Report_DS_v2

Elena Rughi matricola 0001005361
Distributed Systems course
2024/12/16

Abstract

This project aims to implement an addition to a Web App that was developed as a group project (which I was a part of); this addition is oriented towards the course of Distributed Systems, with the goal of examining how the application handles issues typical of distributed systems and implementing improvements.

The project I sought to expand is called Waffles - a "to-do" single page Web application for making lists, alone or in collaboration with friends.

Index

Base project

Concept

The *to-do app* was developed as a single page web application.

> We developed a single-page application for saving all your precious to-do lists, so you can keep track of all the things you need to remember during your day. But here's the thing: each item in a list is associated with a count, so you can use our app also for writing your grocery list, keeping track of the clothes you put into your luggage, and tracking whatever you want to count.
>
> The first thing you'll see logging in is the roll-up of all the fundamental things you wanted to remember or which are essential to remember because they will, are, or were due.
>
> We also provide easy-to-use charts. They allow you to see more clearly how your progression in completing the items in your lists is going. A calendar is also available for grouping the items by the day they'll be due or by the day you told us to remind them.
>
> You don't have to worry about having too many lists or too many to-dos. With our efficient search functionality, you can search between all items you previously tagged and see in which list you inserted them.

> But the most prominent feature is collaboration! You can share your lists with your friends to make them help you. In this way, you can complete the tasks you created together. No account is required! A new member can access the list as an anonymous user: you decide if it can access the list or not. Moreover, the access is only temporary: when the user leaves the page will need to request access to the list again. Whenever anyone modifies a list you're in will receive a notification in real-time about what happened.
>
> This app keeps on giving: a system of achievements is in place, making the organization of your daily tasks fun!

Requirements

The to-do application allows users to create and manage their accounts, create lists populated by items which can also double as tasks, set reminders for specific tasks or events, and collaborate on shared lists.

User data is stored remotely by the web app, and the site contains disclaimers to reflect that.

Personas

University student

Marco is a 24-year-old university student. He studies computer science. He knows it's best to stay organized and keep up with deadlines, thus he needs to quickly visualize not only daily tasks but also upcoming deadlines.

Married couple

Dalia and Alessandro are a happily married couple. They have an 8-year-old son, Pietro. They're both very busy with work and need to efficiently decide who buys groceries and what to buy, who needs to go pay for insurance, who needs to accompany their son to a birthday party. They're also thinking of going on a small holiday, which needs to be organized.

Teenager

Sara is a 17-year-old girl, very sociable and of happy disposition. She goes to school and often goes out with friends to watch a film, eat something together or simply spend time at one of their houses. She especially loves sushi and proposes to go to a Japanese restaurant every chance she has. She's also very interested in city events and loves to organize outings with her friends.

Scenarios

Marco

Marco primarily uses his computer. Once home from lessons, he accesses his account in order to decide what to study.

In the homepage he immediately sees the most important tasks and the tasks with a due date set withing the next few weeks. He opens the calendar on the site, to have a better global view of the tasks: within a week he has to submit a lab assignment, within the next month he has to submit a project. Selecting this last one, the browser redirects him to the list containing all projects he has to do: he sets the project with the nearest deadline as "important"; adds both a tag and a reminder to highlight that the project has a firm deadline and thus must absolutely submit it on time.
Marco is very interested in knowing how much productive he is so from time to times he accesses the report page to see how he's doing: how many tasks he completed in one week, how many of these tasks belong to one list or another etc. He also uses the site to set important reminders like ones for exams.

Dalia and Alessandro

Dalia and Alessandro primarily use their smartphones, as they don't always have access to a computer. For them it is very important to know who added or completed what in their shared lists.
While Alessandro is doing grocery shopping, when Dalia is right about to head out to bring their son to a birthday party, she notices that they're missing butter. She quickly accesses the "Waffles" website from her phone, proceeds to the page containing the lists created by her and then opens the "groceries" list; she then adds "butter", setting the item quantity to 2. Alessandro, who's already looking at the website to see what he needs to buy, receives a notification that alerts him of the fact that his wife added "butter" and also sees the new item appear at the end of the list, after "tomato sauce x1"; he ticks the box next to the tomato sauce to mark that he has already picked that up and then goes on to pick up two sticks of butter.

Sara

Sara mainly uses her smartphone. Thanks to social media and news sites she keeps up to date with the events happening in her city; when she finds an interesting one, she goes on the "Waffles" website, opens the list she shares with her friends and adds the event, specifying the date.
She also uses the website to remember which films she would like to see and she thinks that the achievements present on the website are a fun way to view an estimate of the number of films she has seen, the number of places she has visited etc.
From time to time she goes out to eat with her friends; when they eat in a "All you can eat" Japanese restaurant the website is very useful to keep track of the orders. Sara makes a new list, setting it's name to the restaurant name, then adds her friends to the list: if they already have an account, she adds them through their e-mail address; if they do not have an account, she makes the list public and shares with them a code that they can use to view the list. Sara's friends can add their own order to the list, specifying the quantity and their own name; if someone wants to eat a dish that is already on the list, they edit the quantity and add their name to the order, specifying the quantity they intend to eat.

