European Parliament Online
Lab AP Project

Simone Scaccia - Pasquale Mocerino - Riccardo Gobbato - Chiara Maggi

September 2023

Abstract

The purpose of this report is to describe our work in the context of the project for the laboratory of advanced programming. In detail, the Introduction section illustrates the main idea of the project and the properties of our system, while the Contribution section shows the implementation, describing main components, assumptions, some crucial fragments of the code and a demo. Then, in Comments, it is carried on a performance analysis on the system, also making some relevant considerations. Finally, in Conclusions, our system is compared to existing solutions and some future improvements are proposed.

1 Introduction

1.1 Project idea

This project outlines the objectives of a system ideally commissioned by the European Union. The system is designed to facilitate inter-nation communication, enabling consensus-building and the exchange of locally collected data and public opinions. It empowers citizens from all EU member nations to actively participate in surveys, expressing their views on critical issues and proposing ideas for future referendums. Leveraging a repository of proposals, survey results, and comprehensive, up-to-date objective data covering all European nations, this system aims to provide participating countries with essential resources for the accurate formulation of new referendums, subject to the vote of both national and European citizenship. The document details the technical goals and significance of this transformative initiative.

1.2 Properties of the system

In the context of this system, referendums serve as a pivotal mechanism for citizen engagement and decision-making. Here it is a breakdown of the referendum process:

1. Submission to Citizens
   - A nation can initiate a referendum and present it to its citizens.
   - Citizens have the option to participate in the referendum by casting their votes or choosing to abstain.
2. Citizen-Proposed Referendums
   - Any individual citizen has the opportunity to publicly propose a referendum idea via a dedicated portal.
   - To move the idea forward, it must garner approvals from other citizens, reaching a threshold of 1% of the total number of citizens.
3. Nation-Level Evaluation
   - The nation's authorities assess the proposed referendum idea and decide whether to formulate it.

- A referendum can be designed to have either national or European scope, depending on its nature and relevance.

4. Approval Criteria at the National Level
    - National referendums must meet uniform approval criteria across all nations.
    - These criteria include achieving a majority vote in favor of the referendum and garnering participation from a number of voters exceeding 50% of the total number of citizens in that nation.
5. Outcome of National Referendums
    - When the result of a national referendum aligns with the conditions set forth in step 4, it is considered approved at the national level.
    - If the referendum does not meet these criteria, it is deemed null and void.
6. European Referendums
    - European referendums are distinct in that they are shared with all participating nations.
    - Each national representative assesses whether to present the referendum to their own citizens.
    - If a majority of nations opt to proceed with the referendum, it is made available for voting by all EU citizens.
7. Minimum Voting Threshold
    - Once a referendum is approved for voting, if it fails to attract a sufficient number of voting citizens from a nation, that nation abstains from voting, and its votes are not considered in the final tally.

This streamlined and standardized referendum process ensures that the voice of the citizens is central to decision-making, both at the national and European levels, while maintaining consistency and fairness in the evaluation and execution of referendums.

2 A little bit about E-voting

2.1 Introduction

Considering that voting is an integral part of our project, we have studied how we can relate it to our needs. So we took into account the electronic voting (e-voting) referred to the use of computers in order to cast votes in an election or, like in our case, in supporting a proposal. E-voting is an optimal solution to increase speed, reduce cost and improve the accuracy of the results rather than classic paper based voting. An electronic voting system creates and manages data securely and secretly, so it must meet security requirements such as confidentiality, integrity, fairness, privacy and verifiability.

2.2 E-voting technology

There are several "actors" who take part in the vote collection process:
- the *election* collects data in order to represent the set of participants who are answering to a posed question;
- the *vote* is the participant's answer to the posed question;
- the *candidates* are the predetermined set of available answers;
- an *authority* is an entity responsible for conducting the election;
- the *voting scheme* is the protocol which receives votes in input and produces an output as the sum of the votes cast for each candidate that may result in a decision.

Lastly, the *e-voting scheme* is one that makes use of electronic devices to conduct an election.

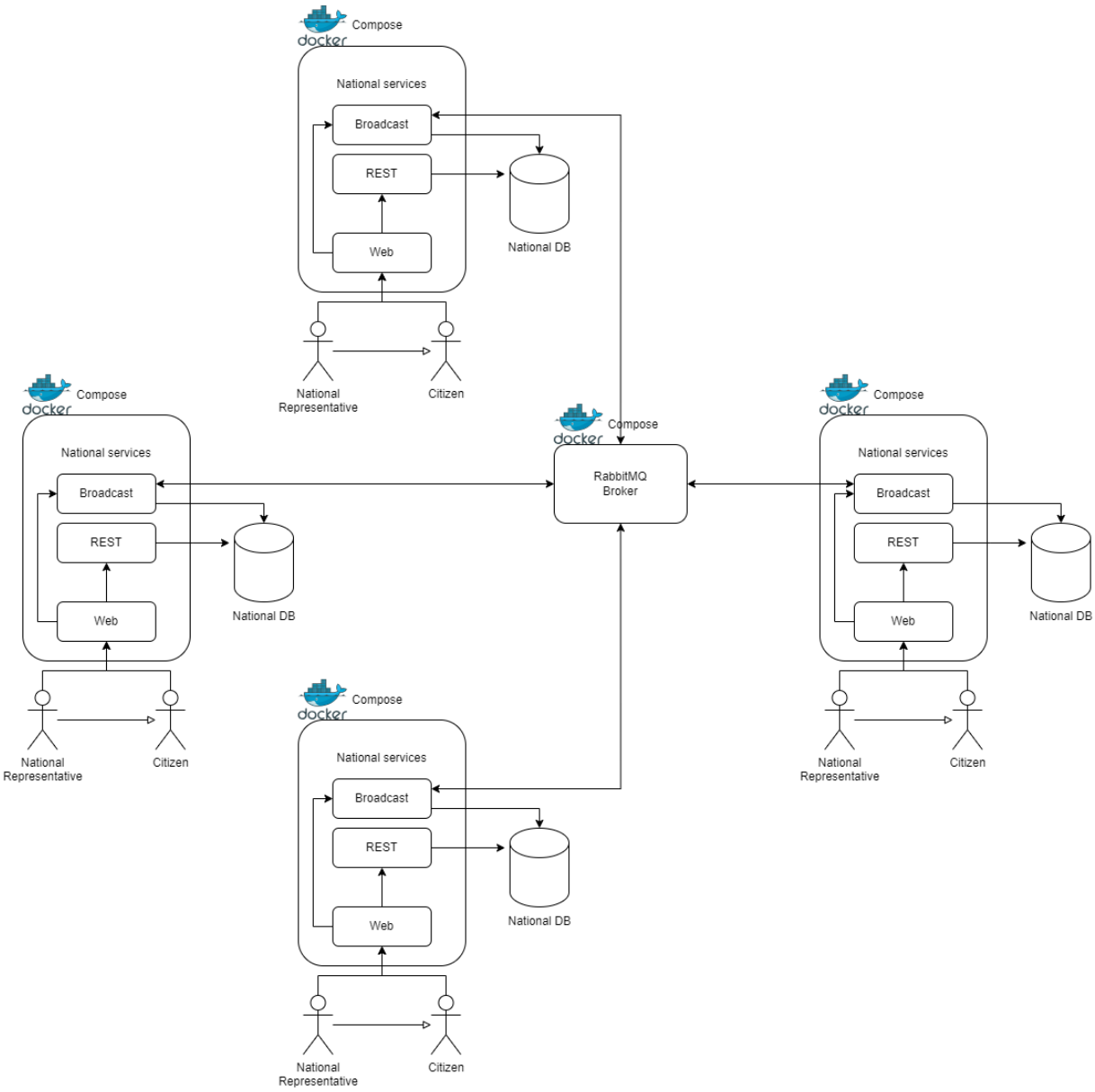
2.3 Online voting

It's a type of e-voting that involves the use of a computer and the Internet or a private network in order to support the voting process. There exist different ways to conduct online voting: *Poll station*, *Kiosk* and *Remote e-voting*. This last one (Internet voting) allows voters to cast their votes remotely from any computer or digital device connected to a public network, such as the Internet.

