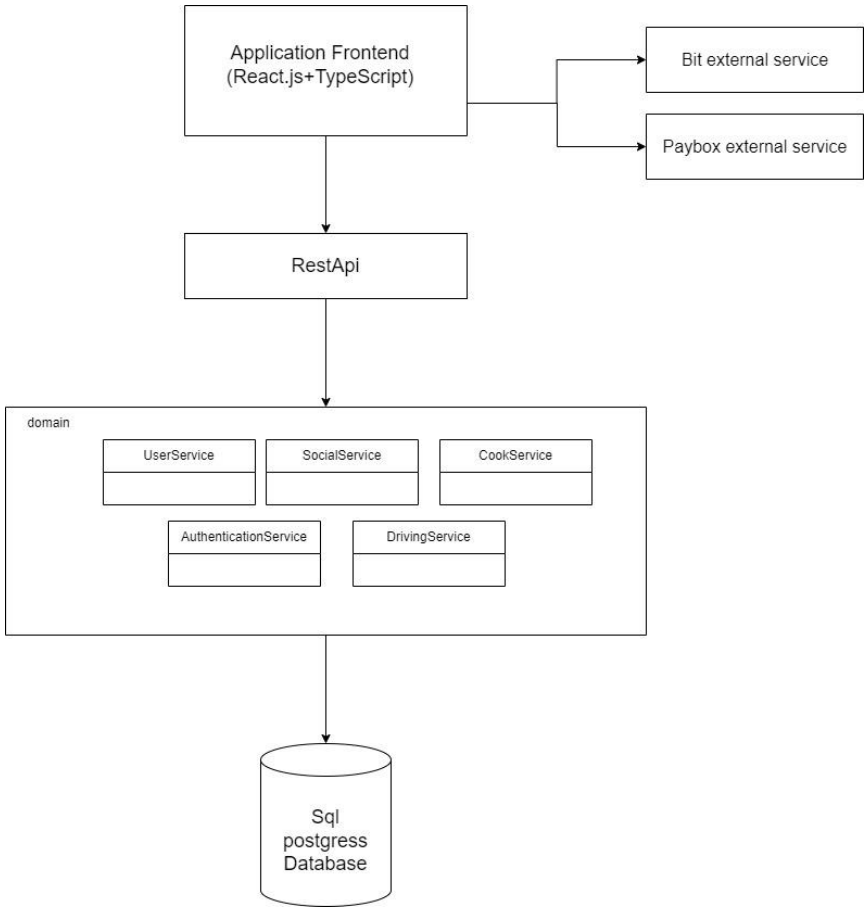
The project utilizes a layered architecture with a React.js and TypeScript frontend communicating with a Spring Boot backend. The backend services are organized into modules responsible for user management, authentication, cooking, social interactions, and driver management. A PostgreSQL database serves as the central data store.

The User Service manages donors and needers, while the Authentication Service handles phone number and password-based authentication. The Cooking Service manages cook constraints and shifts, and the Social Service handles needers' availability for donation. Finally, the Driving Service manages drivers' constraints and routes. All services interact with each other to ensure a cohesive and functional system.

Key Points:

- **Frontend:** React.js + TypeScript, communicates with the backend via Restful APIs.
- **Backend:** Spring Boot, divided into multiple services (User, Auth, Cooking, Driving, Social).
- **Security Layer:** Manages JWT tokens, authentication filters, and role-based access.
- **Database:** PostgreSQL
- **External integrations:** Bit and Paybox for donation processing, and email service for reset password.
- **Donation Features:** The frontend includes QR code and URL features to facilitate donations to the association.

Each service is loosely coupled and communicates via HTTP endpoints. The controller layer serves as an API gateway, while services contain the business logic.



Chapter 3 – Data Model

3.1 Description of Data Object

1. **Individuals**

   This group encompasses all individuals involved in the system, whether they are donors or needers.

1. **Donor:** Represents a registered individual who contributes to the organization.

    - **id:** Unique identifier for each donor (auto-generated, primary key, bigInt)
    - **address:** Mailing address of the donor (Text).
    - **first_name:** First name of the donor (Text).
    - **last_name:** Last name of the donor (Text).
    - **phone_number:** Contact phone number of the donor (unique, Text).
    - **role:** Role of the donor within the organization (e.g., "Donor", "Staff", "Admin") (Text).
    - **email:** Email address of the donor (unique, Text).
    - **status:** Current status of the donor (e.g., "Pending", "Available")(Text).
    - **last_donation_date:** Date of the donor's last contribution (date).
    - **verification_code:** code to reset password (Text).
    - **verification_code_expires_at:** date that the code above is in valid (timestamp).
    - **user_credentials_id:** Foreign key referencing the User Credentials table, linking the donor to their authentication credentials (bigInt).

2. **Needy:** Represents an individual or family in need of assistance.

    - **id:** Unique identifier for each needy individual or family (auto-generated, primary key, bigInt).
    - **address:** Mailing address of the needy individual/family (Text).
    - **first_name:** First name of the needy individual (Text)..
    - **last_name:** Last name of the needy individual (Text)..
    - **phone_number:** Contact phone number of the needy individual (Text).
    - **role:** Role or category of need- express different from donors (Text).

- **confirm_status:** Status of the needy individual's registration or confirmation (e.g., "Pending", "Confirmed", "Active") (Text).

2. Constraints

This group includes entities related to scheduling and availability limitations for cooks and drivers.

- **Cook Constraints:** Represents constraints submitted by donors willing to cook.
    - **constraint_id:** Unique identifier for each cook constraint (auto-generated, primary key, bigInt).
    - **constraints:** Specific constraints or preferences for cooking (e.g., "1 meat 1 vegetetain", "2 without solt") (Text).
    - **date:** Date of the constraint (date).
    - **start_time:** Start time of availability for pick-up the food (Text).
    - **end_time:** End time of availability for pick-up the food (Text).
    - **location:** address for pickup the food (Text).
    - **status:** Status of the constraint (e.g., "Pending", "Approved") (Text).
    - **cook_id:** Foreign key referencing the Donor table (bigInt).
- **Driver Constraints:** Represents constraints submitted by donors willing to drive.
    - **driver_id:** Foreign key referencing the Donor table (co-PK, bigInt)
    - **date:** Date of the driver's availability (co-PK, date).
    - **start_hour:** Start time of availability for driving (Text).
    - **end_hour:** End time of availability for driving (Text).
    - **requests:** Any specific requests or preferences for driving assignments (e.g., "Prefer routes within a certain area")(Text).
    - **start_location:** starting location for driving (Text).

3. Routes & Visits

This group focuses on the logistics of deliveries or visits, including planned routes and individual visit records.

- **Route:** Represents a planned route for food delivery and pickup.
    - **route_id:** Unique identifier for each route (auto-generated, primary key, bigInt).
    - **date:** Date of the planned route (date).
    - **driver_id:** Foreign key referencing the Driver Constraints or Donor table (bigInt).
    - **is_submitted:** Boolean flag indicating whether the route has been published to driver (Boolean).

- **Visit:** Represents a single stop or visit on a route.
    - **visit_id:** Unique identifier for each visit (auto-generated, primary key, bigInt).
    - **additional_notes:** Any additional notes or instructions for the visit (Text).
    - **address:** Delivery or pickup address for the visit (Text).
    - **first_name:** First name of the recipient (Text).
    - **last_name:** Last name of the recipient (Text).
    - **start_hour:** Estimated start time for the visit(Text).
    - **end_hour:** Estimated maximum time for the visit (Text).
    - **status:** Purpose of the visit (e.g., "Start", "Pickup", "Deliver") (Text).
    - **route_id:** Foreign key referencing the Route table (bigInt).

4. Needer Tracking

This group contains entities related to tracking the needs and status of needy individuals.

- **Needer Tracking:** represent the availability of every needer to get food in certain
    - **id:** Unique identifier for each tracking record (auto-generated, primary key, bigInt).
    - **additional_notes:** Any additional notes about the needy individual's needs or preferences (i.e. "leave outside the door", "call when arrive") (Text).
    - **date:** Date of the tracking record (date).
    - **dietary_preferences:** Dietary preferences of the needy individual (e.g., "Vegetarian", "Vegan", "No salt") (Text).

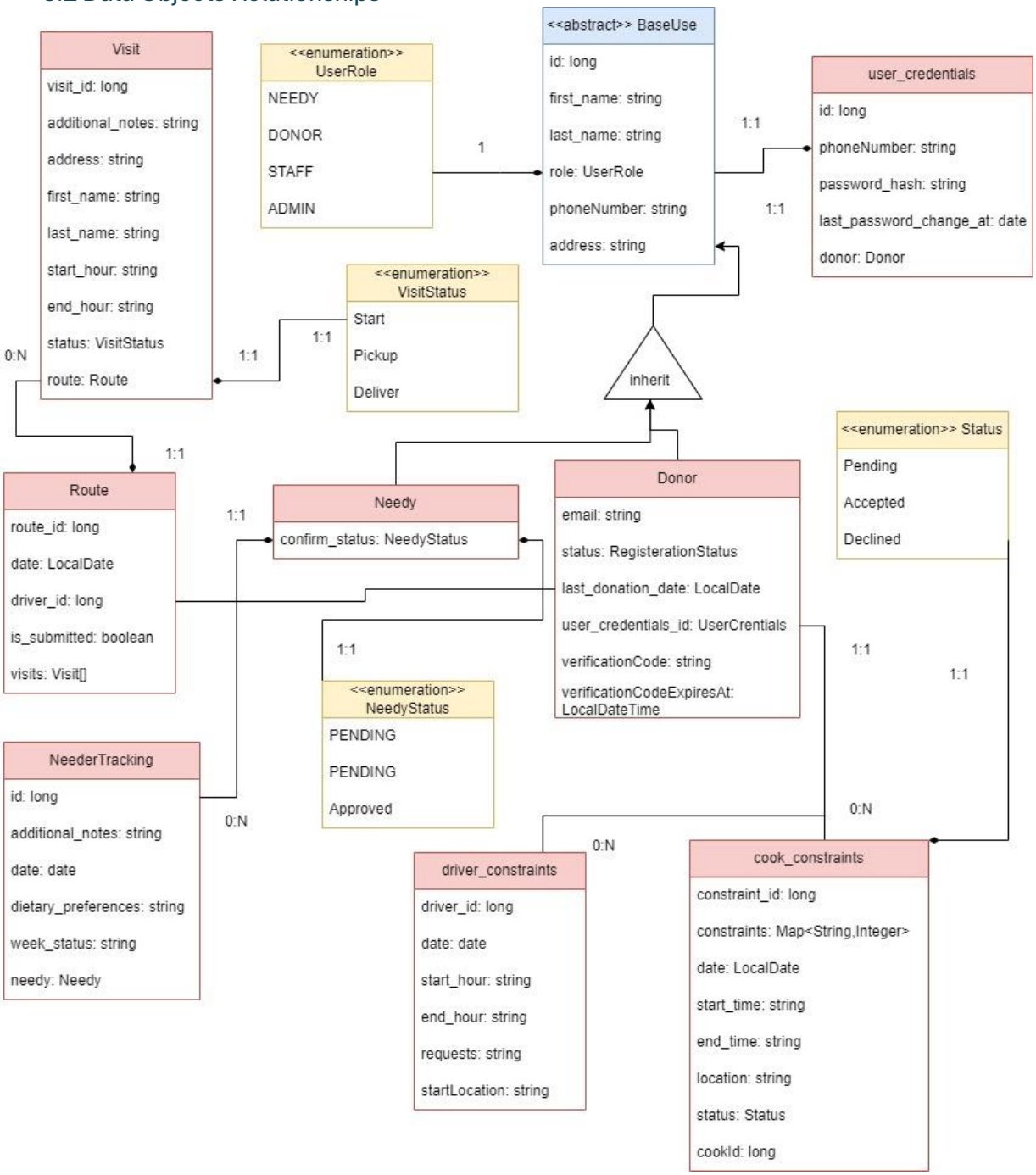
5. Authentication

includes methods for donors to enter the system.

- **User_credentials** manages donors authentication by unique phone number and password
    - **id:** Unique identifier for each user credential record (auto-generated, primary key, bigInt).
    - **phone_number:** Phone number used for user authentication (unique) (Text).
    - **password_hash**: Hashed password for user authentication (Text).
    - **last_password_change_at:** Timestamp of the last password change (date).

- **base_user_id:** Foreign key referencing the corresponding donor entity (bigInt).

3.2 Data Objects Relationships



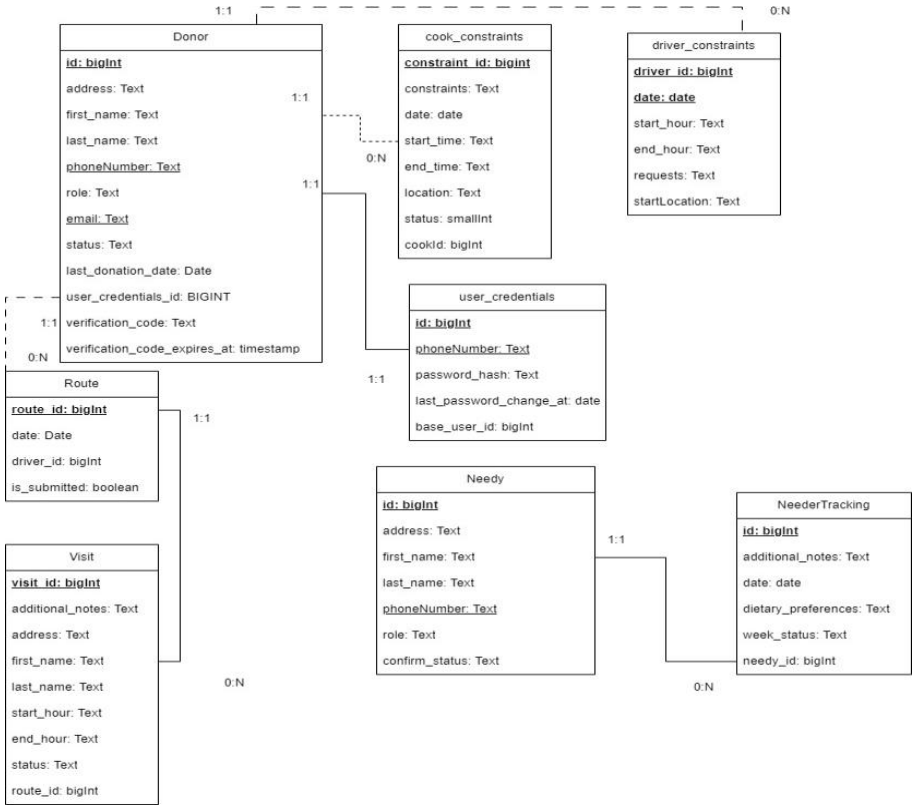
1. **Donor - User Credentials (1:1)**
   Each Donor has one associated set of User Credentials for authentication.
2. **Donor - Cook Constraints (0..N)**

A Donor can have zero or many Cook Constraints, indicating their availability and preferences for cooking.

3. **Donor - Driver Constraints (0..N)**
   A Donor can have zero or many Driver Constraints, indicating their availability and preferences for driving.
4. **Route(1:1) - Visit (0:N)**
   A Route can have one or many associated Visits.
5. **Needy - Needer Tracking (0:N)**
   Each Needy individual has one associated Needer Tracking record per every week.

3.3 Databases

Because Of the OOP nature of this project, we have chosen to use "Object Relational Mapping" (ORM), for mapping our Model Classes into DB entities. Specifically, we are using the "Jakarta ORM" library. As a result, the Data Base Tables and columns are almost an identical reflection of our class's model.
All passwords in the database are encrypted, the passwords are hashed using Bcrypt algorithm with a salt. This ensures secure one-way encryption, no decryption key is stored, as Bcrypt is not reversible.
Credentials are verified by comparing the input password hash to the stored hash.
All passwords logic is handled securely in the Authentication Service.
The following image depicts all entities as tables and the relationships.
In bold marked primary key and in underline marked unique values.



Main transactions:

1. User Management:

- Create Donor:
    - Create a new Donor record.
- Update Donor:
    - Modify existing Donor information (e.g., address, phone number, status).
- Create Needy:
    - Create a new Needy record.

2. Constraint Management:

- Create Cook Constraint:
    - Allow a Donor to submit a new Cook Constraint (availability for cooking).
- Update Cook Constraint:
    - Allow a Donor to modify an existing Cook Constraint.
    - Allow administrators to approve or decline Cook Constraints.
- Create Driver Constraint:
    - Allow a Donor to submit a new Driver Constraint (availability for driving).
- Update Driver Constraint:
    - Allow a Donor to modify an existing Driver Constraint.
    - Allow administrators to approve or decline Driver Constraints.

3. Route Management:

- Create Route:
    - Create a new Route based on a date
- Update Route:
    - Modify an existing Route (e.g., add or remove Visit records, assign driver).
- Delete Route:
    - Remove an existing Route from the system.

4. Visit Management:

- Create Visit:

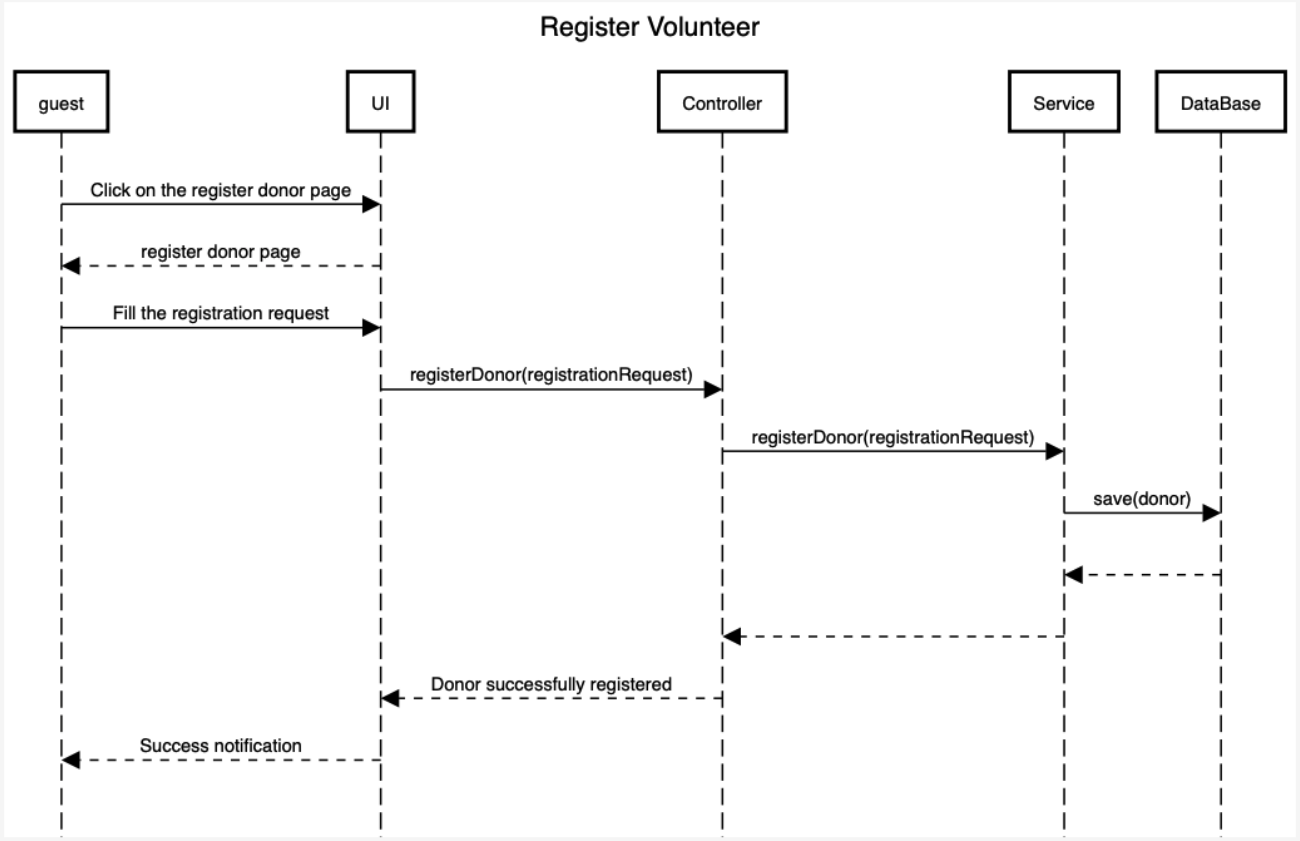
    - Add a new Visit to an existing Route.

5. Needer Tracking Management:
    - Create Needer Tracking Record:
        - Create an initial Needer Tracking record for a new Needy individual.
    - Update Needer Tracking:
        - Update the Needer Tracking record with information such as availability, dietary preferences, and additional notes.
6. Authentication & Authorization:
    - User Registration:
        - Create new UserCredentials for Donor.
    - User Login:
        - Authenticate user credentials and grant access to the system.
    - Role-Based Access Control:
        - Enforce access control rules based on user roles (e.g., Donor, Admin).

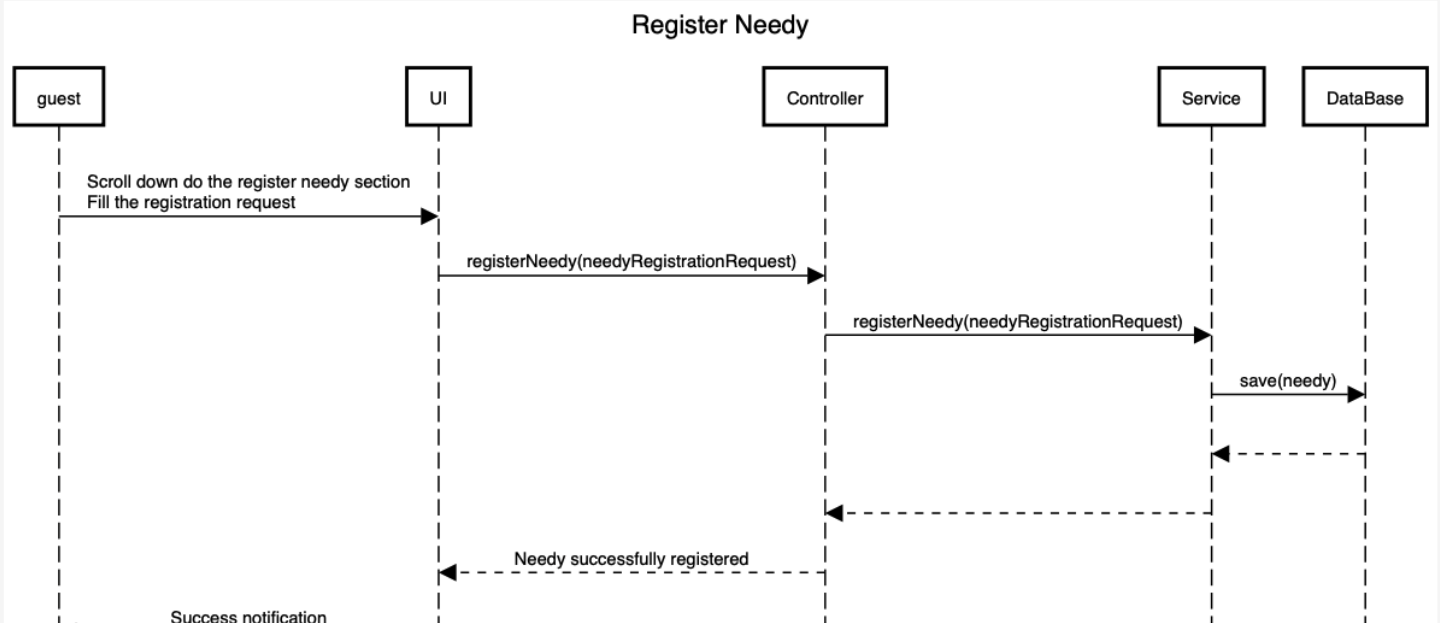
Chapter 4 – Behavioral Analysis:

4.1 Sequence Diagrams:

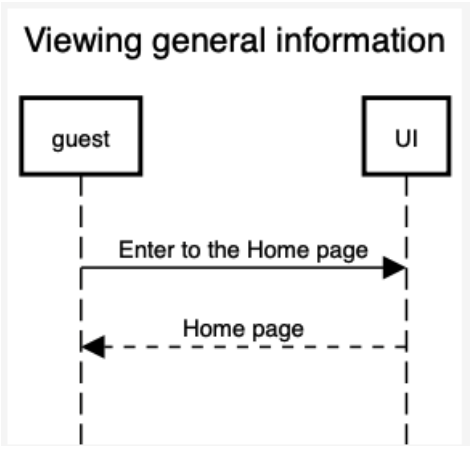
1.



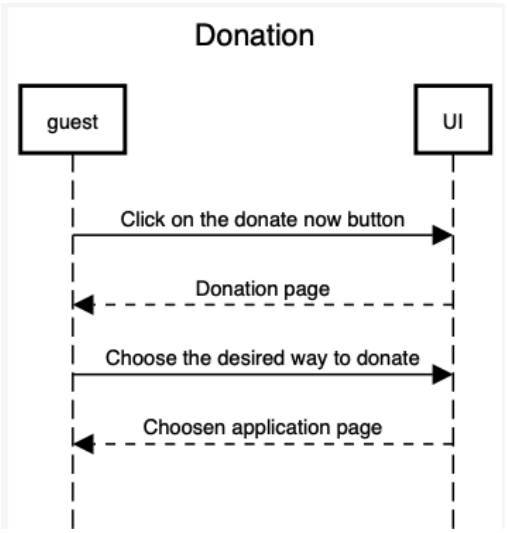
2.



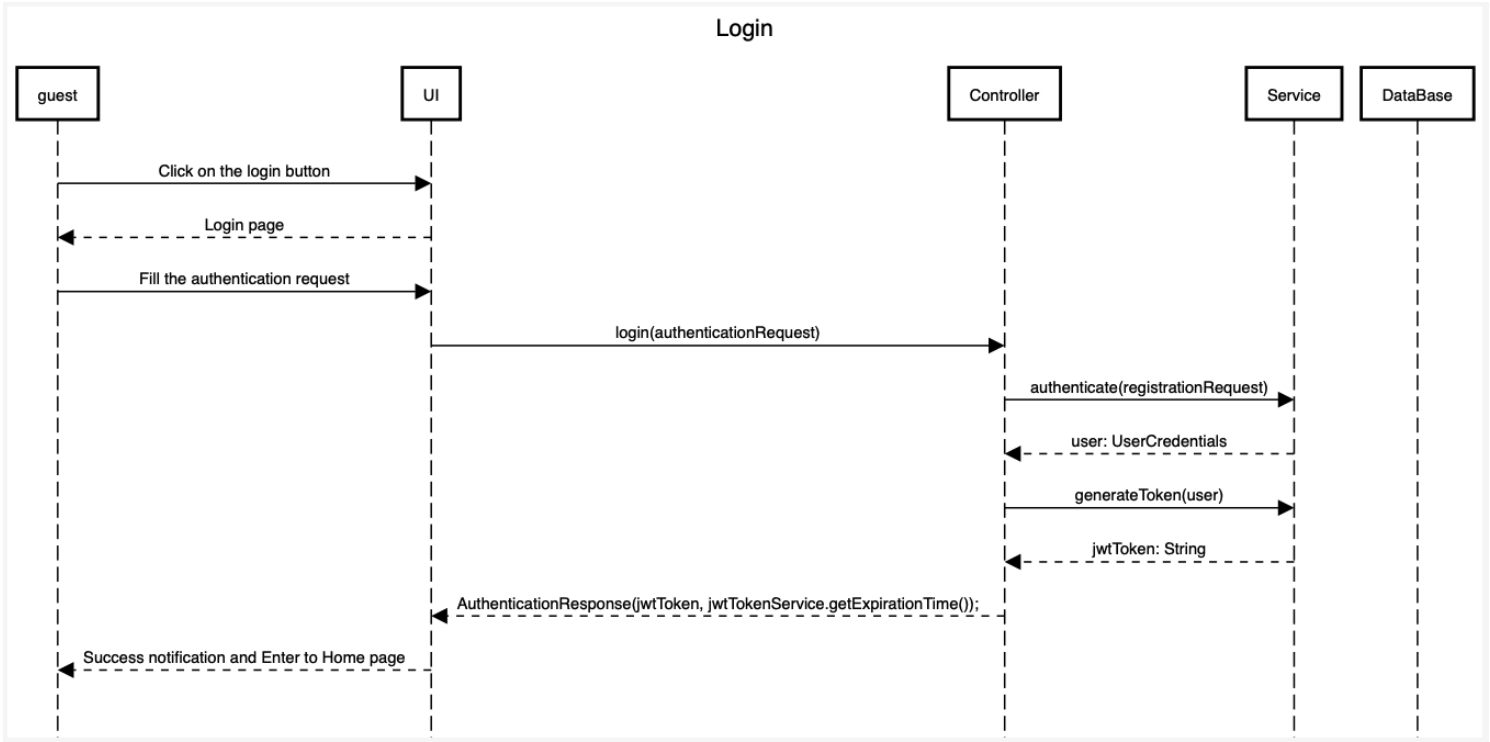
3.