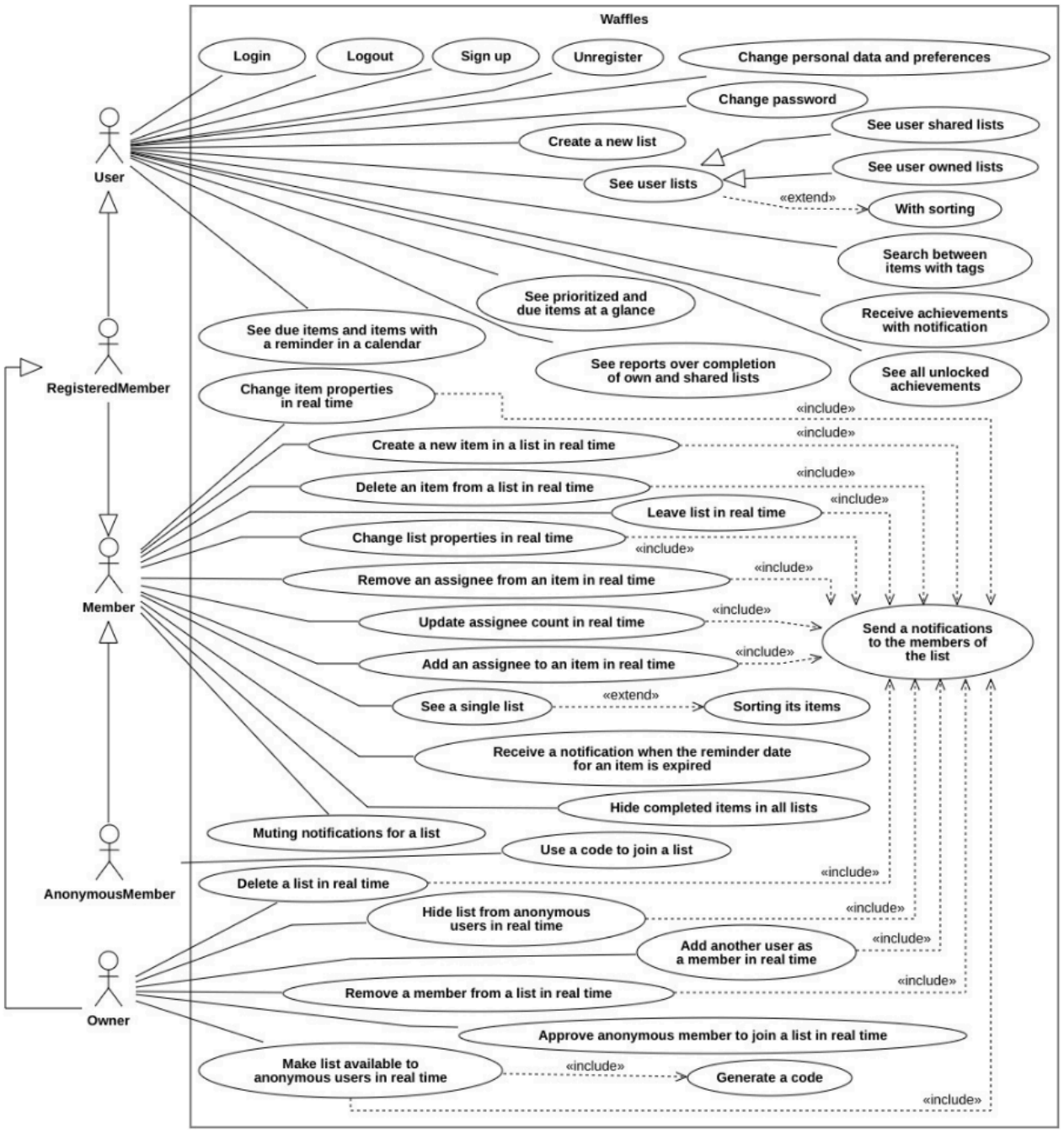
Functional requirements

Following the personas and scenarios, here are the requirements for functionalities available to users:

1. Profile management
    1. Sign up to create an account with:
        1. User name of max 15 characters;
        2. An e-mail, can be used for only one account;
        3. A sufficiently strong password.
    2. Delete an account:
        1. Requires inserting valid password;
        2. Deletes all lists and items created by the user;
        3. Removes account from shared lists;
        4. Closes all active sessions associated with the account.
    3. Access account with credentials, namely e-mail and password;
    4. Log out;
    5. Edit data inserted during sign up phase;
    6. Edit access password;
    7. Edit preferences in regards to receiving notifications;
    8. Receive achievements linked to salient events during use of the application, marked by a notification upon achievement reception;
    9. View all unlocked achievements along with the date.
2. List management
    1. View own lists, both created by user and shared by other users, optionally change viewing order;
    2. Create new list;
    3. Delete owned list:
        1. Deletes items of that list;
        2. If shared list, removes members with whom it was shared.
    4. Edit list properties, which are title and color;
    5. Be able to view items contained in a list, be able to reorder them or hide completed ones;
    6. Add a member to a created list;
    7. Approve access of an anonymous member to an owned list;
    8. Remove a member from an owned list;
    9. Make list accessible to anonymous users who do not have an account:
        1. This generates a unique 6-character code for access.
    10. Make list inaccessible to anonymous users who do not have an account;
    11. Leave a shared list of which user is a member of:
        1. Cannot leave shared list if user is the owner (creator).

12. Receive notifications upon edits to lists user is a part of and their content;
13. Disallow notification pop-up for single lists;
14. Ability to access and edit a list as an anonymous user via access code.

3. List's items management
    1. View items with priority and those not yet completed with a due date in the homepage;
    2. View graphs with the completion rate of the five lists containing items completed most recently;
    3. View items with a due date or a reminder set;
    4. Search items via tags associated to them;
    5. Add an item to a list;
    6. Remove an item from a list;
    7. Edit the properties of an item, namely the title, item count, priority status, optional due date, optional timestamp for the associated reminder and tags;
    8. Receive a notification at the time specified by the reminder set on an item;
    9. Add a member of the list as an assignee to the item, with a count associated to the assignee;
    10. Edit the count associated to the assignee;
    11. Remove an assignee;
4. Real time collaboration
    1. User sees in real time notifications for edits made to their lists, both edits to the list title/color and edits to the items contained in said lists;
    2. Edits must be immediately visible when viewing pages whose displayed information depends on modified lists or their items:
        1. The page that shows all lists must automatically update;
        2. The page that shows an individual list must automatically update;
        3. The page that shows graphs must automatically update;
        4. The page that shows the calendar must automatically update;
        5. The home page must automatically update.
5. Calendar
    1. The calendar page show items with a due date or reminder set;
    2. For each day, correspondent items are show;
    3. If an item has both due date and reminder set, the item is shown in the day of the due date;
    4. The calendar shows both completed items and not completed items;
    5. The calendar shows items belonging to both owned lists and shared lists.



Non-functional requirements

- User's account data must be correctly stored;
- Users must view most up to date data;
- User data treatment must conform to the European Union's “General Data Protection Regulation”, in particular the terms for the treatment of technical cookies;
- The application's user experience should be smooth and reactive.

Implementation requirements

- The web app will be implemented as a "Single Page Application", for better user experience;
- The web app will be implemented with the popular development stack “MongoDB - Express - React - NodeJS”;

- The interfacing between the application and the MongoDB database will be managed via the "mongoose" library, for convenient and secure data validation;
- Account passwords must be stored encrypted with the algorithm “BCrypt” with a *salt* unique to each of them;
- In order to receive timely notifications and allow users to collaborate in real-time the application makes use of the *Websocket* communication protocol, implemented with the Node library "socket.io";
- Final deployment of the application will be achieved through "containerization" of the system via Docker for ease of distribution.

Design

Architecture

The "Waffles" web app follows a client-server architecture:

- Separation of concerns between user interface (client/*user agent*) and application logic (*origin server*). The server component offers services by listening for requests, which are sent by the client component and responded to;
  - Since clients are simple programs that do not execute heavy computation, they are very *portable* across different devices;
  - Since server components are separate from client components, they can be modified and expanded to improve the service without impacting the clients, thus favoring *scalability*.

Since it is a *web* application, the project also implements:

- Uniform interface: communication revolves around standard self-descriptive messages (HTTP methods) and exchange of resources, which are uniquely identified by their respective URIs. The resources exchanged between client and server are agnostic of their final presentation, be it on the client or on the server. User agent and server need only agree to use the same presentation during transfer over network.
- Layered view: components are only aware of other components on the same layer. For example the user agent in this project directly communicates only with the application server and has no knowledge of the database service used by the application server.

The web server communicates with a separate database service; the database is replicated for fault tolerance and has a primary node which is the one that communicates with the web server.

