Final Report Template
Roddit

Final Report for the Distributed Systems Course 2025/2026

Filippo Di Pietro
filippo.dipietro3@studio.unibo.it

Giovanni Babbi
giovanni.babbi@studio.unibo.it

June 4, 2026

This project consists in a social network like web application developed from scratch, following, as inspiration, similar products available on the market such as Reddit, to toggle the usual distributed services problems. On top of it, we developed the project through containerization by using Kubernetes in order to increase the resilience of the application against common runtime problems, as well as adding to the application monitoring services such as Grafana and Chaos Mesh to control and manage its execution.

Disclaimer

During the preparation of this work, the author(s) used the overleaf to allow group components to edit the project report independently. Moreover, the group components already had experience with the platform. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

This project consists of a social network web application, developed locally with the help of containerization. It follows similar principles of other ones of its kind, despite being developed following our personal view. The social allows to:

- Follow some subreddits
- Interact with posts that are posted on their subreddit, using likes and comments
- Create a post
- Comment a comment
- Uploading images (as a post or profile image)

The use cases:

- The users interact with the system every time they want to see some posts, but the client that they are using polls the system for new notifications (every 1 second), and if the user reaches the last displayed post, more posts are asked to the system
- The users interact with the system through the use of a web browser, that could be installed on a mobile device or a personal computer
- The users interact with the system using the buttons provided the html of the page, that instantiate an https connection that leads to computation to the system
- the system stores data into a mongodb database, moreover the database is a cluster sharded in multiple dynamic shards, more disk utilization implies the allocation of a new shard

The distribution is needed, because it allows to scale in number of users, posts, comments

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

- The system allows users to register and authenticate. As part of the acceptance criteria, users must be able to provide their personal data to use the application, and the system saves the session state using cookies, reducing the number of times the user must authenticate. The system must prevent two users from registering with the same username by marking the oldest registration as incomplete. Furthermore, the password entered by the user must be stored in the database in an unclear format.

- The system must display posts by the user and others, as well as any comments made on that post, as well as the number of likes received by that post. As an acceptance criterion, the number of likes persists between user visits. The system must also allow the user to add/remove a like from that post and leave a comment on it.
- The system must allow users to create new subreddits by entering their names, as well as allow them to follow/unfollow them. As part of the acceptance criteria, other users must be able to see and create new posts on those subreddits created by other users. Furthermore, the user who created the new subreddit becomes its follower.
- The system must allow users to search for other users, subreddits, posts, and the content of posts using a dedicated search bar. If the user selects a search term, the system must respond by displaying the resulting entity on the main page, if any, or the comment in which a full or partial mention of the search term was found using the search bar.
- The system must allow users to write new posts on an existing subreddit, specifying the subreddit it belongs to, its title, and its content. The content can be in the form of an image or text. Acceptance criteria include the post being associated with a subreddit, requiring the user to specify the name of an existing one. It is not necessary to be a follower of that subreddit to post there.
- The system must allow users to change their personal information, such as their username, password, username, and bio. After updating their information, the system must display the profile status change for both the affected user and other users.

2.2 Non Functional Requirements

Availability: The service must be highly available, minimizing downtime, thanks also to orchestration via Kubernetes and automatic restart of pods in the event of a failure.

2.3 Implementation Requirements

- The application must be developed and deployed using Kubernetes to simulate a distributed environment across multiple nodes (physical or virtual machines) and manage a shared, distributed database. Kubernetes also allows monitoring of pod and service metrics.
- The application logic must be implemented using the Flask framework, which handles client requests and interaction with the data persistence system.
- A chaos engineering tool, such as Chaos Mesh, must be integrated to test the robustness and resilience of the cluster by simulating controlled failures and abnormal conditions.

2.4 Relevant Distributed System Features

- **Fault tolerance and dependability**: The system must ensure the ability to continue functioning even in the event of partial failures. Specifically, communication between distributed components occurs via HTTP, which can be subject to latency or request loss at the network level; therefore, it is important that the system is designed to handle any retries or temporary call failures. At the infrastructure level, Kubernetes contributes to fault tolerance by automatically restarting pods in the event of an unexpected termination, thus improving service availability. In this way, the system maintains a good level of dependability, reducing the impact of failures and ensuring rapid recovery.

- **Scalability**: The system is designed to be scalable through the use of database sharding, which allows data to be distributed across multiple nodes. This approach allows for increased data volume and load management resulting from increased user activity over time. Specifically, entities such as users, subreddits, and posts are stored in the database, which can grow horizontally without compromising overall system performance.

- **Transparency**: The presence of database sharding is completely hidden from the user, who interacts with the system as if it were a single logical resource, without direct access to the data segmentation. From the developer's perspective, the system offers dashboard-based observability tools that allow monitoring the status of the service at various levels. Specifically, metrics related to both normal operation and critical situations, such as high load or high latency, are analyzed, allowing for effective diagnosis of any issues.

- **Evolvability and maintainability**: The application is monitored via Grafana dashboards that allow you to observe internal traffic and pod status metrics, making it easier to identify anomalies and problems. Furthermore, chaos engineering tools like Chaos Mesh allow you to test the system's resilience by simulating controlled failures and verifying the infrastructure's ability to react. These aspects contribute to maintainability, as they make it easier to diagnose and resolve problems, and to evolvability, as they allow you to introduce changes and improvements in a controlled manner, assessing their impact on the system.

- **Fault tolerance, dependability – availability, reliability, integrity, maintainability, safety** Fault tolerance can be ensured within the system by ensuring that, in the event of an unexpected component termination, they are automatically restarted by the Kubernetes orchestrator. This helps maintain high service availability. Availability is a fundamental requirement: the service must remain accessible at all times, as there are no mechanisms to manage downtime. Regarding reliability and integrity, data loss or corruption must be prevented. Using HTTPS ensures data protection in transit (confidentiality and integrity during communication), but data persistence and consistency must also be ensured at

the database level. Finally, the system must be able to recover quickly from any failures (maintainability), minimizing service interruption.

- **Security, trust**: In the system, user credentials and personal data are stored in a database; specifically, passwords are saved in hashed form using salting and peppering techniques to increase security. To use the services offered by the Flask application, user authentication is required, thus ensuring controlled access to resources. Monitoring dashboards (e.g., via Grafana) are external to the application and require dedicated credentials: this introduces a separation of access domains, increasing the overall level of security and trust in the system.

- **Resource sharing**: In the system, the Flask application accesses a distributed database via sharding, where data is divided across multiple nodes but made available as a single logical resource. Resources such as images can also be shared and accessed by different system components. This approach enables efficient resource management, improving scalability and ensuring coordinated access to data while maintaining its physical distribution.

- **Openness, interoperability, heterogeneity of components**: The system uses Grafana dashboards, which read data from Prometheus; the latter collects metrics from the Flask application for application monitoring. A form of heterogeneity is also observed, as the system is composed of several distinct components (Grafana, Prometheus, and Flask) that collaborate with each other.

- **evolvability, maintainability** : The system must be easily monitored and maintainable, thanks to the use of dashboards and observability tools (Grafana, Prometheus).

