



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare



Deliverable 6.7
“Prototypical implementation of phase 3 and evaluation results”

Work Package 6
“Blockchains and user right management”

Disclaimer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078. Any dissemination of results reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Copyright message

© FeatureCloud Consortium, 2023

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Document information

Grant Agreement Number: 826078			Acronym: FeatureCloud	
Full title	Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare			
Topic	Toolkit for assessing and reducing cyber risks in hospitals and care centres to protect privacy/data/infrastructures			
Funding scheme	RIA - Research and Innovation action			
Start Date	1 January 2019	Duration	60 months	
Project URL	https://featurecloud.eu/			
EU Project Officer	Christos Maramis, Health and Digital Executive Agency (HaDEA)			
Project Coordinator	Jan Baumbach, University of Hamburg (UHAM)			
Deliverable	D6.7 – Prototypical implementation of phase 3 and evaluation results			
Work Package	WP6 – Blockchains and user right management			
Date of Delivery	Contractual	31/12/2023	Actual	19/01/2024
Nature	Report	Dissemination Level	Public	
Lead Beneficiary	SBA			
Responsible Author(s)	Walid Fdhila (SBA) and Rudolf Mayer (SBA)			
Keywords	Blockchains, user rights, consent management, patient consent, GDPR compliance, non-expert users, global discovery, metadata			

History of changes

Version	Date	Contributions	Contributors (name and institution)
V0.1	30/11/2023	First draft	Walid Fdhila (SBA)
V0.2	21/12/2023	Comments	Sandor Fejer (Egnosis)
V0.3	22/12/2023	Addressing comments	Walid Fdhila (SBA)
V0.3	01/12/2023	Comments	Rudolf Mayer (SBA)
V0.3	02/01/2023	Addressing comments and internal review	Walid Fdhila (SBA)
V1.0	19/01/2024	Final edits and approval for submission	Nina Donner (concentris) Jan Baumbach (UHAM)

Table of Content

1 Table of acronyms and definitions	5
2 Objectives of the deliverable based on the Description of Action (DoA)	6
3 Executive Summary	6
4 Introduction (Challenge)	6
5 Methodology	7
6 Results	7
6.1 Overall FeatureCloud Architecture	7
6.2 Certificate Authorities	8
6.2.1 Identity Registration	9
6.2.2 Identity Enrollment	9
6.2.3 Illustration of a FeatureCloud Participant Configuration	9
6.2.4 Identity Registration and Enrollment Process	10
6.3 Network Deployment	13
6.4 Smart Contract Deployment	16
6.4.1 Consent Contract	16
6.4.2 MLStudy Contract	21
6.4.3 Chaincode Deployment	23
6.5 Integration and Client Application of the FeatureCloud Blockchain Network	24
6.5.1 Architecture Overview	24
6.5.2 RESTful Integration with the FeatureCloud platform	26
6.6 Command Line Interface (CLI)	28
6.7 Web Application	32
6.7.1 Web Authentication	32
6.7.2 Home view and consent management Features	36
6.8 FeatureCloud Blockchain operator	46
6.9 Technology Stack used for FeatureCloud blockchain	47
7 Deviations	48
8 Conclusion	48
9 References	48
10 Other supporting figures	49

1 Table of acronyms and definitions

ABAC	Attribute-Based Access Control
AC	Access Control
API	Application Programming Interface
CA	Certificate Authority
CID	Consent Identifier
CLI	Command Line Interface
concentris	concentris research management gmbh
D	Deliverable
gRPC	gRPC Remote Procedure Call
IBAC	Identity-Based Access Control
JSON	JavaScript Object Notation
JWT	JSON Web Token
LPM	Local Project Manager
ML	Machine Learning
MSP	Membership Service Provider
MS	Milestone
NodeOU	Node Organizational Unit
Patients	In this deliverable, we use the term “patients” for all research subjects. In FeatureCloud, we will focus on patients, as this is already the most vulnerable case scenario and this is where most primary data is available to us. Admittedly, some research subjects participate in clinical trials but not as patients but as healthy individuals, usually on a voluntary basis and are therefore not dependent on the physicians who care for them. Thus, to increase readability, we simply refer to them as “patients”.
PID	Patient Identifier
PKI	Public Key Infrastructure
RBAC	Role-Based Access Control
REST	Representational State Transfer
SBA	SBA Research Gemeinnützige GmbH
SDK	Software Development Kit
SMPC	Secure Multi-Party Computation
TLS	Transport Layer Security
UHAM	University of Hamburg
UI	User Interface
YAML	YAML Ain't Markup Language
WP	Work package

2 Objectives of the deliverable based on the Description of Action (DoA)

The objective of this deliverable is based on the description of action, which incorporates aspects from all Objectives 1 to 4 and thus are part of the corresponding tasks 1 to 4 in work package 6.

“Based on this academic research SBA will construct a prototype that can be verified and tested with actual information. Required changes will thereby be fed back into the construction phase.”
“During the prototype phase, it will also be considered which level of detail will be appropriate for non-expert users in order to be able to use the FeatureCloud platform (MUG, TUM).”