Viewing general information
guest
UI
Enter to the Home page
Home page

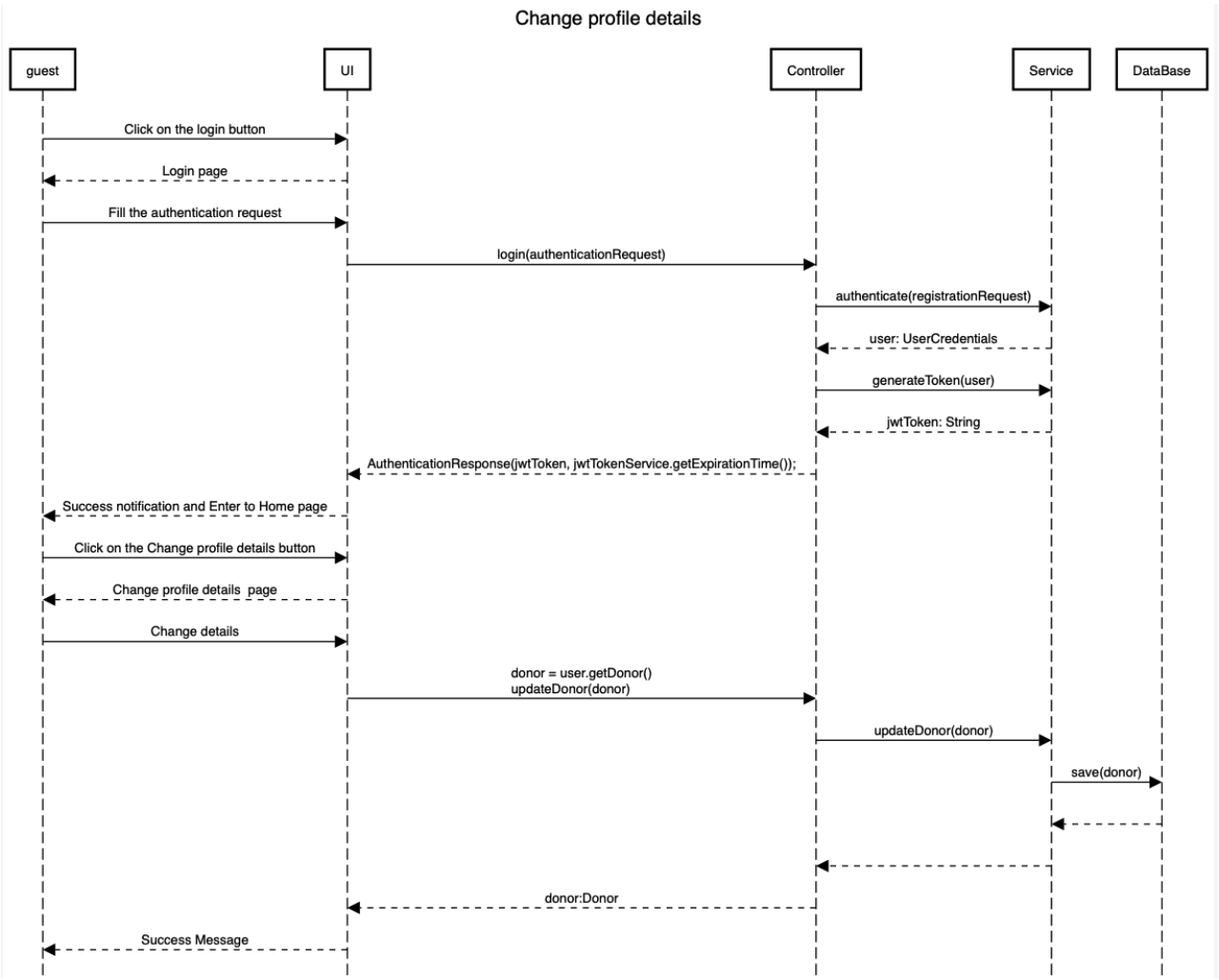
4.


Donation
guest
UI
Click on the donate now button
Donation page
Choose the desired way to donate
Choosen application page

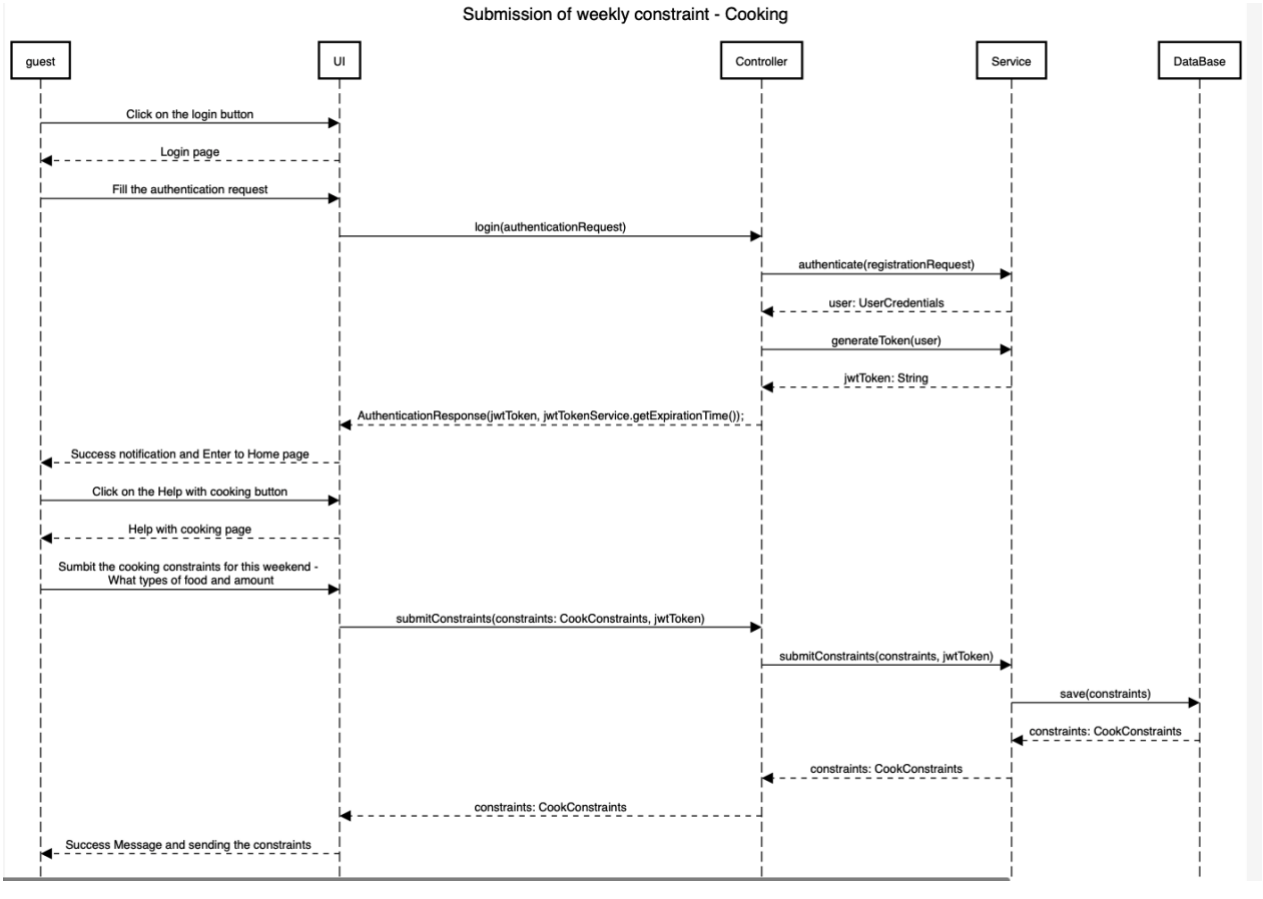
5.

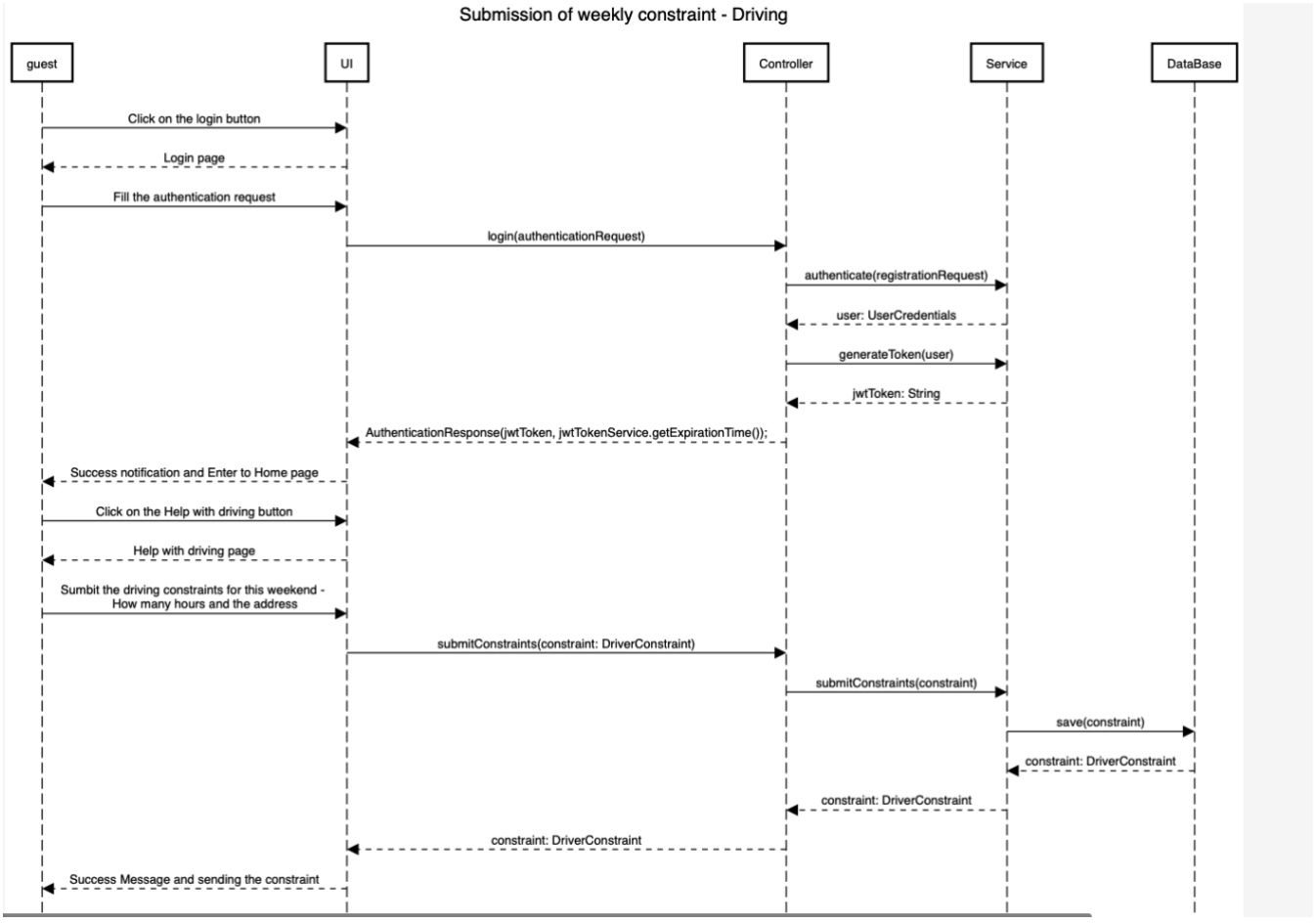

Login
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page

6.

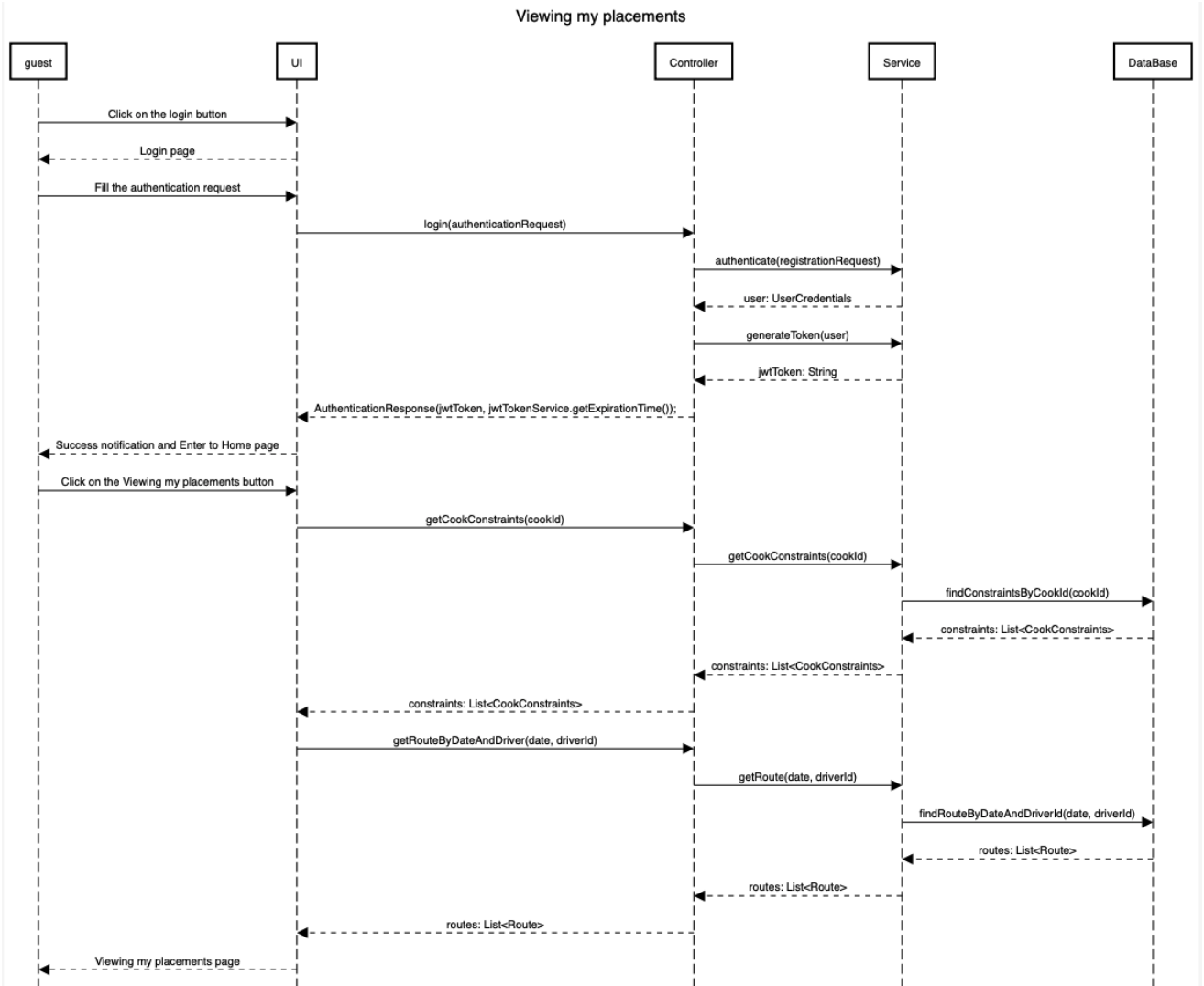

Change profile details
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Change profile details button
Change profile details page
Change details
donor = user.getDonor()
updateDonor(donor)
updateDonor(donor)
save(donor)
donor:Donor
Success Message

7.

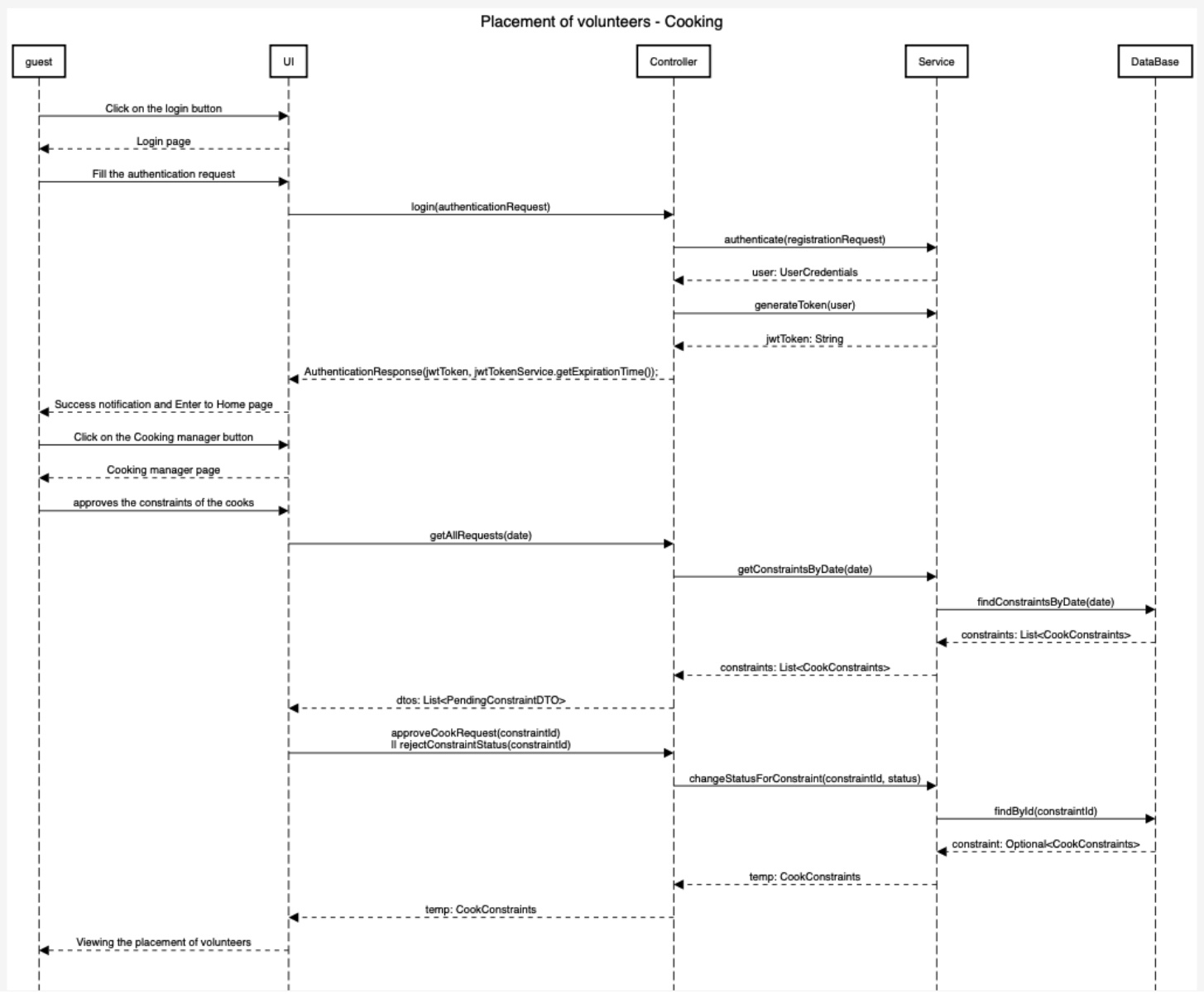

Submission of weekly constraint - Cooking
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Help with cooking button
Help with cooking page
Sumbit the cooking constraints for this weekend -
What types of food and amount
submitConstraints(constraints: CookConstraints, jwtToken)
submitConstraints(constraints, jwtToken)
save(constraints)
constraints: CookConstraints
constraints: CookConstraints
constraints: CookConstraints
Success Message and sending the constraints

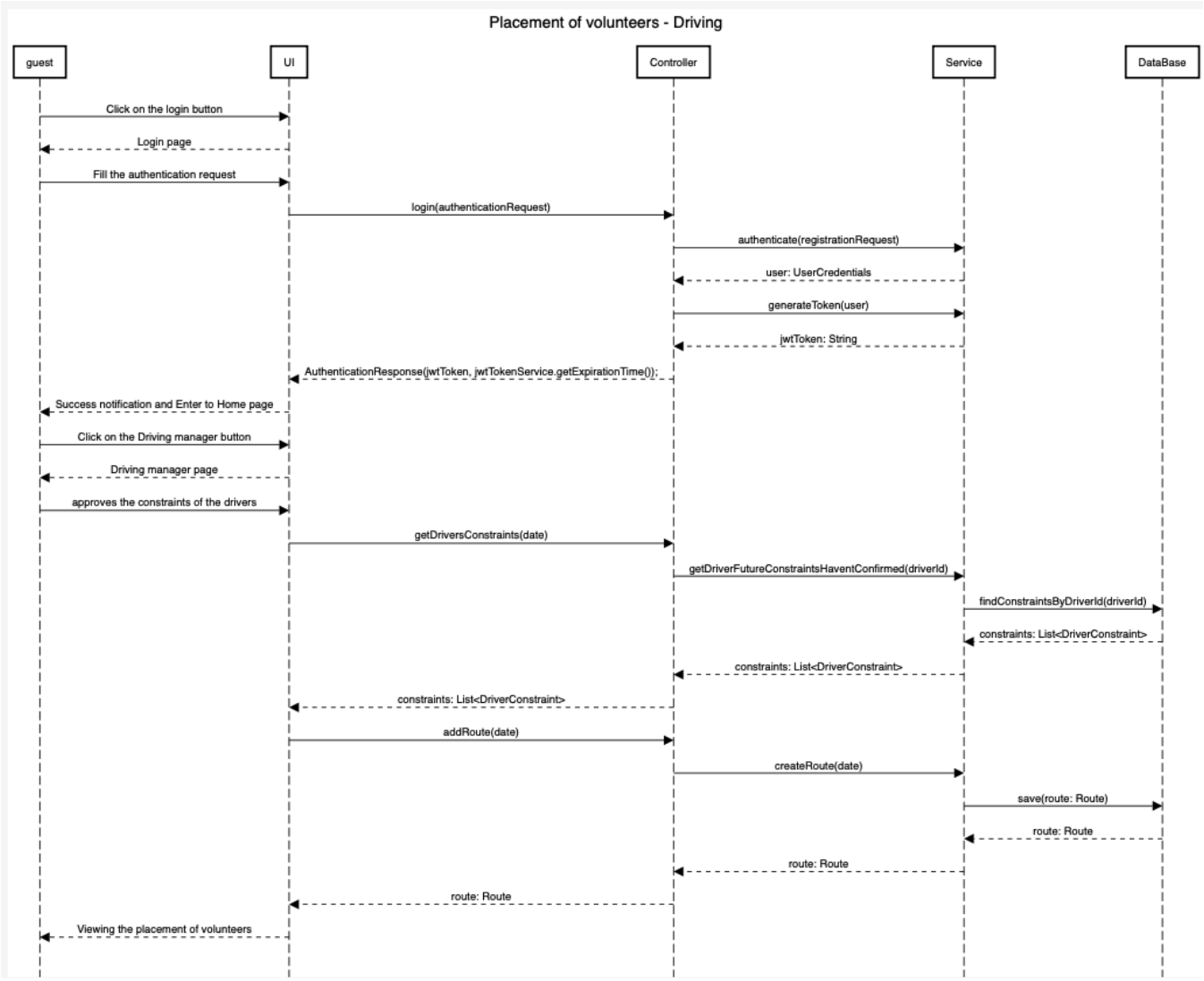

Submission of weekly constraint - Driving
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Help with driving button
Help with driving page
Sumbit the driving constraints for this weekend - How many hours and the address
submitConstraints(constraint: DriverConstraint)
submitConstraints(constraint)
save(constraint)
constraint: DriverConstraint
constraint: DriverConstraint
constraint: DriverConstraint
Success Message and sending the constraint

8.

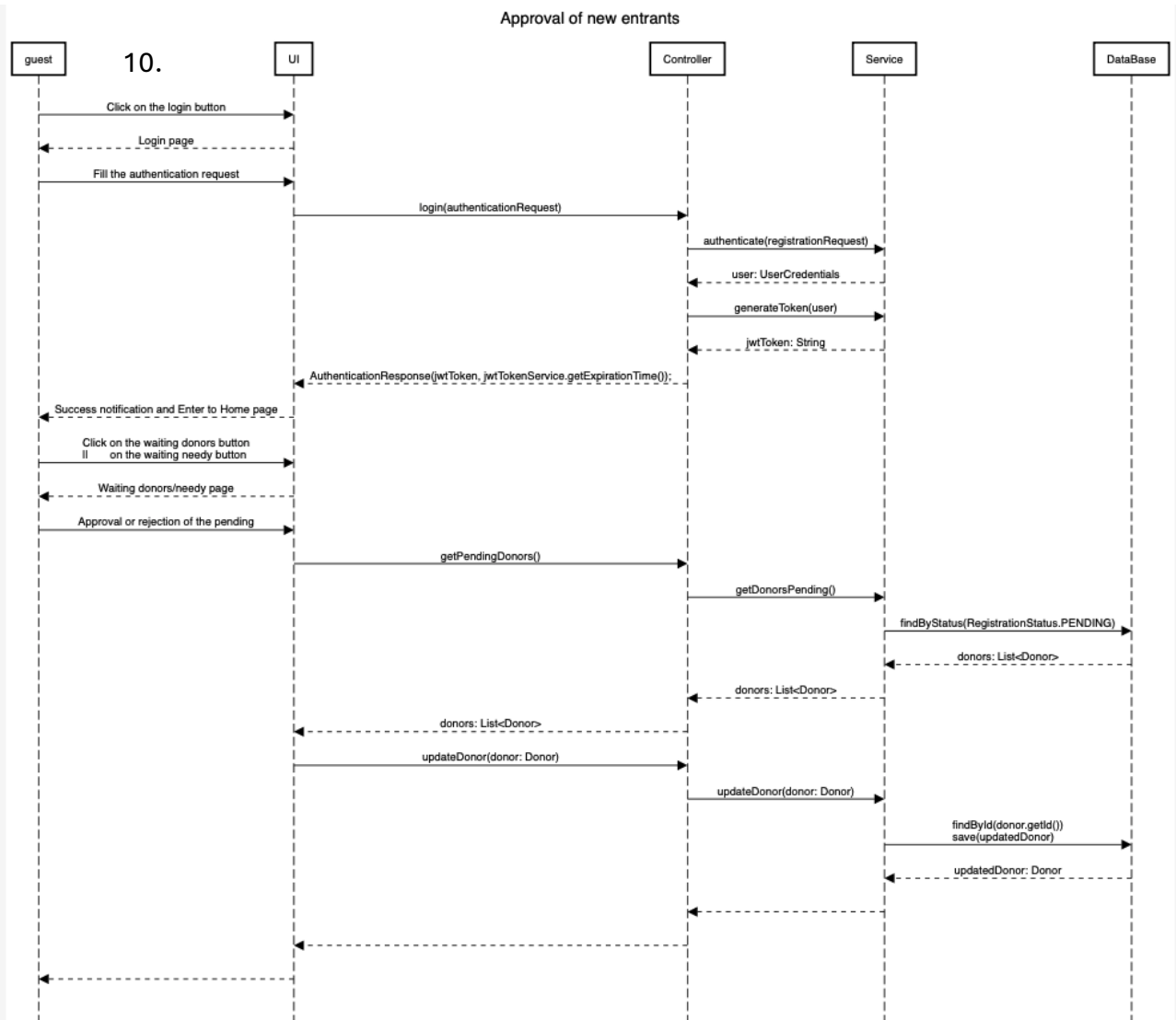

Viewing my placements
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Viewing my placements button
getCookConstraints(cookId)
getCookConstraints(cookId)
findConstraintsByCookId(cookId)
constraints: List<CookConstraints>
constraints: List<CookConstraints>
constraints: List<CookConstraints>
getRouteByDateAndDriver(date, driverId)
getRoute(date, driverId)
findRouteByDateAndDriverId(date, driverId)
routes: List<Route>
routes: List<Route>
routes: List<Route>
Viewing my placements page

9.