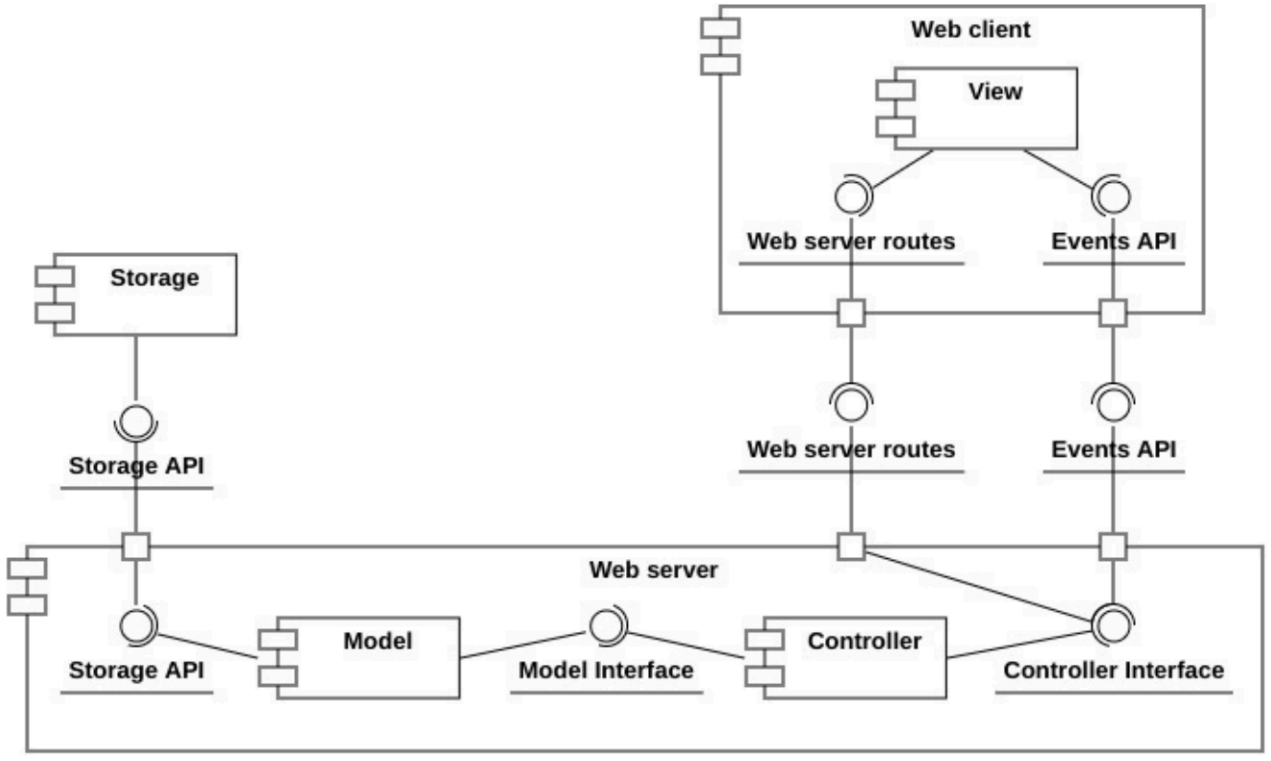
Infrastructure

The to-do web app's infrastructural components are as follows:

- web client -- the user interface capable of making requests towards the web server, each end user of the application can use their browser to run their own user agent or client;
- web server -- hosts the application that exposes the functionality of the web app to the network. The "Waffles" base project makes use of a single server to host the application;
- database -- a separate service allows the web server to host permanent user data.

The web server and database can be hosted either on the same machines or on different ones, the only requirement is that they operate on a shared private network and register themselves on said network as `<service name>:<port>`.

Below the component diagram of the system, in the context of the "MVC" pattern that was applied to the web app. The web client communicates with the web server via standard HTTP methods, however it also creates a dedicated Websocket connection to manage the real-time aspects of the project.



Modelling

There are four **domain entities** in this application:

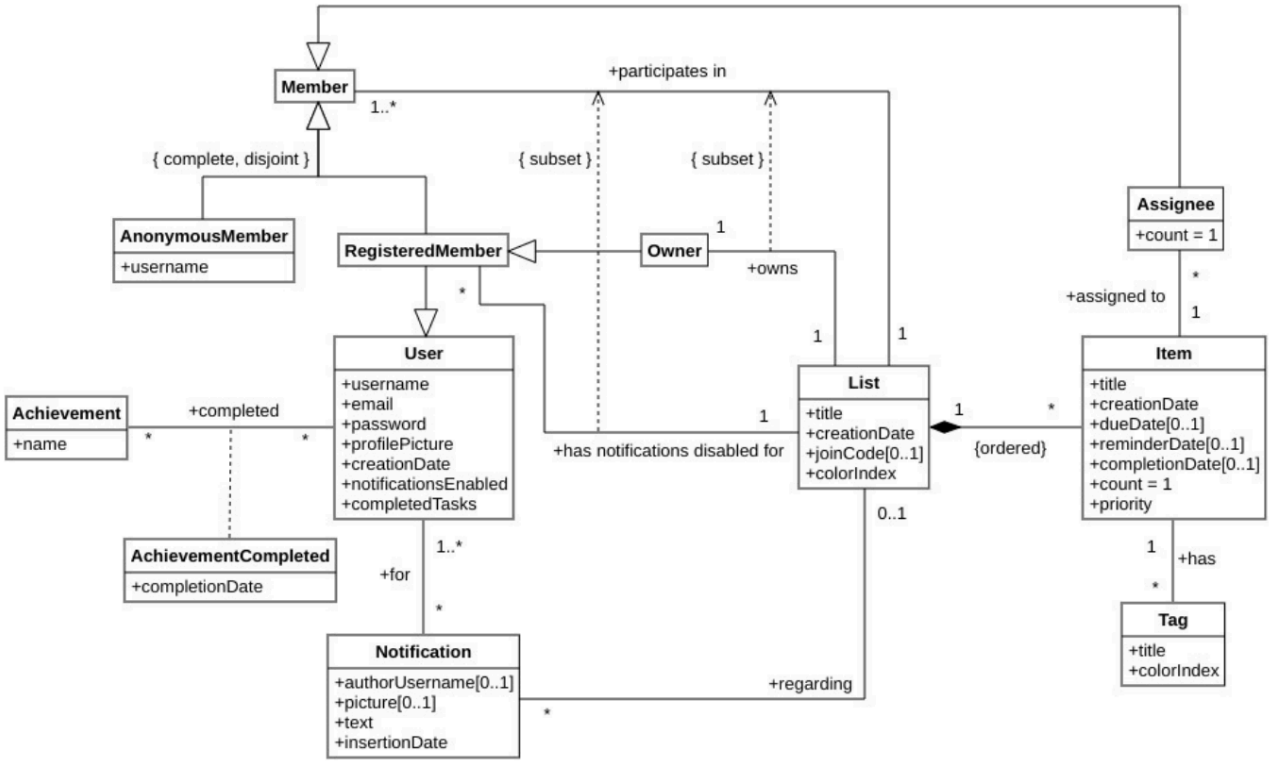
- user -- represents a user who registered an account; contains account information, settings preferences and associated achievements;
- list -- represents a list of items; each list has a title, date of creation, color, list of members and optional access code for anonymous users;
- item -- represents a single list item; each item has a title, date of creation, due date, reminder, date of completion, reference to list it belongs to, item count, tag, assignee

list and priority status;

- notification -- represents a notification composed of text, date of publication, list of interested users and other information depending on the type of notification: notifications are created by the system when:
  - a user obtains an achievement;
  - a shared list is edited by another member user;
  - a shared list changes because another member user deleted their account.

All four entities' data is stored in the database to be accessed by the server and indirectly by the user agent or client.