3 Executive Summary

Deliverable D6.7 is the culmination of Work Package 6's comprehensive efforts, incorporating insights from related tasks and feedback from other Work Packages. D6.7 specifically builds and iterates over the prototypical implementation described in **D6.4** with additional features and an accessible user interface. The final prototype reflects an iterative process that adheres to a comprehensive methodology including the design, development, and deployment of a permissioned blockchain solution using Hyperledger Fabric [1], and encompassing requirement analysis and threat modeling. The proposed blockchain solution enables patients to manage their consents, and participants in federated machine learning of healthcare data to secure their audit processes. A multi-organization test network has been established, featuring certificate authorities, peer nodes, and orderer nodes. While peer nodes maintain ledgers, execute smart contracts, and manage transaction proposals and endorsements, orderer nodes are responsible for ordering transactions, ensuring consensus among peers. The deployment includes multiple smart contracts (chain codes) facilitating the effective management of consents and machine learning studies, made conveniently accessible through REST services. To enhance user accessibility, especially for patients less familiar with technology, a web application has been integrated with security measures in mind. The prototype was also integrated with the FeatureCloud platform.

4 Introduction (Challenge)

Training machine learning models in healthcare poses significant challenges, particularly when data crucial for training is dispersed among several health care providers such as hospitals, each potentially located in different jurisdictions. The conventional approach of transferring and centrally processing data poses legal and privacy concerns. Federated learning, unlike traditional statistical and machine learning methods, allows local execution of ML algorithms directly at the data holders' sites, avoiding the need for centralizing sensitive healthcare data. This approach not only enhances confidentiality and addresses privacy concerns by avoiding the centralization of sensitive data, but also aligns with legal and technical constraints such as GDPR regulations (e.g., Art. 25 GDPR; Data protection by design and by default) [2]. However, this federated process introduces its own set of challenges, including the need for ensuring the integrity of the learning process and preventing malicious behavior. Because the data stays local, outsiders can't easily check if the learning process was performed correctly. This opens the door to problems like hospitals using unauthorized data or faking information. In this context, the focus shifts to mechanisms that can verify the accuracy, integrity and compliance of results without compromising individual data privacy.

In this context, the integration of blockchain technology becomes crucial, providing a potential solution to enhance the security and reliability of federated machine learning processes. Blockchain, with its attributes like traceability, integrity and immutability, can serve as an immutable audit trail. Its decentralized and chronological timestamping capabilities act as a strong defense

against later attempts to tamper with data, significantly improving the overall security of federated machine learning of healthcare data.

5 Methodology

The development of the prototype for securing the audit process and managing consents follows an iterative and incremental methodology. It starts with a comprehensive analysis of requirements, identifying potential threats in the existing system design and implementing mitigations. The initial prototype undergoes continuous refinement to accommodate design, architectural, and integration changes. This iterative approach spans the design, development, and deployment phases, focusing on a permissioned blockchain solution using Hyperledger Fabric. To enhance accessibility, smart contract functions are made conveniently accessible through a web API for patient consent management, and REST services for seamless integration with the FeatureCloud platform.

6 Results

This section outlines the main results of the prototype implementation, building upon the findings and prototype in D6.4, and emphasizing key components. It represents an incremental evolution, refining existing features and introducing novel components essential for enhancing user accessibility. Subsequent sections explore these additions, providing a comprehensive understanding of the latest iteration of the prototype.

For the sake of making the deliverable self-contained, relevant figures or text parts from prior deliverables are incorporated or reused when beneficial.

6.1 Overall FeatureCloud Architecture

In the FeatureCloud approach, participant data and consents are securely stored on-site where they are collected (cf. Figure 1). Throughout federated machine learning (ML) studies, the ML algorithm operates only locally, sharing solely the output model (federated learning, FL). This ensures that confidential patient data and consents remain within each site's storage. Each site employs a FeatureCloud controller overseeing local execution, whereas the coordinator controller orchestrates the global workflow. The FeatureCloud system integrates a global API for project details and an AI store for making FeatureCloud apps available. A trust framework, consisting of accredited entities, sets up governance rules and facilitates the onboarding of new members.

In the following, our attention turns to the FeatureCloud blockchain prototype, where we will discuss its key components in detail.

FeatureCloud participants can choose to either operate their own blockchain nodes or obtain accreditation to leverage the infrastructure. Smart contracts are deployed on peer nodes, handling the management of both patient consents and ML studies. Access to smart contract functions is facilitated through REST services hosted on each participant, either directly or via a web API. A user-friendly interface caters to novice users, enabling them to easily manage their consents. Additionally, patients have the option to handle their own cryptographic keys (e.g., X.509 certificates) and directly interact with the blockchain, bypassing the need to go through a participant in the FL, such as a hospital.