- **Cost and economy**: The system has costs related to the use of a distributed infrastructure based on Kubernetes and multiple components such as sharded databases, monitoring systems, and application services. Adopting sharding increases infrastructure costs, as it requires more nodes to manage data; however, it allows for efficient scaling as the number of users and tasks grows, avoiding bottlenecks and performance degradation. Similarly, the use of monitoring tools introduces resource overhead, but it allows for timely identification of problems and optimization of infrastructure use, reducing long-term operating costs. Overall: the increased complexity and resource usage is justified by the increased scalability, reliability, and load management capacity.

3 Design

3.1 Architecture

The system is based on a data-centered architecture, centered on the data management system. Specifically, the database (organized through sharding) is the central component around which the other system elements, such as the Flask application and monitoring services, revolve. All components interact primarily through access and manipulation

of shared data, rather than through complex direct communications between services. This approach centralizes information management, ensuring consistency and facilitating data access by different components. Furthermore, the use of sharding allows for the benefits of a data-centered architecture to be maintained while improving system scalability. Consequently, the system is highly data-driven, representing the main point of integration and coordination between the various components. Centralization is a deliberate design choice to facilitate database scalability and service monitoring.

3.2 Infrastructure

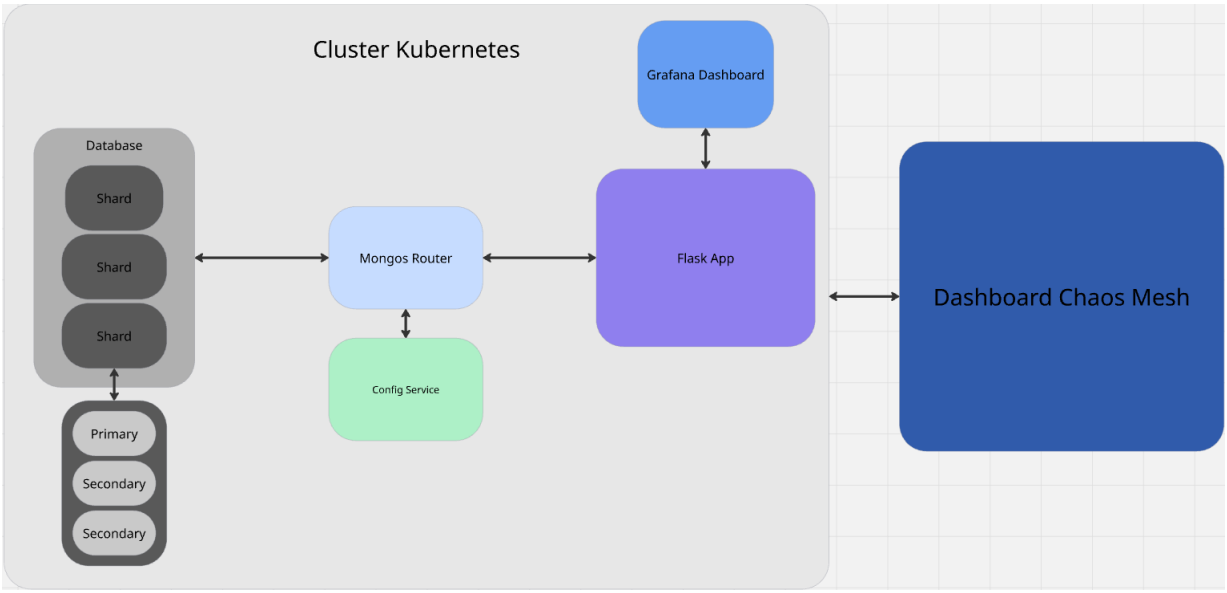

Figure 1: Entity Diagram

- Mongos: Router and load balancer. Has knowledge of the content and current status of the database. It's in charge of handling both the queries directed to the database and its scaling up process.
- Database: Defined using MongoDB, designed as a distributed database divided in shards. The shards are autonomous, independent from one another, entities that contains one portion, or the entirety, of the data stored in the whole database.
- Flask Application: Client. Main application used by the users, it sends queries to the mongos router.
- Grafana Dashboard: Monitoring. It coordinates with the main application through the collected metrics with Prometheus, it shows them and the current status of the application.

- Chaos-mesh Dashboard: Testing. It interfaces directly with the Kubernetes cluster. It allows the execution and management of experiments designed to simulate events that may harm the stability of the services of the cluster. The experiments are developed as yaml files.

The described components are executed in a Kubernetes cluster hosted locally in a single machine. They interact with each other through the Kubernetes integrated DNS server that handles the requests between services.

3.3 Modelling

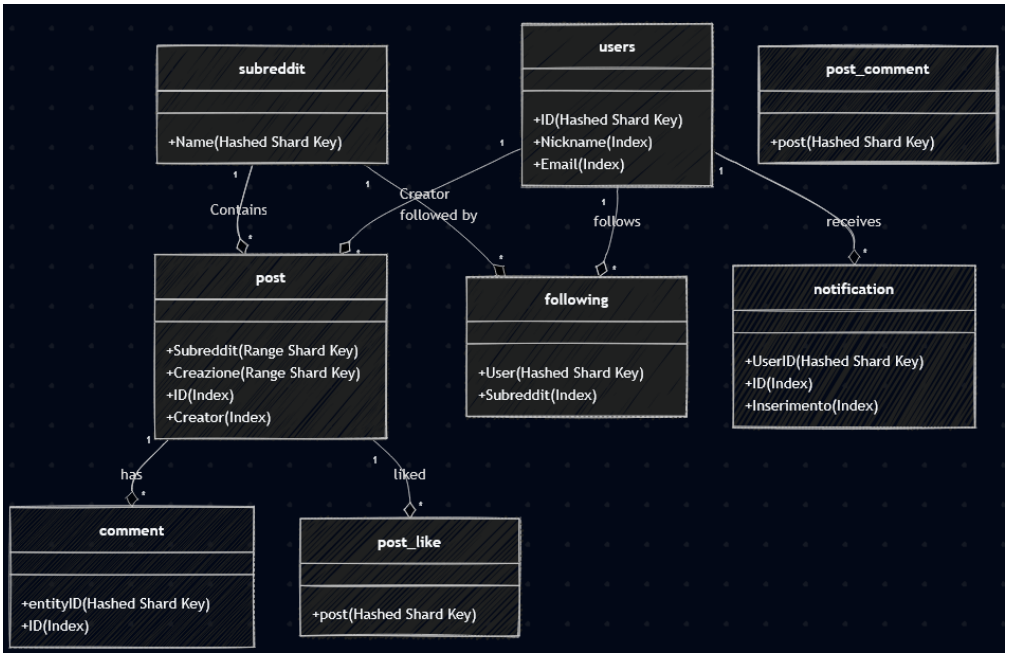

Figure 2: Database class schema

- **Domain entities**: In the system are present the domain entities to handle the main elements managed by the application, such as the authenticated users with their personal information, the subreddits that represent the communities where users can make posts of a given subject, the posts them self which contain the messages or images shared by the users. There are also comments made as a reply to a specific post. These entities define the domain in the system which are then stored in the database.
- **Mapping between domain entities and infrastructural components**: The domain entities are distributed across all components based on their usage and required persistence. Main entities such as users, posts and comments are stored in the distributed database which contains the information related to the current

system status, which grants both scalability and efficient data handling. The Flask application acts as an intermediary between the user and the database: it receives the client requests, handles the user interactions(such as comment and post making) and interacts with the database to read and update its data. The client doesn't have a persistent state and it handles only the visual representation.