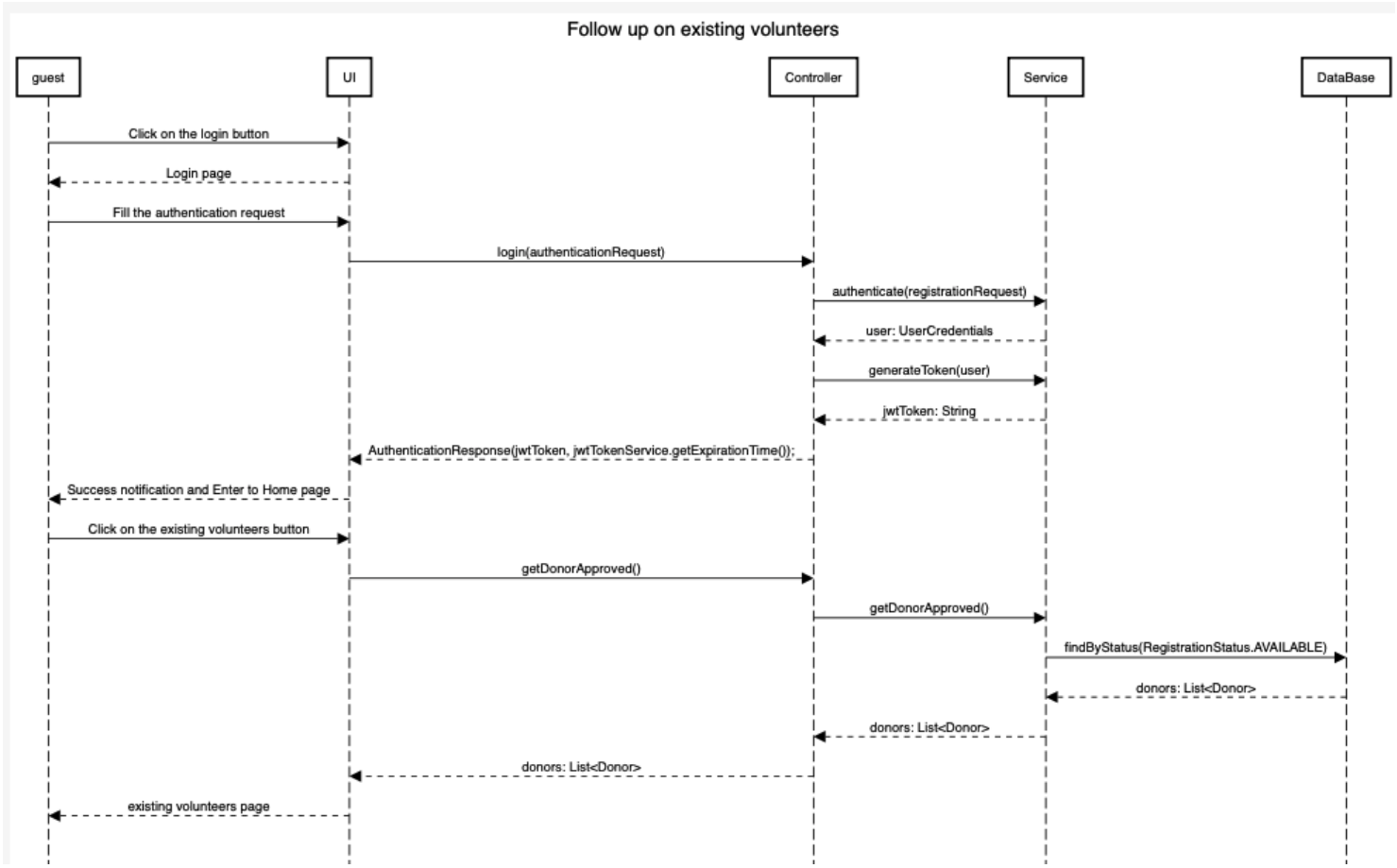

Placement of volunteers - Cooking
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Cooking manager button
Cooking manager page
approves the constraints of the cooks
getAllRequests(date)
getConstraintsByDate(date)
findConstraintsByDate(date)
constraints: List<CookConstraints>
constraints: List<CookConstraints>
dtos: List<PendingConstraintDTO>
approveCookRequest(constraintId)
|| rejectConstraintStatus(constraintId)
changeStatusForConstraint(constraintId, status)
findById(constraintId)
constraint: Optional<CookConstraints>
temp: CookConstraints
temp: CookConstraints
Viewing the placement of volunteers


Placement of volunteers - Driving
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the Driving manager button
Driving manager page
approves the constraints of the drivers
getDriversConstraints(date)
getDriverFutureConstraintsHaventConfirmed(driverId)
findConstraintsByDriverId(driverId)
constraints: List<DriverConstraint>
constraints: List<DriverConstraint>
constraints: List<DriverConstraint>
addRoute(date)
createRoute(date)
save(route: Route)
route: Route
route: Route
route: Route
Viewing the placement of volunteers

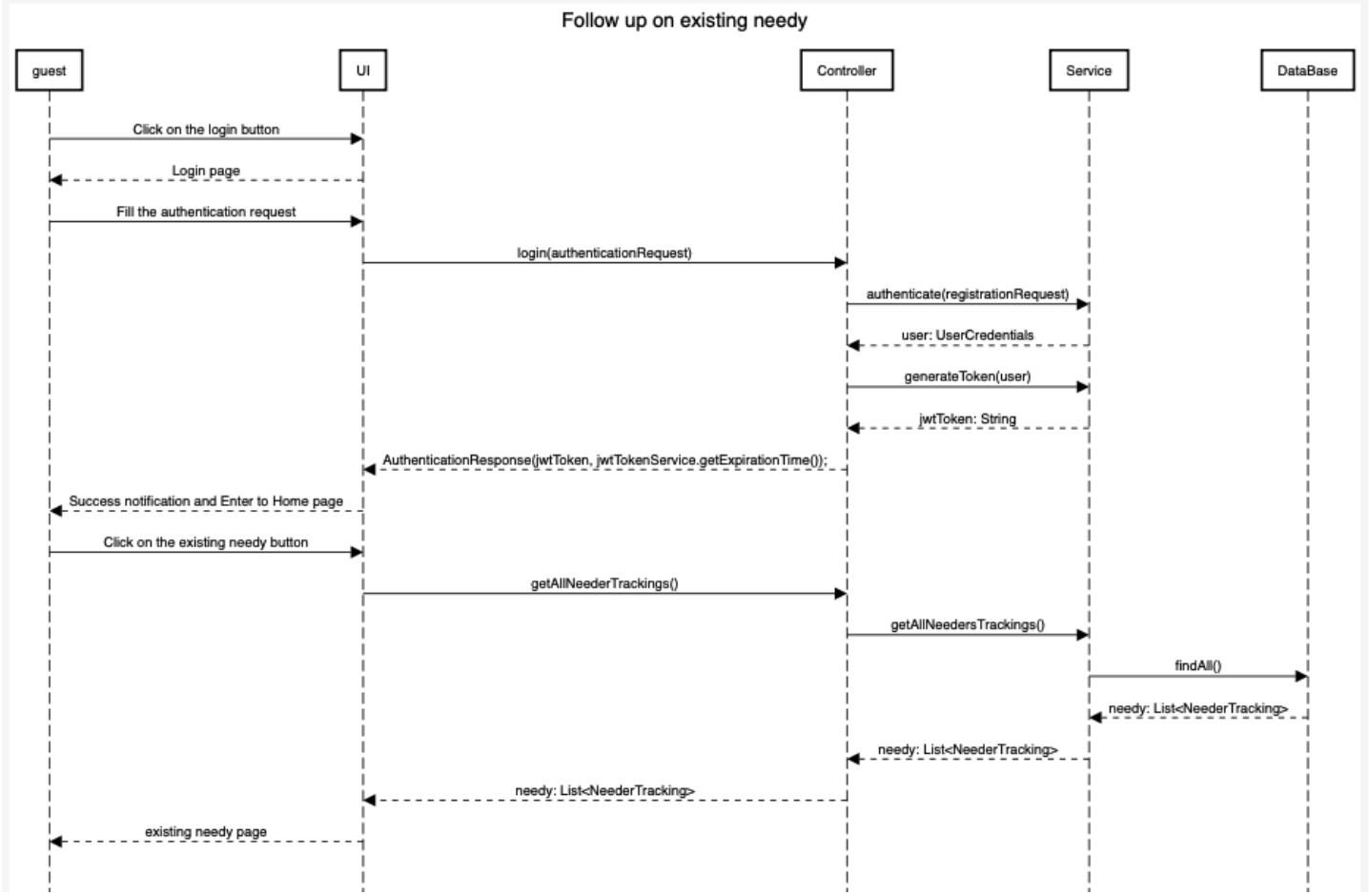
10.


Approval of new entrants
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the waiting donors button
II on the waiting needy button
Waiting donors/needy page
Approval or rejection of the pending
getPendingDonors()
getDonorsPending()
findByStatus(RegistrationStatus.PENDING)
donors: List<Donor>
donors: List<Donor>
donors: List<Donor>
updateDonor(donor: Donor)
updateDonor(donor: Donor)
findById(donor.getId())
save(updatedDonor)
updatedDonor: Donor

11.


Follow up on existing volunteers
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the existing volunteers button
getDonorApproved()
getDonorApproved()
findByStatus(RegistrationStatus.AVAILABLE)
donors: List<Donor>
donors: List<Donor>
donors: List<Donor>
existing volunteers page

12.


Follow up on existing needy
guest
UI
Controller
Service
DataBase
Click on the login button
Login page
Fill the authentication request
login(authenticationRequest)
authenticate(registrationRequest)
user: UserCredentials
generateToken(user)
jwtToken: String
AuthenticationResponse(jwtToken, jwtTokenService.getExpirationTime());
Success notification and Enter to Home page
Click on the existing needy button
getAllNeederTrackings()
getAllNeedersTrackings()
findAll()
needy: List<NeederTracking>
needy: List<NeederTracking>
needy: List<NeederTracking>
existing needy page

4.2 Events:

Event name	Description	Action
New Volunteer Approval	Occurs when coordinators or the association manager approve a new volunteer to the association (driver or cook).	The system updates the volunteer's status to available and notifies the volunteer accordingly.
New Recipient Approval	Occurs when coordinators or the association manager approve a new recipient for the association to assist them with their needs.	The system updates the recipient's status to active in the system and notifies the user accordingly.
System Login	Occurs when a volunteer, coordinator, or association manager enters their details and logs into the system.	The system verifies the login details against existing data. If the details are correct, it allows the login and grants access to the system; otherwise, it notifies the user of the issue.
Donation	Occurs when any user makes a monetary donation.	The system redirects the user to the relevant pages in the chosen donation application for continuing the donation process.
Personal Details Update	Occurs when a logged-in user wishes to update their personal details in the system.	The system receives the updated details and checks if they are valid and within the system's constraints. If valid, it updates the data; otherwise, it notifies the user of the issue.
Submitting Constraints	Occurs when a logged-in user submits their weekly availability constraints for volunteering.	The system verifies that submitted constraints comply with predefined conditions, such as valid time intervals, no overlap with other assignments, and matching required fields. If validation fails, an appropriate error message is displayed.

Creating Work Schedule for Cooks	Occurs when the cook coordinator or association manager receives the weekly constraints of the various cooks and creates assignments for each.	The system receives the assignment and forwards the information to all relevant cooks, so they are aware of their work schedule for the upcoming week.
Creating Work Schedule for Drivers	Occurs when the driver coordinator or association manager receives the weekly constraints of the various drivers and creates assignments for each.	The system receives the assignment and forwards the information to all relevant drivers, so they are aware of their work schedule for the upcoming week.
Monitoring Existing Volunteers	Occurs when a coordinator or the association manager wishes to view and monitor all existing volunteers within the association.	The system displays all available volunteers approved by the association manager in a clear and easily understandable format.
Monitoring Existing Recipients	Occurs when a coordinator or the association manager wishes to view and monitor all existing recipients within the association.	The system displays all available recipients approved by the association manager in a clear and easily understandable format.

4.3 States:

The system operates in various states based on different roles within the system and the interactions between these roles:

1. States by Role:

- **Guest User:**
  Limited to viewing general information about the association and making monetary donations.
- **Volunteer:**
  The system allows them to submit weekly availability constraints, update their personal details within the system, and view their assignments.
- **Coordinator:**
  The system enables them to approve new volunteers and recipients, monitor existing volunteers and recipients, and assign work schedules to each volunteer based on their constraints and the association's needs.
- **Association Manager:**
  The system allows them to perform all available actions within the system.

2. States by Approval – Volunteers:

- **Pending:**
  The volunteer is awaiting approval from the association and is therefore limited in their actions.
- **Approved:**
  The volunteer has been approved by the association and can now submit their various constraints and begin contributing to the association.
- **Rejected:**
  The volunteer has been rejected by the association and is similarly limited in their actions.

3. User's State in the System:

- **Logged in:**
  The user can perform the required actions based on their role within the system.
- **Logged out:**
  The user is limited in their activities, similar to a guest user.

4. System Operation State:

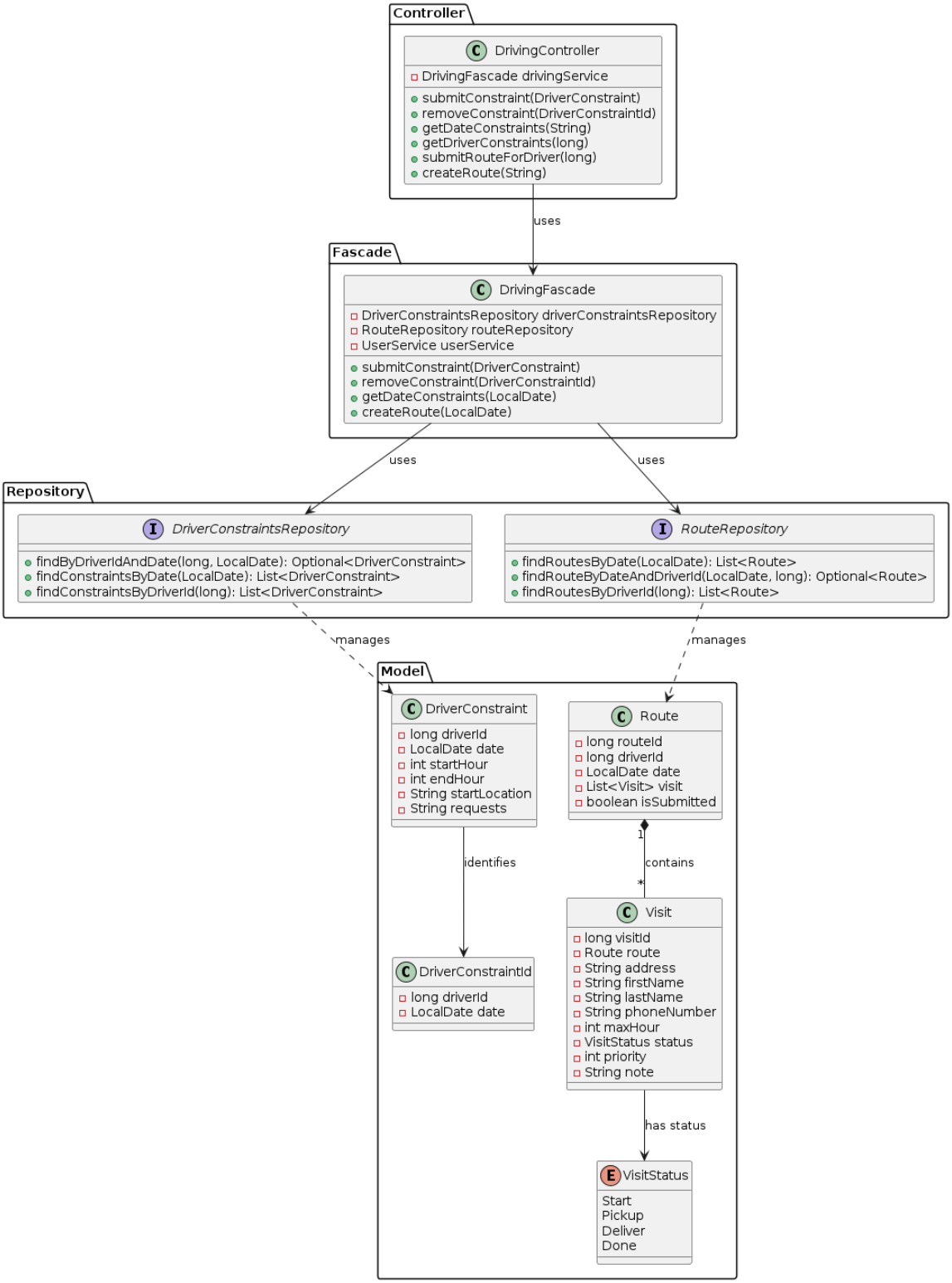
- **Idle State:**
  The system is standing by and not performing any actions.
- **Processing State:**
  The system is handling various requests, such as registration, login, volunteer assignments, etc.
- **Error State:**
  A malfunction has occurred within the system due to a specific action, requiring attention and resolution.