Class diagram of the application:



The **domain events** associated to the above **domain entities** are as follows:

- **user**
  - create account
  - login
  - logout
  - delete account
  - update account information
  - update password
  - enable/disable notifications
  - enable/disable notifications for a single list
  - get an achievement
- **list**

    - create list
    - delete list
    - update list title
    - update list visibility
    - update list color
    - add member to list
    - remove member from list
- **item**
    - create item
    - delete item
    - update item title
    - update item due date
    - update item reminder date
    - change completion state of item
    - add, remove, edit tag of item
    - update count of item
    - update priority of item
    - add assignee to item
    - remove assignee
- **notification**
    - delete a notification for a user
    - delete all notifications for a user

Interaction

- *client-server communication* -- the web server listens for requests incoming from user agents;
- *server-database communication* -- the web server queries the database to retrieve, edit, add, remove data;
- *intra-database communication* -- the database service consists of replicated nodes which communicate among themselves to achieve consistency; there is a primary node which is the one that communicates with the web server.

Behaviour

The web client sends standard stateless requests to the server to load basic static content; however, when a user accesses their account a session is started so that they don't need to log in every time they access the site.
The session data is however not stored in the web server, for it would be too much overhead; session data is stored in the database and it is associated to a cookie, which is then included in client requests to communicate that user is logged in.

The user agent also opens a persistent Websocket connection whenever a logged in user opens the site; this stateful connection is used by the server to push notification to the user, as well as a two-way communication when users are editing the same shared list at the same time.

Data and Consistency Issues

For this web app, the data that needs to be stored corresponds to data about the four domain entities: user account, lists, items and notifications.

The database chosen for this application is a *document-oriented* one; each domain entity is represented with a collection validated by a schema. This type of database was chosen because of the flexibility and convenient direct mapping between modeled entities and storage method.

The database is queried by the web server only; it performs writes in the order they arrive in.

Fault-Tolerance

The database is replicated across three nodes; the replication is managed within the database service. Replication grants redundancy and can increase data availability in environments that require heavy operations.

A replica set is a group of database instances, or nodes, that maintain the same data set; each node can only belong to one replica set.
There is a primary node that receives all write operations and records changes to its data in an *operation log*, which is replicated across the secondary nodes. The changes to the operation log are forwarded asynchronously and applied by the secondary nodes upon reception.

There is a heartbeat mechanism between all nodes of a replica set; if the primary node fails, the secondary nodes will elect a new primary node among themselves.

Security

User accounts are tied to an e-mail that must be unique within the application; users access their account with a password that is stored encrypted with the algorithm “BCrypt” with a *salt* unique to each of them.

There is only one user role: each user has authority over their own account data. When a user creates a list, it is owner of that list. A user can only access their own lists and lists shared with them.
Within a shared list, only the owner of that list can delete the list, add members, remove members, share the list to anonymous users. A user that is member of another user's list can leave said list. When an owner of a shared list deletes said list, the list is deletes for all other members as well.

Availability

In this project, there is only one application server: it can handle only so many requests, and further users would not be able to access it until it is freed up. Or, a maintenance might be necessary and the service would be taken offline as needed, leaving the users without.

As a web application, "Waffles" should keep functioning as well as possible through network failure; in order to manage network partitions, consistency must be in part sacrificed in favor of availability, or vice-versa.
In this project consistency is prioritized: if a request results in an error, an error message is displayed to the user, to notify that the request has not been correctly received.

As for fault tolerance, if the lone web server were to shut down for whatever reason the users would simply be unable to access the service until it has been brought back online.

Implementation

The to-do app makes use of an HTTP server; the server also makes use of the WebSocket protocol for the real-time component of the application.

In-transit data is represented with the standard JSON format.

The database is queried via a library compatible with the implementation of the web server, for ease of modelling data and validation.

Technological details

Web server and client are written in Javascript and ran through Node.js; development was aided by the package manager *npm*.

- web server -- Express
  - mongoose -- library for modeling and querying MongoDB database
  - express-session -- library for managing user sessions
  - socket.io -- library for real-time event-based communication
- web client -- React
  - axios -- promise based HTTP client
- database -- MongoDB

In this project, consistency of data is managed by MongoDB, which has an out-of-the-box implementation to solve common consistency issues.
Of course, data corruption while in transit on the network or an untimely failure of the primary node in a replica set is always a possibility; consistency loss is merely mitigated as much as possible in a distributed environment.

The database is actually implemented by three Docker services, making use of the *replica set* functionality given MongoDB. A replica set is a group of `mongod` instances that maintain the same data set.

A replica set contains several data bearing nodes and optionally one arbiter node. Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.

- The primary node receives all write operations, records all changes in its *operation log*, which is used by the secondary nodes to update themselves *asynchronously*.
- If the primary node fails, a secondary one takes on the role.
- By default, clients read from the primary node.

Expansion project

Goals

The goal of this addition to the *to-do app* project is to evaluate where it is lacking in regard to its distributed aspect and implement some solutions.

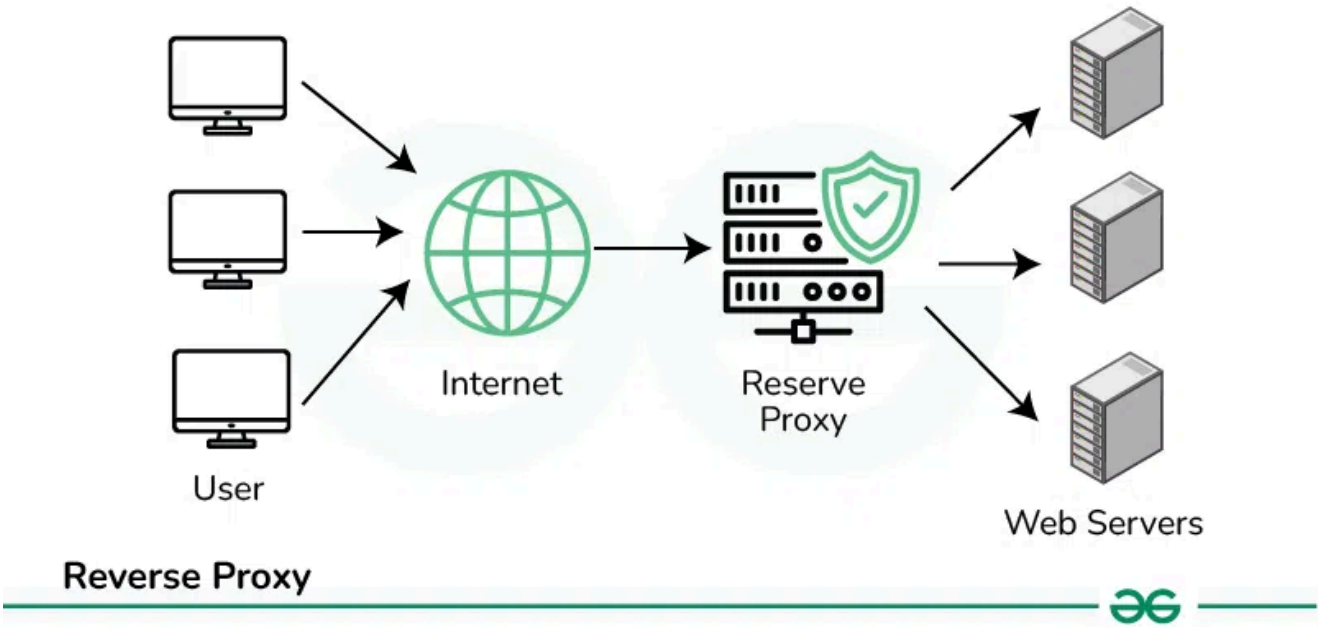
The base *to-do app* already implements database replication; the glaring omission from the web application is redundancy at server level.
At the moment, all user requests are forwarded to one single application server, which could be overwhelmed by excessive traffic or could fail thus rendering the entire web app unavailable util repaired.
By implementing application server replication, availability and fault tolerance would be improved.