- **Domain Events**
  - **registration** on user subscription.
  - **login** on user authentication.
  - **create new subbreddit** when a user creates a new subreddit.
  - **create new post** when a user creates a new post.
  - **like** when a user adds a like to an existing post.
  - **dislike** when a user removes a like to an existing post.
  - **comment** when a user replies to an existing post or another user comment.
  - **follow** when a user follows an existing subreddit
  - **unfollow** when a user stops following a specific subreddit
  - **change settings** on user's settings update.
- **Sorts of messages are exchanged**: *Commands*: They represent requests to alter the current state of the system, such as, the creation of a new post, the addition of a comment and the registration of a new user. These messages are sent from the web browser to the flask app, which contains update operation of the database. *Queries*: These are read requests that don't change the system state. For example, getting a list of posts in a subreddit, displaying comments, or retrieving a user's profile. These operations query the database and return the data to the client.
- **System state**: The system state includes all the persistent and relevant information required for the correct functioning of the application, as well as the state of the distributed infrastructure. Specifically, it includes data related to key domain entities, which are stored in the distributed database. Moreover, it also includes infrastructure information, such as the number of active pods that make up the services within the Kubernetes cluster, as this information directly impacts the system's availability, scalability, and capacity.

3.4 Interaction

- The system components communicate mainly via HTTP/HTTPS protocol according to a client-server model. The clients send requests to the Flask application, which manages the business logic and coordinates access to the distributed database.

- The flask app communicates with MongoDB through the PyMongo driver, which opens a connection pool of persistent connections to the mongos router using the mongos service. When a Flask route is invoked, PyMongo serializes the query parameters into BSON and sends it to the Mongos router. Upon receiving the query, it consults the config server to obtain the chunk map of the relevant collection: a table that associates each range of shard key values with the shard to which it belongs. Depending on the result, two scenarios arise: if the query contains a filter on the shard key, Mongos identifies the exact shard and forwards the request to it (targeted query); if the query does not filter on the shard key, Mongos broadcasts to all shards and aggregates the responses (scatter-gather). The shard involved, the primary, receives the query, reads the documents from disk (from its PersistentVolume), serializes them into BSON, and sends them back to the router. The router aggregates the multi-shard response, returns it to PyMongo, which deserializes it into Python objects, and the Flask app can then construct the HTTP response.

- As for the monitoring system with Grafana, a *pull* model is adopted.
  - The Flask application exposes metrics via an HTTP endpoint called `/metrics`. Each time an application route is invoked, internal (in-memory) counters representing the status and performance of the service are updated.
  - Prometheus periodically queries this endpoint to collect metrics. Specifically, it makes HTTP GET requests every 15 seconds to `/metrics`. The endpoint returns data in standard Prometheus text format.
  - Requests are routed via the cluster's internal DNS to one of the available pods. The collected data is then saved in a time-series database managed by Prometheus.
  - Grafana queries the Prometheus database to display metrics in the form of a dashboard, allowing monitoring of system health and performance.

3.5 Behaviour

- One of the main components responsible for altering the system state is the *autoscaler*, a cluster component responsible for managing the horizontal scaling of the sharded MongoDB database. The autoscaler establishes a connection to the MongoDB cluster and periodically monitors the state of the shards, performing checks at regular intervals (every minute) on the available capacity of each node. When the available space on existing shards drops below a certain threshold, the autoscaler automatically initiates the scaling process:
  - Creates and configures a new shard;
  - Updates the MongoDB cluster configuration so that the new shard is recognized by the system;
  - Enables data balancing (*rebalancing*) across available shards, when supported.

3.6 Data and Consistency Issues

- The application requires the storage of user interaction data in the database. This includes posts published, likes assigned, comments made, and followed subreddits. Furthermore, the information provided during registration on the platform and the created subreddits are also stored.
- The data is stored as documents within the MongoDB database. During data entry, logical relationships between the various entities (users, posts, comments, subreddits) are maintained using primary/foreign key pairs. This preserves the association between interactions and their authors, introducing a form of non-repudiation and traceability of user actions.
- There are two main components that regularly query the MongoDB cluster:
  - The Flask app, which creates read and write queries in response to user actions and routes invoked by the application.
  - The dynamic autoscaler which periodically monitors the capacity of the database shards to determine whether the database needs to be scaled up. If necessary, it requests the Mongos router to add a new shard and initiate a data rebalancing among the existing shards.
- The main information kept outside the database are:
  - temporary authentication and session data used to maintain the user's login between multiple sessions;
  - metrics collected locally by Prometheus and subsequently visualized via Grafana;
  - any images or multimedia files saved in the local filesystem or in dedicated storage, while only the path or reference to the resource is stored in the database.

3.7 Fault-Tolerance

- The MongoDB database is distributed using sharding, which divides the data volume across multiple nodes and improves system scalability. Distribution occurs through shard collection, which allows data to be divided across the cluster's different shards. Each shard is organized as a replica set consisting of a primary node and two secondary nodes. The primary node manages write operations and synchronizes data with the secondary nodes. In the event of a primary node failure or downtime, the secondary nodes automatically initiate an election process to select a new primary, ensuring service continuity and fault tolerance. Furthermore, the dynamic autoscaler can automatically create new shards following the same logical structure.
- All system components, including databases, application services, and monitoring tools, are deployed within a Kubernetes cluster. The deployment is managed in

the form of Kubernetes resources such as Services and Deployments. Kubernetes constantly monitors the status of the pods associated with the various services and, if a pod fails or terminates abruptly, automatically restarts it. This behavior helps improve the system's availability, resilience, and fault tolerance.

3.8 Availability

- **Load balancing**:
  - Kubernetes automatically distributes requests across multiple Flask app pod replicas via *Service*. This prevents overloading a single instance and improves availability and scalability.
  - MongoDB implements a form of data balancing similar to the one mentioned above: the *mongos* router queries the shard containing the collections needed for the queries and returns the records corresponding to the query. Meanwhile, the balancer, running in the background, can redistribute chunks across shards to maintain uniform data usage.

  It allows to:
  - Distribute the load across multiple components
  - Improve performance
  - Increase fault tolerance

- **Behavior in case of network partitioning**: When a network partition occurs, the system behavior depends on the component involved.
  - MongoDB:
    * Each shard is organized as a replica set.
    * If the primary node becomes unreachable but the secondaries maintain quorum, an automatic election is performed to select a new primary.
    * If quorum cannot be achieved, write operations are temporarily blocked to preserve data consistency until the primary is automatically rebooted.
  - On the application side:
    * Kubernetes can detect unreachable or failed pods and automatically restart them.
    * Its Services continue to route traffic only to pods deemed healthy.

3.9 Security

- To use the application, you must register for a user account, which requires some personal information and authentication credentials. This information is then used during the login process and can be changed by the user after authentication. The application does not distinguish between administrator accounts and standard users.

- The monitoring dashboards, such as Grafana and Chaos Mesh, require dedicated credentials to access their respective administrative services and features.
- Access to MongoDB is managed through authentication. Database requests are handled by two separate profiles, one requiring administrator access to the server, and the other requiring access to the individual database. The credentials for the accounts used are created manually within the file sources.