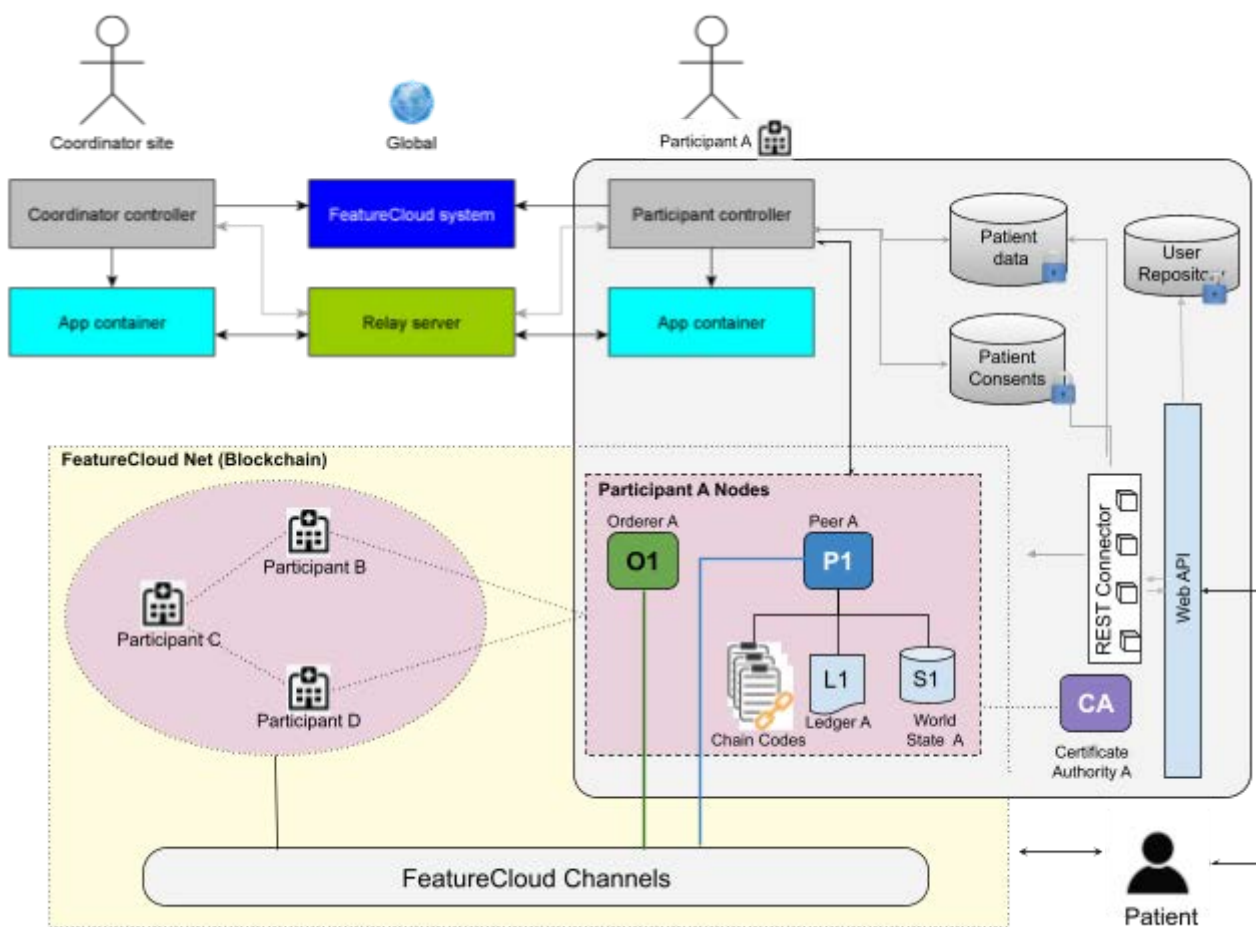


Figure 1. Overall featureCloud architecture

6.2 Certificate Authorities

In a Fabric network, establishing unique and secure identities is fundamental for ensuring secure communication channels, and the integrity and controlled access of participants within the network. These identities play a crucial role in defining roles, permissions, and interactions across the network. In **FeatureCloud**, we have chosen to employ certificate authorities instead of the default keying material generation library, "cryptogen". This approach ensures that each organization within FeatureCloud possesses its own dedicated Certificate Authority (CA) responsible for generating identities for users within that organization. Specifically, we utilize the built-in **Fabric CAs** as root CA providers, thereby facilitating the creation of Membership Service Providers (MSP) structures. MSP are configuration folders that define organizational membership and roles, specifying accepted Certificate Authorities (CAs) and assigning privileges like admin or peer. It plays a crucial role in validating identities, listing members, and determining their roles and permissions within the network.

We have deployed two separate CAs for each FeatureCloud participant: i) an **enrollment CA** responsible for generating identities for organization administrators, MSPs, peer or orderer nodes, and users, and a ii) **TLS CA** which issues identities to nodes for securing communications through Transport Layer Security (TLS). The TLS CA is deployed first, and its root certificate is utilized during the bootstrapping of the enrollment CA. This TLS certificate is exclusively utilized for issuing certificates for nodes. Users typically register and enroll with the enrollment CA, while nodes undergo registration and enrollment with both the enrollment and TLS CAs.

While FeatureCloud currently utilizes X.509 certificates as identities, it is important to highlight that the system is designed to be adaptable, allowing for the potential integration of alternative identity forms such as decentralized identifiers and verifiable credentials, as detailed in deliverable **D6.4**. Furthermore, as outlined in D6.4, there is an option to leverage trusted Certificate Authorities (CAs) external to an organization or participant, rather than relying solely on locally deployed CAs. It's worth noting that patient identities generated by local CAs differ from global identities assigned by external CAs, such as those held by health ministries or insurance companies.

Once a CA admin is enrolled, the generation of client and node identities within the FeatureCloud blockchain network involves two key steps: Registration and Enrollment.

6.2.1 Identity Registration

In the registration process, vital information about the entity, whether it's a user or a node, is submitted to the Certificate Authority administrators. The administrators verify the information, assign roles, affiliations, and attributes, and issue a unique enrollment ID and secret, establishing a distinctive identifier for the entity within the network.