3 Contribution

3.1 Overview of the main components

Overview of the main components:



The system is composed of different parts, and each of them will be a different deployment in the distributed system. This architecture has been founded on Docker. The deployed components are grouped into services for specific nations and those that are available to all nations. The latter is a standalone service not affiliated with any nation. We will discuss its application and purpose. Above, there is the architecture of the system. We will explain each component in detail.

National services:

- Web: The service uses the REST API to manipulate national data and ensure its persistence through the web interface. It also uses the Broadcast API to exchange messages with other nations. In this version of the system, we will use the Broadcast and REST API to demonstrate the core functionalities since the Web service has not been fully implemented.
- Broadcast API: This servant is a programming interface that enables the simultaneous distribution of data, messages, or content to multiple recipients or channels, facilitating efficient and real-time communication across various platforms or devices.
- REST API server: This component is a web-based communication protocol that allows clients to request and manipulate resources on a server using standard HTTP methods.
- PostgreSQL databases: The database serves as a repository for national data, encompassing citizen credentials, European referendums, and associated consensus data structures.

Standalone services:

- RabbitMQ broker: This node is needed for making asynchronous communication possible between the servers in charge of dispatching the messages between the organization workers. Each nation establishes its own queue on this broker and designates a routing key. When a nation desires to send a broadcast message, it selects a routing key that matches all the nations, allowing them to receive the message in their respective queues. Once the message has been read, it will be automatically removed from the queue.

3.2 Components communication and useful links

For development purposes, all the system's deployments were made accessible within the local network, allowing developers to test each component at the localhost address. This setup eliminated the need for multiple machines and simplified the testing process.

To achieve this, various host entries were created under the same remote address, which were named in a way that reflected the functionality they exposed. This naming convention made it easier to remember and identify the addresses of different services, eliminating the need to search for their respective IP addresses with each deployment.
These components seamlessly communicated with each other through the network established by the Docker deployment, enabling efficient testing and development.

The link for the different services are the following. We have divided into the different nation that we have implemented for the simulation.
For Italy:

- http://localhost:8080/ ← web-ita
- http://localhost:8081/ ← rest-ita
- http://localhost:8082/ ← broadcast-ita
- psql-ita : not visible

For France:

- http://localhost:8084/ ← web-fra
- http://localhost:8085/ ← rest-fra
- http://localhost:8086/ ← broadcast-fra
- psql-fra : not visible

For Germany:

- http://localhost:8088/ ← web-ger
- http://localhost:8089/ ← rest-ger
- http://localhost:8090/ ← broadcast-ger
- psql-ger : not visible

Then, there is the address to manage RabbitMQ:

- http://localhost:15672/ ← RabbitMQ admin

3.3 Services and Dependencies

All the services are Spring-Boot project.
Spring Boot is an open-source framework for building Java applications, designed to simplify development with minimal configuration. It provides an embedded web server, auto-configuration, and seamless integration with the Spring ecosystem. Spring Boot applications can be run as standalone Java applications and are suitable for various use cases, including web applications and microservices. It emphasizes Java-based configuration and properties files, reducing the need for XML configuration. Spring Boot is actively maintained and has a rich ecosystem of libraries and extensions, making it a popular choice for modern Java development.

To allow the correct implementation, we have imported different dependencies for the services. In particular:

1. web service:
    - Thymeleaf: this dependency simplifies the setup of Thymeleaf, a popular templating engine, for rendering dynamic web pages in Spring Boot web applications.
    - Spring Boot Starter Web: this dependency is used for building web applications with Spring Boot.
    - Spring DevTools: this dependency is primarily used during development to improve the developer experience.
    - Spring Boot Starter Test: this dependency is used for testing Spring Boot applications.
2. rest service:
    - Spring Data JPA: this dependency simplifies the setup of a Java Persistence API (JPA) data source and provides common configurations for working with relational databases
    - Spring Boot Starter Web: this dependency is used for building web applications with Spring Boot.
    - Spring DevTools: this dependency is primarily used during development to improve the developer experience.
    - Spring Boot Starter Test: this dependency is used for testing Spring Boot applications.
    - PostgreSQL: this is the JDBC driver for the PostgreSQL database.
3. broadcast service:
    - Spring Boot Starter AMQP: this dependency provides everything you need to integrate and work with RabbitMQ.
    - Spring Boot Starter Web: this dependency is used for building web applications with Spring Boot.
    - Spring DevTools: this dependency is primarily used during development to improve the developer experience.
    - Spring Boot Starter Test: this dependency is used for testing Spring Boot applications.
    - Spring Rabbit Test: this dependency is specifically for testing RabbitMQ-related components in Spring applications.

- Google Gson: this dependency provides functionality for working with JSON (JavaScript Object Notation) data.

The plugins used for every service are:

- org.springframework.boot: used to package the Spring Boot application.
- org.apache.maven.plugins: used to configure Surefire (Maven's test plugin) to skip tests during the build with <skipTests>true</skipTests>.

3.4 Docker deployment

To allow a correct use of Docker, we have implemented a docker-compose for each nation, plus a docker-compose for RabbitMQ.

The docker compose for the Rabbit MQ Broker is structured as follow:

1. version: Specifies the version of Docker Compose used for this configuration.
2. name (epo-all): Sets the name of the Docker Compose project. This is used to group and manage related containers.
3. services: Defines the services to be run as containers (in this case, rabbitmq).
4. Configuration for the rabbitmq service:
    - container name: Assigns a name to the RabbitMQ container.
    - image (rabbitmq:3-management): Specifies the Docker image to be used for RabbitMQ, including the RabbitMQ Management Plugin for web-based management.
    - ports: Maps the container ports for RabbitMQ (5672 for AMQP and 15672 for the management interface) to the corresponding host ports.
    - networks: Specifies the networks to which this service is connected. It is connected to three networks: network-ita, network-fra, and network-ger.
5. networks: Defines three custom Docker networks: network-ita, network-fra, and network-ger. These networks are named and can be used to connect containers to specific network segments. In this case, they are used for RabbitMQ but could potentially be used for other services as well.

This Docker Compose configuration sets up a RabbitMQ container with the RabbitMQ Management Plugin enabled, allowing to manage RabbitMQ through a web interface. The container is connected to three custom networks, which we have used to facilitate communication between RabbitMQ and other containers in different networks.

Meanwhile, the docker-compose for the different nations is the following one and it has a common pattern:

1. version: as before.
2. name (epo-ita, epo-ger or epo-fra): Specifies the name of the Docker Compose project. This can be used to group and manage related containers.
3. services: Defines all the containerized services required for the application. In this case, there are four services: web, rest, psqlDB, and broadcast.
4. Configuration for the web service:
    - image: Specifies the Docker image used for this service.
    - build: Specifies the path to the Dockerfile folder to build the image.
    - container name: Assigns a name to the container.
    - ports: Maps the container port (8080) to the host port (8080, 8084 or 8088).

- depends on: Indicates that this service depends on the rest service.
- networks: Specifies the network to which the service is connected.