4 Implementation

- HTTP/HTTPS is used as the primary network protocol for managing Flask application routes and for communication between the client and application server. HTTP requests are used for read and update requests, such as authentication, post creation, content retrieval, and user interactions.
- Data exchanged via the Flask APIs is typically represented in JSON format, which is automatically serialized and deserialized by the framework when handling HTTP requests and responses.
- For monitoring and management components (such as Grafana, Prometheus, and Chaos Mesh), connections between services rely on the underlying TCP/IP protocols, while exposed APIs and endpoints continue to predominantly use HTTP.
- Communication with MongoDB occurs via the database's native protocol, which uses BSON (*Binary JSON*) as the data serialization format. The Flask app communicates with the *mongos* router, which then forwards queries to the appropriate shards in the distributed MongoDB cluster. The database is NoSQL and is queried in JavaScript.
- The autoscaler-rbac (role-based access control) manages the roles and permissions needed to be assigned to the autoscaler service pod so that it can obtain information on the status of other pods and perform scale up.

4.1 Technological details

- The system was built using Flask, a Python micro framework that allows for the rapid implementation of REST APIs and HTTP routes needed to manage user interactions, such as authentication, post creation, comments, and content retrieval. It easily supports monitoring tools like Prometheus, allowing metrics to be exposed via dedicated endpoints (`/metrics`). An additional advantage is its compatibility with containerized and Kubernetes-orchestrated environments: Flask can be easily deployed within replicable pods.

5 Validation

- To test most of the behavior of the flask app components we relied on manual testing using log messages and checks of the data inserted into the database, as well as runtime monitoring of the application behavior.
- For cluster robustness testing and for autoscaling, horizontal scaling, and fault tolerance mechanisms, we relied on Chaos-Mesh by running experiments aimed at simulating some of the most common events.
  - The events taken into consideration are: sudden termination of a service pod, increase in the service's computational load and increase in the latency in a service's responses.
  - The targets of these experiments are the Mongos router as the main access point to MongoDB and the Flask app as it is the main application of the developed service.
  - For the three experiments, three different yaml sources were developed for each of the mentioned target services:
    - Chaos-heavy-load: The experiment simulates situations where the targeted service pod reaches 100% CPU usage.
      - Flask app, having developed a yaml source to manage its hpa, after reaching the specified threshold (70%), a replica of the service is created.
      - Mongos router, It doesn't have a file to create additional replicas, so only one replica of the service pod is active. During testing, the service remained running, but some requests timed out, and the autoscaler log reported "service unreachable." As soon as the experiment was terminated, the queries were executed normally.
    - Chaos-high-latency: The experiment will introduce a 500ms latency increase to all messages sent to the targeted service.
      - Flask app, When running the application without using port forwarding, response times were visibly higher.
      - Mongos router, It was necessary to check the response time between the Mongos router and the shard, which also took over 500ms per request.
    - Chaos-pod-termination: The experiment simulates the termination of a currently running pod of the targeted service, which happens once every 2 minutes.
      - Flask app, The service remained unavailable for about ten seconds, once the application pod became operational again, the web service also became operational again.

· Mongos router, Once finished, it took about the same amount of time as the Flask app to become operational again. If the shard status check was in progress during that time, the autoscaler would respond with a service unreachable. Once the router was restarted, it would still be able to connect to the shards normally and interact with the autoscaler.

6 Deployment

6.1 Deployment commands

For a better view of the commands it's preferable to follow the README in the Github Repo.

If you are going to execute these commands on a Windows machine, please be sure to have a Docker Desktop process active in the background.

Firstly we start Kubernetes

```
minikube start --container-runtime=containerd
```

Next we install Heml with admin privileges

```
choco install kubernetes-helm
```

or with Winget:

```
winget install Helm.Helm
```

Add required repos for prometheus/grafana and chaos mesh + update

```
helm repo add prometheus-community https://prometheus-community.github.io/
helm repo add chaos-mesh https://charts.chaos-mesh.org
helm repo update
```

Install the prometheus/grafana stack

```
helm install my-release prometheus-community/kube-prometheus-stack
```

Creates a new separated namespace and installs chaos-mesh stack in it.

```
helm install chaos-mesh chaos-mesh/chaos-mesh -n chaos-mesh --create-name
```

We create the mongodb shard

```
kubectl apply -f mongodb-service.yaml
kubectl apply -f mongo-configsvr.yaml
kubectl apply -f autoscaler-rbac.yaml
kubectl apply -f autoscaler-configmap-avg.yaml
kubectl apply -f autoscaler-deployment.yaml
kubectl apply -f mongodb-shard.yaml
kubectl apply -f mongos-deployment.yaml
```

Set up the database schema with sharding policies, you need to substitute ¡pod_of_mongos_router¿ with the name of the mongos pod

```
kubectl cp src/database/init-db.js <pod_of_mongos_router>:/tmp/init-db.js
kubectl exec -it <pod_of_mongos_router> -- mongosh /tmp/init-db.js
```

Set up flask

```
kubectl apply -f flask-service.yaml
kubectl apply -f flask-deployment.yaml
kubectl apply -f flask-hpa.yaml
```

Set up the grafana service and monitor

```
kubectl apply -f webapp-service.yaml
kubectl apply -f flask-monitor.yaml
```

For starting the deployment on port 5000, type:

```
kubectl port-forward deployment/flask-app 5000:5000
```

Set up the port-forwarding for visualizing grafana dashboard from web browser

```
kubectl port-forward deployment/my-release-grafana 3000:3000
```

Get the Grafana password required to login in the browser.
NOTE: The command may vary depending if you are on Linux or Windows.
Linux Version:

```
kubectl get secret --namespace default my-release-grafana -o jsonpath="{.
```

Windows Version:

```
$pass = kubectl get secret my-release-grafana -o jsonpath="{.data.admin-p
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String
```

Creation of local admin user to interact with chaos mesh dashboard

```
kubectl create serviceaccount chaos-admin -n chaos-mesh
kubectl create clusterrolebinding chaos-admin-binding --clusterrole=cluste
kubectl create token chaos-admin -n chaos-mesh
```

Set up the port-forwarding for visualizing chaos mesh dashboard from web browser

```
kubectl port-forward -n chaos-mesh svc/chaos-dashboard 2333:2333
```

Once the port forwarding is enabled you can login on the Chaos-Mesh dashboard using the token obtained with the previous command and as service account chaos-admin.
Enable metrics-server to allow hpa

```
minikube addons enable metrics-server
```

Execution of the experiments (DO NOT EXECUTE ALL AT ONCE)

```
kubectl apply -f chaos-high-latency-flask.yaml
kubectl apply -f chaos-high-latency-mongos.yaml
kubectl apply -f chaos-heavy-load-flask.yaml
kubectl apply -f chaos-heavy-load-mongos.yaml
kubectl apply -f chaos-pod-termination-flask.yaml
kubectl apply -f chaos-pod-termination-mongos.yaml
```

To check if experiments are working:

- for chaos-high-latency-flask.yaml:

```
minikube service flask-service --url
curl -o nul -s -w "tempo_totale:%{time_total}s\n" <url obtained>
```

- for chaos-high-latency-mongos.yaml:

```
kubectl exec -it <mongos pod> -- mongosh --quiet --eval "var s=Da
```

- for chaos-heavy-load-flask.yaml:

```
kubectl get hpa -w
```

- for chaos-heavy-load-mongos.yaml, you may use these commands in two different consoles:

```
kubectl logs -f <mongos pod>
kubectl logs -f deployment/mongodb-autoscaler
```

- for both chaos-pod-termination-flask.yaml and chaos-pod-termination-mongos.yaml:

```
kubectl get pods -w
```

6.2 Deployment scripts

- **Dockerfile**: a Dockerfile for a Python 3.11 Flask app:
  - Installs dependencies from requirements.txt (no cache, keeping the image lean)
  - Sets PYTHONDONTWRITEBYTECODE and PYTHONUNBUFFERED for cleaner container logging
  - Runs Flask on 0.0.0.0:5000 (accessible from outside the container)
  - Exposes port 5000
- **mongodb-service.yaml**: this YAML defines two Kubernetes Services:
  - *shard1*: headless service (clusterIP: None): exposes individual pods with stable DNS names like shard1-0.shard1, required for the MongoDB replica set on a StatefulSet. Port 27018.