6.2.2 Identity Enrollment

Enrollment is the process where certificates are created and handed to the user of the identity. After successful registration, entities, whether users or nodes, use their own Fabric CA clients with the provided enrollment ID and secret to trigger the Certificate Authority. The process yields a public/private key pair encoded with the roles and attributes assigned by the CA admin. The enrolled identity can now actively participate, sign transactions, and fulfill its designated role, maintaining the security integrity of the network. Importantly, private keys remain secure, accessible only to the user of the identity, maintaining a robust security posture within the network.

A key advantage of **client-initiated enrollment** is the increased confidentiality surrounding private keys. Traditionally, administrators are actively involved in the enrollment process, potentially exposing them to sensitive cryptographic material. With client-initiated enrollment, the entity manages the process autonomously, ensuring that administrators remain unaware of the private keys associated with the enrolled identity. This confidentiality adds an extra layer of security to the overall network architecture.

As discussed in previous deliverables (D6.2, D6.3, D6.4), in FeatureCloud, there are two fundamental designs catering to users with varying technical expertise. Users with limited tech experience can opt for client-initiated enrollment, allowing them to manage their cryptographic keys. Conversely, novice users, like those relying on hospitals for key management, can leverage administrator-initiated enrollment. In the latter case, the associated keys remain on the participant site, and an accessible user interface with a classical login/password mechanism enables patients to access provided services, such as smart contracts for managing consents.

6.2.3 Illustration of a FeatureCloud Participant Configuration

Figure 2 provides a sample configuration file of a FeatureCloud participant. This file defines a set of identities and their associated attributes necessary for the registration and enrollment processes. This configuration includes specified **roles** such as patient, auditor, and doctor, along with corresponding Node Organizational Units (NodeOUs). **NodeOUs**, integral to Membership Service Providers (MSPs), play a crucial role in categorizing nodes into Fabric specific roles, i.e **clients**, **admins**, **peers**, and **orderers**. This classification enforces an organizational structure, allowing each node to fulfill its distinct responsibilities within the FeatureCloud blockchain network.

Within this organizational framework, roles offer a more detailed breakdown within the NodeOU structure, enabling a client node, for instance, to assume roles like patient or auditor.

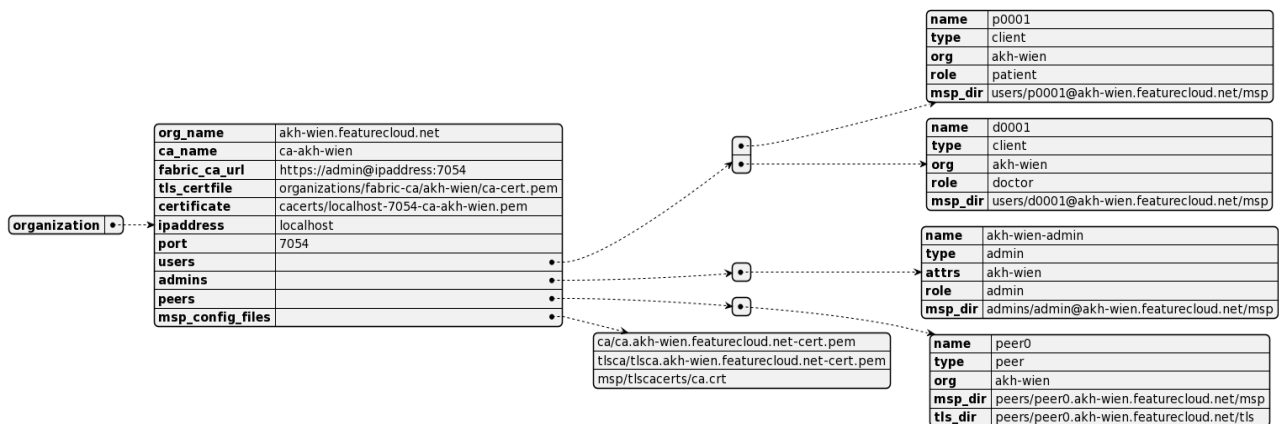


Figure 2. Sample configuration file of a FeatureCloud participant

6.2.4 Identity Registration and Enrollment Process

In FeatureCloud, participants initiate the identity creation process by enrolling the Certificate Authority (CA) admin (cf. Figure 3). Subsequently, the generation of Membership Service Provider (MSP) configuration takes place, outlining the roles and responsibilities of nodes within the organization. Following this, entities, representing users/clients, org admins, or peer and orderer nodes, undergo registration, leading to the creation of their dedicated MSPs. This includes the generation of Peer MSPs, which are customized to accommodate the specific attributes and roles assigned to each entity.

As mentioned earlier, following the registration (not depicted in Figure 3), the enrollment process in FeatureCloud can be carried out either by the admin or by the entities themselves, depending on the specific use case. Note that peer and orderer nodes also need to be registered with the TLS CA of their respective organizations for securing the communication within the network. Network participants can choose to include only peer nodes, orderer nodes, or a combination of both.