Chapter 5 – Object-Oriented Analysis

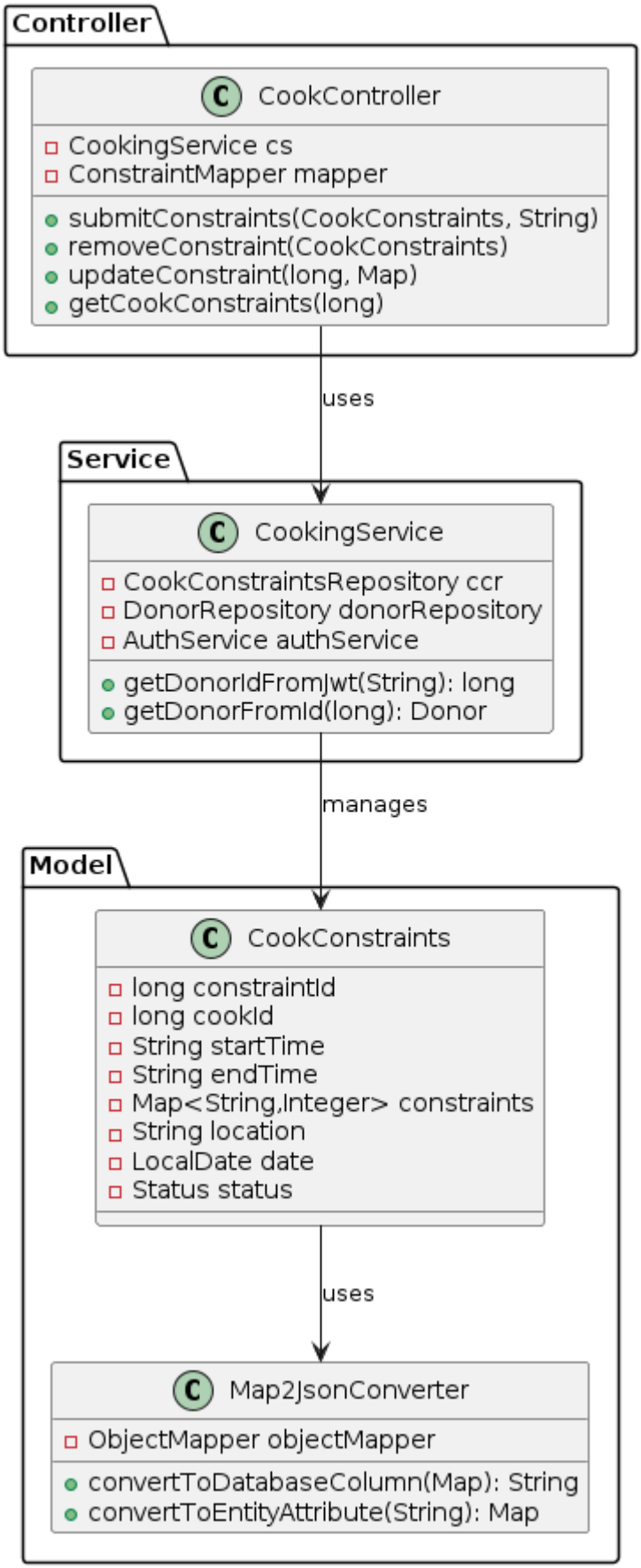
5.1 Class Diagram

5.1.1 Backend structure

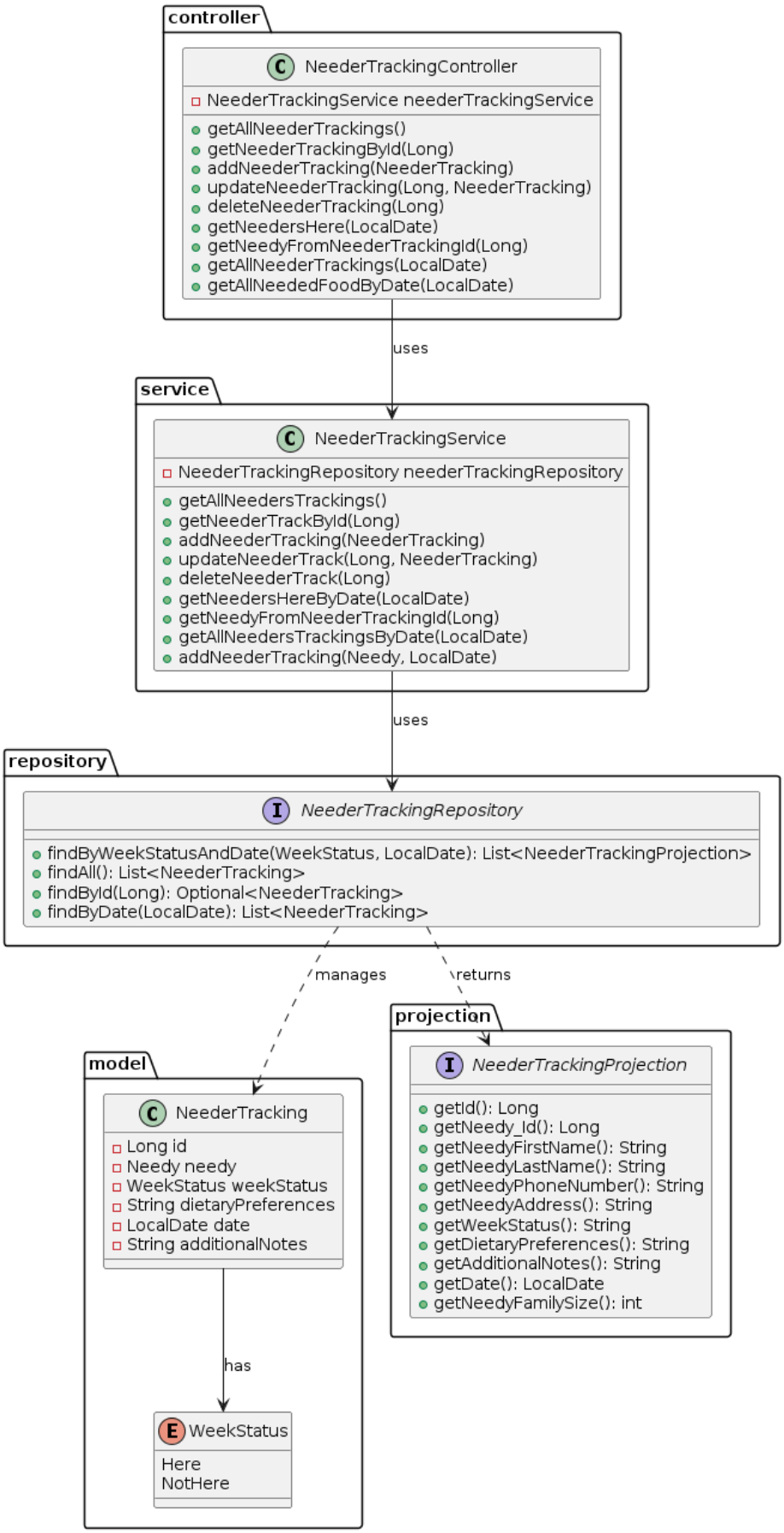
- Driving Package



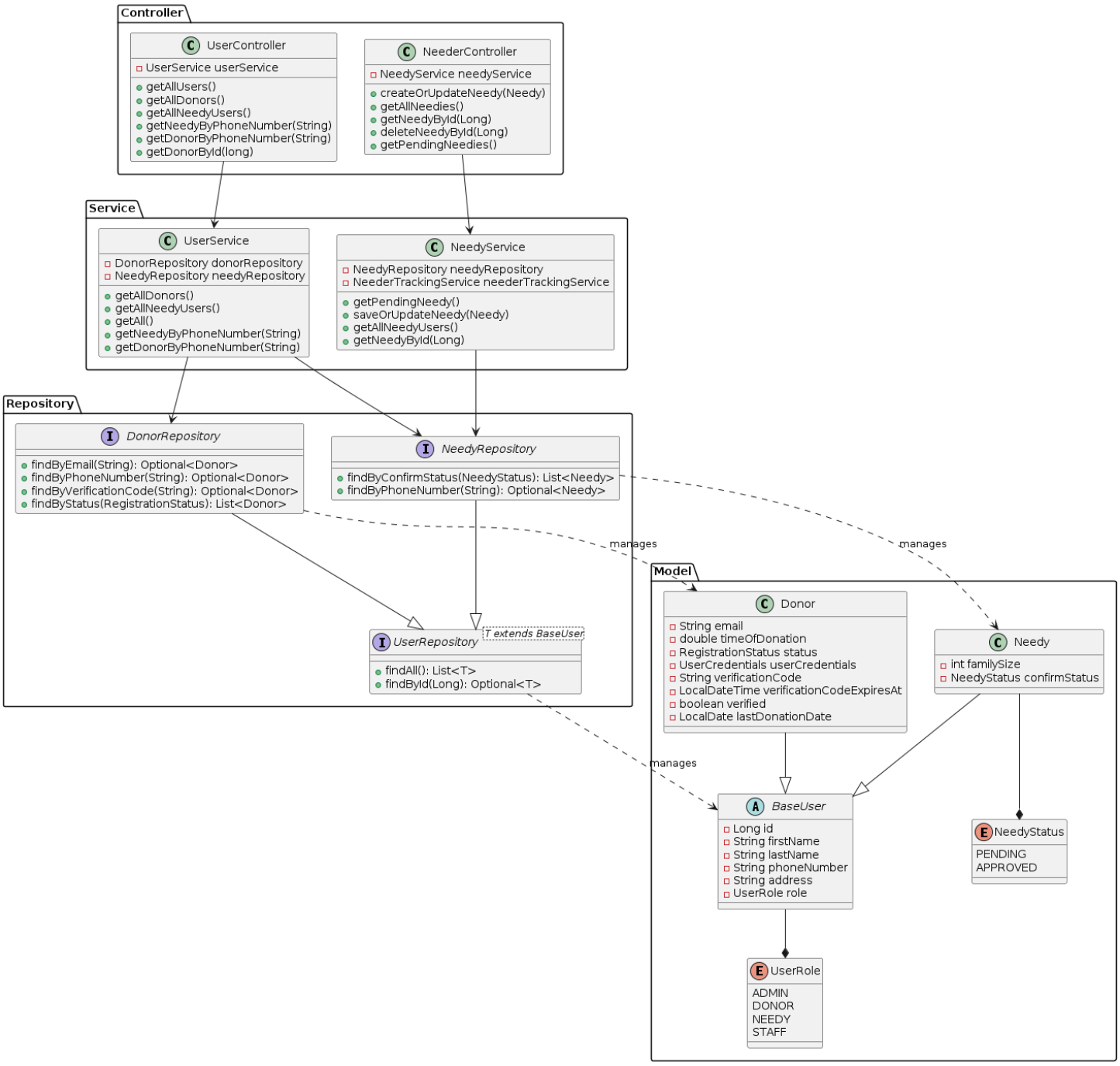
- Cooking Package



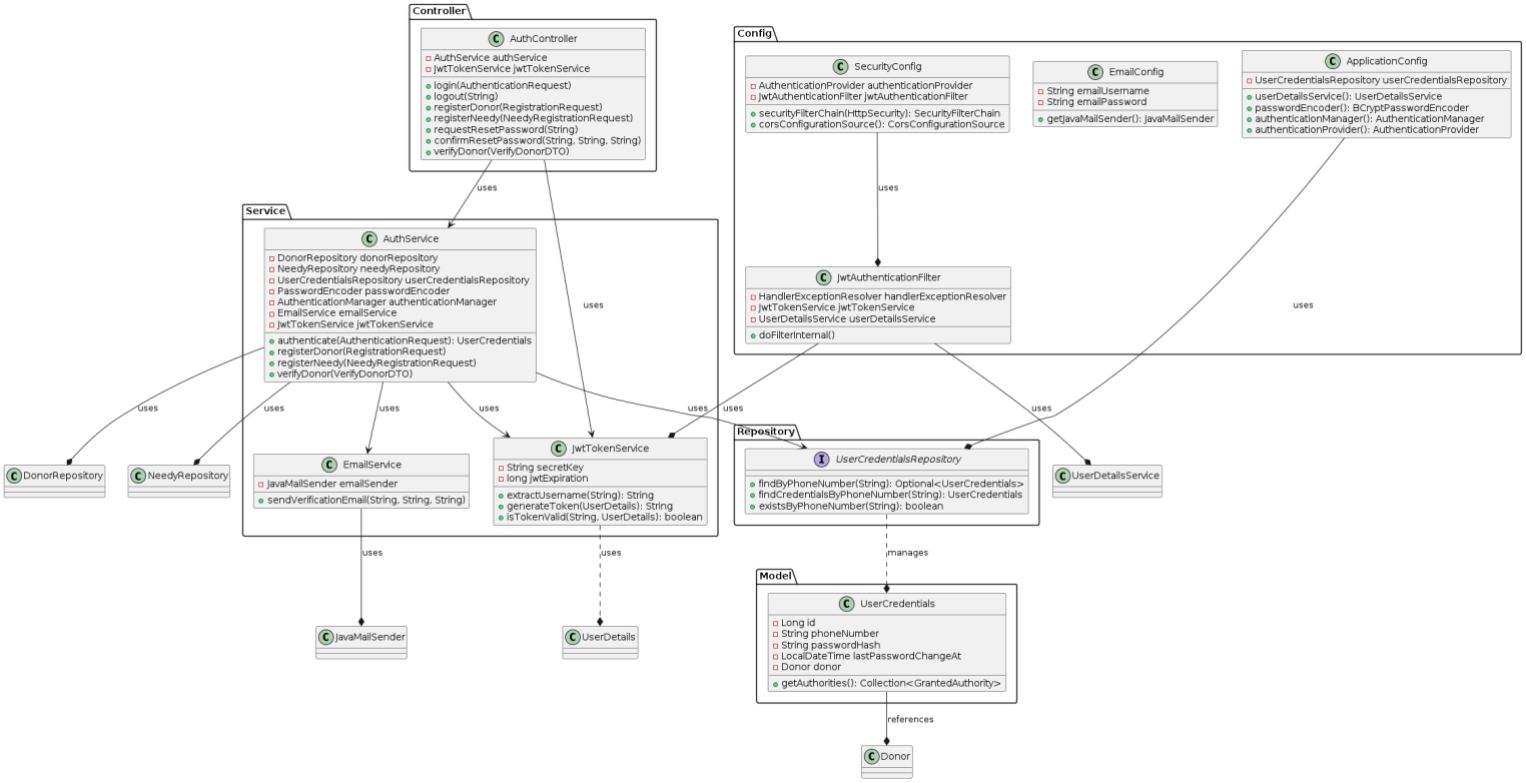
- Social Package



5.1.2 Backend Users management

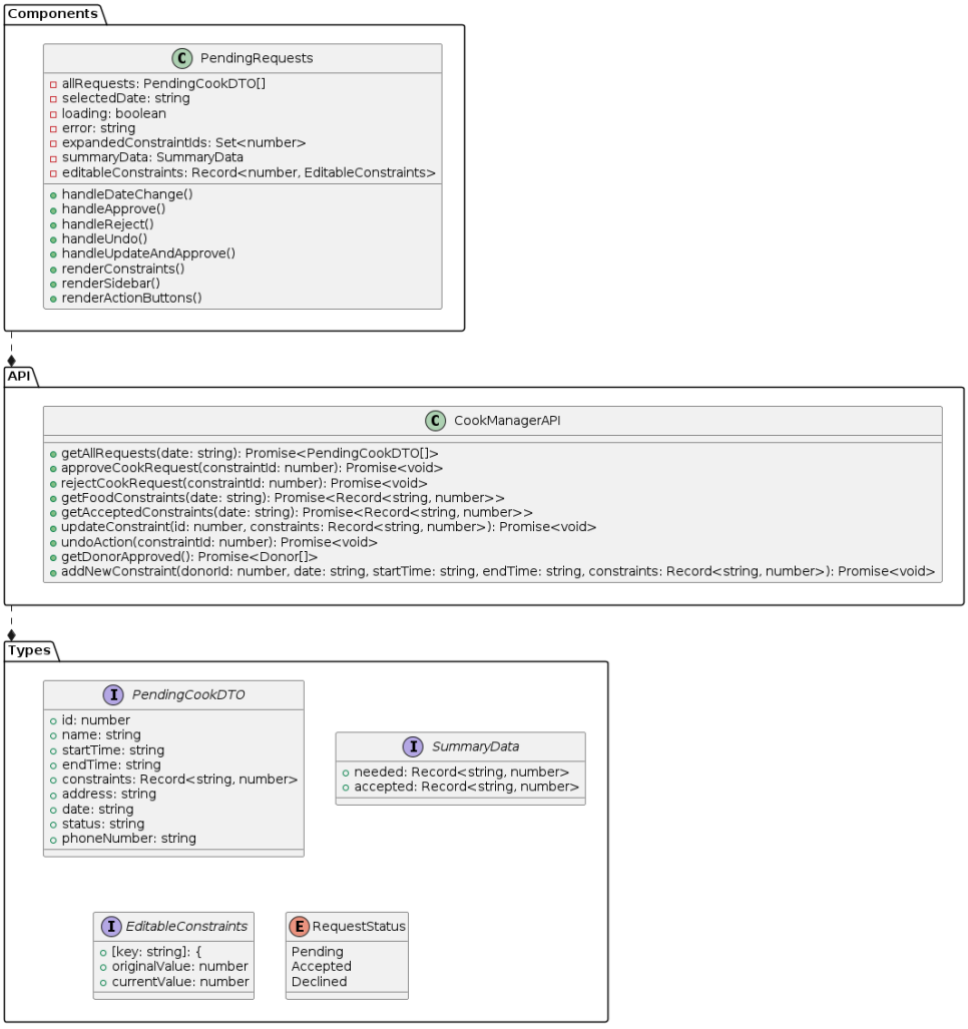


5.1.3 Security and Authentication Structure

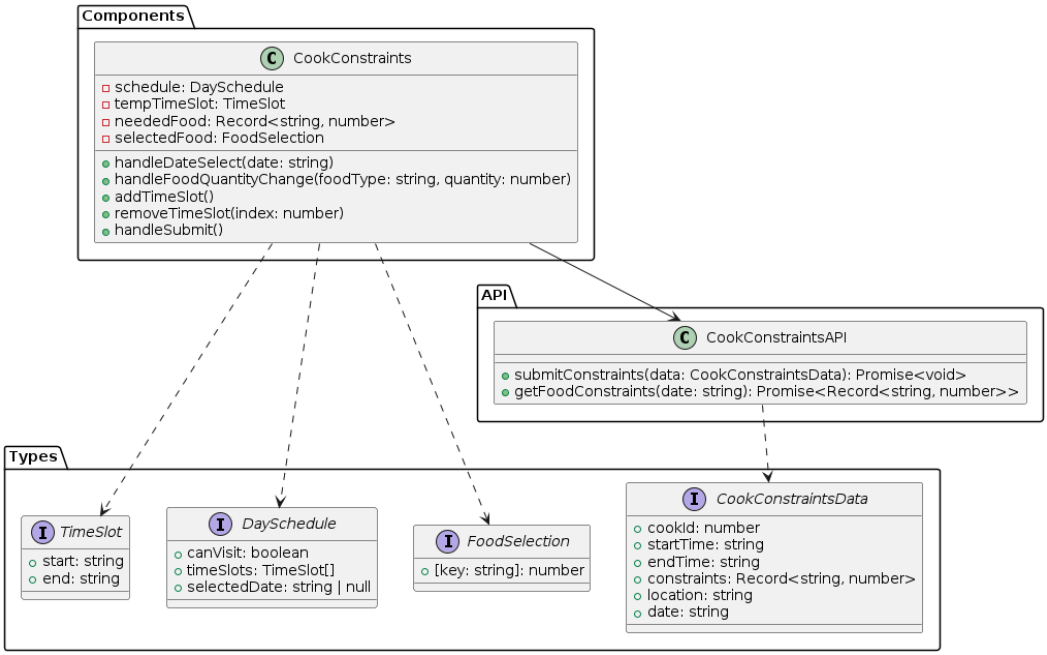


5.1.4 UI

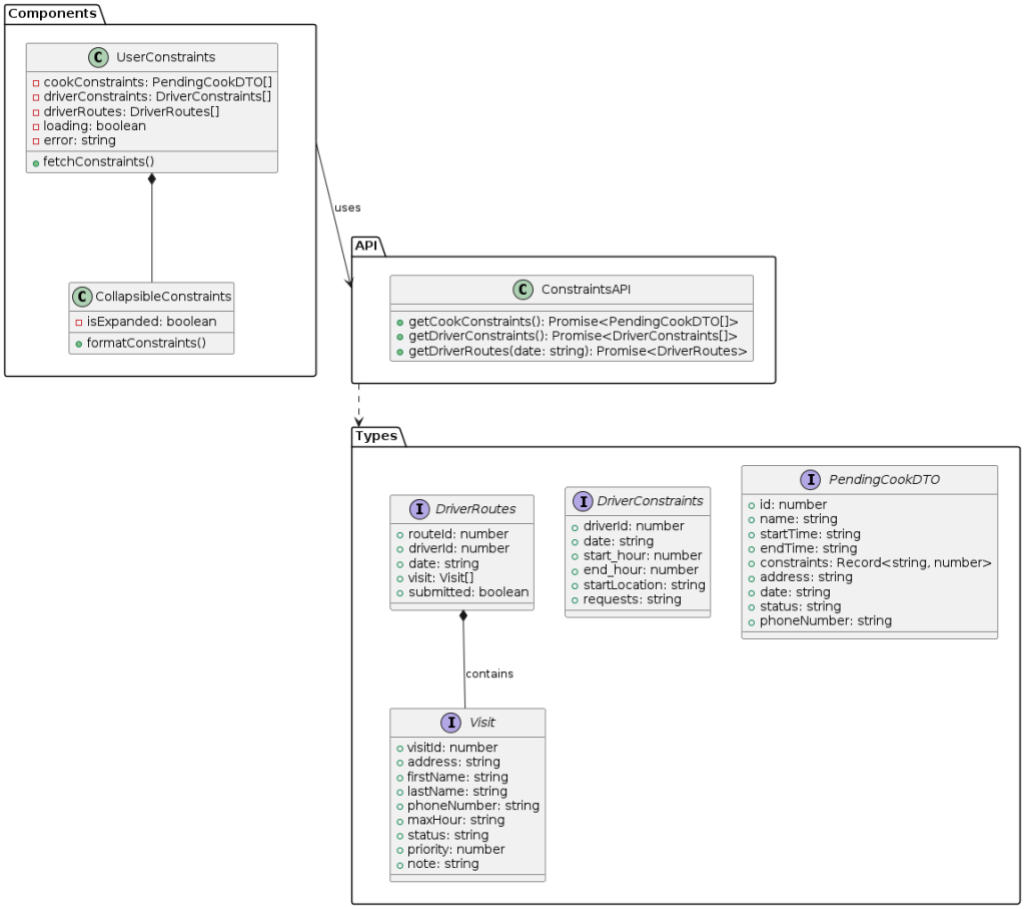
- Cooking Manager Page



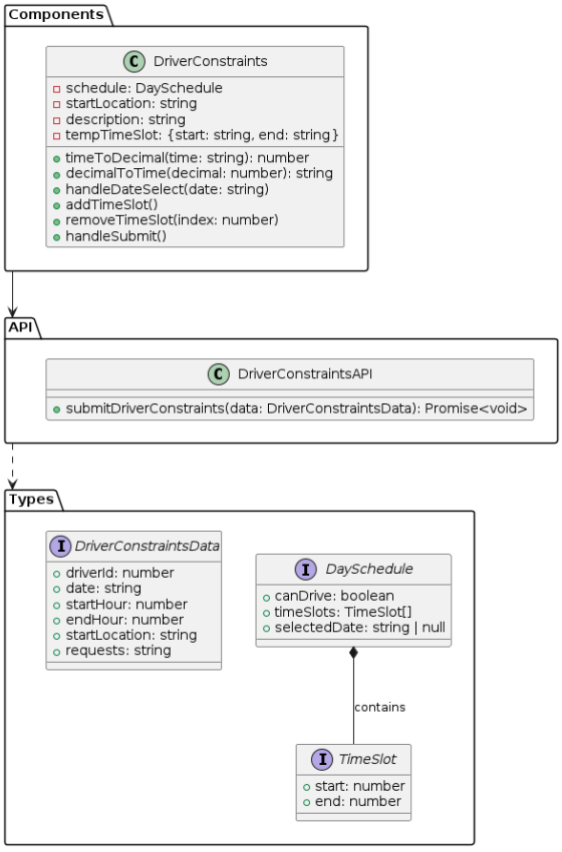
- Cook Donor Constraints Uploading Page



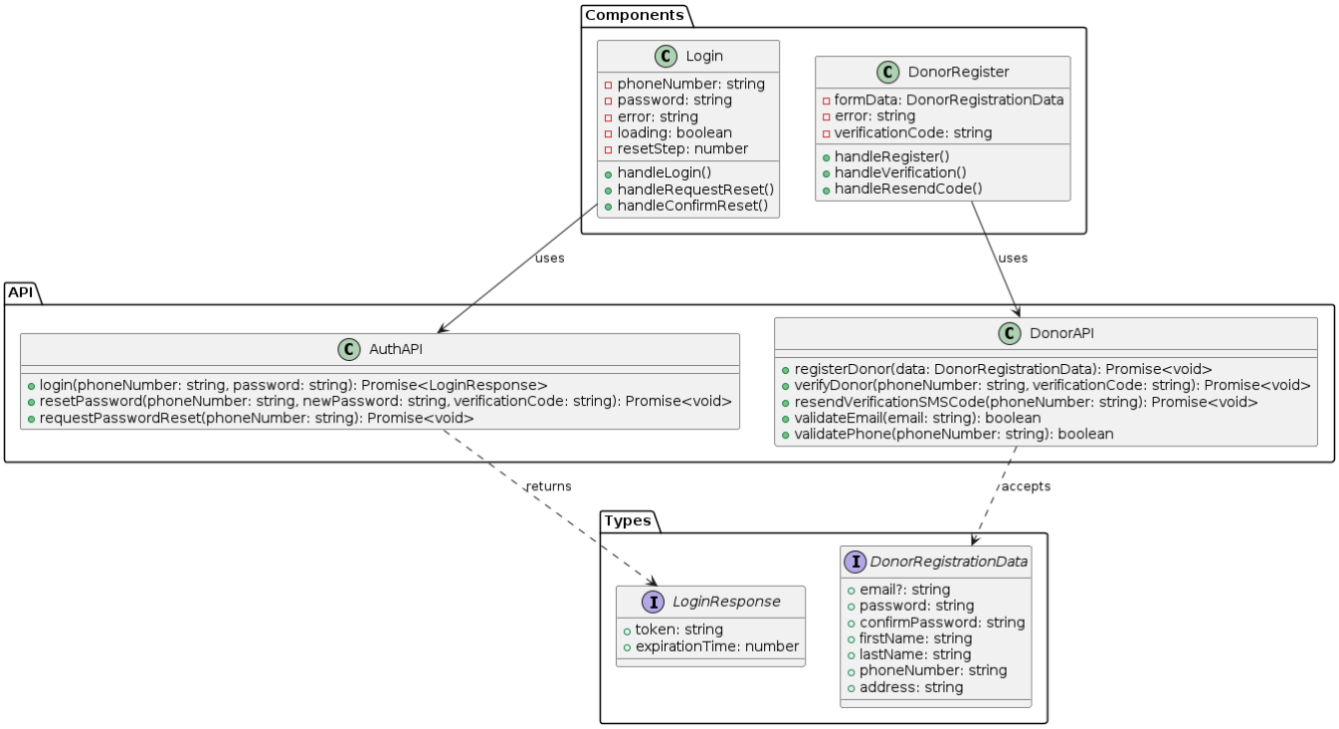
- Constraints View for Donor Page



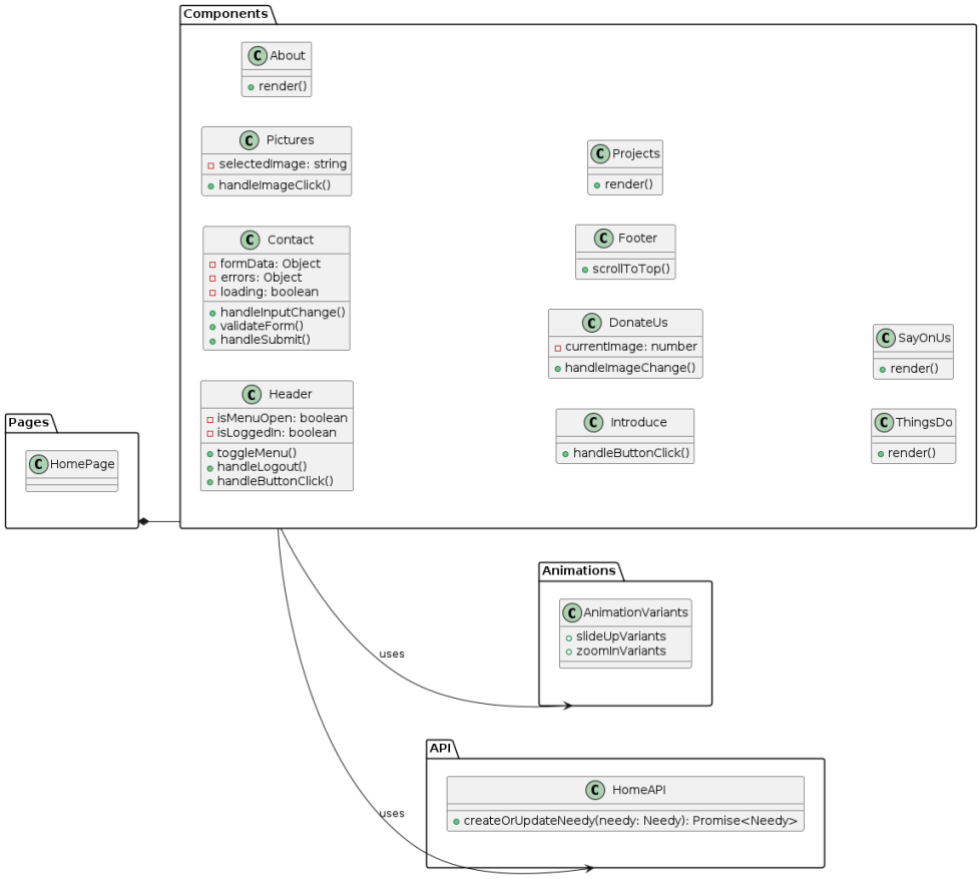
- Drivers Constraints Uploading Page



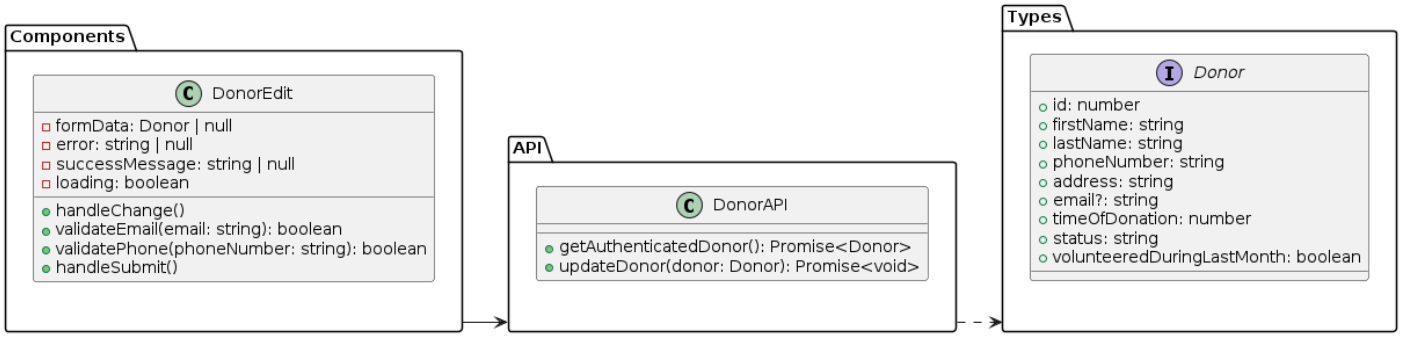
- Login and Registration Pages



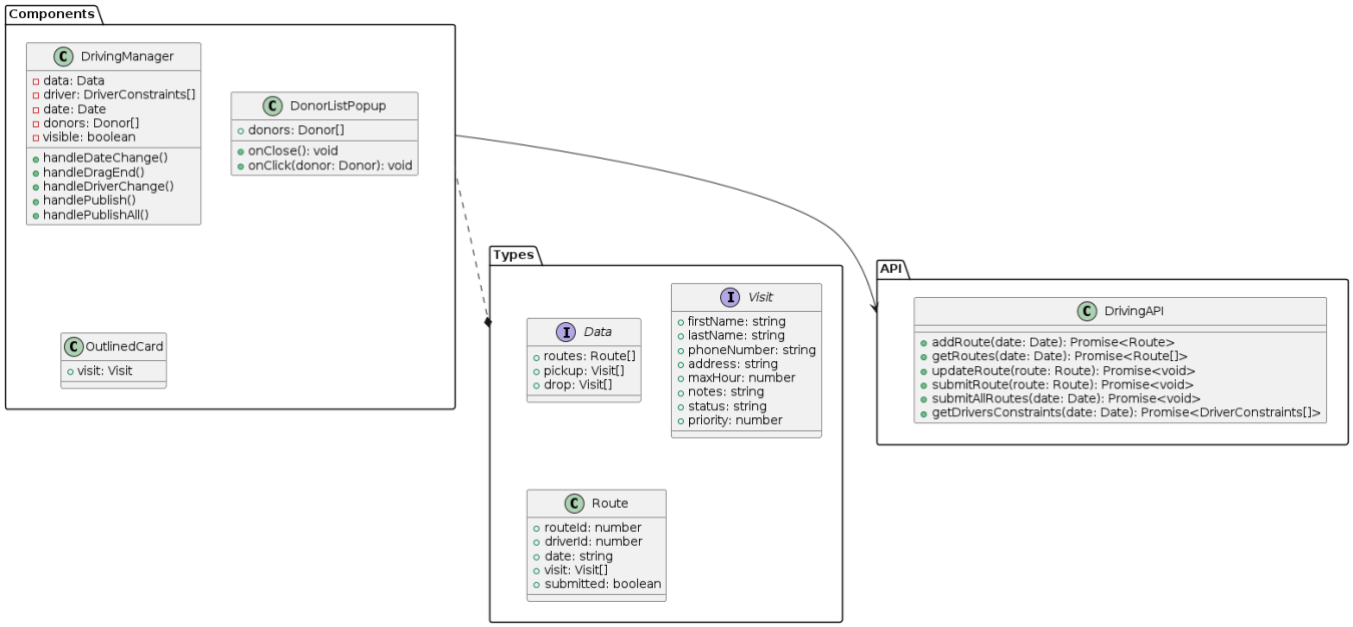
- Home Page



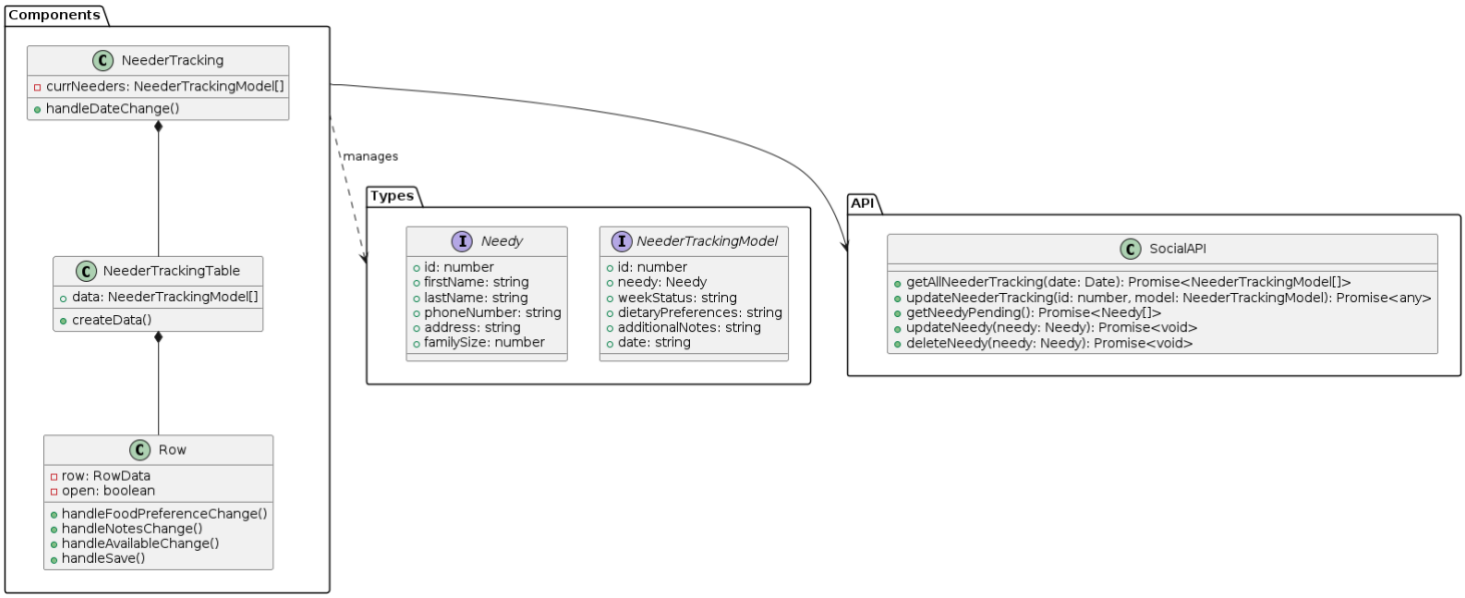
- User Change Details Page



- Driver Manager Page



- Social Manager Page



5.2 Class Description

5.2.1 Backed Structure

Most of the classes follow spring boot notations and follow spring boot architecture.

DrivingController: This class contains public methods and is responsible for receiving HTTP requests regarding the driver's repository.

DrivingFascade: This class acts as the service layer. The controller class calls this class, this class does all the algorithmic behavior and submits and request data from the different database tables.

DriverConstraint: This class represents the model that is being held in the database.

Route: This class represents the model that is being held in the database.

DrivertConstraintRepository: This class is responsible for communication with the database. It saves and fetches securely information from the database. This repository extends the spring boot JPA repository interface.

CookingController: This class contains public methods and is responsible for receiving HTTP requests regarding the cooking and chef's repository.

CookingService: This class acts as the service layer. The controller class calls this class, this class does all the algorithmic behavior and submits and request data from the different database tables.

CookConstraints: This class represents the model that is being held in the database.

CookConstraintsRepository: This class is responsible for communication with the database. It saves and fetches securely information from the database. This repository extends the spring boot JPA repository interface.

Map2JsonConverter: Since there are fields in the model that are stored as a "MAP", the system has to convert it into a json format to be able to save it in the database.

ApplicationConfig: This class is the base class for future security methods. It follows the spring boot design using "Bean" to implement one time each needed object in one place, instead of spreading it across the whole project.

EmailConfig: This class is the base for sending email messages to clients. It defines the username, password and different related protocols.

JwtAuthenticationFilter: This class is responsible for token handling, and setting the details to the token.

SecurityConfig: This class is responsible for the base authorization management. It sets the roles that are allowed to enter each package and each area at the site.

AuthController: This class contains public methods and is open to HTTP requests regarding login/logout, verifying tokens and new users, and all security methods overall.

AuthService: This class is the one that is responsible for the whole logic of verifying users and implementing security measures.

EmailService: This class is the one that is responsible for the whole logic of sending emails to clients when needed. It uses the setting that were defined in "EmailConfig".