    - *mongos*: standard ClusterIP service: single endpoint that load-balances traffic across ‘mongos‘ pods. Port 27017.

- **mongo-configsvr**: this YAML defines three resources:
    - *config-svc*: headless service for the config server pods. Enables stable DNS (configsvr-0.config-svc...). Port 27019
    - *configsvr*: StatefulSet that runs a single MongoDB config server (–configsvr –replSet=configRepl) on port 27019 with 5Gi persistent storage. A readiness probe pings it with mongosh every 5s before marking the pod ready.
    - *init-configrs*: One-off init job that:
        1. Waits until configsvr-0 responds to ping
        2. Calls rs.initiate() to bootstrap the configRepl replica set
        3. Checks rs.status().ok — if it’s already initialized, skips; otherwise retries initiation
- **autoscaler-deployment.yaml**: It defines the env variables used by the other yaml files and creates defines the container in which the autoscaler will be executed.
    - serviceAccountName: mongodb-autoscaler, it binds the account properties defined in the rbac file to allow the pod to interact with the others.
    - containers.args: defines to install the required dependencies, such as the mongo driver and kubernetes, and executes the py script defined in the autoscaler-configmap.yml.
    - env: it defines the environment variables required for the yml files, and the volume mount for the py script.
- **autoscaler-rbac.yaml**: it defines 3 different resources:
    - ServiceAccount: the identity to connect the account rules define in this yaml to the previous one.
    - Role: it defines what is allowed to do with which resources.
        - StatefulSet: allows the autoscaler to read existing StatefulSets and create/modify new ones when it decides to scale up.
        - Service: When a new shard is created, a dedicated Headless Service is needed to allow internal communication between the pods in the replica set.
        - Pod: Read-only permissions. The autoscaler must monitor the status of the pods.
        - PVC: Minimum read permissions. The autoscaler reads PVCs to calculate the currently occupied storage and decide whether it’s required to scale up.

    * RoleBinding: it connects the previously defined roles to the ServiceAccount metadata name, to be used on the other yaml files.

- **autoscaler-configmap-avg.yaml**: it defines the python script of the autoscaler. By defining it inside the yaml file it's easier to execute and to edit. Since it's executed inside a dedicated container its execution is independent from the rest of the environment.
    * The script is an infinite loop that every n seconds checks if it's required to perform the scale up, with a global try/except to avoid crashing in case of transient errors.
    * evaluate_scaling(): main method of the loop:
        · Cooldown: before doing anything, check if at least 300 seconds have passed since the last scale-up. If not, skip the cycle.
        · Reading active shards: asks the mongos router the list of currently registered shards using the listShards command. If none are found, the loop is skipped.
        · Storage measurement: For each shard, it connects directly to its primary node, without querying the router, and runs dbStats to read the dataSize of the roddit database. It then calculates the average per shard, not the total.
        · scale up criteria: checks the average per shard, if it's lower than a specified threshold (current threshold, 10 MB). If it's lower, it doesn't do anything. Otherwise, it creates a new shard. The current max shard limitation is to prevent the autoscaler to create an unlimited number of shards.
- provision_new_shard(): event sequence performed during scale up process:
    * Creates a Kubernetes Service with no IP that allows DNS resolution of individual pods in the new shard.
    * Creates a StatefulSet with three replicas, each with its own 500Mb PVC. The container starts mongod in –shardsvr mode with the replica set name equal to the shard name.
    * Wait for pods to be ready, polls every 10 seconds for pod status, with a 5-minute timeout. Only when all three pods of the new shard are ready the procedure continues.
    * It connects directly to the first pod fo the shard and initializes the MongoDB replica set, defining the three members with their internal DNS hostnames. Then, it waits 15 seconds for the primary to be elected.
    * Connects to MongoDB and registers the new shard with addShard. From this point onward, MongoDB can automatically start moving chunks to the new shard using its internal balancer.

  - In order to prevent the autoscaler to initiate the same shard multiple times, it tracks the currently processed ones in a set.

- **mongodb-shard.yaml**: this YAML defines two resources
  - *shard1*: StatefulSet that runs 3 MongoDB shard members (–shardsvr –replSet=shard1) on port 27018, each with 10Gi persistent storage. Pods get stable DNS names shard1-0, shard1-1, shard1-2
  - *init-shard1rs*: One-off init job that:
    1. Waits for all 3 pods to respond to ping
    2. Checks rs.status().ok on shard1-0 — if already initialized, skips
    3. Otherwise calls rs.initiate() with all 3 members; exits with code 1 on failure so Kubernetes retries the job (restartPolicy: OnFailure)

- **mongos-deployment.yaml**: this YAML defines two resources
  - *mongos*: deployment that runs the MongoDB query router with two init containers that block startup until both configsvr-0 and shard1-0 are reachable. Once they are, mongos starts pointing at configRepl as its config server. A readiness probe ensures traffic only reaches it after it's responsive.
  - *add-shard*: One-off job that:
    1. Waits for mongos to be reachable
    2. Checks if shard1 is already registered via sh.status()
    3. If not, calls sh.addShard() to register the shard1 replica set with the cluster

- **flask-service.yaml**: A NodePort Service that exposes the Flask app:
  - Routes external traffic from node port 30001, cluster port 80, pod port 5000
  - Selects pods with label app: flask

- **flask-deployment.yaml**: A Deployment running a single Flask pod:
  - Image: dippi9845/flask-app:1.0, always pulled fresh
  - Exposes port 5000
  - CPU capped at 50m (0.05 cores), memory between 128Mi–256Mi

- **flask-hpa.yaml**: A HorizontalPodAutoscaler for the flask-app deployment:
  - Scalable from 1 to 5 replicas based on CPU usage (the maximum of 5 replicas was set to avoid consuming too many hardware resources during testing)
  - Adds pods when average CPU utilization exceeds 70%, scales down when below

- **webapp-service.yaml**: It defines the endpoint to be queried by the other files.

- **flask-monitor.yaml**: It's a custom Prometheus Operator resource. It specify the Prometheus Operator to query the flask app at the given port every 15 seconds and make a GET request on /metrics resource.
- **mongodb-exporter.yaml**: it performs two different tasks:
  - MongoDB doesn't expose the metrics in native Prometheus format. The Percona container acts as a translator, it connects directly to the shard primary, queries MongoDB with its internal commands and re-exposes them in Prometheus format.
  - Just like the webapp-service, it also creates a stable endpoint within the cluster that the ServiceMonitor can reach
- **mongodb-servicemonitor.yaml**: Just like the flask-monitor.yaml, it tells Prometheus to scrape the mongodb-exporter service on the metrics port every 15 seconds.