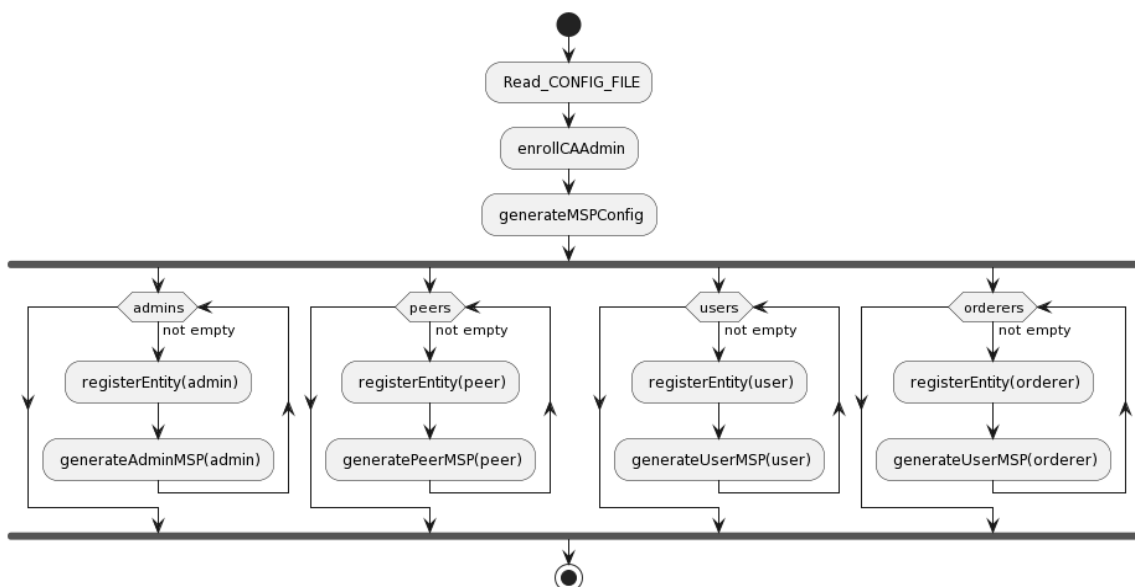


Figure 3. Identities Registration process with a CA

The script segments presented in Figures 4-6 showcase the process of creating organizational identities. This involves the registration of new entities, including peers, admins, and users, as well as the generation of their corresponding Membership Service Providers (MSPs). The function is designed to receive a JSON configuration file, encapsulating essential configuration details (cf. Figure 4).

```
function createOrg() {
    CONFIG_FILE="$1" # Must be a json config file
    ...
    CONFIG=$(jq '.' $CONFIG_FILE)
    # Extract values from the parsed JSON
    org_name=$(echo $CONFIG | jq -r '.org_name')
    org_base_dir=$(echo $CONFIG | jq -r '.org_base_dir')
    fabric_ca_url=$(echo $CONFIG | jq -r '.fabric_ca_url')
    ...
    infoln "Creating organization: $org_name"
    mkdir -p "${PWD}/${org_base_dir}"
    enrollCAAdmin
    generateMSPConfig
    copyCertificates

    entities=($peers $users $admins)
    for entity in "${entities[@]}; do
        registerEntity "$entity"
        generatePeerMSP "$entity"
    done
}
```

Figure 4. Organization Identities creation script snippet

```
function registerEntity() {
    local user_info="$1"
    local user_name=$(echo $user_info | jq -r '.name')
    local user_password=$(echo $user_info | jq -r '.password')
    local user_type=$(echo $user_info | jq -r '.type')
    local user_attrs=$(echo $user_info | jq -r '.attrs')

    infoln "Registering $user_name with org= $user_attrs :ecert"
    export FABRIC_CA_CLIENT_HOME=${PWD}/${org_base_dir}
    fabric-ca-client register --caname $ca_name --id.name $user_name --id.secret $user_password
    --id.attrs "org=$user_attrs:ecert" --id.type $user_type --tls.certfiles ${PWD}/${tls_certfile}
    { set +x; } 2>/dev/null
}
```

Figure 5. Entity registration script snippet

```
# Function to generate MSP and certificates for peers
function generatePeerMSP() {
    local peer_info="$1" # Pass user data as an argument
    local peername=$(echo $peer_info | jq -r '.name')
    ...

    infoln "Generating the $peername MSP"
    fabric-ca-client enroll -u https://$peername:$peer_password@$ipaddress:$port --caname $ca_name -M
        "${PWD}/${org_base_dir}/${peer_msp_dir}" --tls.certfiles ${PWD}/${tls_certfile}

    cp "${PWD}/${org_base_dir}/msp/config.yaml" "${PWD}/${org_base_dir}/${peer_msp_dir}/config.yaml"

    if [ -n "$peer_tls_dir" ]; then
        infoln "Generating the $peername-tls certificates, use --csr.hosts to specify Subject Alternative Names"
        fabric-ca-client enroll -u https://$peername:$peer_password@$ipaddress:$port --caname $ca_name -M
            "${PWD}/${org_base_dir}/${peer_tls_dir}" --enrollment.profile tls --csr.hosts $peername.$org_name
            --csr.hosts $ipaddress --tls.certfiles ${PWD}/${tls_certfile}
    ...
}
}
```