Design

Replicating the web server, or application server, is easy enough: host the same server on different machines on the same internal network, through which they all can access the separate database service which is exposed on the network.

In order to access the web app from an external network however another solution is needed, for it is unthinkable to require end users to directly contact individual servers.

I chose to add a light-weight reverse proxy web server as an intermediary between user requests and the *to-do* web application's inner workings.



Infrastructure

A reverse proxy server acts as an intermediary between the external network and the private network on which the resources are hosted, in this case the application servers and the database.

Reverse proxies are a single point which can filter traffic incoming into the private network to enact security measures, and hides the inner structure of the application. For example, it hides the number of devices hosting the application, and their addresses.

For this project I added one reverse proxy server that also acts as a load balancer; it communicates with a pool of upstream proxied application servers hosted on the application's private network.

The services sharing the private network are:

- the reverse proxy;
- the application services (replicas);
- the database services (replicas).

Since they belong to the same network the services discover-able by each other.

Fault-Tolerance

The proxy server passively performs health checks: since all requests now pass through the reverse proxy server, it notices whenever a request is not being fulfilled and tries to resume failed connections. If the failed transaction cannot resume, the proxy server marks the server as unavailable and redirects the traffic to another available server of the replica set.

The time for which a server is marked as unavailable and the maximum number of failed transaction attempts before being marked unavailable are among the configuration options available for the proxy server.

A proxy server can actively perform health checks by periodically sending special health-check requests to each proxied server and verifying the correct response.

Availability

For the expansion to the *to-do* app I also added a **load balancing** functionality to the reverse proxy: it maintains a list of *upstream* proxied servers, and routes traffic to them following the configured load balancing method.
For this project, the chosen load balancing method is **hashing** based on the IP address associated to a user's request, since the web application makes use of sessions thus requiring that the client communicates with the same server whenever possible.
The proxy server can also be configured to allow for dynamically adding new replica servers to the *upstream* server pool if traffics requires it.

The proxy server can **cache static content**, so as to offload the web servers of their repetitive responses.
Additionally, the proxy server can **buffer responses** from proxied servers: a response is stored in the internal buffers and is not sent to the client until the whole response is received. Buffering helps to optimize performance with slow clients, which could otherwise waste proxied server time if the response is passed from the proxy server to the client synchronously.

Finally, if an upstream server is no longer responsive, shows remarkable delay in responding or returns faulty responses, the reverse proxy can label it as faulty and redirect traffic previously assigned to it to the other functioning servers in the pool.

Implementation

NGINX

NGINX is easily configured via its `nginx.conf` file, which is read during the creation of the NGINX Docker container.

The `nginx.conf` file in the project is as follows:

```
upstream loadbalancer {
  hash $remote_addr$http_x_forwarded_for consistent;

  server nodeapp1:8080 fail_timeout=600s;
```

```
  server nodeapp2:8080 fail_timeout=600s;

  keepalive 4;
}

server {
  listen 80;
  server_name localhost;

  location / {
    add_header ProxyServer $upstream_addr;
    proxy_set_header Host $host;
    proxy_http_version 1.1;
    proxy_set_header Upgrade upgrade;
    proxy_set_header Connection "";
    proxy_pass http://loadbalancer;
  }
}
```

The *upstream* module defines the list of proxied servers and the load balancing method.

The load balancing method is set to *hash* because the web application makes use of sessions, since its users can login to their accounts to save their data.

I set the hash input to be a concatenation of the remote address of the client and the value of the optional header `X-Forwarded-For`, which I needed for testing purposes to mock requests coming from geographically separate clients.

In the *server* module the port on which the NGINX server receives requests is set; the `location /` block defines that requests sent to root (in the case of this SPA it is all of the requests) are proxied to the load balancer defined in the block above.

Lastly I needed to explicitly set the rule for the Express server to trust proxy servers:

```
const app = express();
...
app.enable('trust proxy');
```

Replicating Node servers

In the Docker Compose file, after adding the NGINX service, I simply replicated the Node application service by using the same image but defining multiple Node services.

The NGINX service depends on the replicated Node services, so it does not start until all replicated services are ready.

All services are added to the same network, as well as the database services.

Validation

In remaining within the scope of this project, the tests I conducted is limited to the proxy server expansion implemented on top of the base web app project.

- What was tested:
  - Interaction between end user and web application as a whole (fault tolerance);
  - Interaction between reverse proxy and proxied servers (load balancing).
- Test environment:
  - For convenience, testing was conducted locally with the use of Docker containers simulating services hosted on separate machines;
  - Different clients accessing the web app were simulated with the use of `X-Forwarded-For` headers, which are typically used to preserve the originating IP address for requests that pass through forward proxies;
  - Client connecting from localhost always connects to *nodeapp1*, because of the hash method of load balancing;
  - The proxied servers are two replicas of the Node service, named *nodeapp1* and *nodeapp2*.
- How the tests were automated:
  - The Postman application was used for capturing client requests as Postman Collections, which were then exported as a JSON files into the project folder `/test/test_collections`.
  - These collections also contain pre-request and post-response JS scripts for conducting tests on the responses, in order to verify behavior;
  - Newman, a CLI implementation of Postman; the Newman library for Node.js was used to run the Postman collections from a Node script, which can be launched with `npm test`.

As an aside while the project is running, with the command:

```
docker container logs -f nginx
```

one can view in real time the NGINX access logs.

Fault tolerance

To-do app with NGINX proxy server can be tested from regular browser use: log in, navigate, create a list, write some data.

Then, simulate a server failure with the command:

```
docker container stop nodeapp1
```

And keep using the site. The user does not get logged out, and after NGINX recognizes that *nodeapp1* is unavailable it redirects traffic to *nodeapp2* without the user noticing.

The session is maintained because server side sessions are managed with the Node library `connect-mongodb-session` which automatically stores session data in a collection on the database, separate from Node servers.

I also wrote this test as a pair of Postman Collections: in the first collection the user logs in, verifies there are only the pre-populated lists, then creates List A, verifies it has been saved. Then I manually stop the `nodeapp1` service, and run the second collection in which, without logging back in, the test verifies that List A is still present, then creates List B, verifies it has been saved.

The session survives `nodeapp1`'s failure and the user does not need to log in again because, in the first collection, the request:

```
POST http://localhost:9000/socket
```

returns the session cookie, which I then save as a Postman global variable:

```
const cookie = pm.cookies.toJSON();
pm.globals.set('cookie', cookie[0].value);
```

thus being available in the second collection run, where by setting the cookie header the request is linked to the already open session:

```
pm.request.headers.upsert({
        key: 'Cookie',
        value: pm.globals.get('cookie')
});
```

And subsequent create list requests work for the correct account.

I tried running this test automatically with Newman, however running two separate collections outside of the Postman environment conflicts with the Websocket connection towards NGINX that is used to retrieve the session cookie.

The JSON files of the collections can be imported into the Postman application in order to run them.

Load balancer

I set NGINX to add a `ProxyServer` header containing the address of the upstream proxied server to every response, otherwise it is hidden by the NGINX server:

```
[/nginx/nginx.conf]

...
server{
        ...

        location / {
                  add_header ProxyServer $upstream_addr;
                  ...
        }
}
```

Then I created a Postman collection that simply accesses the web page:

```
GET http://localhost:9000/
```

with a collection script that then sends it 10 more times, each time with a different `X-Forwarded-For` header to mock 10 clients with different IP addresses accessing the web page.