7 User Guide

To start using the social firstly we need to login to the platform, otherwise you need to register

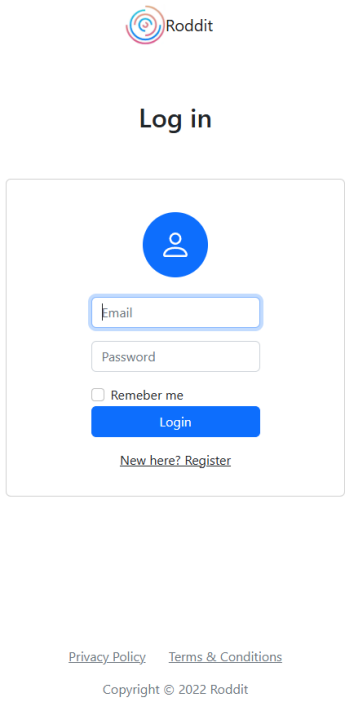

Figure 3: This is image shows the main entrance for the website

7.1 Registration

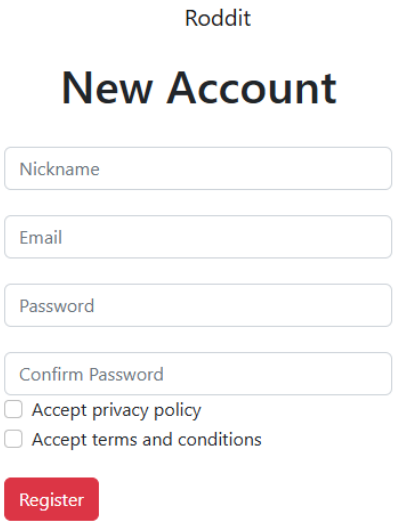

Figure 4: This is image shows the page of registration, to register is asked: the nickname to use in the website, an email, a password and a confirm of the password inserted. You need to accept privacy policy and terms and condition in order to register

7.2 Main Page

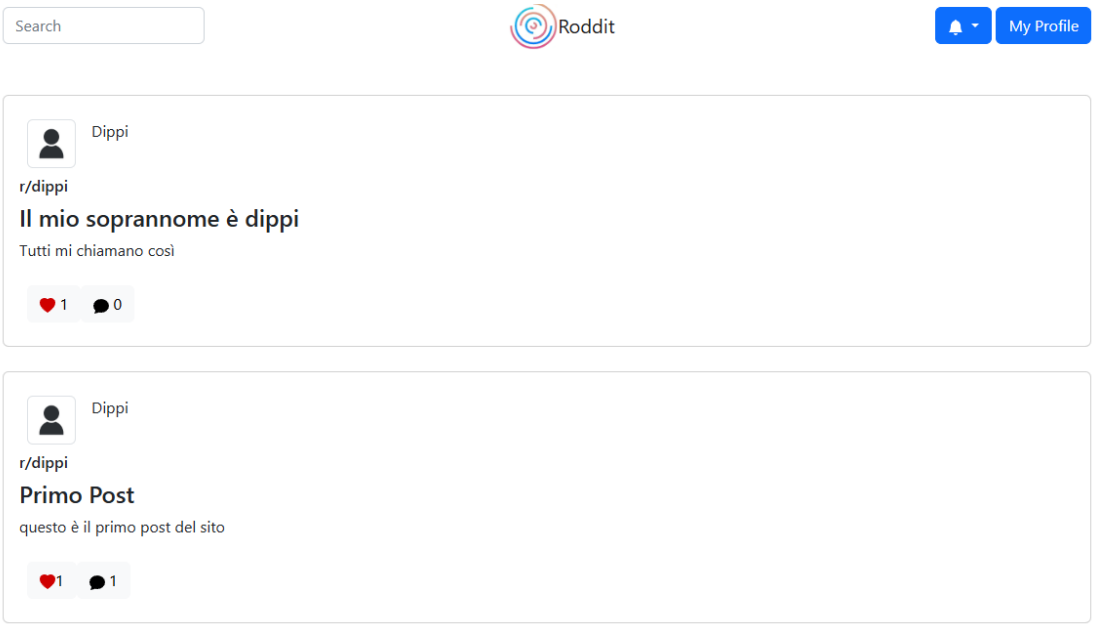

Figure 5: This is the main image shows the post of the subrreddit that you follow, each post you can put a like or leave one or more comments. In the top you can see there is a searchbar, the button for the notifications, and the a button to see you profile and more

7.3 Comment

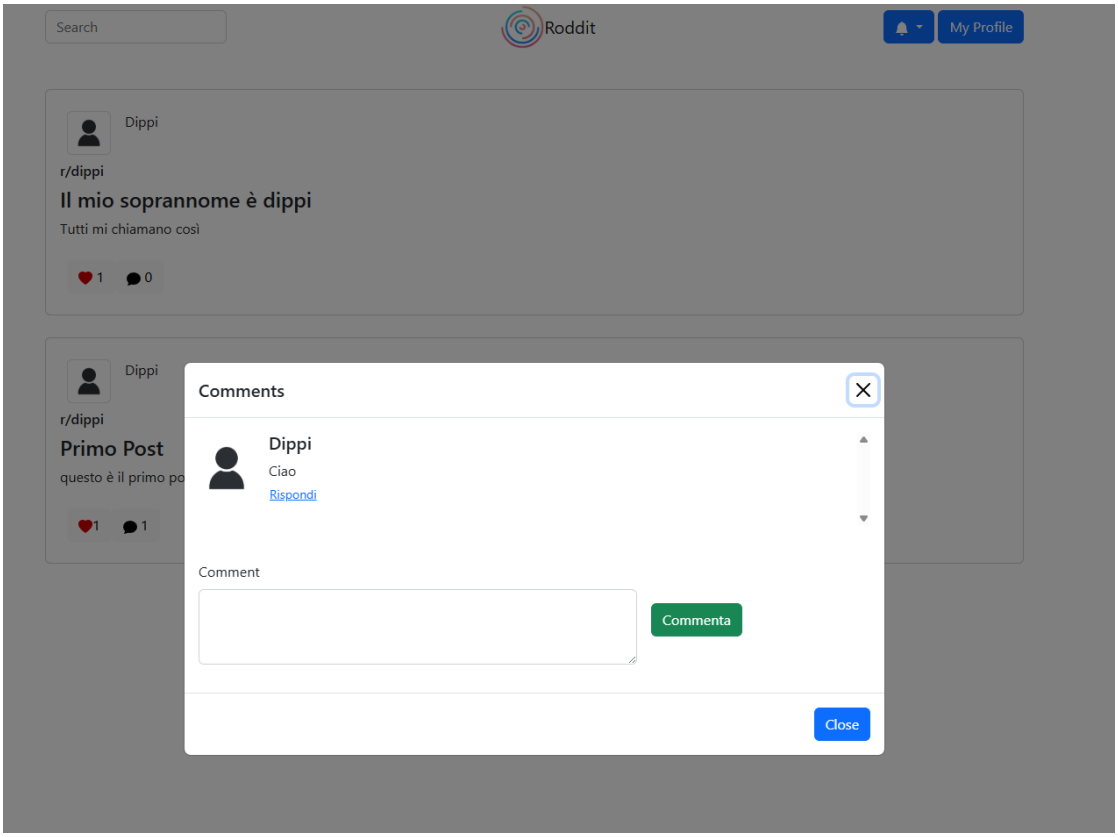

Figure 6: Here as you can see there is the possibility to leave a comment or to respond to another comment