Figure 6. MSP generation script snippet

Figures 7-8 present the client implementation dedicated to enrolling organizational identities. Figure 7 provides part of the Java code responsible for executing the identity enrollment process. The subsequent figure complements this by revealing a snippet of the script used to invoke the client application. The enrollment process is possible via command-line invocation of the JAR file, but a REST service implementation is also available.

```
public void enrollIdentity(final Wallet wallet, final String credentialPathString, final String
    identityLabel) {
    try {
        final Path credentialPath = Paths.get(credentialPathString).toAbsolutePath();
        final Path certificatePath = credentialPath.resolve(certificateFilename).toAbsolutePath();
        final Path privateKeyPath = credentialPath.resolve(privateKeyFilename).toAbsolutePath();
        final X509Certificate certificate = readX509Certificate(certificatePath);
        final PrivateKey privateKey = readPrivateKey(privateKeyPath);
        final Identity identity = Identities.newX509Identity(mspId, certificate, privateKey);
        if (wallet.get(identityLabel) != null) {
            logger.log(Level.SEVERE, "Wallet already contains identity with label " + identityLabel);
            throw new IllegalStateException("Wallet already contains identity with label " +
                identityLabel);
        }
        wallet.put(identityLabel, identity);
    } catch (IOException e) {
        logger.log(Level.SEVERE, "Error reading credential files: " + e.getMessage(), e);
        throw new IllegalArgumentException("Error reading credential files: " + e.getMessage(), e);
    } catch (CertificateException e) {
        logger.log(Level.SEVERE, "Error parsing certificate file: " + e.getMessage(), e);
        throw new IllegalArgumentException("Error parsing certificate file: " + e.getMessage(), e);
    } catch (InvalidKeyException e) {
        logger.log(Level.SEVERE, "Error parsing private key file: " + e.getMessage(), e);
        throw new IllegalArgumentException("Error parsing private key file: " + e.getMessage(), e);
    }
}
```

Figure 7. Client implementation snippet of the enrollment process

```
# Extract the file name from the file path
KEYSTORE_FILE_NAME=$(basename "$KEYSTORE_FILE")
# Enroll the user using the dynamically obtained keystore file
java -jar enrollUser.jar -w "./wallet" \
-c "../fc-network/organizations/peerOrganizations/
uni-wien.featurecloud.net/users/User1@uni-wien.featurecloud.net/msp" \
-m "UniWienMSP" \
-cert "signcerts/cert.pem" \
-key "keystore/$KEYSTORE_FILE_NAME" \
-id "User1@uni-wien.featurecloud.net"
```

Figure 8. Identity enrollment script snippet ¹

The provided screenshots in Figure 9 illustrate a sample Membership Service Provider (MSP) folder structure for a participant. In this scenario, the participant assumes responsibility for overseeing all associated private keys on behalf of users.

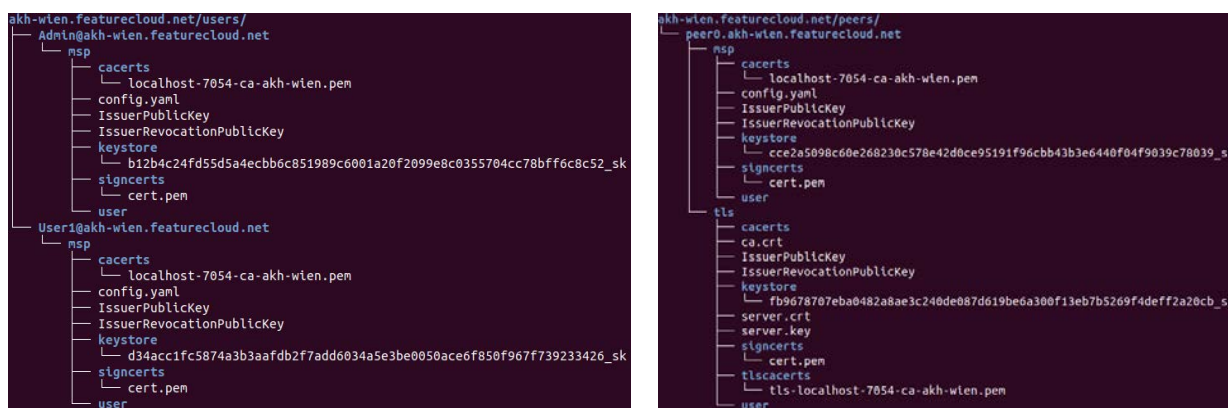


Figure 9. Sample MSP structure

6.3 Network Deployment

Once cryptographic identities are established, participants in the network start the process of configuring and deploying their peer and orderer nodes. The distinction between these two types of nodes lies in their roles and functionalities. Peer nodes engage in maintaining the ledger, endorsing transactions, and committing blocks to the blockchain, while orderer nodes focus on managing the ordering service, i.e., defining the consensus algorithm, and organizing the sequence of transactions.

This deployment process involves detailed configuration steps for each node type. Participants configure network and database settings (e.g., peer address, external endpoints, tls, couchDB), logging configurations, and paths to cryptographic materials. Additionally, each participant designates an anchor peer, a pivotal element facilitating cross-organizational communication and endorsing transactions.