Each response contains the `ProxyServer` header set by the NGINX server, the value of this header for each response is saved to then verify that both proxied servers have received requests.

This test can be executed from the project folder with the command:

```
npm test
```

(Might still require Node v. 16.1.1)

Websockets

By testing the web application with the browser and network inspector I was able to confirm that Websocket communication still functions correctly through the NGINX proxy server.

Release

Docker Compose was used to ship the application, which is formed by the following services:

- `mongodb` -- the database service;
- `nodeapp` -- the application service, implemented with Node.js, depends on the database service;
- `nginx` -- the reverse proxy service, depends on `nodeapp` and `mongodb`.

Each service was individually uploaded to Docker Hub.

The compose file launches three MongoDB services initialized as a replica set; then the first application service is launched, immediately connects to the replica set and executes a script to populate the database with starting data. After this, the second application service is launched, after which the NGINX service finally runs. All services are part of the same network.

Deployment

Pull the required images from DockerHub

1. Clone the GitHub repository.
2. Move to the main folder of this repo with `cd todo-app-sd`.
3. Deploy the application using `make up`.

When the messages

> "nodeapp1 started"
>
> "nodeapp2 started"
>
> "Connection with the database established"

appear on-screen, the installation of this app onto your system is complete. If you want to use it, reach `localhost:9000` from your browser to access the app.

The app may take some seconds to make those messages appear, so be patient for a bit.

If the app exits unexpectedly during this phase, please retry. I noticed that from time to time there are problems with the `node-modules` which are solved by starting the app a second time. This may be due to the fact that the Node dependencies used in this project are not the most recent and thus may have unsolved bugs.

Build your images

If you'd rather build the Docker images locally instead:

1. Clone this repo.
2. Move to the main folder of this repo with `cd todo-app`.
3. Compile and deploy the application using `make`.

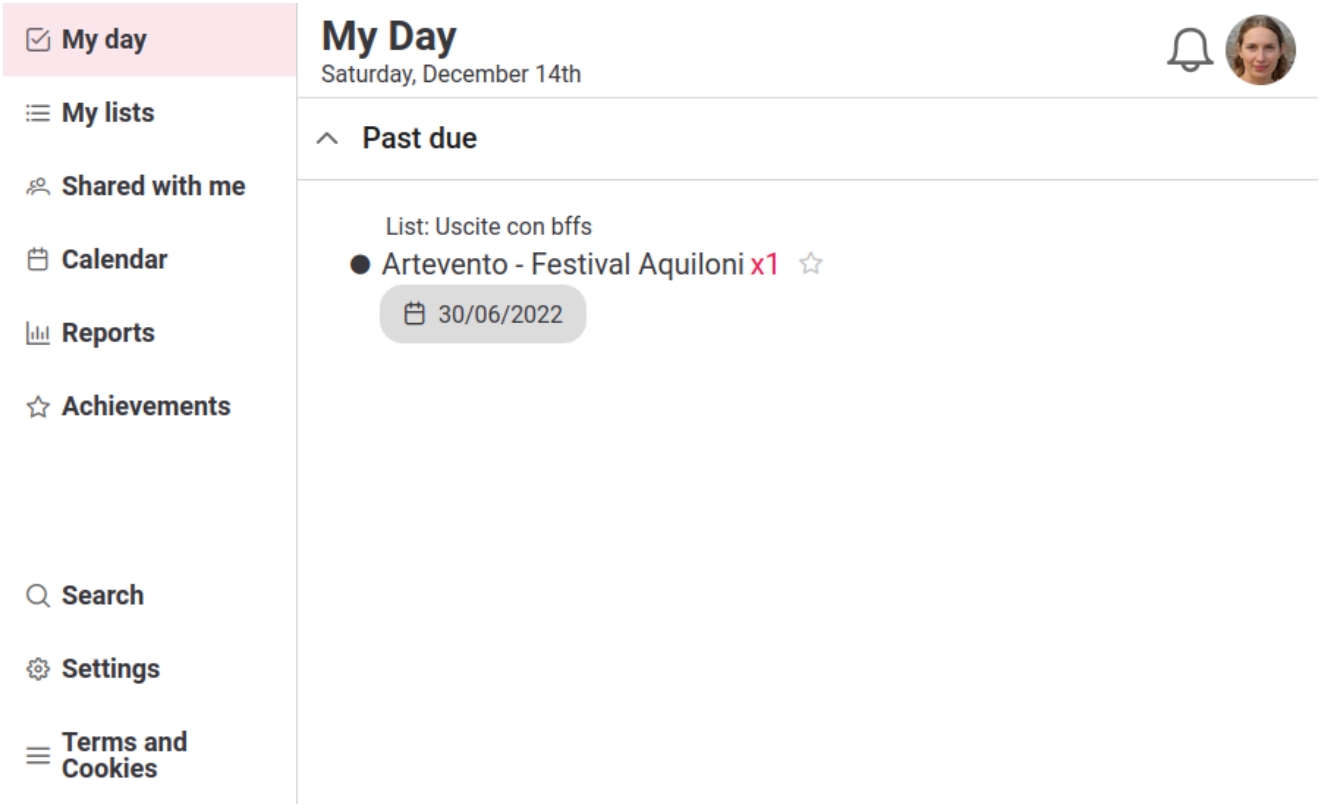
User Guide

The database is pre-populated with some accounts showcasing use cases. The accounts are as follows:

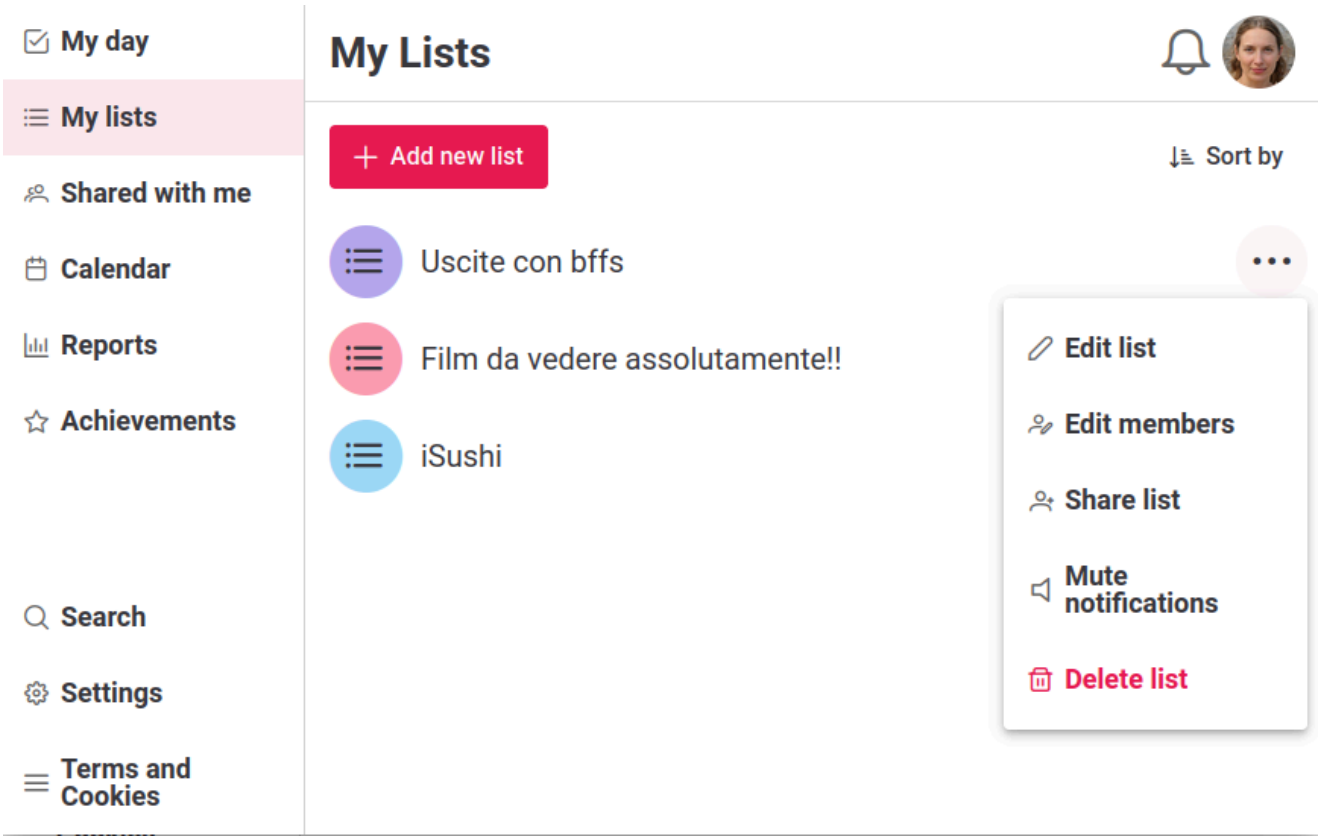
Username	Email	Password
sara	sara.camporesi@mail.com	Password1
lucy	lucia.hu@mail.com	Password1
chiarina	chiara.lombardi@mail.com	Password1
marco_98	marco.venturi@mail.com	Password1
ale	alessandro.raggi@mail.com	Password1
dalia	dalia.giunchi@mail.com	Password1

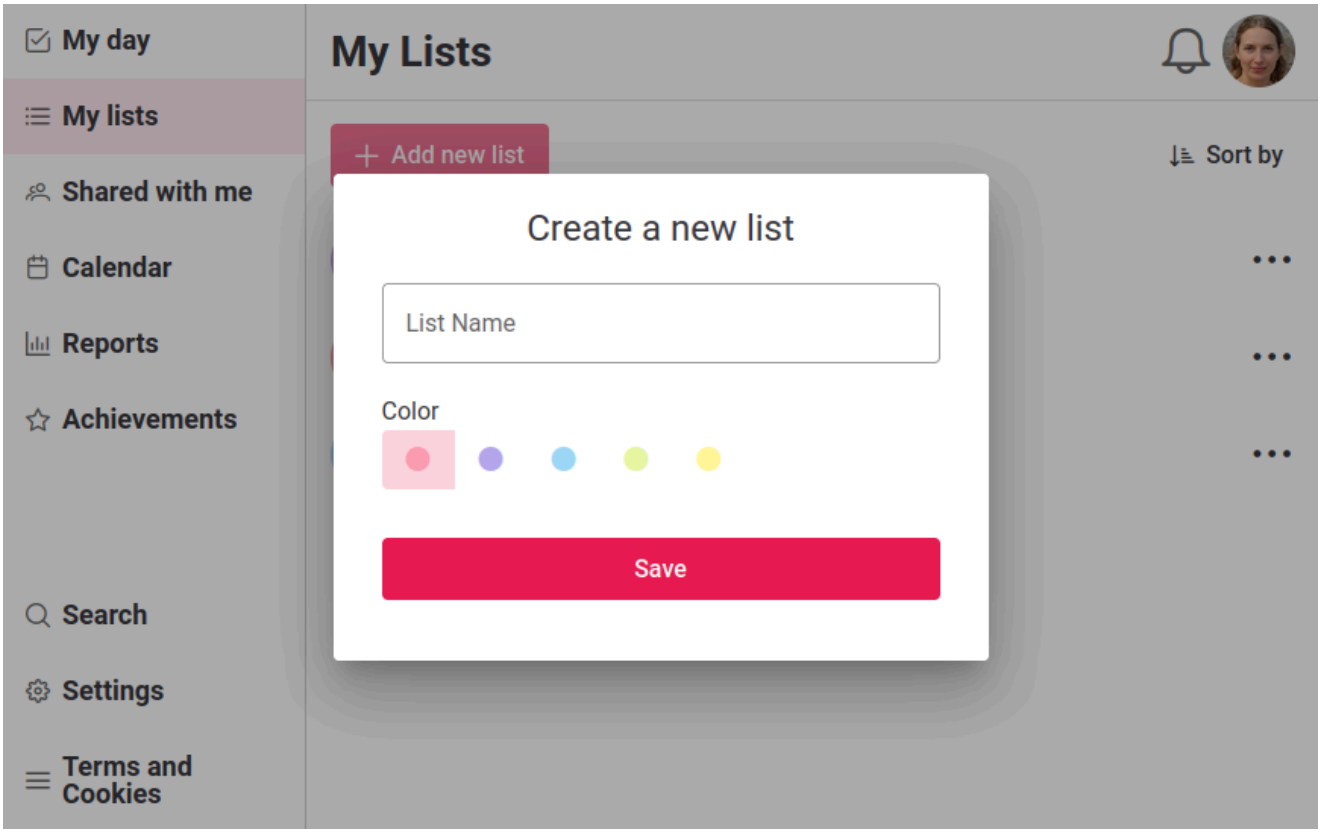
If you want to shut down the application, you only need to execute the command `make down` and Docker will shut down all containers and delete them.