7.4 Profile

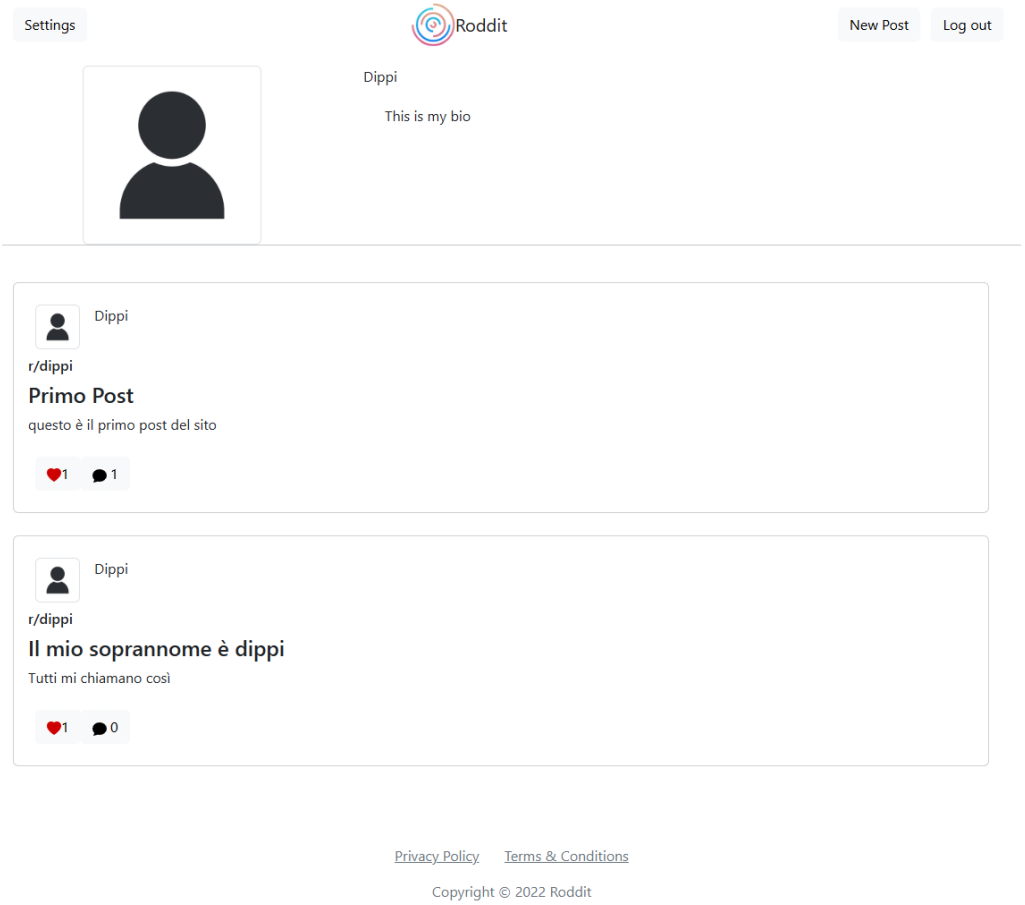

Figure 7: This page shows your profile, with your post, your biography , and allows you to create a new post or to change the settings of your profile

7.5 Settings

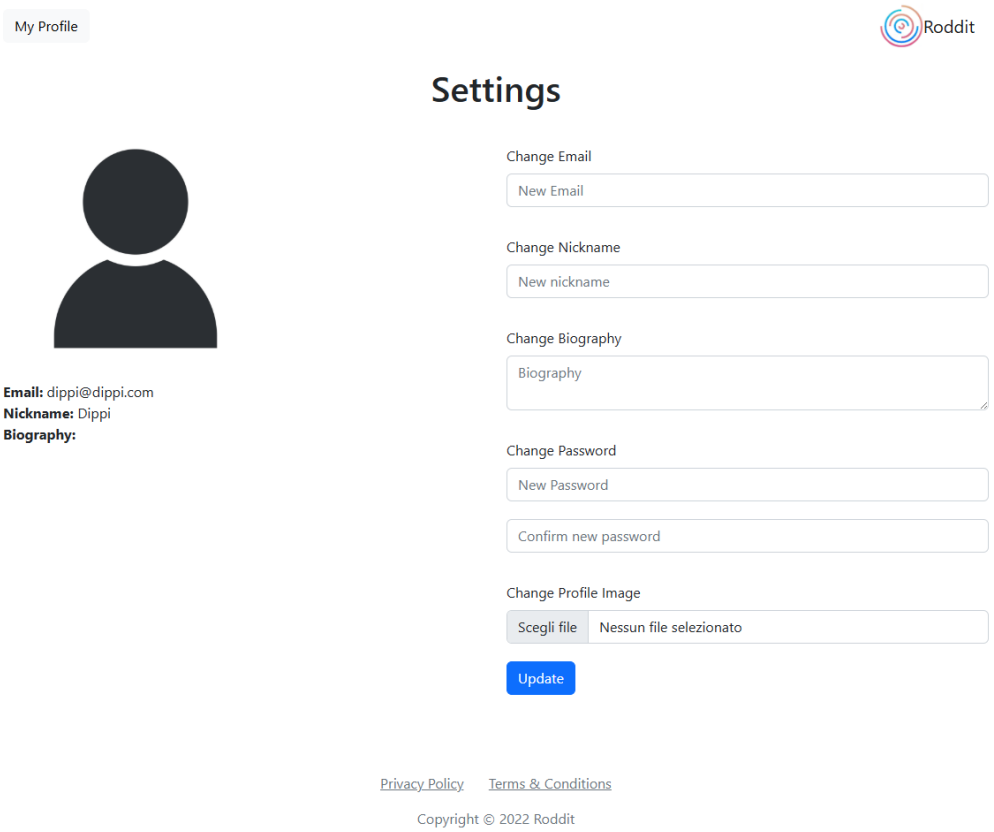

Figure 8: This page allows you to change settings of your personal profile

7.6 Create new post

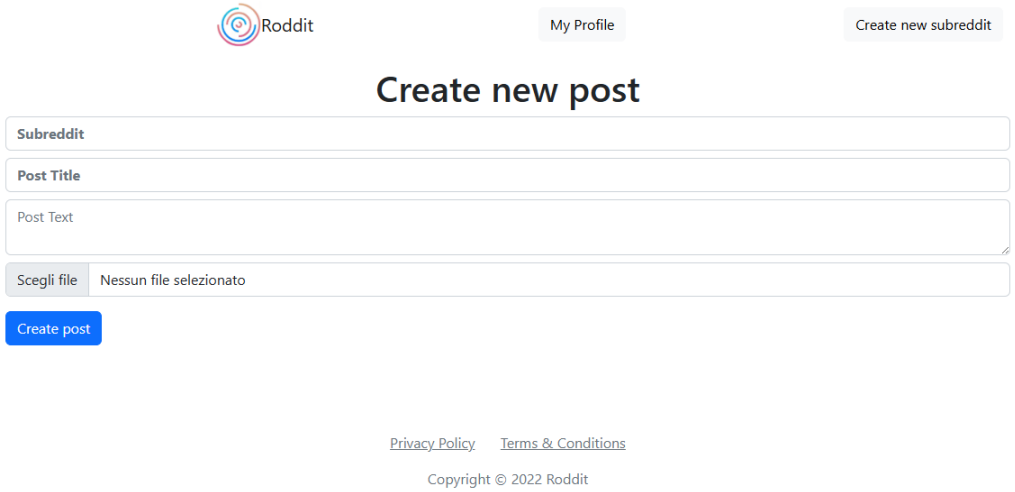

Figure 9: This page allows you to create a new post by specifying the subreddit, and giving a title and a body text and, optionally, an image. It's required to specify the name of an existing subreddit.