5. Configuration for the rest service:
   - image: Specifies the Docker image used for this service.
   - build: Specifies the path to the Dockerfile folder to build the image.
   - container name: Assigns a name to the container.
   - ports: Maps the container port (8080) to the host port (8081, 8085 or 8089).
   - environment: Sets some environment variables required for the REST service.
   - depends on: Indicates that this service depends on the psqlDB service.
   - networks: Specifies the network to which the service is connected.
6. Configuration for the psqlDB service:
   - image: Specifies the Docker image used for this service (PostgreSQL image).
   - container name: Assigns a name to the container.
   - ports: Maps the container port (5432) to the host port (5433).
   - environment: Sets some environment variables to configure the PostgreSQL database.
   - networks: Specifies the network to which the service is connected.
7. Configuration for the broadcast service:
   - image: Specifies the Docker image used for this service.
   - build: Specifies the path to the Dockerfile folder to build the image.
   - container name: Assigns a name to the container.
   - environment: Sets some environment variables for the broadcast service.
   - ports: Maps the container port (8080) to the host port (8082, 8086 or 8090).
   - depends on: Indicates that this service depends on the ‘rest‘ service.
   - networks: Specifies the network to which the service is connected.
8. networks: Defines a network named network-ita (or network-fra, or network-ger), which appears to be connected to an external RabbitMQ service. This allows containers to communicate with each other through this network.
9. volumes: used to mount a volume for PostgreSQL database data.

Ports are mapped to allow access to the services from an external host.

In the end, for each service in every nation there is a docker file to allow the creation of the jar file and the container.
For example, we explain the docker file of the service 'rest'. It is a multi-stage build for a Java application using Maven and OpenJDK 17. We used to build and package a Java application into a Docker image. There are two main parts:

- Build Stage:
  - It starts from the official Maven image.
  - It copies the application source code and the 'pom.xml' file into the '/home/app' directory inside the Docker container.
  - It then runs 'mvn clean package' to build the Java application using Maven.
- Package Stage:
  - It starts from the official OpenJDK 17 image ('openjdk:17-jdk-alpine'), which is a minimal image containing the Java Runtime Environment (JRE).

- It copies the JAR file generated in the build stage ('rest-0.0.1-SNAPSHOT.jar') from the build stage's '/home/app/target' directory to '/usr/local/lib/app-1.0.0.jar' in the current image.
- It sets the entry point for the Docker container to run the Java application using the JAR file ('app-1.0.0.jar').

This Dockerfile separates the build process from the final runtime image, resulting in a smaller runtime image that only contains the necessary JRE and the application JAR file.

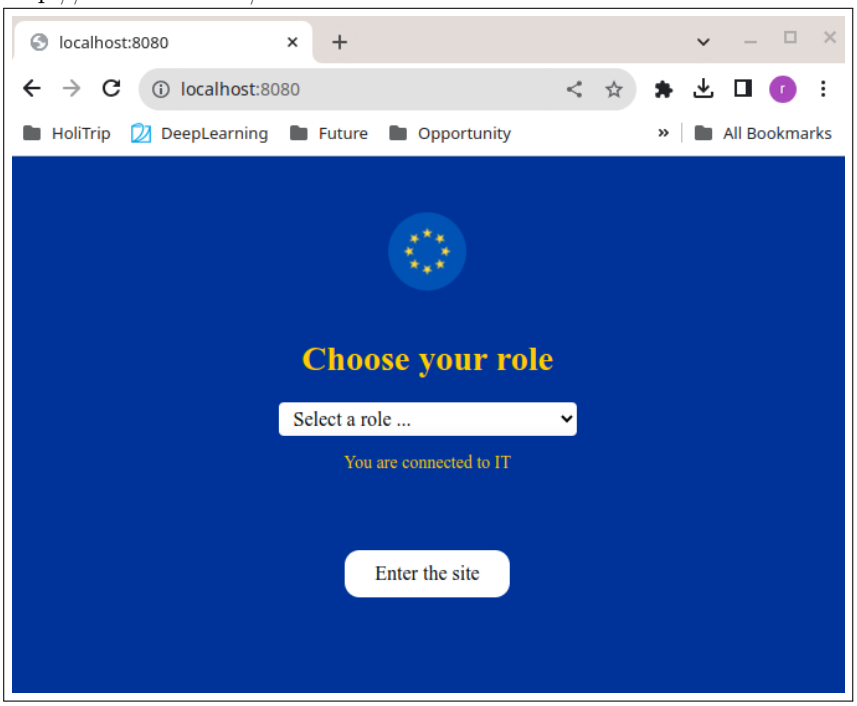
3.5 Demo and code

In this section we analyse a demo execution in order to better show the system working principle. We also show some relevant code fragments to better understand the execution. The code is fully available as a part of the larger EPO project accessible from the Github repository [2].

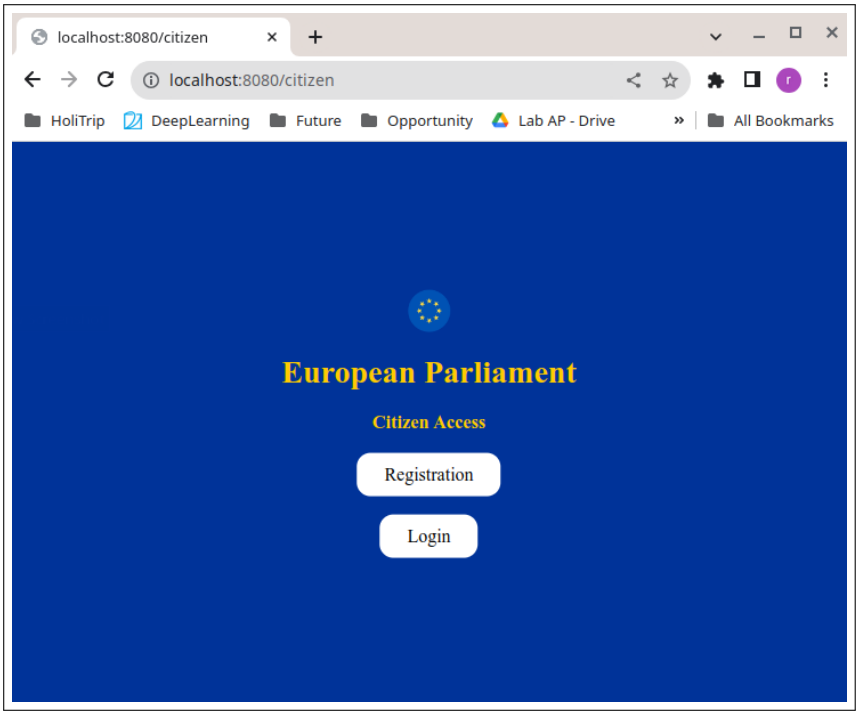
3.5.1 Frontend

The web interface is composed by several pages, based on the User Stories written in the initial implementation of the platform.