JwtTokenService: This class is the one that is responsible for the whole logic of creating and deleting new tokens and verifying current token.
UserCredentials: This class is a model that holds sensitive data about the user. It implements the spring boot “UserDetails” class.
UserCredentialsRepository: This class is responsible for handling the userCredentials model and using it to insert and fetch data from the database. This repository extends the spring boot JPA repository interface.
NeederController: This class contains public methods and is responsible for receiving HTTP requests regarding needy users. It handles CRUD operations and tracking functionalities with proper error handling and cross-origin request support.
NeedyService: This class acts as the service layer for needy user operations. It handles business logic, synchronization of tracking data, and communicates with repositories for data persistence.
BaseUser: This abstract class serves as the base model for all user types. It defines common user attributes and uses Lombok annotations for automatic getter/setter generation.
Donor: This class extends BaseUser and represents the donor model in the database. It includes specific donor attributes like email, donation timing, and verification details with proper JPA annotations.
Needy: This class extends BaseUser and represents the needy user model in the database. It contains specific attributes for needy users like family size and confirmation status.
DonorRepository: This interface extends UserRepository and is responsible for donor-specific database operations. It provides methods for finding donors by various attributes. This repository extends the spring boot JPA repository interface.
NeedyRepository: This interface extends UserRepository and handles needy user database operations. It includes methods for finding needy users by status and phone number. This repository extends the spring boot JPA repository interface.
NeederTrackingController: This class handles HTTP requests for tracking needy users' attendance and dietary preferences. It implements CRUD operations with role-based access control and comprehensive error handling.
NeederTrackingService: This class manages business logic for needy user tracking, including attendance records, dietary preferences, and additional notes. It handles data validation and persistence through the repository layer.
NeederTracking: This class represents the model for tracking needy users' attendance and preferences in the database. It maintains relationships with the Needy entity and includes status tracking, dietary preferences, and notes.
NeederTrackingRepository: This interface extends JpaRepository for NeederTracking entity operations. This repository extends the spring boot JPA repository interface.

5.2.2 UI Structure

CookConstraintsRestAPI: This TypeScript module handles cooking-related API calls using Axios. It manages chef constraints, food requirements, and communicates with the backend server.

SubmitConstraints (data: CookConstraintsData): Promise<void>

- This method sends to the backend the new constraints that should be submitted to the user.
- Preconditions:
  - data.cookId must be positive.
  - data.startTime and data.endTime must be valid time strings.
  - data.constraints must not be empty.
  - localStorage must contain valid string.
- Postconditions:
  - Response status is 200 if successful.

GetFoodConstraints (date: string): Promise<Record<string, number>>

- This method fetches the food constraints for a specific date for a specific user.

- Preconditions:
  - date must be valid ISO date string.
  - localStorage must contain valid token.
- Postconditions:
  - Returns mapping of food types to quantities in a record form.
  - All quantity values are non-negative

CookManagerRestAPI: This TypeScript module handles cooking-related API calls using Axios for managing cooking constraints, requests, and donor interactions.

getAllRequests (date: string): Promise<PendingCookDTO[]>

- Fetches all pending cooking requests for a specific date.
- Preconditions:
  - date must be valid ISO string
  - localStorage must contain valid token
- Postconditions:
  - Returns array of pending cook requests with valid fields
  - Each request contains id, name, time fields, and constraints

approveCookRequest (constraintId: number): Promise<void>

- Approves a pending cooking request.

- Preconditions:
    - constraintId must be positive
    - localStorage must contain valid token
- Postconditions:
    - Request status updated to approved in backend
    - Returns void if successful

rejectCookRequest (constraintId: number): Promise<void>

- Rejects a pending cooking request.
- Preconditions:
    - constraintId must be positive
    - localStorage must contain valid token
- Postconditions:
    - Request status updated to rejected in backend
    - Returns void if successful

getFoodConstraints (date: string): Promise<Record<string, number>>

- Retrieves required food quantities for a specific date.
- Preconditions:
    - date must be valid ISO string
    - localStorage must contain valid token
- Postconditions:
    - Returns mapping of food types to required quantities
    - All quantities are non-negative

getAcceptedConstraints (date: string): Promise<Record<string, number>>

- Fetches and aggregates all accepted cooking constraints.
- Preconditions:
    - date must be valid ISO string
    - localStorage must contain valid token
- Postconditions:
    - Returns summed constraints for each food type

- All constraint values are non-negative

updateConstraint (id: number, constraints: Record<string, number>): Promise<void>

- Updates existing cooking constraints.
- Preconditions:
  - id must be positive
  - constraints must not be empty
  - localStorage must contain valid token
- Postconditions:
  - Constraints updated in backend
  - Returns void if successful

undoAction (constraintId: number): Promise<void>

- Reverts previous action on a constraint.
- Preconditions:
  - constraintId must be positive
  - localStorage must contain valid token
- Postconditions:
  - Action undone in backend
  - Returns void if successful

getDonorApproved (): Promise<Donor[]>

- Retrieves list of approved donors.
- Preconditions:
  - localStorage must contain valid token
- Postconditions:
  - Returns array of approved donors
  - Each donor has valid ID and approval status

addNewConstraint (donorId: number, date: string, startTime: string, endTime: string, constraints: Record<string, number>): Promise<void>

- Creates new cooking constraints for a donor.
- Preconditions:

    - donorId must be positive
    - date must be valid ISO string
    - startTime must be before endTime
    - constraints must not be empty
    - localStorage must contain valid token
- Postconditions:
    - New constraint created in backend
    - Returns void if successful

DonorRegRestAPI: TypeScript module handling donor registration operations.
validateEmail (email: string): boolean

- Validates email format using regex.
- Preconditions:
    - email must be string
- Postconditions:
    - Returns true if email matches pattern
    - Returns false otherwise

validatePhone (phoneNumber: string): boolean

- Validates Israeli phone format.
- Preconditions:
    - phoneNumber must be string
- Postconditions:
    - Returns true if matches 05XXXXXXXX pattern (The association is based in Beer Sheva. As such, there is no need at this stage to support international phone numbers outside of Israel. In future versions, this restriction may be removed to support broader geographical use).
    - Returns false otherwise

registerDonor (data: DonorRegistrationData): Promise<void>

- Registers new donor.
- Preconditions:
    - All required fields in data populated

    - Password matches confirmPassword
    - Valid email if provided
    - Valid phone number
- Postconditions:
    - Creates donor account if successful
    - Throws error if user exists

verifyDonor (phoneNumber: string, verificationCode: string): Promise<void>

- Verifies donor using SMS code.

- Preconditions:
    - Valid phone number
    - Non-empty verification code
- Postconditions:
    - Verifies donor if code correct
    - Throws error if verification fails

resendVerificationSMSCode (phoneNumber: string): Promise<void>

- Resends verification SMS.
- Preconditions:
    - Valid phone number
- Postconditions:
    - Sends new code if successful
    - Throws error if fails

LoginRestAPI: TypeScript module handling authentication operations.
Login (phoneNumber: string, password: string): Promise<LoginResponse>

- Authenticates user login.
- Preconditions:
    - Valid phone number
    - Non-empty password
- Postconditions:
    - Returns token and expiration if successful

- Throws error if credentials invalid

resetPassword (phoneNumber: string, newPassword: string, verificationCode: string): Promise<void>

- Resets user password.
- Preconditions:
  - Valid phone number
  - Non-empty new password
  - Valid verification code
- Postconditions:
  - Updates password if successful
  - Throws error if code invalid

requestPasswordReset (phoneNumber: string): Promise<void>

- Initiates password reset.
- Preconditions:
  - Valid phone number
- Postconditions:
  - Sends reset code if user exists
  - Throws error if request fails

ConstraintsViewRestAPI: TypeScript module managing API interactions for viewing driver and cook constraints.

getCookConstraints (): Promise<PendingCookDTO[]>

- Retrieves cook constraints for current date.
- Preconditions:
  - Valid token in localStorage
- Postconditions:
  - Returns array of cook constraints
  - Each constraint has valid id, times, and constraints fields

getDriverConstraints (): Promise<DriverConstraints []>

- Fetches driver constraints and filters based on routes.
- Preconditions:
  - Valid token in localStorage

    - Valid driver ID retrievable
- Postconditions:
    - Returns filtered array of driver constraints
    - Excludes constraints matching existing route dates

getDriverRoutes (date: string): Promise<DriverRoutes>

- Retrieves driver routes for specific date.
- Preconditions:
    - Valid token in localStorage
    - Valid date string
    - Valid driver ID retrievable
- Postconditions:
    - Returns driver routes with visits
    - Each visit contains valid address and status information

EditDonorDetailsRestAPI: TypeScript module for managing donor profile operations.

getAuthenticatedDonor (): Promise<Donor>

- Retrieves authenticated donor's profile data.
- Preconditions:
    - Valid token in localStorage
- Postconditions:
    - Returns donor data with all required fields
    - Throws error if authentication fails

updateDonor (donor: Donor): Promise<void>

- Updates donor's profile information.
- Preconditions:
    - Valid token in localStorage
    - All required donor fields populated
    - Valid donor ID
- Postconditions:

- Updates donor information in backend
- Throws error if update fails

DrivingRestAPI: TypeScript module for managing driving operations and routes.

getRoutes (date: Date): Promise<Route []>

- Fetches routes for given date.
- Preconditions:
  - Valid date object
  - Valid token in localStorage
- Postconditions:
  - Returns array of routes with valid IDs and visits

setDriverIdToRoute (routeId: number, driverId: number): Promise<Route>

- Assigns driver to route.
- Preconditions:
  - Valid routeId and driverId
  - Valid token in localStorage
- Postconditions:
  - Updates route with driver assignment
  - Returns updated route

addRoute (date: Date): Promise<Route>

- Creates new route.
- Preconditions:
  - Valid date object
  - Valid token in localStorage
- Postconditions:
  - Creates route in backend
  - Returns new route object

updateRoute (route: Route): Promise<void>

- Updates existing route.
- Preconditions:

    - Valid route object with ID
    - Valid token in localStorage
- Postconditions:
    - Updates route in backend

submitRoute/submitAllRoutes

- Submits single/all routes.
- Preconditions:
    - Valid route ID/date
    - Valid token in localStorage
- Postconditions:
    - Marks routes as submitted

DonorRestAPI: TypeScript module for donor management.
getPendingDonors (): Promise<Donor []>

- Fetches pending donor applications.
- Preconditions:
    - Valid token in localStorage
- Postconditions:
    - Returns array of pending donors

updateDonor/deleteDonor

- Manages donor records.
- Preconditions:
    - Valid donor object/ID
    - Valid token in localStorage
- Postconditions:
    - Updates/removes donor record

SocialRestAPI: TypeScript module for social tracking.
getAllNeederTracking/updateNeederTracking

- Manages needy tracking records.
- Preconditions:
    - Valid date/tracking object

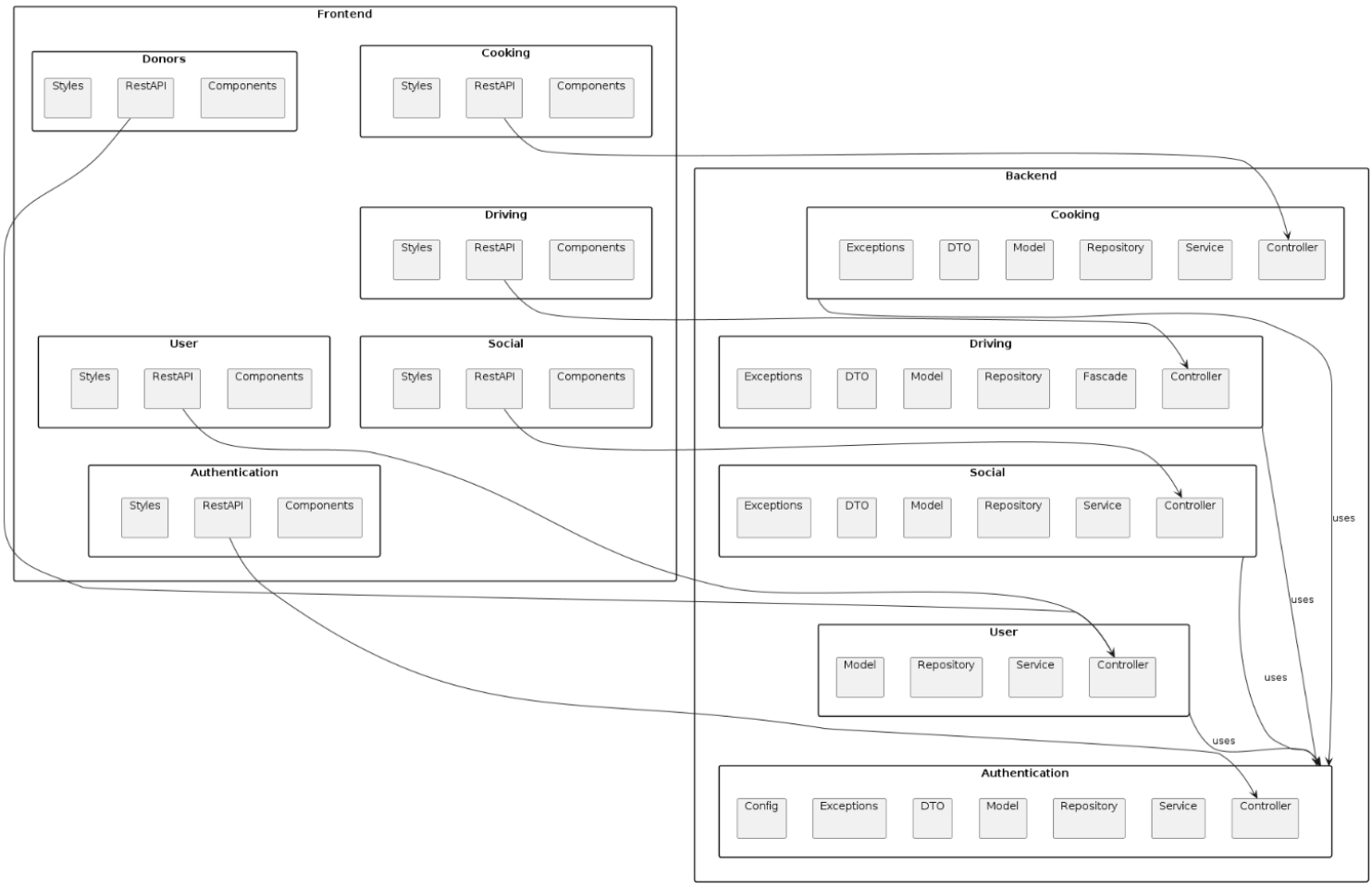
- Valid token in localStorage

- Postconditions:
  - Returns/updates tracking records

getNeedyPending/updateNeedy/deleteNeedy

- Manages needy applications.
- Preconditions:
  - Valid needy object/ID
  - Valid token in localStorage
- Postconditions:
  - Returns/updates/removes needy records

5.3 Packages



5.4 Unit Testing

We tested almost every controller, service, model and repository that we built with dedicated unit tests that aimed to test both success and failing situations with over 90% test coverage across the whole project.

AuthController Tests:

- Login Tests

Test Case	**Scenario**	**Expected Result**
Success	User provides valid phone number and password	- Status: 200 OK - Returns JWT token and expiration time
Failure	User provides invalid credentials	- Status: 400 Bad Request - Returns "Invalid credentials" message

- Donor Registration Tests

Test Case	**Scenario**	**Expected Result**
Success	New donor registration with valid details	- Status: 200 OK - Returns "Donor successfully registered" message
Failure	Registration with existing donor details	- Status: 409 Conflict - Returns "Donor already exists" message

- Needy Registration Tests

Test Case	**Scenario**	**Expected Result**
Success	New needy registration with valid details	- Status: 200 OK - Returns "Needy successfully registered" message

Failure	Registration with existing needy details	- Status: 400 Bad Request - Returns "Needy already exists" message

- Password Reset Tests

Test Case	**Scenario**	**Expected Result**
Success	Valid phone number and new password	- Status: 200 OK - Returns "Password reset successfully" message
Failure	Non-existent phone number	- Status: 400 Bad Request - Returns "User not found" message

- Donor Verification Tests

Test Case	**Scenario**	**Expected Result**
Success	Valid phone number and verification code	- Status: 200 OK - Returns "Donor successfully verified" message
Failure	Invalid verification code	- Status: 400 Bad Request - Returns "Invalid verification code" message

- Logout Tests

Test Case	**Scenario**	**Expected Result**
Success with Token	Valid JWT token in header	- Status: 200 OK - Returns "logged out successfully" message
Success without Token	No token provided	- Status: 200 OK - Returns "No token provided. nothing to invalidate" message
Failure	Invalid token	- Status: 400 Bad Request - Returns "Invalid token" message

- Request Password Reset Tests

Test Case	Scenario	Expected Result
Success	Valid phone number	- Status: 200 OK - Returns "Verification code sent to your phone number" message
Failure	Invalid phone number	- Status: 400 Bad Request - Returns "Phone number not found" message

- Confirm Password Reset Tests

Test Case	Scenario	Expected Result
Success	Valid verification code and new password	- Status: 200 OK - Returns "Password has been reset successfully" message
Failure	Invalid verification code	- Status: 401 Unauthorized - Returns "Invalid verification code" message

- Resend Verification Code Tests

Test Case	Scenario	Expected Result
Success	Valid email address	- Status: 200 OK - Returns "Email code resend successfully" message
Failure	Invalid email address	- Status: 400 Bad Request - Returns "Email not found" message

- Get User Role Tests

Test Case	Scenario	Expected Result
Success	Valid JWT token	- Status: 200 OK - Returns user role (e.g., "ADMIN")
Failure	Invalid token	- Status: 400 Bad Request - Returns "Invalid token" message

- Get User ID Tests

Test Case	Scenario	Expected Result

Success	Valid JWT token	- Status: 200 OK - Returns user ID
Failure	Invalid token	- Status: 400 Bad Request - Returns "Invalid token" message

- UserCredentialsRepoTest

Test Case	**Scenario**	**Expected Result**
shouldFindByEmail	Search for user by valid phone number	- Successfully finds user - Returns user with matching phone number
shouldReturnCredentialsByEmail	Get credentials by valid phone number	- Returns non-null user credentials - Returns credentials with matching phone number
shouldCheckIfEmailExists	Check if phone number exists	- Returns true for existing phone number
shouldReturnEmptyWhenEmailDoesNotExist	Search for non-existent phone number	- Returns empty Optional
shouldReturnFalseIfEmailDoesNotExist	Check if non-existent phone number exists	- Returns false

- EmailServiceTest

Test Case	**Scenario**	**Expected Result**
testSendVerificationEmail_Success	Send verification email with valid parameters	- Email is created and sent successfully - Verifies email sender interactions

testSendVerificationEmail_ThrowsException	Email sending fails	- Throws MessagingException - Verifies email sender attempted to send

- JwtTokenServiceTest

Test Case	**Scenario**	**Expected Result**
testGenerateToken	Generate JWT token with user details	- Returns non-null, non-empty token - Token contains correct username
testIsTokenValid_ValidToken	Validate a legitimate token	- Returns true for valid token and matching user details
testExtractClaim	Extract specific claim from token	- Successfully extracts and matches role claim
testGenerateTokenWithClaims	Generate token with custom claims	- Token contains correct custom claims - Claims can be extracted correctly
testIsTokenValid_NullToken	Validate null token	- Returns false
testIsTokenValid_NullUserDetails	Validate token with null user details	- Returns false
testExtractUsername	Extract username from token	- Returns correct username from token

- UserCredentialsTest

Test Case	**Scenario**	**Expected Result**
testGetAuthorities_WithValidDonorAndRole	Get authorities for valid donor	- Returns one authority with correct role

testGetAuthorities_WithNullDonor	Get authorities with no donor	- Returns empty collection
testGetAuthorities_WithNullUserCredentialsInDonor	Get authorities with null credentials	- Returns empty collection
testIsAccountNonExpired	Check if account is expired	- Returns true (accounts don't expire)
testIsAccountNonLocked	Check if account is locked	- Returns true (accounts don't lock)
testIsCredentialsNonExpired	Check if credentials expired	- Returns true (credentials don't expire)
testIsEnabled_WhenDonorIsVerified	Check enabled status for verified donor	- Returns true
testIsEnabled_WhenDonorIsNotVerified	Check enabled status for unverified donor	- Returns false
testSettersAndGetters	Test basic property setters/getters	- All properties are set and retrieved correctly

- AuthServiceTest

Test Case	**Scenario**	**Expected Result**
testAuthenticate_Success	Authenticate with valid credentials	- Returns valid user credentials - Successfully authenticates

authenticate_UserNotFound	Authenticate non-existent user	- Throws InvalidCredentialException
registerDonor_ValidRequest	Register new donor	- Successfully registers donor - Saves donor and credentials
registerDonor_DonorAlreadyExists	Register existing donor	- Throws UserAlreadyExistsException
registerNeedy_ValidRequest	Register new needy	- Successfully registers needy
registerNeedy_NeedyAlreadyExists	Register existing needy	- Throws UserAlreadyExistsException
confirmPasswordReset_ValidRequest	Reset password with valid code	- Successfully resets password - Updates credentials
resetPassword_ValidRequest	Reset password directly	- Successfully updates password
sendVerificationEmail_Success	Send verification email	- Successfully sends email
verifyDonor_Success	Verify donor with valid code	- Successfully verifies donor
getUserRoleFromJWT_Success	Get role from valid JWT	- Returns correct user role
getUserIDFromJWT_Success	Get ID from valid JWT	- Returns correct user ID
generateVerificationCode_ValidRequest	Generate verification code	- Returns valid 6-digit code

- CookControllerTests

Test Case	Scenario	Expected Result
removeConstraint_Success	Remove valid constraint with admin role	- Status: 200 OK - Constraint successfully removed
removeConstraint_Failure	Remove non-existent constraint	- Status: 400 Bad Request
updateConstraint_Success	Update valid constraint with new values	- Status: 200 OK - Constraint successfully updated
updateConstraint_Failure	Update fails due to runtime error	- Status: 400 Bad Request
getCookConstraints_Success	Get constraints for valid cook ID	- Status: 200 OK - Returns list of constraints
getCookConstraints_Failure	Get constraints for invalid cook ID	- Status: 400 Bad Request
getCookHistory_Success	Get history for valid cook ID	- Status: 200 OK - Returns cook's constraint history
getCookHistory_Failure	Get history for invalid cook ID	- Status: 400 Bad Request
getAllAcceptedConstraintsByDate_Success	Get accepted constraints for date	- Status: 200 OK - Returns list of accepted constraints
getAllAcceptedConstraintsByDate_Failure	Get accepted constraints fails	- Status: 400 Bad Request
getPendingConstraintsByDate_Success	Get pending constraints for date	- Status: 200 OK - Returns list of pending constraints
getPendingConstraintsByDate_Failure	Get pending constraints fails	- Status: 400 Bad Request

acceptConstraintStatus_Success	Accept a pending constraint	- Status: 200 OK - Constraint status changed to accepted
acceptConstraintStatus_Failure	Accept constraint fails	- Status: 400 Bad Request
rejectConstraintStatus_Success	Reject a pending constraint	- Status: 200 OK - Constraint status changed to declined
rejectConstraintStatus_Failure	Reject constraint fails	- Status: 400 Bad Request
undoConstraint_Success	Undo constraint status change	- Status: 200 OK - Constraint status reset to pending
undoConstraint_Failure	Undo constraint fails	- Status: 400 Bad Request
addConstraint_Success	Add new constraint	- Status: 200 OK - New constraint added successfully
addConstraint_Failure	Add constraint fails	- Status: 400 Bad Request

- CookingServiceTests

Test Case	**Scenario**	**Expected Result**
getDonorIdFromJwt_Success	Get donor ID from valid JWT	- Returns correct donor ID
getDonorIdFromJwt_InvalidToken	Get donor ID with invalid token	- Throws RuntimeException
getDonorFromId_Success	Get donor with valid ID	- Returns correct donor object

getDonorFromId_NotFound	Get donor with invalid ID	- Throws UserDoesntExistsException
submitConstraints_WithToken_Success	Submit constraints with valid token	- Returns saved constraint object
submitConstraints_WithToken_InvalidToken	Submit with invalid token	- Throws RuntimeException
submitConstraints_Direct_Success	Direct constraint submission	- Returns saved constraint object
submitConstraints_UserDTO_Success	Submit via UserDTO	- Returns saved constraint object
submitConstraints_UserDTO_UserNotFound	Submit with invalid user	- Throws UserDoesntExistsException
updateConstraint_Success	Update existing constraint	- Constraint updated with new values
updateConstraint_ConstraintNotFound	Update non-existent constraint	- Throws IndexOutOfBoundsException
removeConstraint_Success	Remove existing constraint	- Constraint successfully removed
removeConstraint_NotFound	Remove non-existent constraint	- Throws CookConstraintsNotExistException
getCookConstraints_Success	Get constraint	- Returns list of constraints

	s for valid cook	
getCookConstraints_Empty	Get constraints with no results	- Returns empty list
getLatestCookConstraints_Success	Get recent constraints	- Returns list of future constraints
getLatestCookConstraints_InvalidToken	Get recent with invalid token	- Throws RuntimeException
getAcceptedCookByDate_Success	Get accepted constraints	- Returns list of accepted constraints
getAcceptedCookByDate_Empty	Get accepted with no results	- Returns empty list
getPendingConstraints_Success	Get pending constraints	- Returns list of pending constraints
getPendingConstraints_Empty	Get pending with no results	- Returns empty list
getConstraintsByDate_Success	Get constraints by date	- Returns list of constraints
getConstraintsByDate_Empty	Get constraints with no results	- Returns empty list

changeStatusForConstraint_Success	Change constraint status	- Returns updated constraint
changeStatusForConstraint_NotFound	Change status of non-existent constraint	- Throws CookConstraintsNotExistException

- ConstraintMapperTest

Test Case	**Scenario**	**Expected Result**
toDTO	Map valid constraint to DTO	- All fields correctly mapped - Returns complete DTO object
toDTO_WithNullValues	Map constraint with null values	- DTO created with null fields - No exceptions thrown
toDTO_WithEmptyConstraints	Map constraint with empty constraints	- DTO created with empty constraints map - Map exists but is empty

- DrivingControllerTest

Test Case	**Scenario**	**Expected Result**
submitConstraint_Success	Submit valid driver constraint	- Status: 200 OK - Constraint saved successfully
submitConstraint_Failure	Submit invalid constraint	- Status: 400 Bad Request

removeConstraint_Success	Remove existing constraint	- Status: 200 OK - Constraint removed successfully
removeConstraint_Failure	Remove non-existent constraint	- Status: 400 Bad Request
getDateConstraints_Success	Get constraints for valid date	- Status: 200 OK - Returns list of constraints
getDateConstraints_Failure	Get constraints with invalid date	- Status: 400 Bad Request
getDriverConstraints_Success	Get constraints for valid driver	- Status: 200 OK - Returns driver's constraints
getDriverConstraints_Failure	Get constraints for invalid driver	- Status: 400 Bad Request
submitRouteForDriver_Success	Submit valid route for driver	- Status: 200 OK - Route submitted successfully
submitRouteForDriver_Failure	Submit invalid route	- Status: 400 Bad Request
createRoute_Success	Create new route	- Status: 200 OK - Route created successfully

createRoute_Failure	Create route with invalid data	- Status: 400 Bad Request
setDriverIdToRoute_Success	Assign driver to route	- Status: 200 OK - Driver assigned successfully
setDriverIdToRoute_Failure	Assign with invalid data	- Status: 400 Bad Request
removeRoute_Success	Remove existing route	- Status: 200 OK - Route removed successfully
removeRoute_Failure	Remove non-existent route	- Status: 400 Bad Request
submitAllRoutes_Success	Submit all routes for date	- Status: 200 OK - All routes submitted
submitAllRoutes_Failure	Submit with invalid data	- Status: 400 Bad Request
getRoute_Success	Get route by ID	- Status: 200 OK - Returns route details
getRoute_Failure	Get non-existent route	- Status: 400 Bad Request
getRouteByDateAndDriver_Success	Get route for specific driver and date	- Status: 200 OK - Returns matching route

getRouteByDateAndDriver_Failure	Get with invalid parameters	- Status: 400 Bad Request
viewHistory_Success	Get all route history	- Status: 200 OK - Returns historical routes
viewHistory_Failure	History retrieval fails	- Status: 400 Bad Request
getRoutes_Success	Get routes for valid date	- Status: 200 OK - Returns list of routes
getRoutes_Failure	Get routes with invalid date	- Status: 400 Bad Request
getDriverFutureConstraintsHaventConfirmed_Success	Get unconfirmed future constraints	- Status: 200 OK - Returns list of constraints
getDriverFutureConstraintsHaventConfirmed_Failure	Invalid driver ID	- Status: 400 Bad Request
updateRoute_Success	Update existing route	- Status: 200 OK - Route updated successfully
updateRoute_Failure	Update with invalid data	- Status: 400 Bad Request

---

- DriverConstraintTest

Test Case	Scenario	Expected Result
saveAndFindDriverConstraint	Save and retrieve constraint	- Constraint saved successfully - All fields match expected values
driverConstraintNotFound	Find non-existent constraint	- Returns empty Optional
deleteDriverConstraint	Delete existing constraint	- Constraint successfully deleted - Not found after deletion
findConstraintsByDate	Find constraints by date	- Returns correct constraints - All fields match expected values
findConstraintsByDriverId	Find constraints by driver	- Returns correct constraints - All fields match expected values
findConstraintsByDriverIdNotFound	Find constraints for non-existent driver	- Returns empty list
findConstraintsByDateNotFound	Find constraints for date with no entries	- Returns empty list