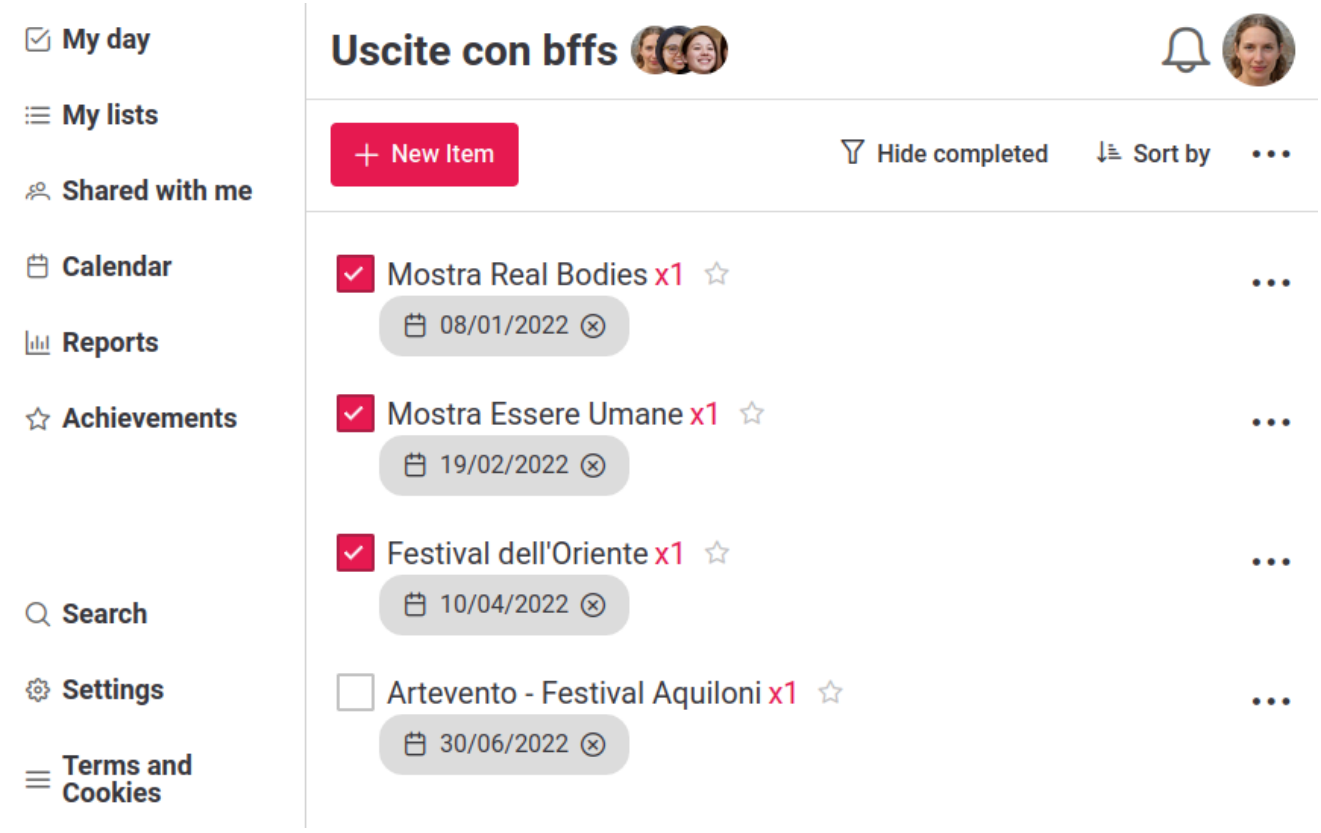
Upon accessing the application site via `localhost:9000`, log in with one of the accounts listed above.
The first page displayed is "My Day", where items with due date set to current day or next few days are displayed, as well as unchecked items with past due dates.
On the top right there is the notification center and the account drop down menu, from where one can access account settings.

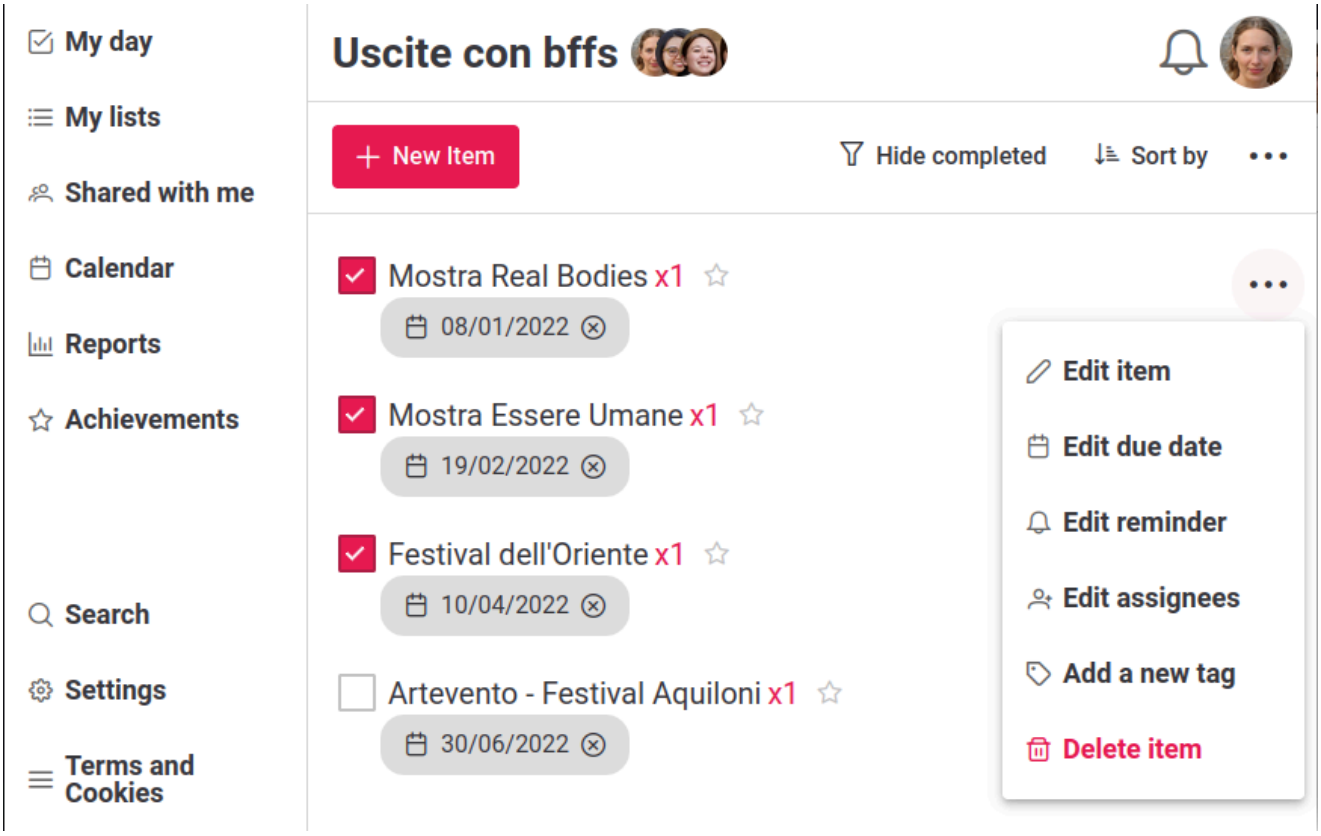


On the left side-bar the other pages a user with an account can access. From the "My lists" page the user can view the lists created by them, edit them, create a new one, view a specific list, edit the list items.


My day
My lists
Shared with me
Calendar
Reports
Achievements
Search
Settings
Terms and Cookies
My Lists
Add new list
Sort by
Uscite con bffs
Film da vedere assolutamente!!
iSushi
Edit list
Edit members
Share list
Mute notifications
Delete list


My day
My lists
Shared with me
Calendar
Reports
Achievements
Search
Settings
Terms and Cookies
My Lists
Add new list
Sort by
Create a new list
List Name
Color
Save





Closing remarks

I did not observe noticeable additional slowness with all requests passing through the NGINX server compared to without. The implementation is lightweight.

The add-on project with NGINX improved the web application in the context of distributed systems:

- the architecture is now more decoupled, allowing for maintenance and expansion to the Node servers without impacting user experience
- load balancing
- fault tolerance

All the while maintaining the responsiveness of a single page application, without requiring reloading the page or logging into the account again.

Now there is however a new single point of failure: the NGINX proxy server. All traffic passes through it.
This is a familiar situation in distributed systems, and can thus be comfortably managed; future expansions of this project might add replicas of the NGINX proxy server, with new consensus challenges to solve.

With the project as is, in a realistic scenario, all that is left to do is to put it in production and analyze the traffic in order to perfectly configure the number of replica servers, decide where they should be located for optimal response time coverage, and fine tune all the minute settings of NGINX.