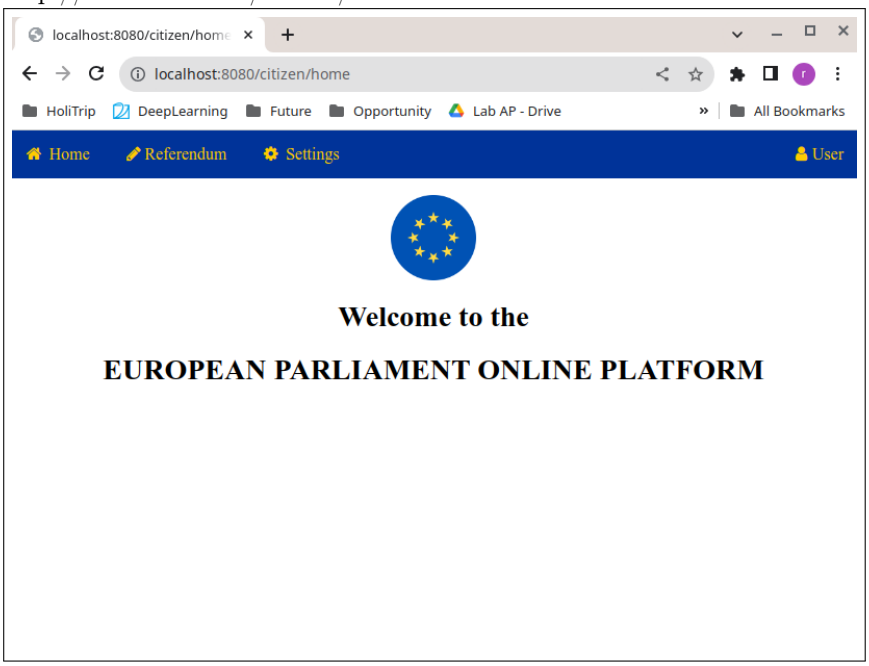
- http://localhost:8080/



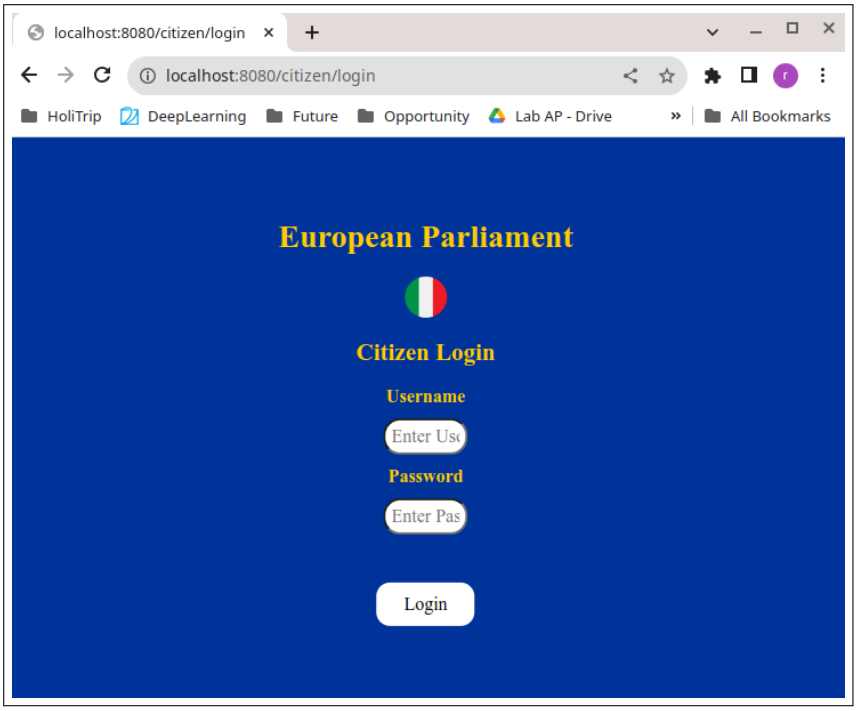
- http://localhost:8080/citizen



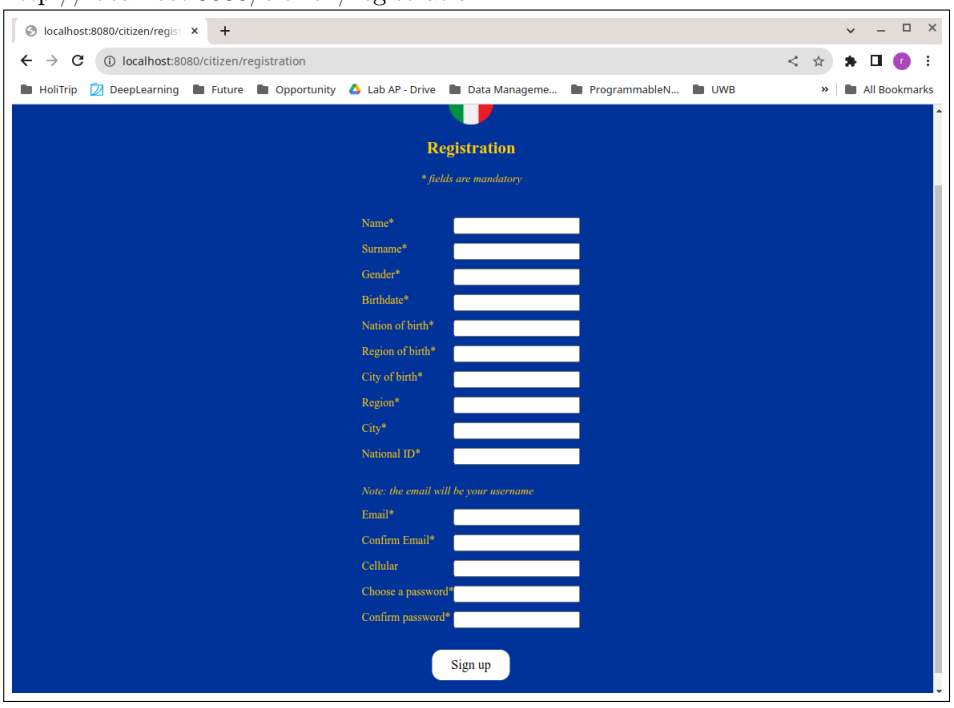
- http://localhost:8080/citizen/home



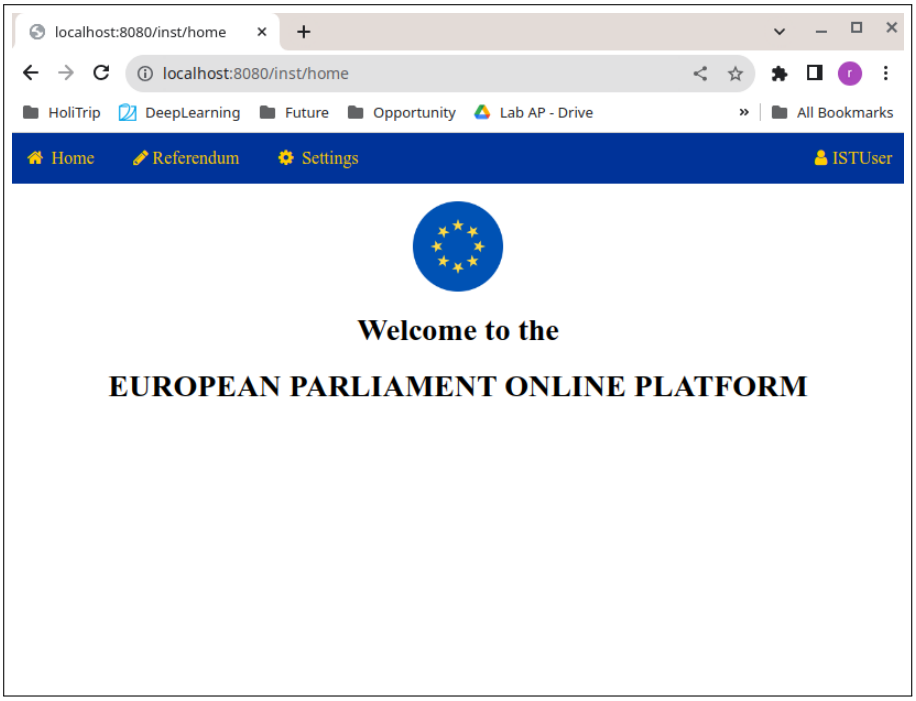
- http://localhost:8080/citizen/login



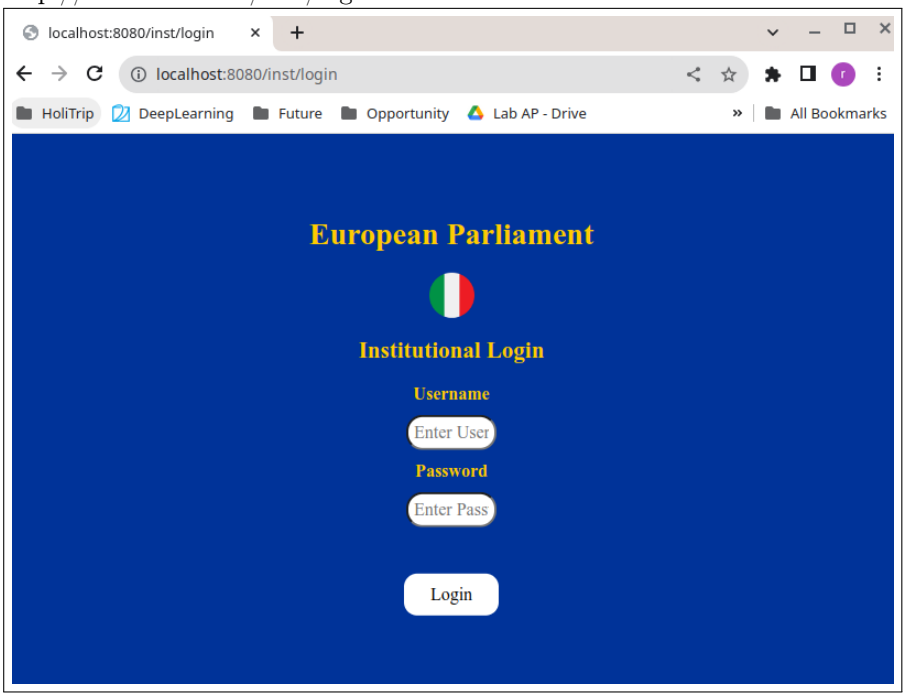
- http://localhost:8080/citizen/registration



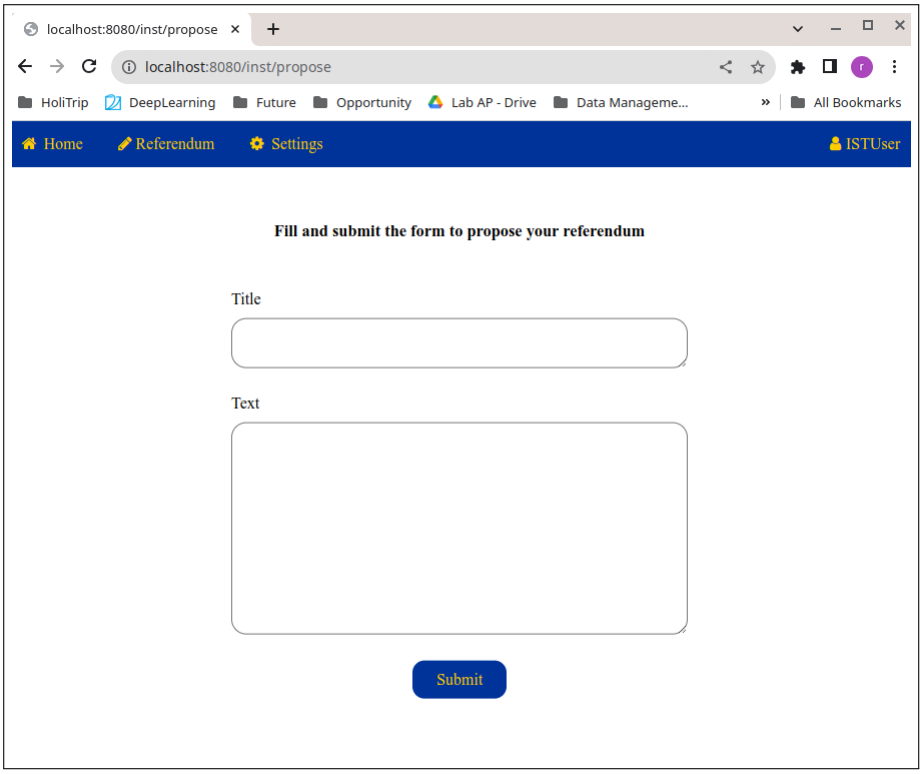
- http://localhost:8080/inst/home



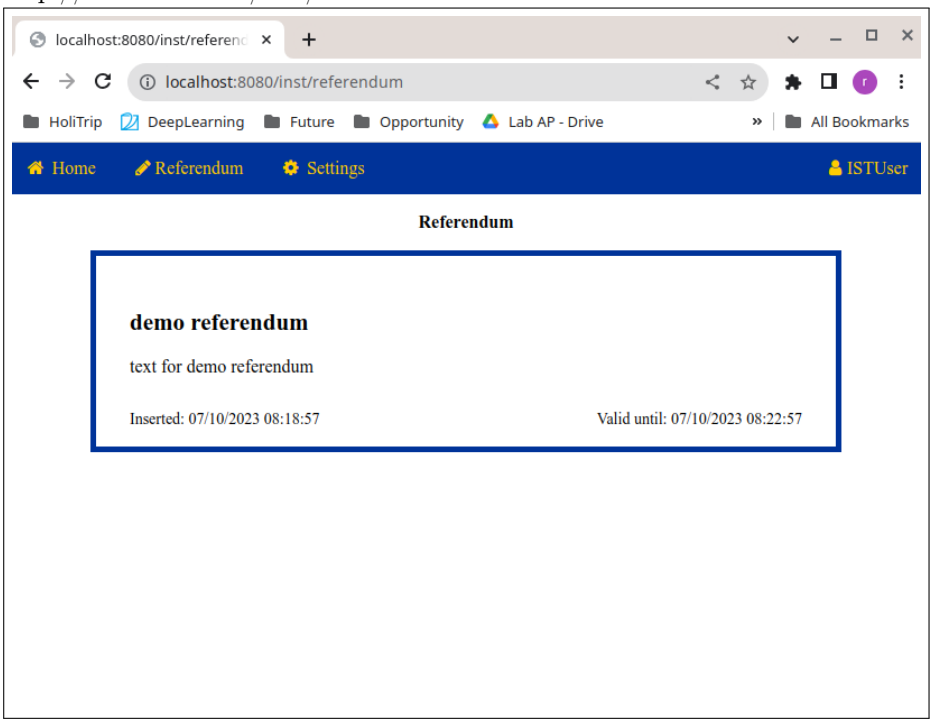
- http://localhost:8080/inst/login



- http://localhost:8080/inst/propose



- http://localhost:8080/inst/referendum



3.5.2 Backend

The main components of the distributed system are outlined through the example of a European Referendum. The consensus implementation used is the flooding consensus since we are simulating three correct nations. For simplicity, each Referendum has three possible outcomes: null if the majority of participants do not vote, true if the majority of the nation votes true, and false if the majority votes

false:

- A national representative can propose a European Referendum by submitting the title and argument through a POST API at /europeanReferendumProposal to local Broadcast services. i.e.

```
{
    "title": "This is a demo referendum",
    "argument": "Lorem ipsum dolor sit amet, consectetur adipiscing elit"
}
```

Output:

```
broadcast-ita  |
broadcast-ita  | Sending an European Referendum to all. Title: title11, Argument: argument11
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a new European Referendum by the Nation ita
broadcast-ita  | Title: title11, Argument: argument11
broadcast-ita  | Please answer to this referendum by: 13/09/2023 14:48:53
broadcast-ita  |
```

Code:

```
//for status 1 - messages for knowing the proposals
@PostMapping("/europeanReferendumProposal")
public String sendEuropeanReferendumProposal(@RequestBody Referendum referendum) {
    System.out.println("\nSending an European Referendum to all. Title: " + referendum.getId().getTitle() + ", Argument: " + referendum.getArgument() + "\n");
    referendum.setNationCreator(resourceMapping.getQueueName());
    rabbitTemplate.convertAndSend(BroadcastApplication.topicExchangeName, "foo.bar.baz", referendum.toString());
    return "Message sent: " + referendum.toString();
}
```