---

- RouteTest

Test Case	Scenario	Expected Result
createFindRoute	Create and find route	- Route saved successfully - Retrieved route matches saved data
deleteRoute	Delete existing route	- Route successfully deleted
updateRoute	Update route details	- Route updated successfully - Changes persisted correctly

findAllRoute	Find all routes	- Returns correct number of routes
findByDate	Find routes by date	- Returns routes for specific date
submitRoute	Submit route	- Route marked as submitted
addVisit	Add visit to route	- Visit added successfully - All visit details correct
removeVisit	Remove visit from route	- Visit removed successfully
setVisit	Set new visit list	- Visit list updated successfully
setters	Test all setter methods	- All fields updated correctly - Changes persisted

---

- DrivingFascadeTest

Test Case	**Scenario**	**Expected Result**
submitConstraint	Submit new constraint	- Constraint saved successfully - Returns saved constraint
removeConstraint	Remove existing constraint	- Constraint removed successfully
removeConstraintNotFound	Remove non-existent constraint	- Throws DriverConstraintsNotExistException
getDateConstraints	Get constraints for date	- Returns correct constraints list
getDriverConstraints	Get driver's constraints	- Returns driver's constraints list
submitRouteForDriver	Submit route for driver	- Route marked as submitted
submitRouteForDriverNotFound	Submit non-existent route	- Throws RouteNotFoundException

createRoute	Create new route	- Route created successfully
setDriverIdToRoute	Assign driver to route	- Driver ID set correctly
removeRoute	Remove existing route	- Route removed successfully
getRoute	Get route by ID	- Returns correct route
getHistory	Get route history	- Returns all routes
submitAllRoutes	Submit all routes for date	- All routes marked as submitted
addVisit	Add visit to route	- Visit added successfully
removeVisit	Remove visit from route	- Visit removed successfully
getDriverFutureConstraintsHaventConfirmed	Get unconfirmed constraints	- Returns correct constraints
getRoutesByDriverId	Get routes for driver	- Returns correct routes list
getRoutes	Get routes for date	- Returns correct routes list
updateRoute	Update route details	- Route updated successfully

- NeederTrackingControllerTest

Test Case	**Scenario**	**Expected Result**

getAllNeeders	Get list of all needer trackings	- Status: 200 OK - Returns list with correct fields
getAllNeeders_EmptyList	Get empty list of needer trackings	- Status: 200 OK - Returns empty list
addNeeder	Add new needer tracking	- Status: 201 Created - Returns created tracking
addNeeder_InvalidInput	Add invalid needer tracking	- Status: 400 Bad Request
getNeederTrackingById	Get tracking by valid ID	- Status: 200 OK - Returns correct tracking
getNeederTrackingById_NotFound	Get non-existent tracking	- Status: 400 Bad Request
updateNeederTracking	Update existing tracking	- Status: 200 OK - Returns updated tracking
updateNeederTracking_NotFound	Update non-existent tracking	- Status: 400 Bad Request
deleteNeederTracking	Delete existing tracking	- Status: 204 No Content
deleteNeederTracking_NotFound	Delete non-existent tracking	- Status: 400 Bad Request
getNeedersHere	Get present needers for date	- Status: 200 OK - Returns list of present needers
getNeedersHere_EmptyResult	Get present needers with no results	- Status: 200 OK - Returns empty list
getAllNeederTrackingsByDate	Get trackings for specific date	- Status: 200 OK - Returns list of trackings

getAllNeederTrackingsByDate_EmptyResult	Get trackings with no results	- Status: 200 OK - Returns empty list
getAllNeededFoodByDate	Get food requirements by date	- Status: 200 OK - Returns food counts by type
getAllNeededFoodByDate_EmptyList	Get food requirements with no data	- Status: 200 OK - Returns empty map

- NeederTrackingRepositoryTest

Test Case	**Scenario**	**Expected Result**
saveAndFindNeeder	Save and retrieve needer tracking	- Tracking saved successfully - All fields match expected values
findByWeekStatus	Find trackings by week status	- Returns correct number of trackings - All have specified status
deleteNeederTracking	Delete existing tracking	- Tracking successfully deleted - Not found after deletion
updateNeederTracking	Update existing tracking	- Updates saved successfully - New values persisted
entityGraphFindById	Find tracking with needy data	- Returns tracking with associated needy - All relationships loaded
findByDate	Find trackings by date	- Returns trackings for specific date - Date matches expected value

- NeederTrackingServiceTest

Test Case	**Scenario**	**Expected Result**
getNeederById_Success	Get tracking by valid ID	- Returns correct tracking - All fields match expected values

getNeederById_NotFound	Get non-existent tracking	- Throws NeederTrackingNotFoundException
addNeeder	Add new tracking	- Tracking saved successfully - Returns saved tracking
updateNeederTrack_Success	Update existing tracking	- Updates applied successfully - Returns updated tracking
updateNeederTrack_NotFound	Update non-existent tracking	- Throws NeederTrackingNotFoundException
deleteNeederTrack_Success	Delete existing tracking	- Tracking deleted successfully
deleteNeederTrack_NotFound	Delete non-existent tracking	- Throws NeederTrackingNotFoundException
getAllNeedersTrackings	Get all trackings	- Returns list of all trackings
getNeedersHereByDate	Get present needers for date	- Returns list of present needers - Correct status filter applied
getAllNeedersTrackingsByDate	Get trackings for date	- Returns trackings for specific date
getNeedyFromNeederTrackingId_Success	Get needy info from tracking	- Returns correct needy data - Maps to DTO properly

getNeedyFromNeederTrackingId_NotFound	Get needy from invalid tracking	- Throws NeederTrackingNotFoundException
addNeederTracking_Success	Add new tracking with needy	- Tracking saved successfully - Correct relationships established

- DonorRepoTest

Test Case	Scenario	Expected Result
testFindByEmail	Find donor by valid email	- Returns correct donor - All fields match expected values
testFindByVerificationCode	Find donor by verification code	- Returns correct donor - All fields including verification code match

---

- NeedyRepoTest

Test Case	Scenario	Expected Result
testFindByPhoneNumber	Find needy by valid phone number	- Returns correct needy user - All fields match expected values
testFindByConfirmStatus	Find needy by confirmation status	- Returns list of needy users with matching status - Correct count and field values

---

- UserControllerTest

Test Case	Scenario	Expected Result
getAllUsers	Get all users (donors and needy)	- Status: 200 OK - Returns combined list of all users

getAllDonors	Get all donors	- Status: 200 OK - Returns list of donors
getAllNeedyUsers	Get all needy users	- Status: 200 OK - Returns list of needy users
getNeedyByPhoneNumber_Success	Get needy by valid phone	- Status: 200 OK - Returns correct needy user
getNeedyByPhoneNumber_NotFound	Get needy by invalid phone	- Status: 400 Bad Request
getDonorByPhoneNumber_Success	Get donor by valid phone	- Status: 200 OK - Returns correct donor
getDonorByPhoneNumber_NotFound	Get donor by invalid phone	- Status: 400 Bad Request
getDonorById_Success	Get donor by valid ID	- Status: 200 OK - Returns correct donor
getDonorById_NotFound	Get donor by invalid ID	- Status: 400 Bad Request
getDonorPending	Get pending donors	- Status: 200 OK - Returns list of pending donors
getDonorApproved	Get approved donors	- Status: 200 OK - Returns list of approved donors
updateDonor_Success	Update existing donor	- Status: 204 No Content
updateDonor_Failure	Update non-existent donor	- Status: 400 Bad Request
deleteDonor_Success	Delete existing donor	- Status: 204 No Content
deleteDonor_Failure	Delete non-existent donor	- Status: 400 Bad Request

- UserServiceTest

Test Case	Scenario	Expected Result
getAllDonors	Get list of all donors	- Returns complete list of donors
getAllNeedyUsers	Get list of all needy users	- Returns complete list of needy users
getAll	Get all users (donors and needy)	- Returns combined list of all users
getNeedyByPhoneNumber_Success	Find needy by valid phone	- Returns correct needy user
getNeedyByPhoneNumber_NotFound	Find needy by invalid phone	- Throws UserDoesntExistsException
getDonorByPhoneNumber_Success	Find donor by valid phone	- Returns correct donor
getDonorByPhoneNumber_NotFound	Find donor by invalid phone	- Throws UserDoesntExistsException
getDonorById_Success	Find donor by valid ID	- Returns correct donor
getDonorById_NotFound	Find donor by invalid ID	- Throws UserDoesntExistsException
getDonorsPending	Get list of pending donors	- Returns list of pending donors
getDonorsApproved	Get list of approved donors	- Returns list of approved donors
updateDonor_Success	Update existing donor	- Updates donor successfully
updateDonor_NotFound	Update non-existent donor	- Throws UserDoesntExistsException
deleteDonor_Success	Delete existing donor	- Deletes donor successfully

deleteDonor_NotFound	Delete non-existent donor	- Throws UserDoesntExistsException
setDonationToDonor_Success	Set donation date for donor	- Updates donation date successfully
setDonationToDonor_NotFound	Set donation for non-existent donor	- Throws UserDoesntExistsException

---

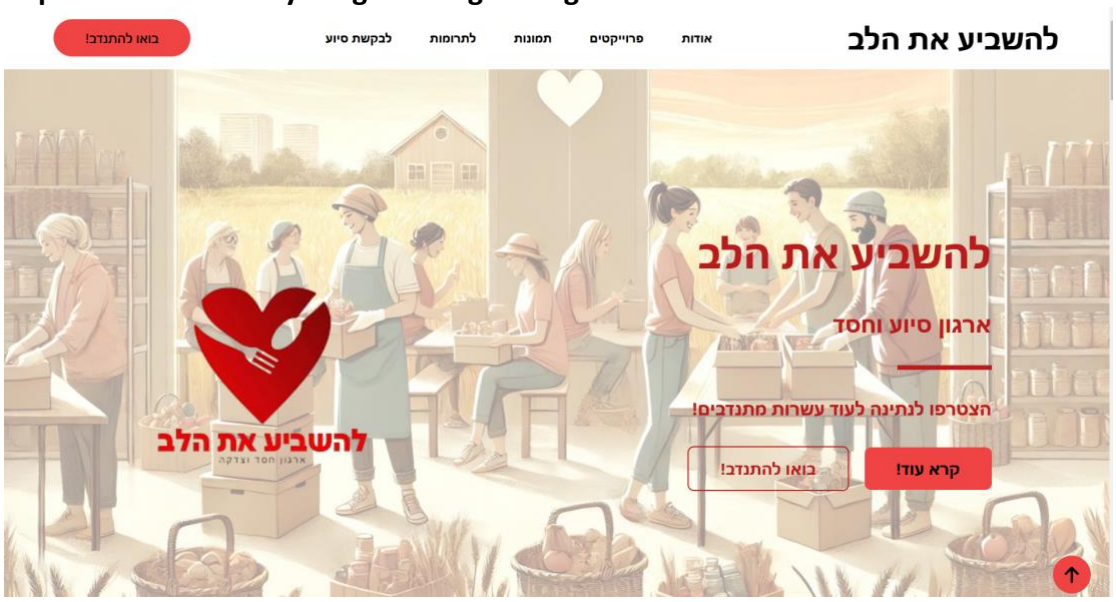
- NeedyServiceTest

Test Case	**Scenario**	**Expected Result**
getPendingNeedy	Get pending needy users	- Returns list of pending needy users
saveOrUpdateNeedy	Save new needy user	- Returns saved needy user
getAllNeedyUsers	Get all needy users	- Returns list of all needy users
getNeedyById	Get needy by valid ID	- Returns correct needy user
deleteNeedyById_Success	Delete existing needy	- Deletes successfully
deleteNeedyById_Failure	Delete non-existent needy	- Throws IllegalArgumentException
getNeedyByPhoneNumber	Get needy by phone	- Returns correct needy user

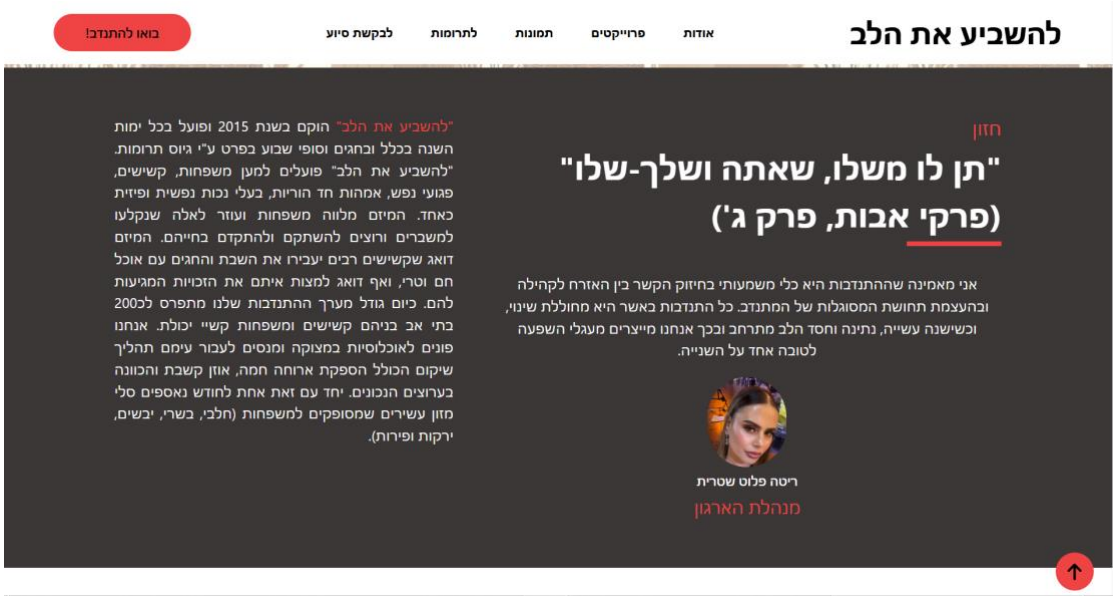
getNeedyUsersTrackingByData_AllApprovedHaveTracking	Get tracking data	- Returns tracking data for all approved users
getNeedyUsersTrackingByData_SomeApprovedMissingTracking	Get tracking with missing data	- Creates missing tracking entries
getNeedyUsersTrackingByData_NoApprovedNeedy	Get tracking with no approved users	- Returns empty list

Chapter 6 – User Interface Draft

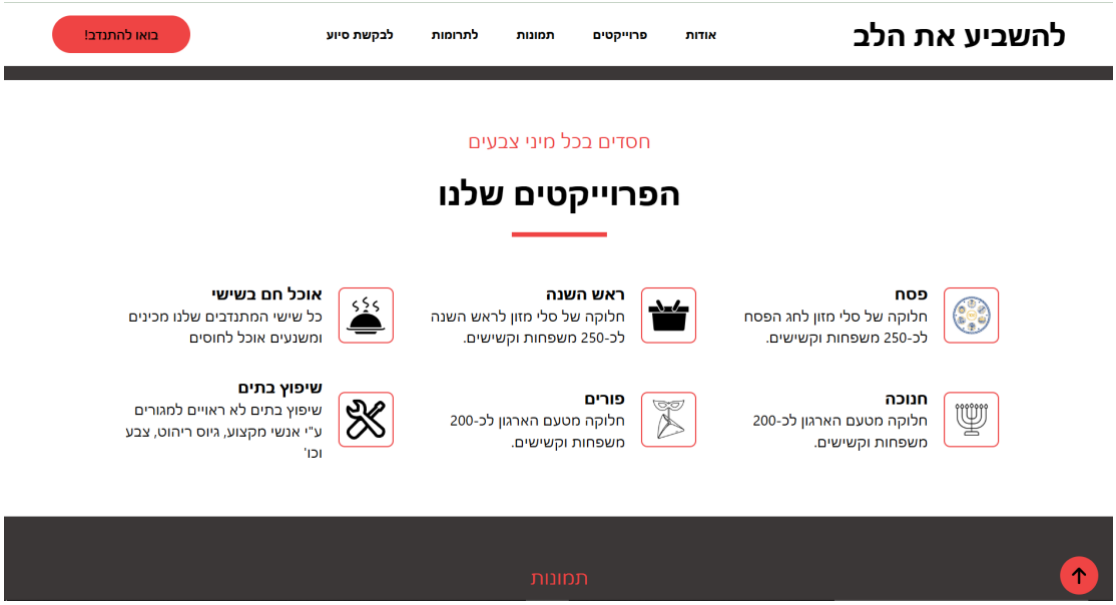
Home Screen: Here we display all the components of the home screen as it's impossible to fit everything in a single image



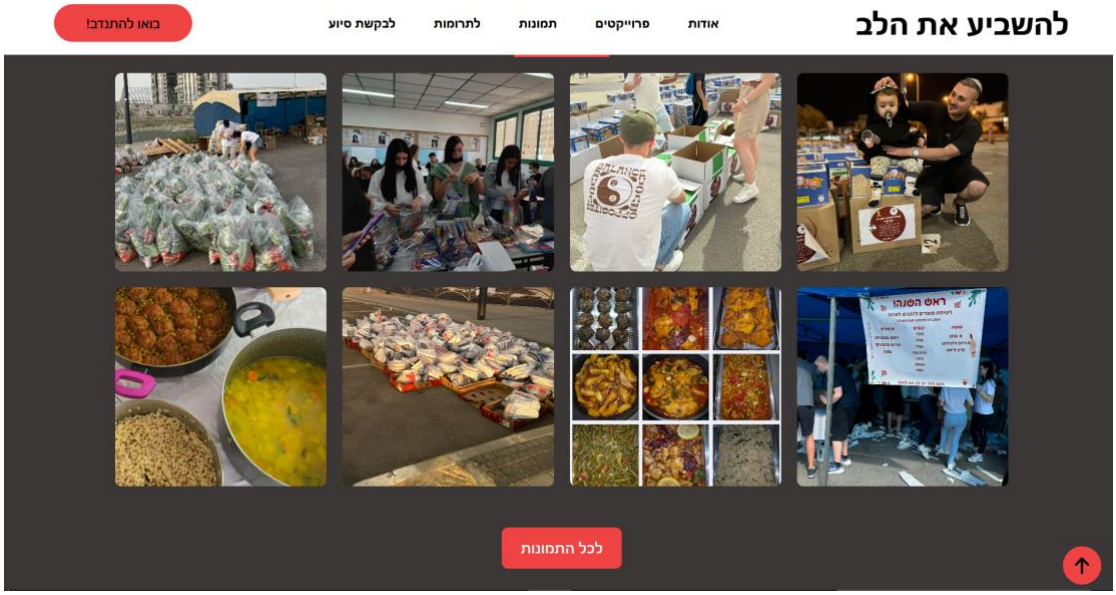
Here is an explanation about the organization, and the director writes her vision.



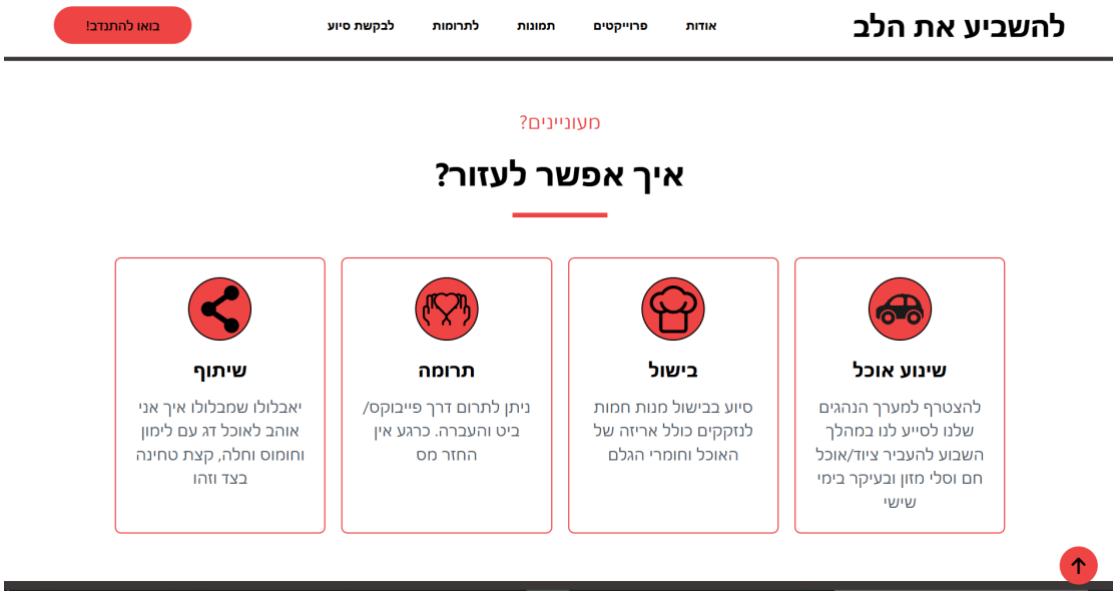
Here the organization's activities are described.



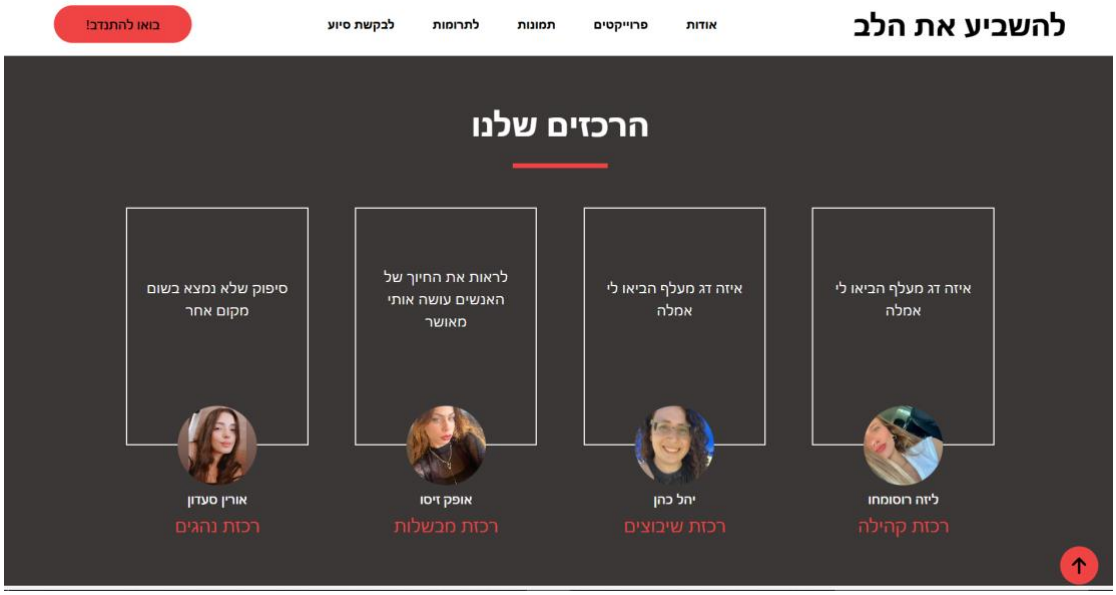
Here are photos from the organization's activities.



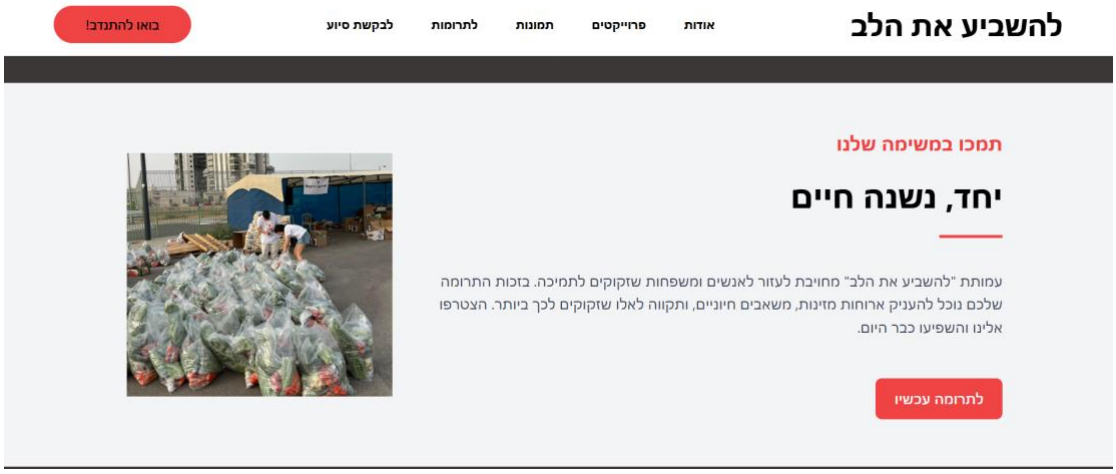
Here we show the user how they can help the organization.



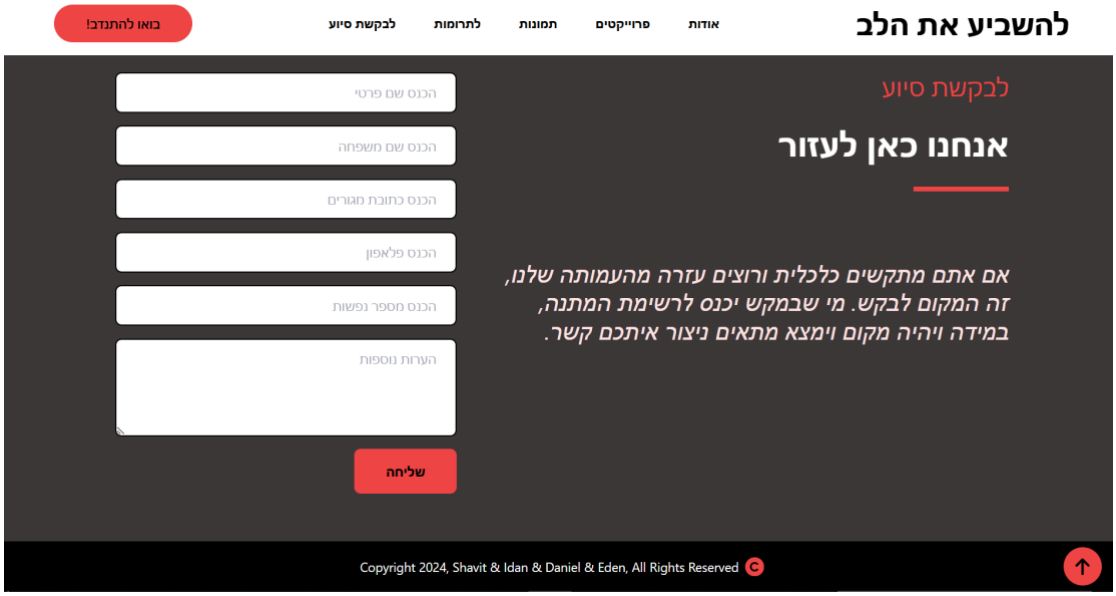
Here the organization's team is presented, along with their statements about the organization.



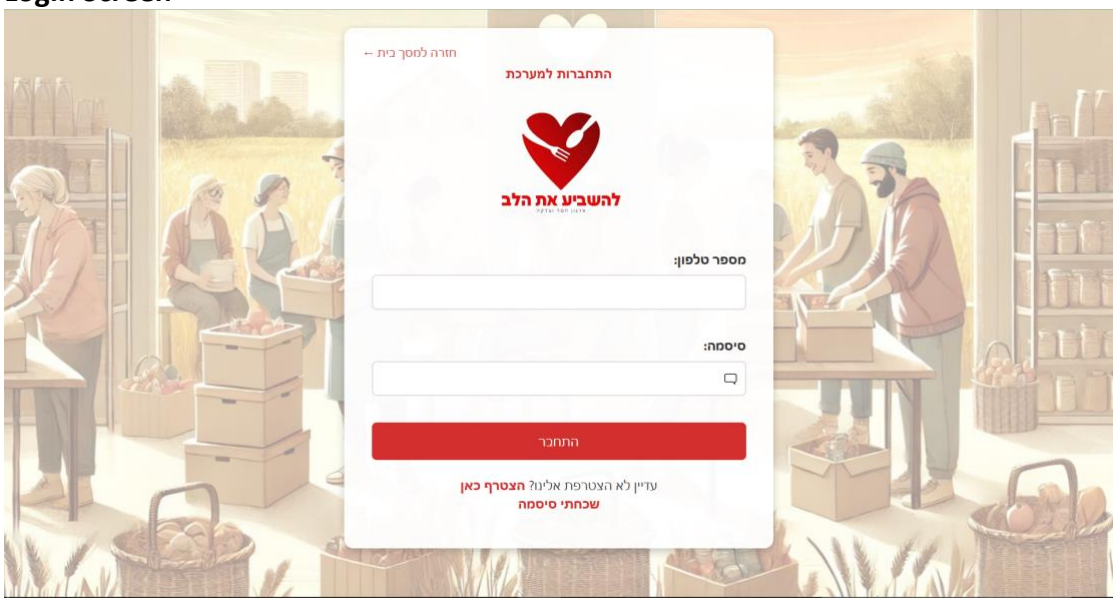
From here, we provide others the opportunity to support the organization by making a financial donation.