7.7 Create new subreddit

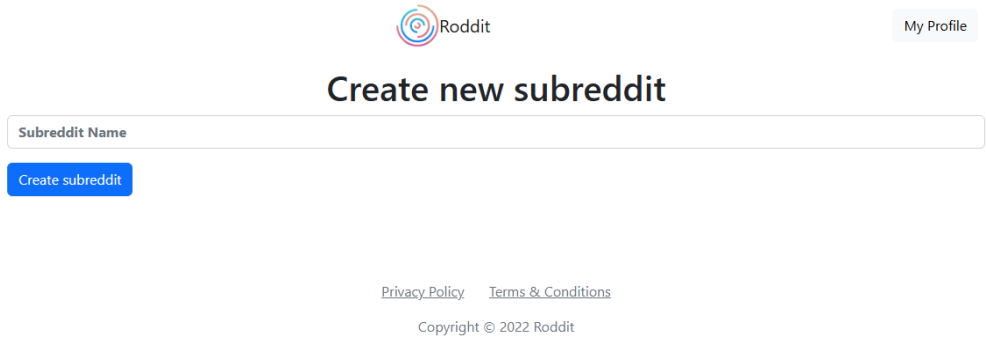

Figure 10: this page allows you to create a new subreddit

7.8 Searchbar

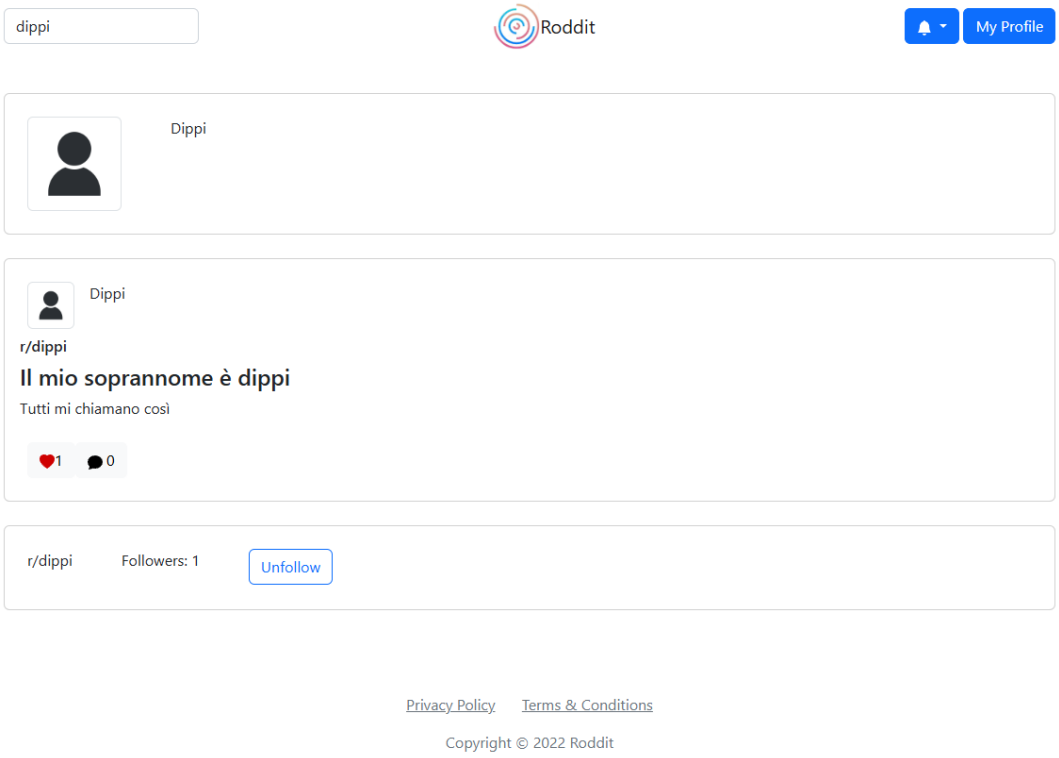

Figure 11: The searchbar allows you to search for: text inside a post, profiles, and subreddit

8 Self-evaluation

- Filippo Di Pietro:
  - Built the entire frontend, its strength is being responsive, weakness is simplicity
  - Built the entire backend, its strength is being versatile, weakness is almost entirely a single file
  - Building a single shard with 3 replicas is a strength that is fault-tolerant, but a single shard does not allow for the division of large amounts of data.
- Giovanni Babbi:
  - Contributed to the conversion of the initial database structure from Cassandra to MongoDB. The database structure appears rudimentary and very simple, not taking full advantage of the engine's capabilities.

  - Contributed to the development of the comment section of the Flask app posts.
  - Created the services required to use Grafana and Chaos Mesh dashboards. This allows the use of high-level tools to monitor and influence the behavior of the Flask app using YAML files. Dashboard setup remains a highly manual and poorly automated process. These tools are primarily for educational purposes, as they don't fully utilize the dashboard's capabilities.
  - Developed experiments using Chaos Mesh to test the robustness of the Kubernetes cluster. They effectively allow you to evaluate the behavior of the Kubernetes cluster and manage its execution using the dedicated Chaos Mesh dashboard. The experiments developed are very simple and only exploit a fraction of what the platform has to offer.
  - Developed the autobalancer service with dedicated logic. This dedicated service, through interaction with the Mongos router, can efficiently scale up based on current memory consumption, with a high level of feedback. The implemented logic remains inefficient, and the waiting times required for the balancing process remain too high for an application scenario.
  - Contributed to manual testing of Kubernetes cluster deployment.

- An individual section is required for each member of the group
- Each member must self-evaluate their work, listing the strengths and weaknesses of the product
- Each member must describe their role within the group as objectively as possible.

It should be noted that each student is only responsible for their own section.

9 Future works

- Add fault tolerance mechanisms to the Mongos router service, as it is currently the only point of access to the centralized database and if it experiences problems during execution or terminates abruptly, the web service will stop working.
- Adding an additional entity within the project structure, a server that sits between the flask app and the mongos router, for two reasons:
  - Security: The server address is exposed externally instead of the Mongos router address for handling queries to the centralized database. It is made the server the primary access point to the Mongos router and fault tolerance mechanisms, such as pod health checks and multiple pod replications, are added internally.

  - Caching: The server would be implemented with a cache to store any incoming requests on the Mongos router. This way, if the router is unreachable, the servers can still receive queries and forward them to the router once it's back up and running.

- Add parallel reading and writing mechanisms to manage any server replica caches and any Mongos replicas.

- Implement functions related to the management of private chats between two or more users, such as the implementation of direct messaging between two users with the ability to send multimedia content and the ability to create and manage groups of multiple users.

- Add server replication scale-up mechanisms between Flask App and Mongos Router based on metrics collected by Prometheus.