- Each nation will receive the Referendum proposal and automatically enter it into the first consensus, storing the Referendum and initializing the data structures in the local database. Output:

```
broadcast-fra  |
broadcast-fra  | Recived a new European Referendum by the Nation ita
broadcast-fra  | Title: title11, Argument: argument11
broadcast-fra  | Please answer to this referendum by: 13/09/2023 14:48:53
broadcast-fra  |
```

```
broadcast-ger  |
broadcast-ger  | Recived a new European Referendum by the Nation ita
broadcast-ger  | Title: title11, Argument: argument11
broadcast-ger  | Please answer to this referendum by: 13/09/2023 14:48:53
broadcast-ger  |
```

Code:

```
public void receiveMessage(String message) {

    try {
        Referendum referendum = Referendum.toReferendum(message);
        System.out.println("\nRecived a new European Referendum by the Nation " + referendum.getNationCreator());
        System.out.println("Title: " + referendum.getId().getTitle() + ", Argument: " + referendum.getArgument());
        System.out.println("Please answer to this referendum by: " + referendum.getDateEndConsensusProposal() + "\n");

        CheckTime myThread = new CheckTime(
                referendum.getId().getTitle(),
                referendum.getId().getDateStartConsensusProposal(),
                referendum.getDateEndConsensusProposal(),1,this);
        myThread.start();

        myThread = new CheckTime(
                referendum.getId().getTitle(),
                referendum.getId().getDateStartConsensusProposal(),
                referendum.getDateEndResult(),2,this);
        myThread.start();

        myThread = new CheckTime(
                referendum.getId().getTitle(),
                referendum.getId().getDateStartConsensusProposal(),
                referendum.getDateEndConsensusResult(),3,this);
        myThread.start();

        // store the referendum proposal
        HttpEntity<Referendum> referendumEntity = new HttpEntity<Referendum>(referendum);
        restTemplate.postForObject(resourceMapping.getUrlReferendum(), referendumEntity, String.class);

        // post consensus data structures
        ConsensusReferendum consensusReferendum = new ConsensusReferendum(
                referendum.getId().getTitle(),
                referendum.getId().getDateStartConsensusProposal(),
                2 // first consensus
        );
        HttpEntity<ConsensusReferendum> consensusReferendumEntity = new HttpEntity<ConsensusReferendum>(consensusReferendum);
        restTemplate.postForObject(resourceMapping.getUrlConsensusReferendum(), consensusReferendumEntity, String.class);
```

- The national representative can respond to the Referendum proposal by broadcasting the proposed message in round one. To send this message another POST API is implemented in the local Broadcast service. The URL is /europeanReferendumFirstConsensus and the body requested includes the ID of the referendum (i.e. title, creation date) and the proposal value:

```
{
    "title": "This is a demo referendum",
    "dateStartConsensusProposal": "14/09/2023 16:31:45",
    "answer": true
}
```

- When a nation receives a consensus message updates its local data structures e.g. proposals and received sets. If the local receivedFrom set contains all the correct processes, the nation can decide a value, broadcasting the decision. Output:

```
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation ger
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 14:45:53
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: ,true,
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: ,true,, ReceivedFrom: ,ger,
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation fra
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 14:45:53
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: ,true,true
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: ,true,true, ReceivedFrom: ,ger,fra
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Sending the proposal answer: true to all, for the following Referendum:
broadcast-ita  | Title: title11, Creation date: 13/09/2023 14:45:53
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation ita
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 14:45:53
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: true,true,true
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: true,true,true, ReceivedFrom: ita,ger,fra
broadcast-ita  |
broadcast-ita  | Decision taken: true
broadcast-ita  |
broadcast-ita  | Sending the proposal decision: true to all.
broadcast-ita  |
```

Code:

```
} catch (Exception e) {

    ReferendumMessage referendumMessage = ReferendumMessage.toReferendumMessage(message);
    System.out.println("\nRecived a consensus message by the Nation " + referendumMessage.getNationSourceAnswer());
    System.out.println("Title: " + referendumMessage.getTitle() + ", Creation Date: " + referendumMessage.getDateStartConsensusProposal());
    System.out.println(referendumMessage.printMessage());

    // get consensus data structure from the database
    ConsensusReferendum consensusReferendum;
    Referendum referendum;
    try {
        referendum = HttpRequest.getReferendum(
                referendumMessage.getTitle(),
                referendumMessage.getDateStartConsensusProposal(),
                resourceMapping
        );
    } catch (NotFoundException nFE) {
        System.out.println("Error: referendum not already received\n");
        return;
    }
    try {
        consensusReferendum = HttpRequest.getConsensusReferendum(
                referendumMessage.getTitle(),
                referendumMessage.getDateStartConsensusProposal(),
                this.resourceMapping);
    } catch (NotFoundException nFE) {
        System.out.println("Discarding message, decision is already taken\n");
        return;
    }

    // check if the status of the message is coherent with the status of the referendum, to avoid obsolete messages
    if(consensusReferendum.getStatus() != referendumMessage.getStatus()) {
        System.out.println("Discarding message: recived message status is different from the current status for the referendum: " +
                referendumMessage.getTitle() + ", " +
                referendumMessage.getDateStartConsensusProposal() + "\n"
                );
        return;
    }
```

```
        // check the type of the message
        if(referendumMessage.getIsDecision() &&
           consensusReferendum.isCorrect(referendumMessage.getNationSourceAnswer())
        ) {
            this.computeDecision(referendumMessage.getAnswer(), referendumMessage.getStatus(), referendum);
            return;
        }
        // compute the proposal

        // update consensusReferendum values
        consensusReferendum.updateProposals(
                referendumMessage.getProposals(),
                referendumMessage.getRound()
        );
        consensusReferendum.updateRecivedFrom(
                referendumMessage.getNationSourceAnswer(),
                referendumMessage.getRound()
        );

        // put new values in the database
        HttpRequest.putConsensusReferendum(consensusReferendum, resourceMapping);
        System.out.println("Local consensus data structurues. Correct: " + consensusReferendum.getCorrect() + ", Round: " + consensusReferendum.getR

        // check decision condition. We already know that the condition is not already taken
        if(consensusReferendum.checkCorrectSubsetOfReceiveFrom()) {
            if(consensusReferendum.checkReceivedFromNotChanged()
            ) {
                // take decision
                Boolean decision = consensusReferendum.decide();
                System.out.println("Decision taken: " + decision + "\n");

                this.computeDecision(decision, referendumMessage.getStatus(), referendum);

            } else {
                consensusReferendum.incrementRound();
                // update ConsensusReferendum
            }
        }

    }

    latch.countDown();

}
```

- Assuming the first consensus on the Referendum proposal is true, we can proceed with presenting the Referendum to each nation's citizens individually. For the sake of simplicity, we will not include a voting component in this demonstration and assume that all nations will answer "yes" to the Referendum. This implies that the majority of citizens have also answered "yes" to the Referendum. If a nation does not respond to the Referendum by the end date, the system implements a timeout using a thread to make a decision. The agreement regarding uniforms is only met if the broadcast communication is of the URB type.

- When the vote ends, the date is defined during the creation of the Referendum, a second consensus starts and all the nations automatically send its result.