For orderer nodes, further configuration is necessary. This includes defining the consensus algorithm to employ, configuring batch sizes, and establishing policies for transaction ordering. In the network channel configuration, participants finalize details such as policies, access control

¹ **Disclaimer:** The organization names used in this deliverable (e.g.m, uni-wien) are illustrative examples and do not represent real participants in the FeatureCloud Blockchain network. They are included for demonstration purposes.

rules, and endorsing peer specifications. A channel in Hyperledger Fabric is an isolated ledger exclusive to a set of organizations, inaccessible to entities outside that channel. A Fabric network can support multiple distinct channels. A well-structured network channel is imperative for regulating interactions among diverse participants within the network. In FeatureCloud, the Raft consensus algorithm is used for ordering service nodes, following a “leader and follower model, in which a leader is dynamically elected among the orderer nodes in a channel [3].

```

volumes:
  peer0.uni-wien.featurecloud.net:
networks:
  test:
    name: featurecloud_net
services:
  peer0.uni-wien.featurecloud.net:
    container_name: peer0.uni-wien.featurecloud.net
    image: hyperledger/fabric-peer:latest
    labels:
      service: hyperledger-fabric
    environment:
      - FABRIC_CFG_PATH=/etc/hyperledger/peerconfig
      - FABRIC_LOGGING_SPEC=INFO
      - CORE_PEER_TLS_ENABLED=true
      - CORE_PEER_PROFILE_ENABLED=true
      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      - CORE_PEER_ID=peer0.uni-wien.featurecloud.net
      - CORE_PEER_ADDRESS=peer0.uni-wien.featurecloud.net:11051
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/fabric/msp
      - CORE_PEER_LISTENADDRESS=131.129.1.7:11051
      - CORE_PEER_CHAINCODEADDRESS=peer0.uni-wien.featurecloud.net:11052
      - CORE_PEER_CHAINCODELISTENADDRESS=131.129.1.7:11052
      - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.uni-wien.featurecloud.net:11051
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.uni-wien.featurecloud.net:11051
      - CORE_PEER_LOCALMSPID=UniWienMSP
      - CORE_METRICS_PROVIDER=prometheus
      - CHAINCODE_AS_A_SERVICE_BUILDER_CONFIG={"peername":"peer0uni-wien"}
      - CORE_CHAINCODE_EXECUTETIMEOUT=300s
    volumes:
      - ../organizations/peerOrganizations/uni-wien.featurecloud.net/peer0.uni-wien.featurecloud.net:/etc/hyperledger/fabric
      - peer0.uni-wien.featurecloud.net:/var/hyperledger/production
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
  ports:
    - 11051:11051
  networks:
    - test

```

Figure 10. Sample docker file for a peer node

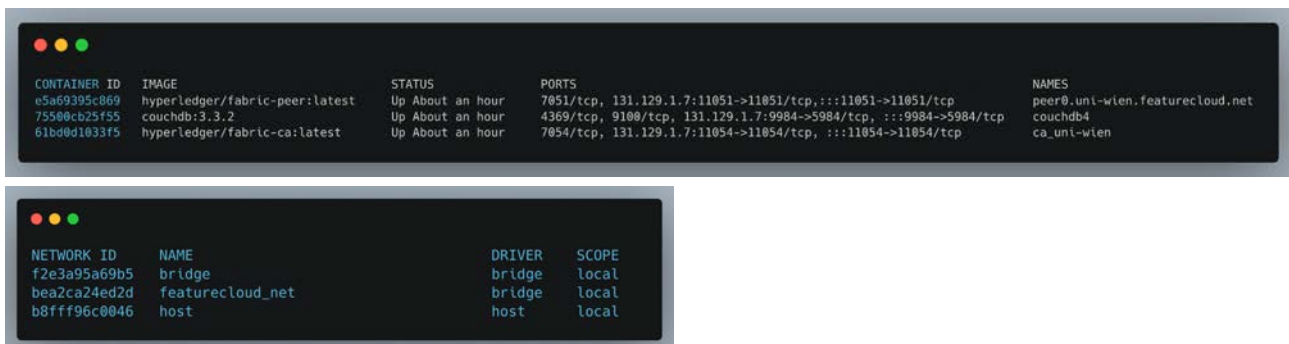
The deployment process extends to defining and enforcing endorsement policies for chaincode. This step ensures that the required number of peers endorses transactions, thereby validating their legitimacy. These configuration and policy-setting steps are very important and collectively contribute to the network's overall security and reliability. Below, we outline the essential steps for initiating the blockchain network, assuming the prior generation of organizational definitions, configuration files, and Membership Service Providers (MSPs).

Blockchain Network Initialization Process Overview:

- 1) In the initial step, **the channel administrator is designated**, involving the selection of at least one peer organization, to exchange private transactions across multiple organizations.

This requires the creation of peer organization MSP definitions, with at least one specification in the configtx.yaml file.

- 2) Following the administrator designation, the **orderers** and **peer nodes** are **independently started** by their respective organizations. Notably, this initialization occurs without channels. Communication between nodes remains inactive until a channel is created in the subsequent steps. A visual representation of this process is shown in Figure 10, extracted from the Dockerfile of one participant “uni-wien.featurecloud”.
- 3) The next phase involves the **generation of the genesis block** for the new channel. This task is performed by the designated administrator peer using the configtxgen tool provided by Hyperledger Fabric and the configuration file “configtx.yaml”.
- 4) With the genesis block created, the **activation of the channel** is effectively carried out by the first orderer node that receives the channel join request. Full operational status is not achieved until a quorum of consenters (orderer nodes participating in the consensus mechanism on a channel) is established, allowing additional nodes to join from either the genesis block or the latest config block.
- 5) The final stage revolves around **peers joining the channel**. This includes the process of joining peers, configuring anchor peers, and, if necessary, submitting a channel configuration update for the inclusion of a new peer organization. For those initially included, peers can fetch the channel genesis block using the peer channel fetch command. Subsequently, they join the channel, facilitating the construction of the blockchain ledger from the ordering service.