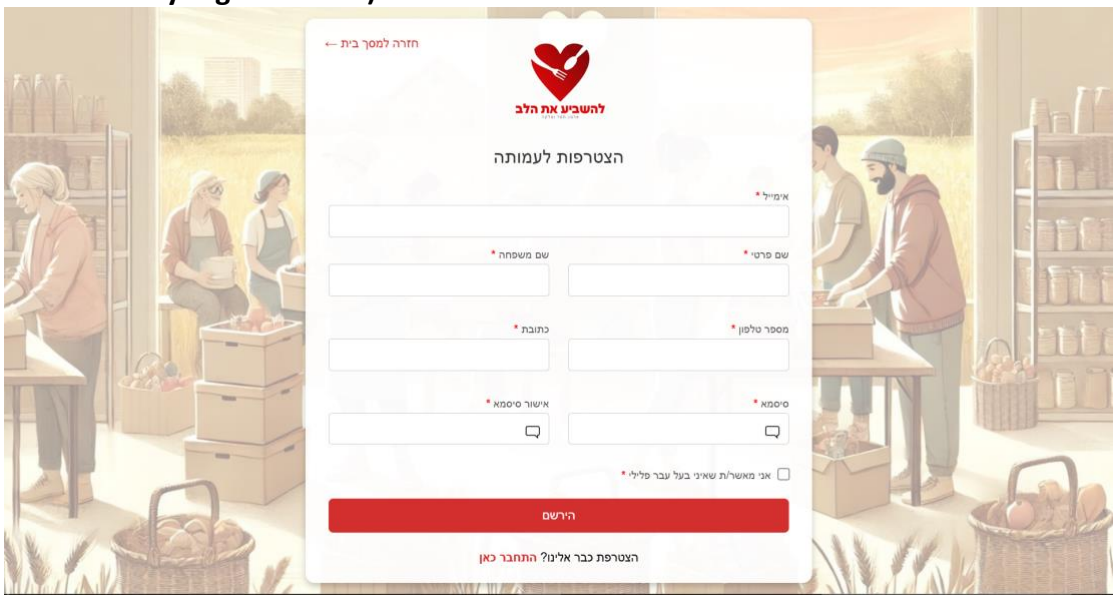
A form intended for those in need who seek assistance from the organization. The details are sent to the coordinator (shown later on another screen) to decide whether to approve the request.



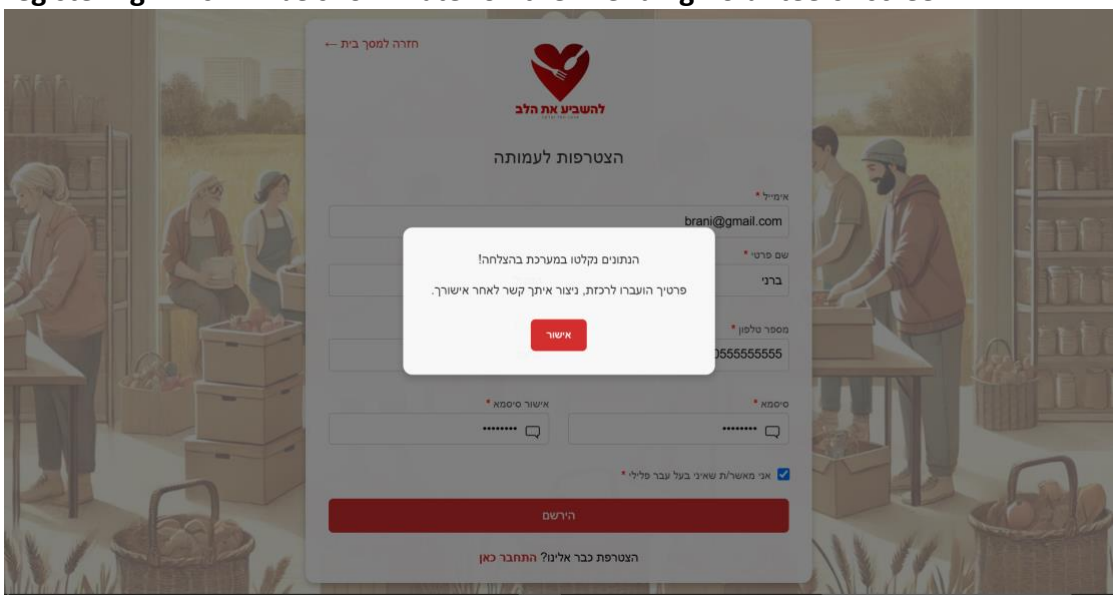
Login Screen



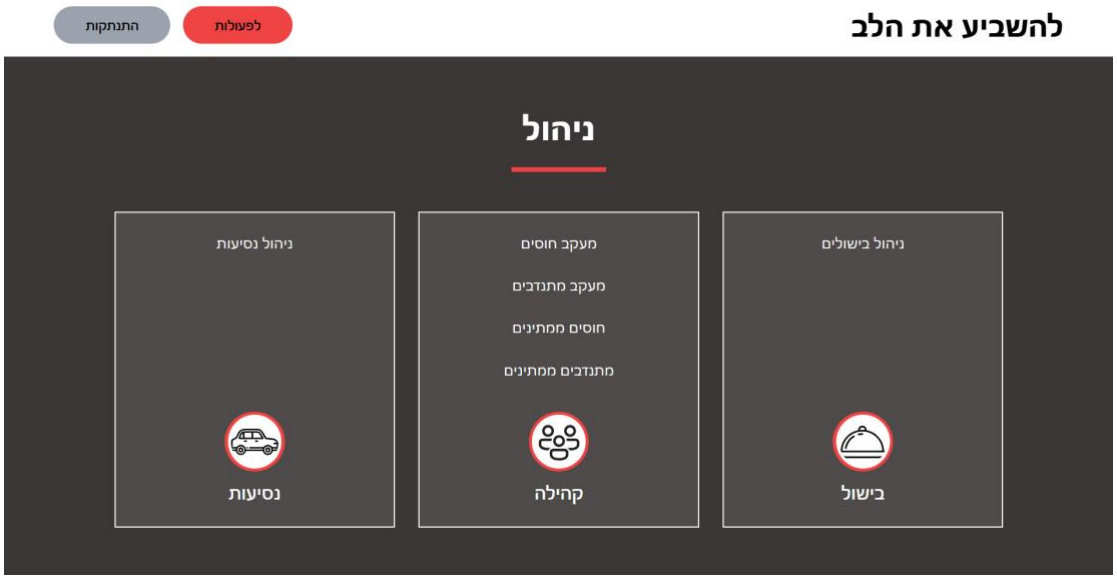
Registration Screen (A guest can join as a volunteer from here, or the coordinator can manually register them).

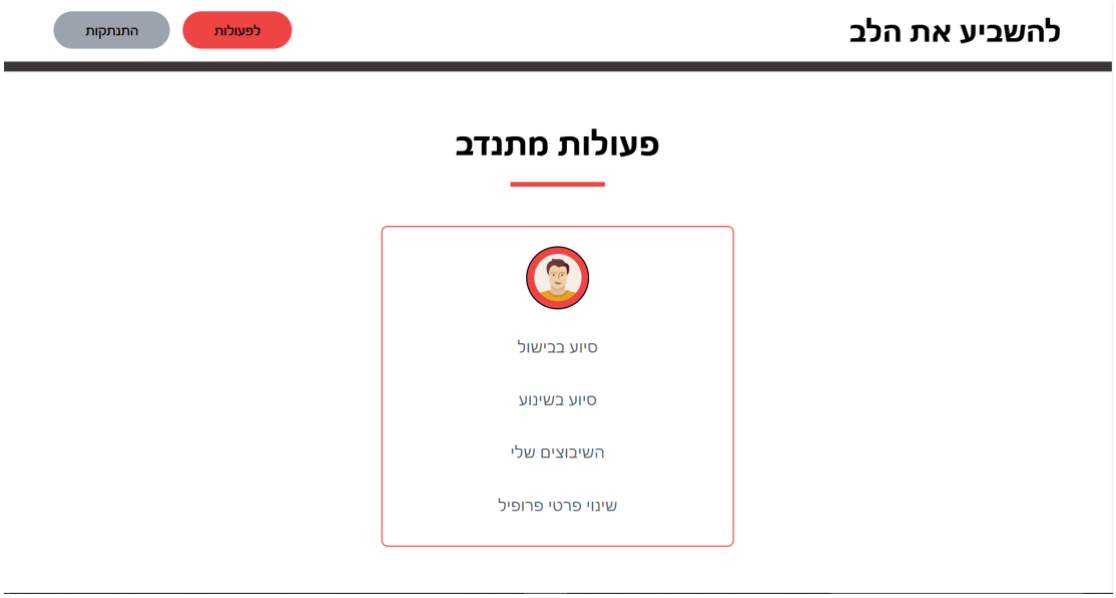


The director wants to approve each volunteer before they can use the system after registering. This will be shown later on the "Pending Volunteers" screen.

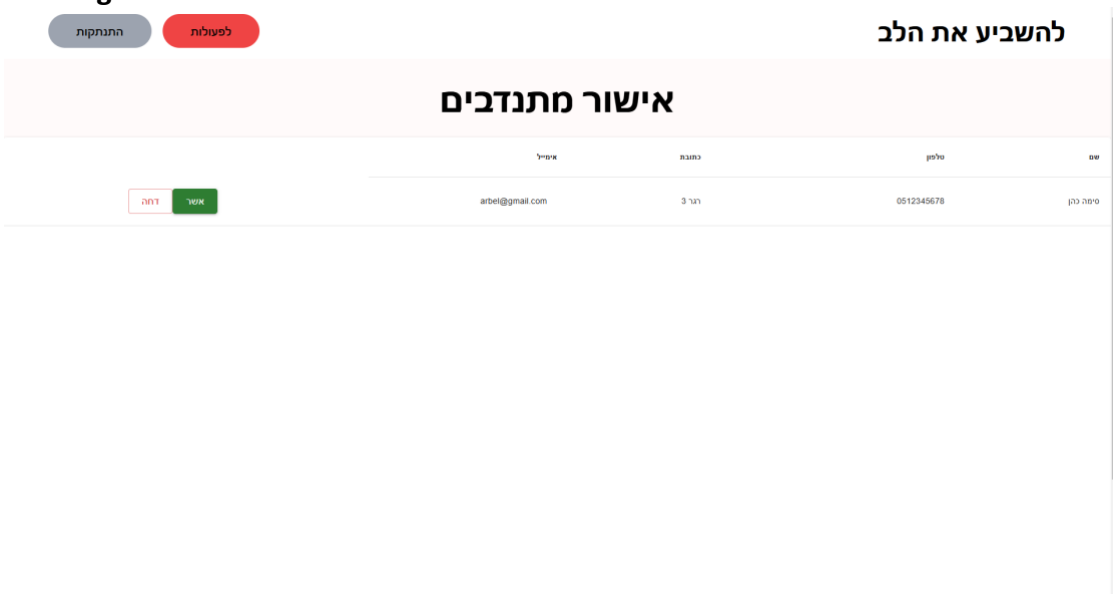


Actions Screen (If a non-manager volunteer logs in, they will only see volunteer-specific actions).

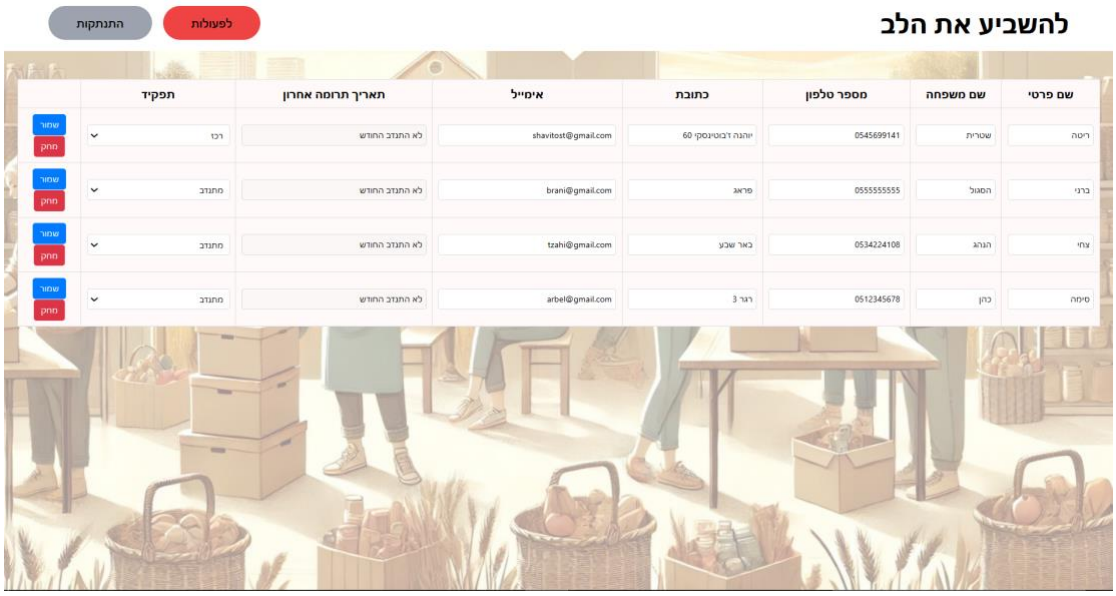




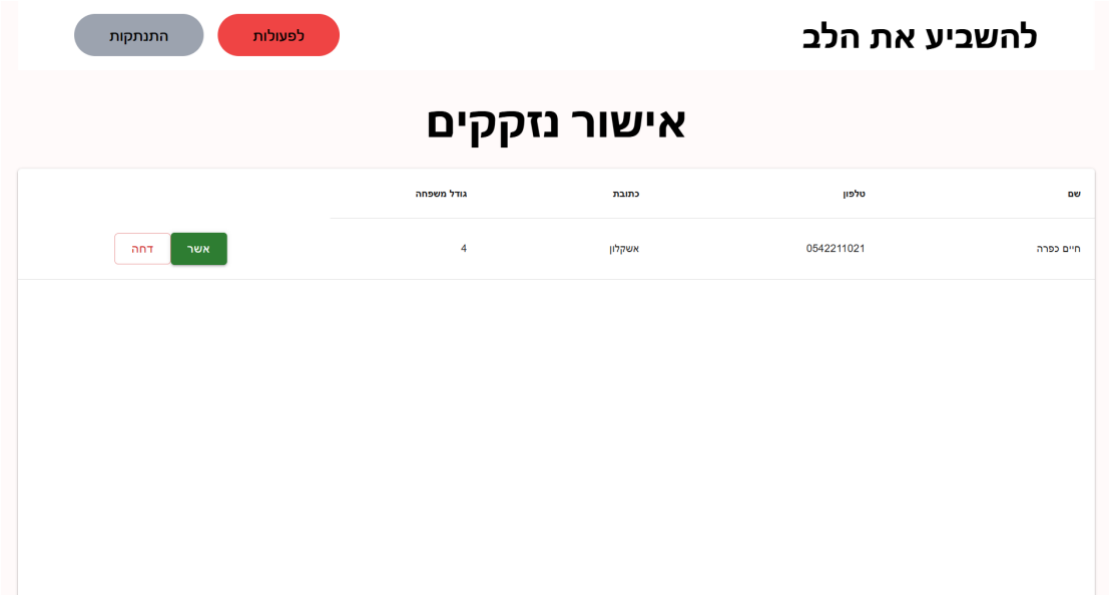
Pending Volunteers Screen



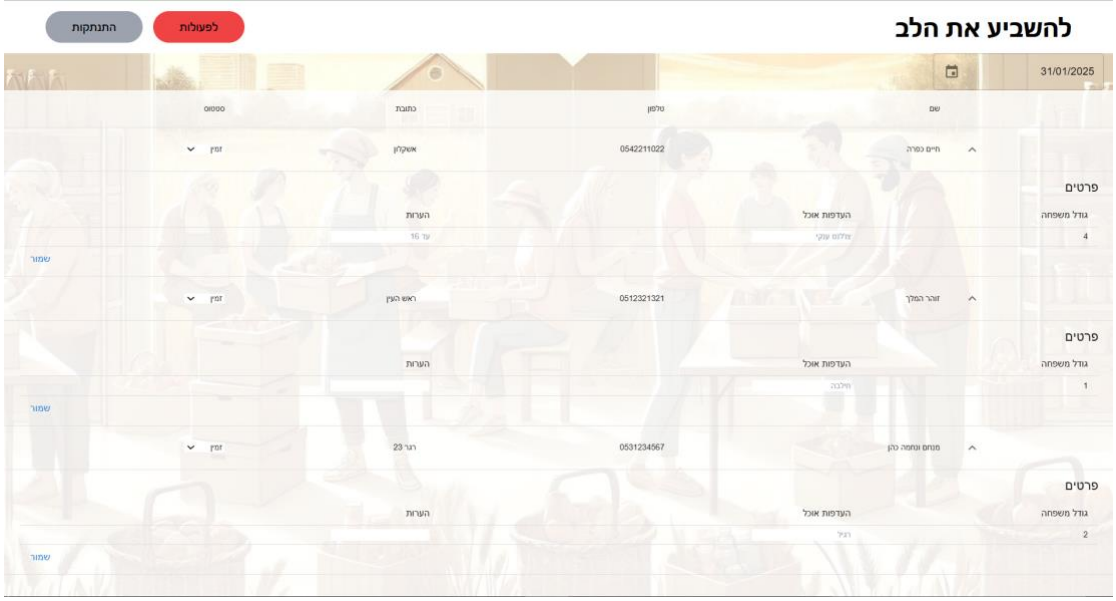
After approval, Sima enters the Volunteer Tracking Screen.



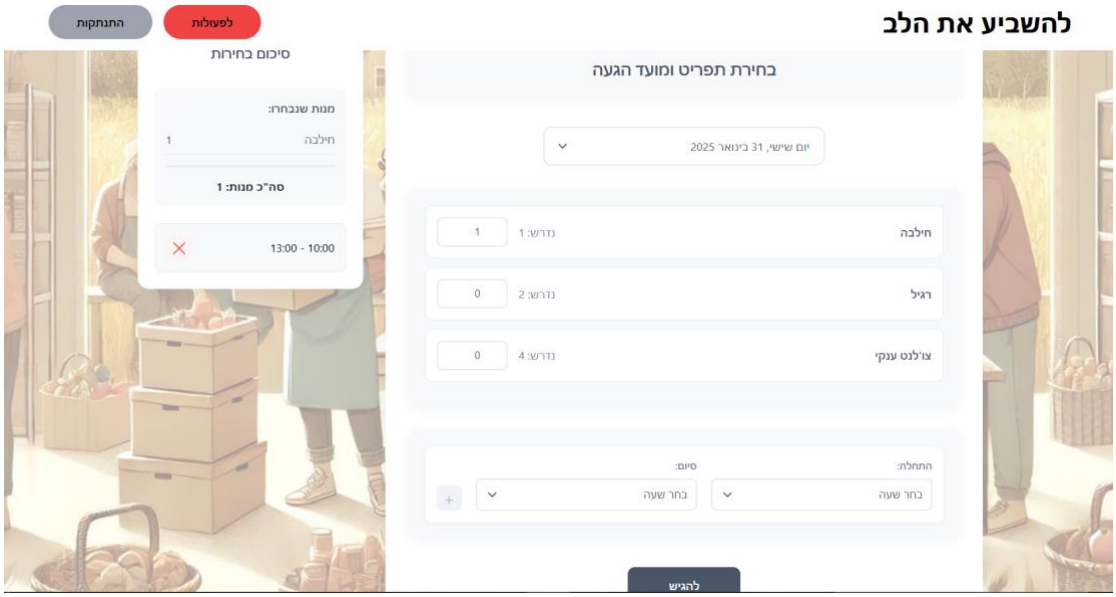
Pending Recipients Screen (They filled out the form on the home page for assistance. The details are sent to the manager on this screen for approval).



Recipient Tracking Screen. On this screen, the community coordinator updates information about approved recipients, whether they are active this week, and other details. The requirements are sent to the Cooking Management Screen.

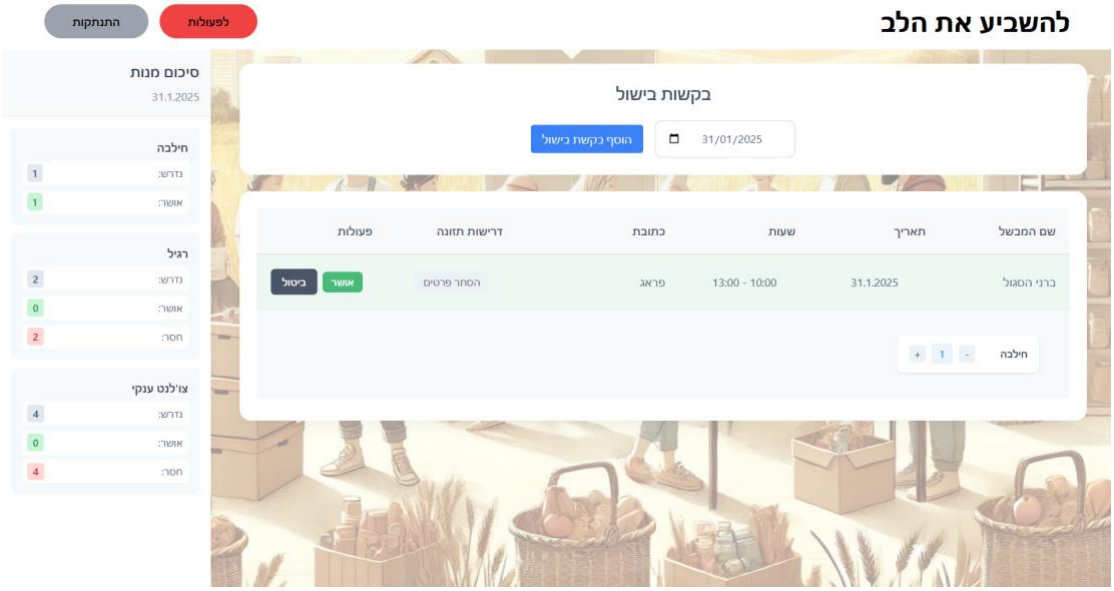


Chef's Request Submission Screen: Here, the chef chose to prepare "Hilba" and requested that it be picked up between 10 and 13.

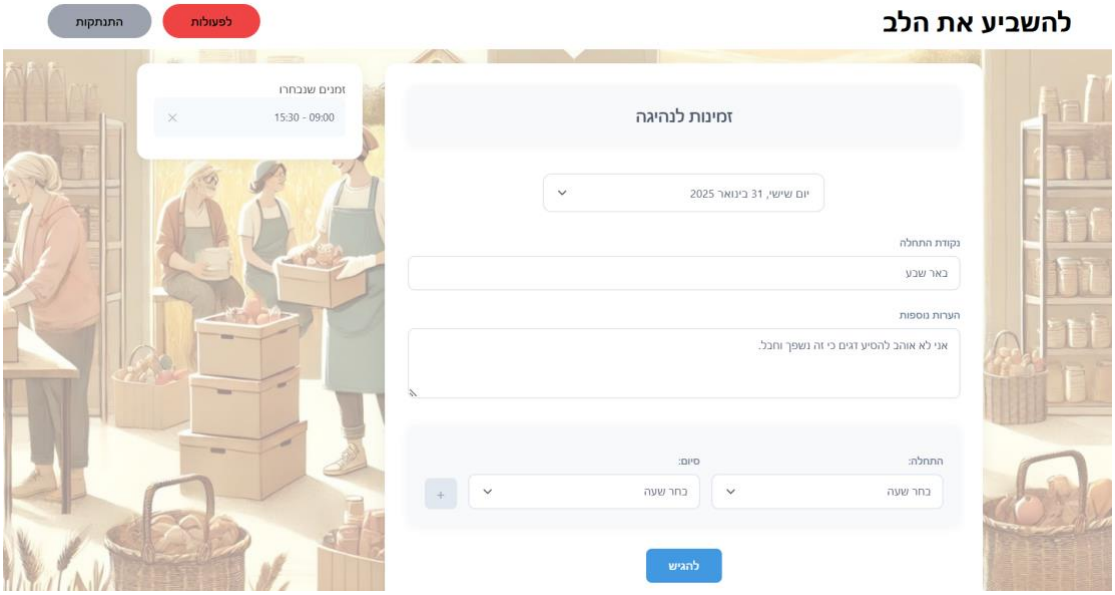


Cooking Management Screen:
On the left side of the screen is a summary of the meals required for the upcoming week, and on the right, the list of chefs who want to volunteer.
Here, the director approved Barney to cook a dish called "Hilba 1" (selected from the recipients' requests).

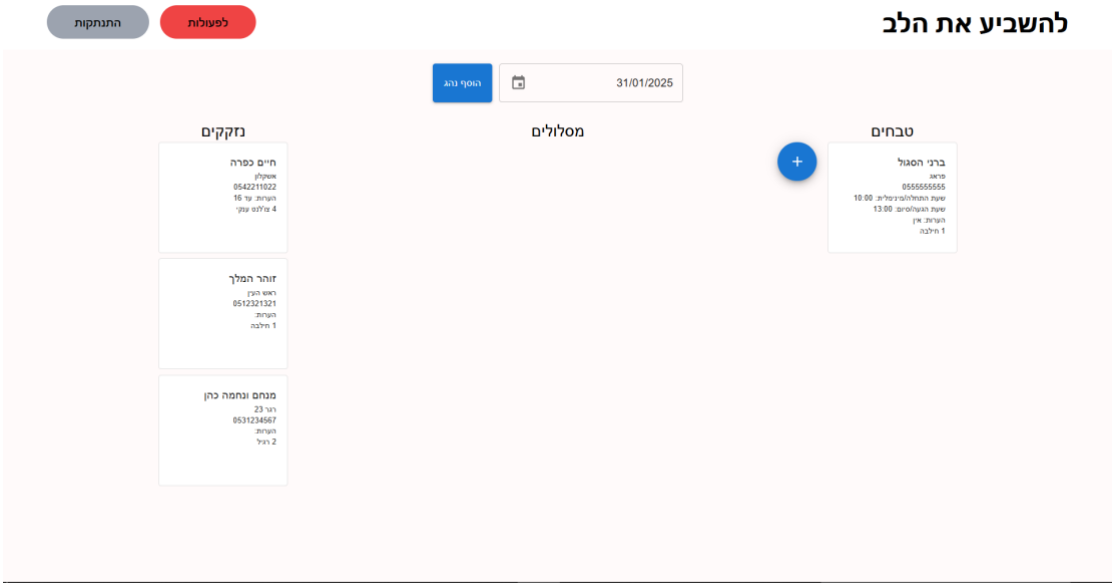
It can be picked up from his address in Prague between 10 and 13.



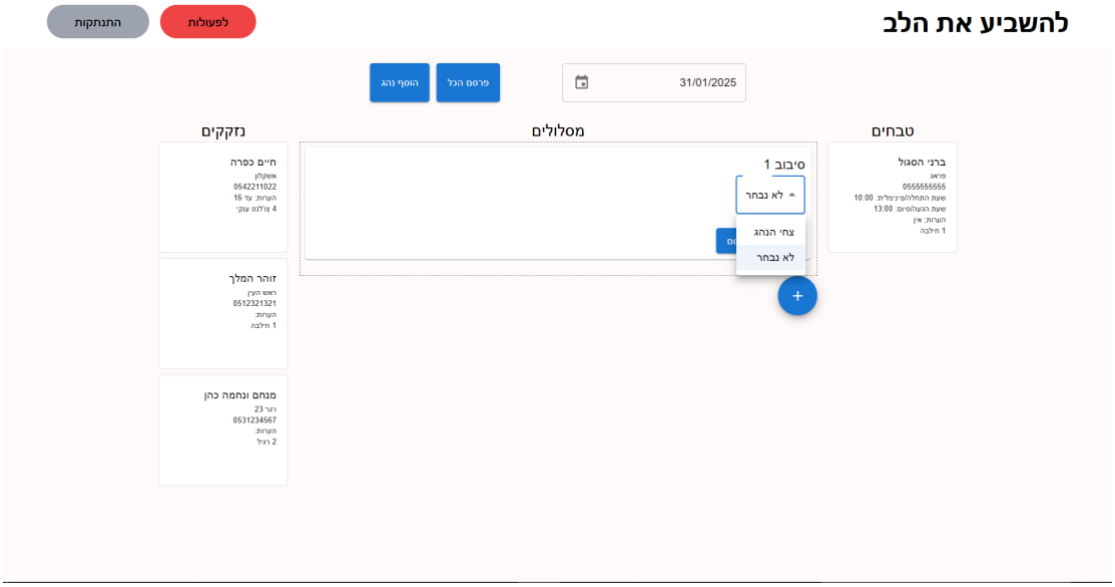
Driver's Request Submission Screen.