- Finally after the flooding of the messages a decision to the Referendum is taken. Output:

```
broadcast-ita  | CheckTime timeout, situation : 1 (end consensus proposals), status: 3. Already decided
broadcast-ita  |
broadcast-ita  | CheckTime timeout, situation : 2 (end citizen votes), send local referendum result
broadcast-ita  |
broadcast-ita  | Sending the voting answer: true to all, for the following Referendum:
broadcast-ita  | Title: title11, Creation date: 13/09/2023 15:50:38
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation fra
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 15:50:38
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: ,,true
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: true,,true, ReceivedFrom: ,,fra
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation ger
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 15:50:38
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: ,true,
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: true,true,true, ReceivedFrom: ,ger,fra
broadcast-ita  |
broadcast-ita  |
broadcast-ita  | Recived a consensus message by the Nation ita
broadcast-ita  | Title: title11, Creation Date: 13/09/2023 15:50:38
broadcast-ita  | Type : PROPOSAL, Round :1, Proposals: true,,
broadcast-ita  |
broadcast-ita  | Local consensus data structurues. Correct: ita,ger,fra, Round: 1, Proposals: true,true,true, ReceivedFrom: ita,ger,fra
broadcast-ita  |
broadcast-ita  | Decision taken: true
broadcast-ita  |
broadcast-ita  | Sending the proposal decision: true to all.
```

4 How to deploy the system for the first time

The following code has been tested for linux and Windows 11. Actually, it should be valid also for MAC-OS.

4.1 Installation and launching

The only requirement is Docker. All the data and programs will be installed downloading and launching the Github code:

1. Clone the repository executing the following CLI command:

```
git clone https://github.com/RicGobs/EPO-European-Parliament-Online.git
```

2. Now, you have to launch the program. If you have Linux you can deploy it with:

```
chmod u+x command_start.sh
./command_start.sh
```

   If you are using Windows or IOS, launch in four different terminal the following commands:

```
sudo docker compose -f docker-compose-all.yaml up --build

# Wait until rabbitmq container ends the initialization

sudo docker compose -f docker-compose-ita.yaml up --build
sudo docker compose -f docker-compose-fra.yaml up --build
sudo docker compose -f docker-compose-ger.yaml up --build
```

4.2 Possible errors

It is possible to have some errors. We will give you some useful commands to solve the easiest issues:

Sometimes, the port neeeded for the program are already used; so use the following commands:

```
# to check which ports are in use
sudo lsof -i -P -n | grep LISTEN

# to kill the process running in port 5672
sudo fuser -k 5672/tcp

# to kill the process running in port 5432
sudo fuser -k 5432/tcp
```

Sometimes, docker gives some problems with old containers; so, to eliminate old containers that can create conflicts:

```
sudo docker container rm /rabbitmq
sudo docker container rm /sender
sudo docker container rm /receiver
sudo docker container rm /web
sudo docker container rm /rest
```

In the end, to clean your system from all docker containers and volumes:

```
sudo docker system prune --all --force
```

to check if the system works and in particular the POST:

```
#Post request from terminal:
curl -X POST localhost:8082/europeanReferendumBroadcast -H "Content-type:application/
    json" -d "{\"title\": \"Samwise Gamgee\",\"status\": \"1\",\"argument\": \"yoo\",\"
    firstNation\":\"yoyoyo\"}"


curl -X POST localhost:8086/europeanReferendumBroadcast -H "Content-type:application/
    json" -d "{\"title\": \"Samwise Gamgee\",\"status\": \"1\",\"argument\": \"yoo\",\"
    firstNation\":\"yoyoyo\"}"

```

```
curl -X POST localhost:8090/europeanReferendumBroadcast -H "Content-type:application/
    json" -d "{\"title\": \"Samwise Gamgee\",\"status\": \"1\",\"argument\": \"yoo\",\"
    firstNation\":\"yoyoyo\"}"
```

An alternative, it is to use the POSTMAN program (you have to download it).

5 Challenges of a distributed system

The described implementation is simplified in some design choices for practical reasons. However, when scaling the system up in a real-world scenario, we must keep into account the impact of these simplifications. Here we list some relevant challenges of a real EU system:

- RabbitMQ Broker. For the sake of simplicity, in our implementation we used a single broker shared by all nations, however in reality multiple RabbitMQ brokers are needed, precisely one for each nation. This implies the need to correctly handle the network communication between these brokers, with a greater latency.
- Temporal and spatial concurrency. Every nation accesses its own database, so there are disjointed instances of data with different schemes. Data dissemination implies difficulties in the access of data at different locations and times.

  Other data problems are related to the law field, but must be kept into account in the implementation of a real system because they have an impact on the system implementation. One is related to the heterogeneity of data, so the need to legally define a common scheme of data for a better aggregation. Another context-related problem is the possibility of restrictions due to government policies which impose not to share some sensitive data about the nation.
- Failures. Since some nodes may crash, we need to cope with failures in advance. By using some mechanisms of failure detection and by adapting the algorithms to the presence of faulty processes, it is possible to keep the system fault tolerant and to preserve its safety.
- Unpredictable latencies. The communication over long distances implies the possibility of slight delays if compared to a local system. However, by fixing some end times to the execution of the consensus, we can solve the problem. These parameters must be properly finetuned to guarantee our goal.
- Byzantine tolerance. In the consideration of a real scenario, we must consider the possibility of Byzantine processes. Given the partial synchrony of the system, some adjustments are possible by using cryptography and by making some assumptions on the number of possible Byzantine processes.

5.1 Assumptions for the distributed environment

We have made different assumptions about the system, resulting in various problems and ways of thinking:

- Processes: processes in the context of this example represent individual entities representing different nations, such as "ita" for Italy, "fra" for France, and "ger" for Germany. The parameter N is utilized to indicate the potential for any number of processes, and scalability is discussed in the conclusion.
- Timing Assumptions: In the face of inherent asynchrony, our approach incorporates a strategic use of timeouts. This transformation introduces partial synchrony, setting bounds on response times, thereby alleviating consensus challenges within the system.
- Clock Synchronization: Our system utilizes External Synchronization through default frameworks like Spring Boot, relying on UTC (Coordinated Universal Time). This approach ensures that stringent time constraints are not a critical factor in our system's operation.

- Failure Detector: Implementing a failure detector by taking advantage of REST API is straightforward and can significantly enhance consensus mechanisms. This approach capitalizes on the inherent synchrony of REST API communication. However, we have not yet implemented this approach for simplicity. Therefore, we assume that all processes (nations) are correct. Nevertheless, we acknowledge the possibility of failures in our discussion.

- Link Abstractions, Broadcast Communication, and Consensus: To ensure proper system functioning, the communication medium between servants of different countries is crucial. In this case, we utilize a RabbitMQ broker that provides a publish and subscribe mechanism. We will delve into the guarantees it offers and how we can effectively utilize them to achieve our goals.

  - Link Abstractions: As stated in this guide [5], we can achieve reliable message delivery even in the face of various failures. It can be concluded that RabbitMQ offers a perfect point-to-point link due to the following features it provides:
    * PL1: Reliable delivery: In case a correct process p sends a message m to a correct process q, q will eventually receive m.
    * PL2: No duplication: No process delivers the same message more than once.
    * PL3: No creation: If process q receives a message m from sender p, then m was previously sent to q by process p.
  - Broadcast Communication: The guarantees of broadcast communication depend on the broker's design choices in our distributed system. Our concept aims to establish a decentralized network of countries without a central governing entity. This implies that there is no central intermediary responsible for facilitating message exchanges. Instead, each nation is expected to create its own queue to receive messages from other countries. This design offers a fully decentralized system, which is advantageous. However, one of its disadvantages is that each nation must maintain a pool of connections with all other nations' brokers. Assuming 100KB per TCP connection [3], and 27 European countries [6], the memory usage is not massive but could become problematic if the pool grows too large. The most significant disadvantage of this type of broadcast communication is that it is best-effort communication, as the sender can crash while sending a message to the connection pool, so the agreement property can't be reached. RB4: Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process. To enrich the properties of Broadcast Communication, we can reason as follows: the idea is to relax the distribution of the brokers and design one single active broker at a time. If a process, regardless of whether it is correct or faulty, sends a message to the broker, the message will be delivered to every correct process through the reliable delivery of the broker. This fulfills the uniform agreement requirement of the Uniform Reliable Broadcast.: URB4 Uniform agreement: If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process. The advancements in broadcast communication technology will impact the consensus algorithm as well. We should address the issues with this design. One challenge is figuring out where to put the broker because we don't have a separate entity from the participating nations. We might have to place the broker in one of the nations, but this could create security issues since all the traffic would go through that nation and it would be a single point of failure. To address this, we could create a list of backup brokers located in different nations that activate if the primary broker fails. Additionally, we can detect broker crashes by using a failure detector that relies on a synchronous API provided by the broker.
  - Consensus: In order to determine the outcome of a European Referendum based on the collective will of all nations, a consensus primitive must be used. Depending on the method of communication, we can employ various consensus primitives. If we choose the BEB design, we cannot use flooding consensus because a crash after a process's decision may lead to disagreement between processes. Therefore, we need a uniform consensus primitive like the flooding uniform consensus. Our application requires all nations, faulty or not, to reach the same decision for the Referendum. However, using the flooding uniform consensus results in a disadvantage of using n^3 messages instead of n^2 messages without crashes. By utilizing the URB design, we can effectively overcome the issue caused by flooding consensus.

The uniform agreement property guarantees that any message sent by a faulty process will also be delivered by all the correct processes. As a result, the flooding consensus will be concluded in a single round, ensuring uniform agreement on the values decided. This method employs n^2 even in the event of crashes.

5.2 Performance evaluation

IT capacity planning consists in estimating the storage, hardware, software and connection infrastructure resources required over some future period of time to correctly support service provisioning.

We have set the system parameters before. About the resource parameters, we have focused on the latency of RabbitMQ. The latency of the read/write operations (in database) is irrelevant compared to RabbitMQ's latency. Now it is interesting to show our perfomance model about the response time, the throughput and the utilization:

- Response Time: we focused on max response time of the system (so what the user see in the worst case). We have the time to send message, the time to conclude a consensus and the time to conclude both consensus. We have put a timeout for every actions. We want to give enough time to citizens to vote, in fact a voting nation takes more time than a consensus, so we will give a time x for voting and a time t for each consensus; so we will have a response time of $x + 2 \cdot t$ of response time. The sending of the message is irrelevant.
  In our simulation with the prototype, we have used a timeout of 2 minutes both for x and t.
- Throughput: there are different possible situations, using a number n of nations:
  * flooding consensus in the Best Case with No failures: one communication round, so $2 \cdot n^2$ messages
  * flooding consensus in the Worst case with $n - 1$ failures: n^2 messages exchanged for each communication step and at most n rounds, so n^3 messages
  * flooding uniform consensus: n^2 messages exchanged for each communication step and at most n rounds, so n^3 messages

  About the weight of the ReferendumMessage Object , we have 4 string, 2 boolean and 2 integer as attributes, so:
  * String title (max 50 characters): 50*2 Byte
  * Integer status: 4 Byte
  * String nationSourceAnswer: 3*2 Byte
  * Boolean answer: 2 Byte
  * String proposals: 4n + (n-1) Byte
  * Integer round: 4 Byte
  * Boolean isDecision: 2 Byte
  * String dateStartConsensusProposal (dd/MM/yyyy HH:mm:ss): 19*2 Byte
  * Overhead added by rabbit: 736 Byte ([4])

  So, for a total of 892 Byte + 4n + (n-1) Byte.
- The Utilization, in particular the use of the database. We are using the following size: integer is 4 byte, string is 2 byte per char, and boolean is 2 byte. We have two type of objects:
  * ConsensusReferendum, with 2 integer, 5 string and 1 boolean as attributes:
    · Integer status: 2 Byte
    · ConsensusReferendumId id: String title (max 50 characters): 50*2 Byte, and String dateStartConsensusProposal (dd/MM/yyyy HH:mm:ss): 19*2 Byte
    · String correct: 3n + (n-1) Byte
    · Boolean decision: 2 Byte

- Integer round: 4 Byte
- String proposals: 4n + (n-1) Byte
- String receivedFrom: 3n + (n-1) Byte

So, for a total of 146 Byte + 10n + 3(n-1) Byte.

* Referendum, with 4 integer and 7 string as attributes:
  - ReferendumId id: String title (max 50 characters): 50*2 Byte, and String dateStartConsensusProposal (dd/MM/yyyy HH:mm:ss): 19*2 Byte
  - Integer status: 4 Byte
  - Integer votesTrue: 4 Byte
  - Integer votesFalse: 4 Byte
  - Integer population: 4 Byte
  - String argument: max 1 MegaByte
  - String nationCreator: 3*2 Byte
  - String dateEndConsensusProposal: 19*2 Byte
  - String dateEndResult: 19*2 Byte
  - String dateEndConsensusResult: 19*2 Byte

  So, for a total of 274 Byte + 1 Megabyte.

The total utilization is 420 Byte + 1 Megabyte + 10n + 3(n-1) Byte. So the text of the Referendum is the bottleneck (also knowing that, for now, n max is 27).

After the implementation, we have analysed that we can improve two different things, to reduce the use of resources: put proposals as integer, and use the standard of 2 characters for identify the nations.

6 Conclusions

The main purpose of this project is to experiment the consensus in a distributed voting system across nations. Despite the implementation of a distributed algorithm, which overcomes the limitation of the legacy centralized voting environment and make the system more secure by design, security is not the main focus of our system. But this is a crucial aspect in the real world.

In fact, the vast majority of recent e-voting systems are based on blockchain, which guarantees the confidentiality (absence of unauthorized disclosure of information), the integrity (absence of unauthorized system alterations) and the validity (respected logical constraints) of the voting process, as well as availability for authorized actions. In detail, a common used method to is the Proof-of-Voting (PoV), preferred over the common PoW for its lower cost and lower power consumption, by keeping security satisfied. An example of these systems is [1], where hierarchical voting roles are defined, each one associated to a smart contract. The method used to reach consensus is PoV, useful also to identify Byzantine processes in order to remove them from the blockchain. The main idea is to assure that only a specific voter could have cast his own vote. Other recent systems seem to be all different implementations of the same blockchain-based solution idea. So, it seems natural to think that a future improvement of our system can be to integrate some security measures in order to avoid attacks to the system functioning principle.

Other future improvements, already known from the previous analysis, are the implementation of a failure detector, with the removal of no failure assumption, and the use of multiple brokers rather than a single one, which can constitute a single point of failure.

References

[1] Ketulkumar Govindbhai Chaudhari. "E-voting system using proof of voting (PoV) consensus algorithm using block chain technology". In: *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering* 7.11 (2018). Accessed: 14/09/2023, pp. 4051–4055.

[2] *Project Github Repository*. Accessed: 14/09/2023. URL: https://github.com/RicGobs/EPO-European-Parliament-Online.

[3] *RabbitMQ Reasoning About Memory Use*. https://www.rabbitmq.com/memory-use.html#breakdown-connections. Accessed: 14/09/2023.

[4] *RabbitMQ Reasoning About Memory Use*. https://www.rabbitmq.com/memory-use.html#message-memory-usage. Accessed: 14/09/2023.

[5] *RabbitMQ Reliability Guide*. https://www.rabbitmq.com/reliability.html. Accessed: 14/09/2023.

[6] *Wikipedia European Union*. https://en.wikipedia.org/wiki/European_Union#Member_states. Accessed: 14/09/2023.