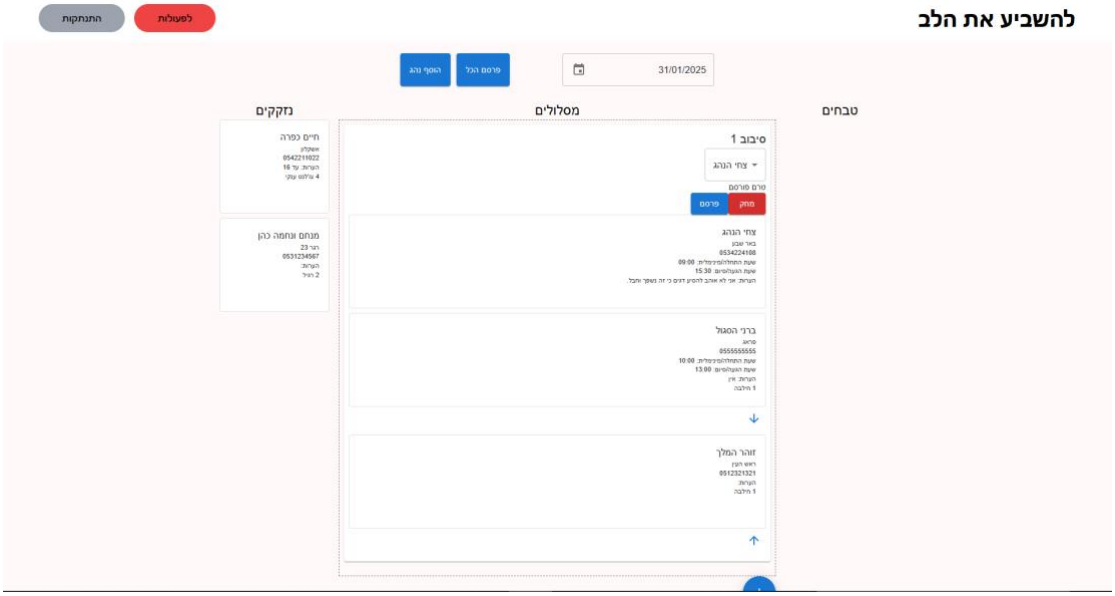
Ride Management Screen.



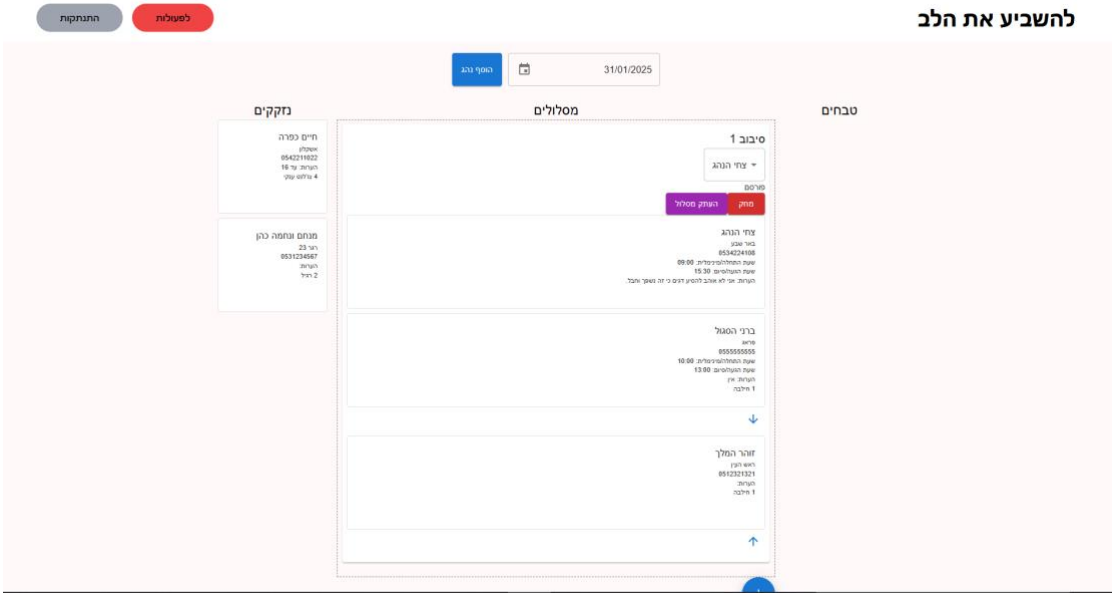
Driver Selection: Here, we can see Tzahi, who submitted a request stating he can drive.



We assigned a route for Tzahi the driver to pick up Hilba from Barney and deliver it to Zohar.



After publishing, the route can be quickly copied so the director can easily send it to the driver via WhatsApp.



This is what is copied to the clipboard:

מסלול הנסיעה שלך:

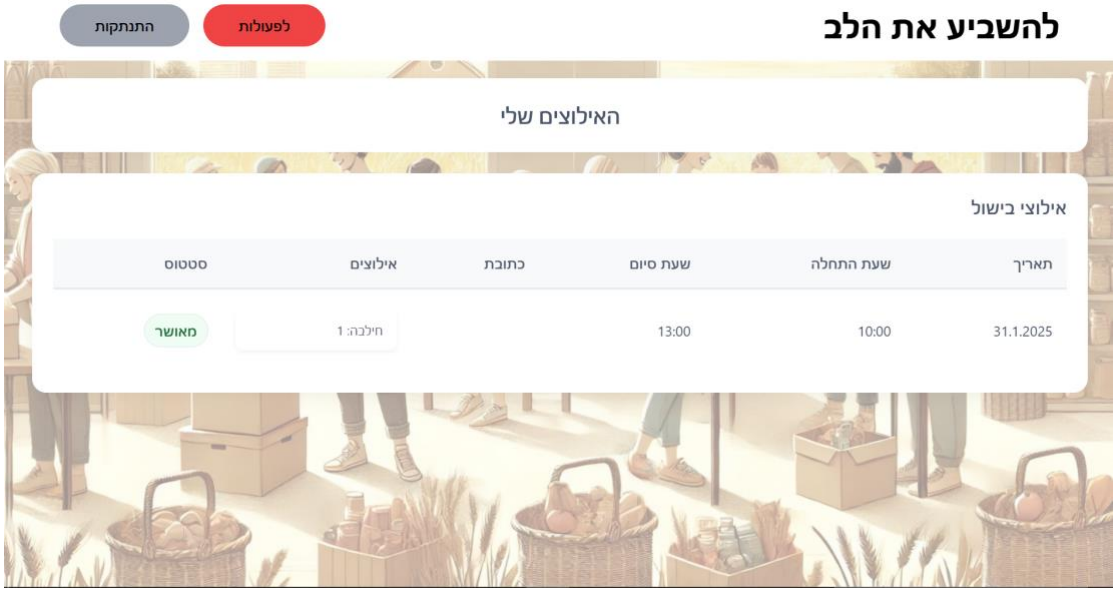
תחנה 1: התחלה
שם: צחי הנהג
כתובת: באר שבע
טלפון: 0534224108

הערות: אני לא אוהב להסיע דגים כי זה נשפך וחבל.

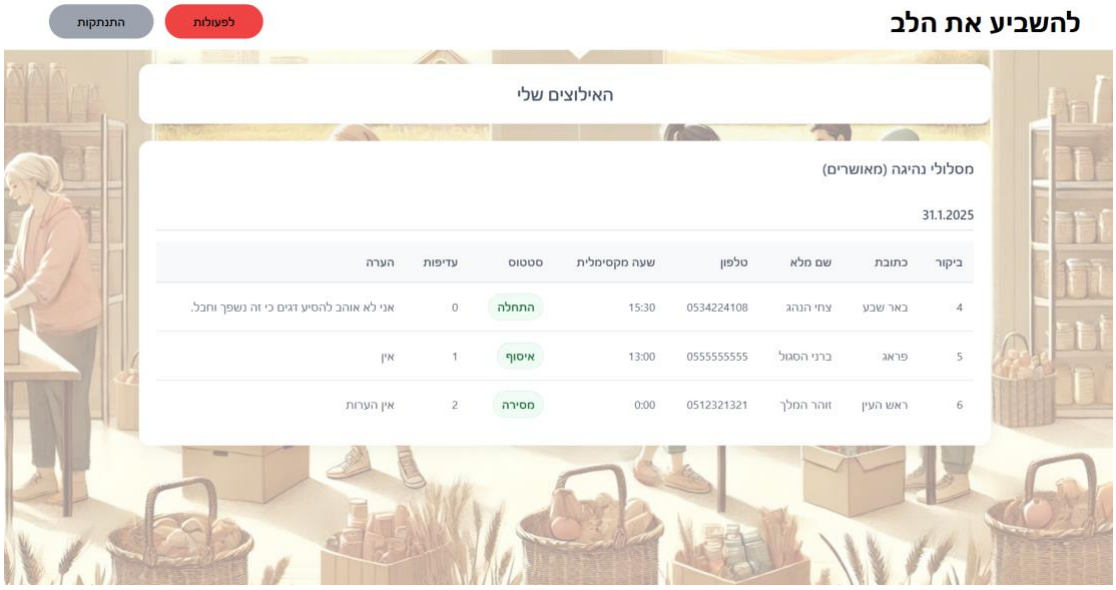
תחנה 2: איסוף
שם: ברני הסגול
כתובת: פראג
טלפון: 0555555555
הערות: אין

תחנה 3: חלוקה
שם: זוהר המלך
כתובת: ראש העין
טלפון: 0512321321
שבת שלום! 💔

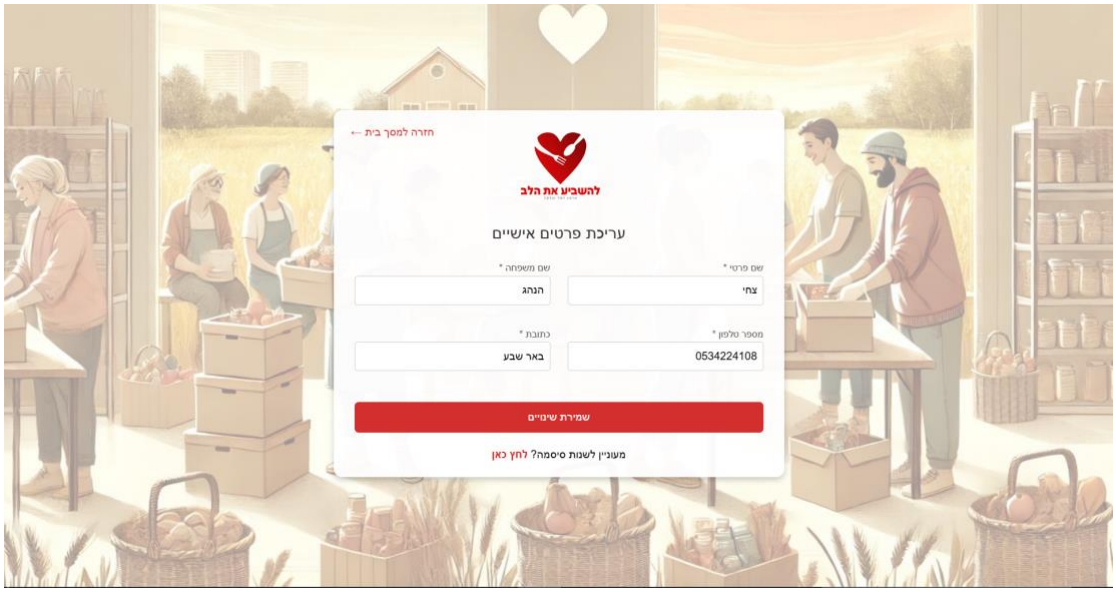
My Assignments Screen.
Here volunteers can see what tasks they have been assigned.
We can see Barney's assignment to prepare "Hilba 1."



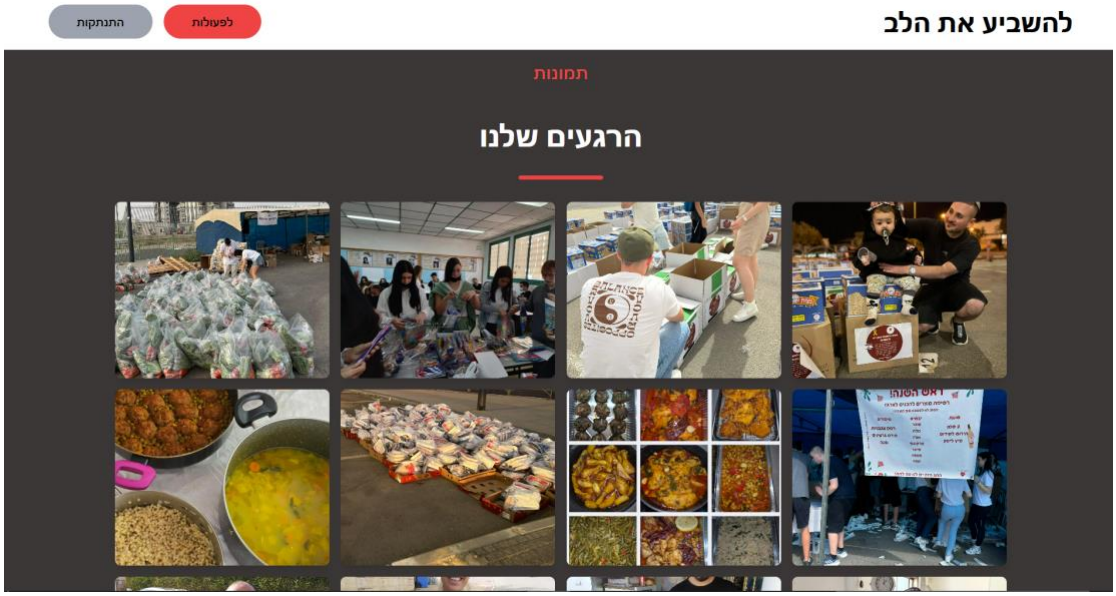
Here, we can see Tzahi the driver's assignment.



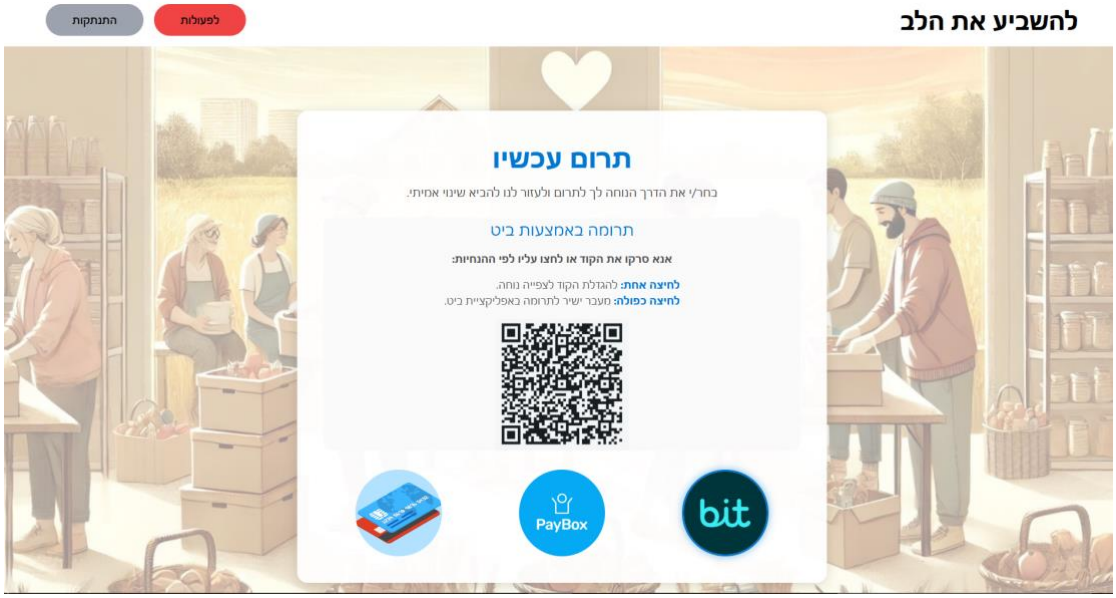
Edit Profile Details Screen.



All Photos Page.

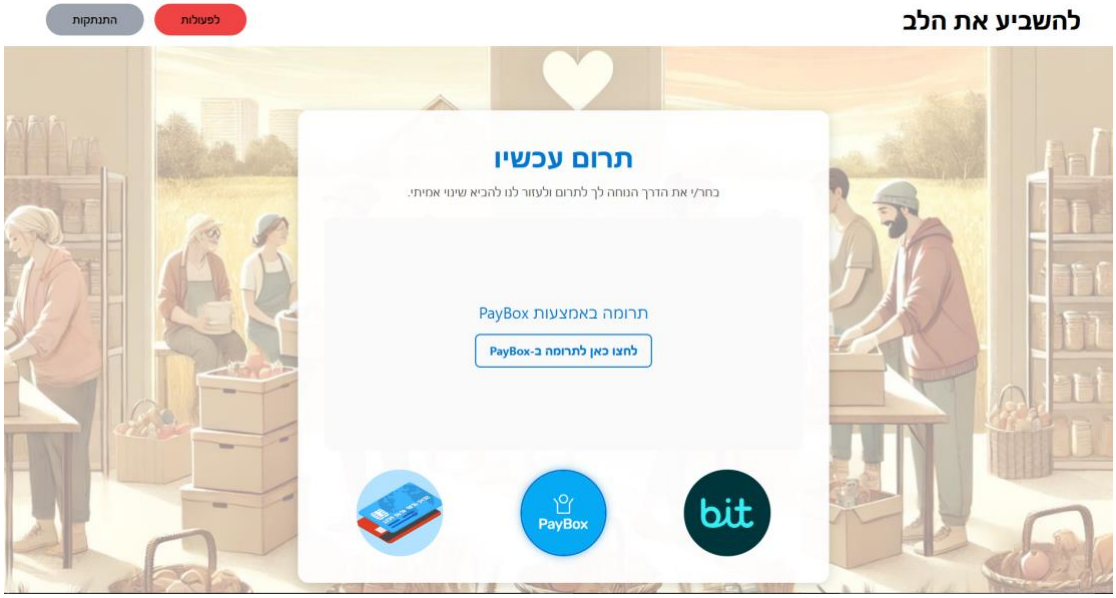


Donations Page.

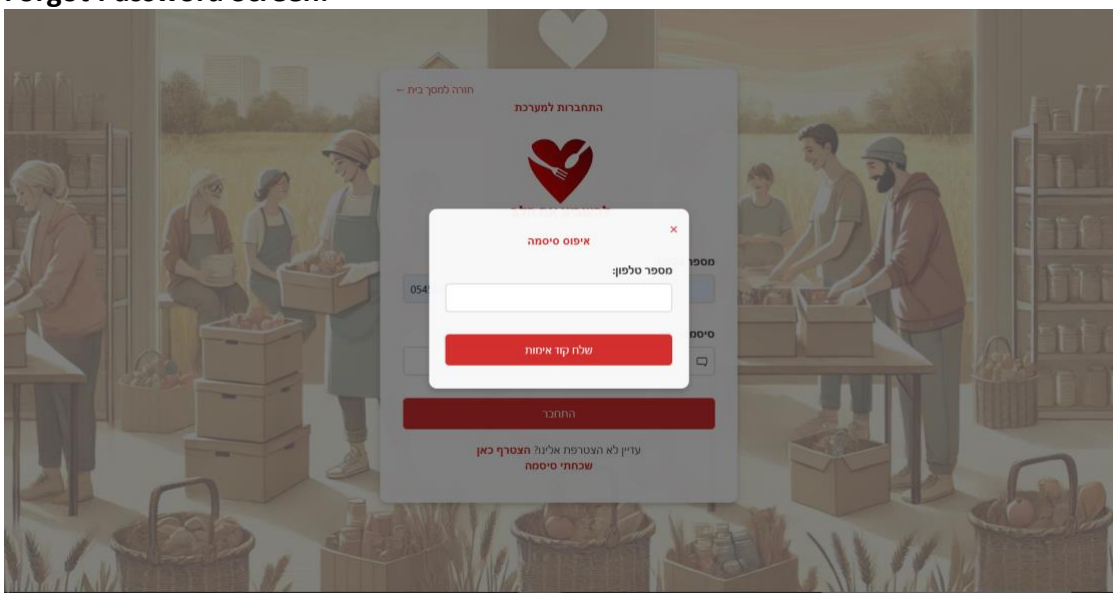


Donation via Bit.

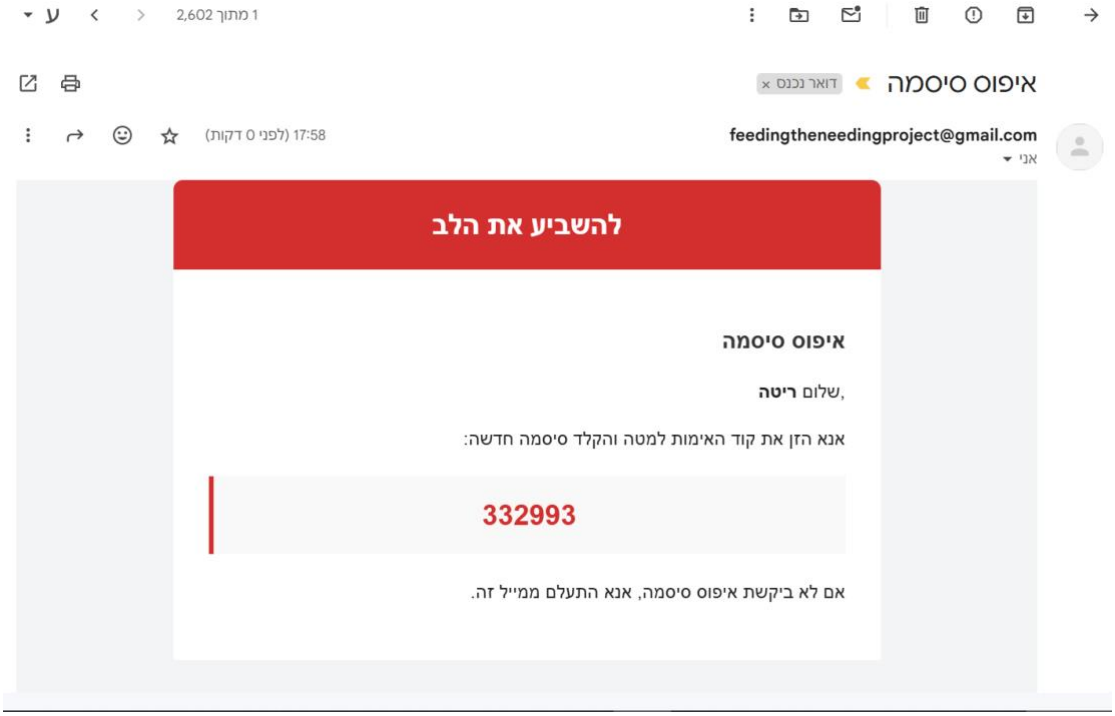
Donation via PayBox.



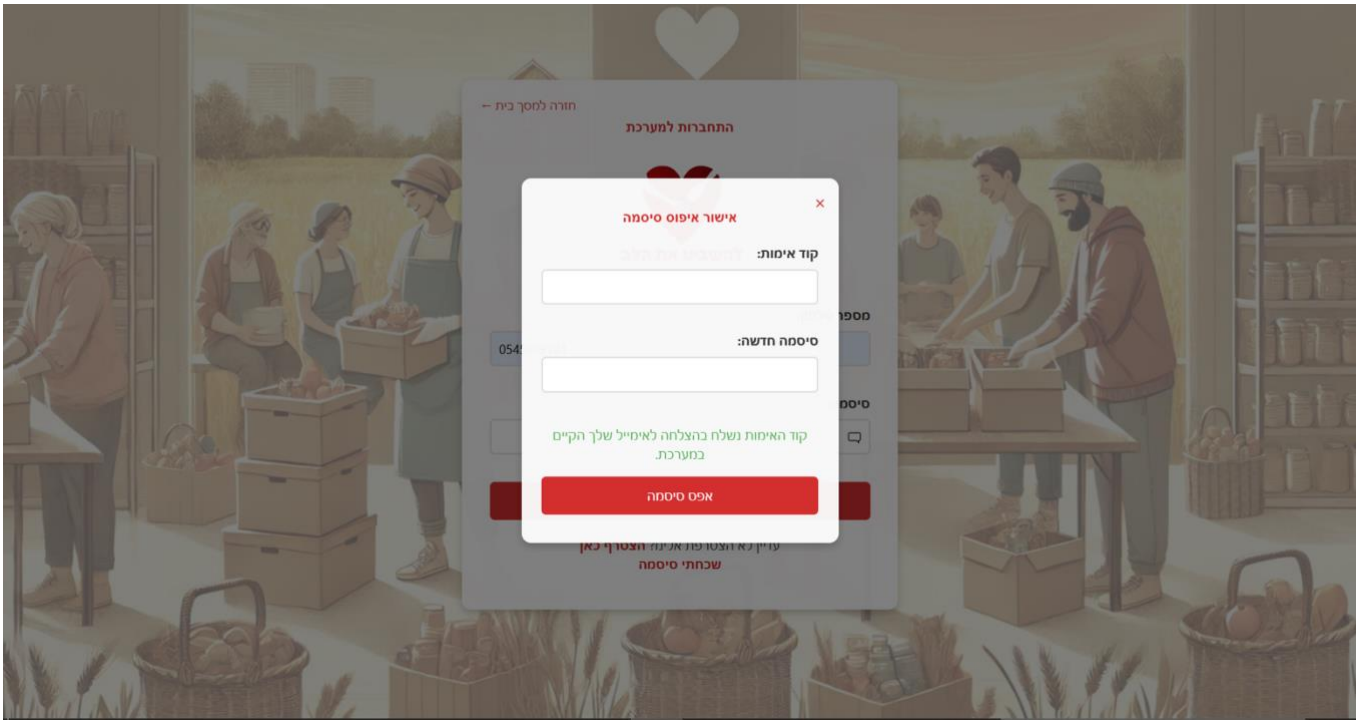
Forgot Password Screen.



A verification code is sent to the email after entering the phone number.



Here we put the validation code and set our new password.



Chapter 7 – Testing

The project employs a comprehensive testing strategy to ensure code quality and system reliability. Unit tests are implemented at the module level, including:

- **Service Tests:** These tests isolate service logic by mocking dependencies like repositories, enabling focused testing of service methods.
- **Repository Tests:** These tests validate database interactions, ensuring that data is stored and retrieved correctly.
- **Controller Tests:** These tests verify the behavior of REST API endpoints, including input validation, output formatting, and proper service calls (using mocked services).

Beyond unit tests, the project includes higher-level testing:

- **Authorization Tests:** These tests verify that access control mechanisms are functioning correctly, ensuring that only authorized users can access specific resources or perform certain actions:
    1. **None members** have access only to login and register.
    2. **Donors** can access resources and perform actions related to their donations, such as submitting constraints or updating their profile information.
    3. **Staff/Admin** have access to a broader range of functionalities, including managing needers and donors accounts, processing donations, and viewing sensitive data.
- **Concurrency Tests:** These tests simulate concurrent user interactions to identify and address potential race conditions or data inconsistencies.
- **System Tests:** Evaluate the entire system's behavior, covering critical user scenarios.
    1. **Current Implementation:** System tests are currently conducted through the REST API, simulating user interactions without the React frontend.
    2. **Future Plan:** Extend system testing to include e2e tests using Selenium to simulate real user interactions with the application's UI.

Continuous Integration

GitHub Actions are utilized to automate the testing process. Every code change triggers a build and test pipeline, including:

- **Unit Tests:** All unit tests are executed.
- **System Tests:** all tests are run.
- **React Frontend Checks:** The React frontend is built and checked for errors.

Non-Functional & Security Testing

The following non-functional and security aspects will be tested:

- **Security Tests:**
  - Authorization and role-based access control enforcement.
  - Token expiration and renewal.
  - Password reset verification.
- **Performance Tests:**
  - Load testing with concurrent users.
  - Response time tracking for key actions (login, assignment loading).
- **Availability & Reliability:**
  - System recovery scenarios (e.g., database restart).
- **Compliance:**
  - Secure password hashing (bcrypt).
  - Audit logging for sensitive operations.