CONTAINER ID	IMAGE	STATUS	PORTS	NAMES
e5a69395c869	hyperledger/fabric-peer:latest	Up About an hour	7051/tcp, 131.129.1.7:11051->11051/tcp, :::11051->11051/tcp	peer0.uni-wien.featurecloud.net
75500cb25f55	couchdb:3.3.2	Up About an hour	4369/tcp, 9100/tcp, 131.129.1.7:9984->5984/tcp, :::9984->5984/tcp	couchdb4
61bd0d1033f5	hyperledger/fabric-ca:latest	Up About an hour	7054/tcp, 131.129.1.7:11054->11054/tcp, :::11054->11054/tcp	ca.uni-wien

NETWORK ID	NAME	DRIVER	SCOPE
f2e3a95a69b5	bridge	bridge	local
bea2ca24ed2d	featurecloud_net	bridge	local
b8fff96c0046	host	host	local

Figure 11. Deployed nodes

This process will result in the deployment of nodes from each organization, rendering the network fully operational. Illustrated in Figure 11 are the nodes belonging to the participant "uni-wien" within the featureCloud blockchain network. These figures primarily showcase the containers for its corresponding Certificate Authority (CA), peer, and database. In this example, "uni-wien" is also integrated into the featurecloud_net network. When deploying the network across multiple hosts, such as through Kubernetes or Docker Swarm, meticulous attention must be paid to the precise addresses, ports, and name resolution. At this stage, the network is fully operational, with all nodes successfully joined to the FeatureCloud channel (cf. Figure 12). However, it's important to note that no chain code has been deployed as of yet. The next crucial step (see Section 6.4) involves the deployment of chain code to enable smart contract execution and interaction among the interconnected nodes.

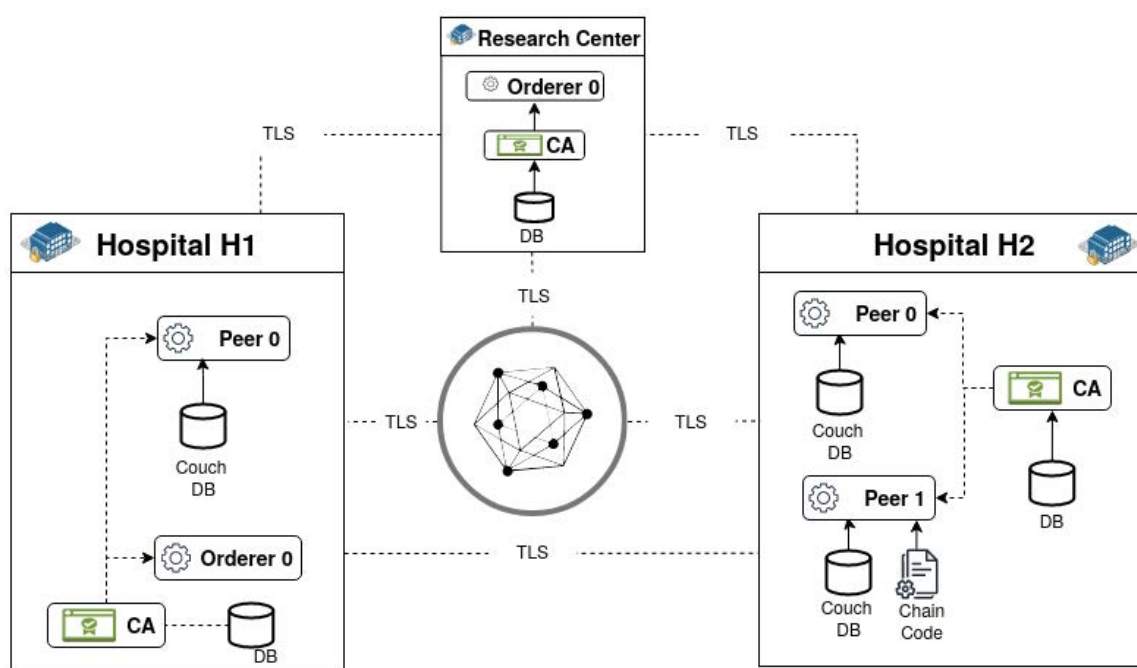


Figure 12. The FeatureCloud blockchain network

6.4 Smart Contract Deployment

In this following section, we offer a brief overview of FeatureCloud's main smart contracts and outline enhancements made to their design, building upon insights from prior deliverables, particularly D6.4. For a comprehensive understanding of the core functionalities, please refer to earlier deliverables (D6.2, D6.3, D6.4). The focus in this deliverable is on clarifying and presenting the improvements made to the design.

6.4.1 Consent Contract

A consent in FeatureCloud serves as an explicit approval from a patient for having their healthcare data used in designated studies. Patients can issue consents through digital means or paper signatures, and in certain cases, trusted third parties can digitally sign consents on behalf of patients. While the actual consents are stored locally at each participant's site for efficiency, they are also committed to the blockchain to establish a transparent and traceable record.

The consent contract, functioning as a smart contract, coordinates operations related to patient consents for healthcare data usage, encompassing actions like consent issuance, update, and revocation. This orchestration enables transparent tracing and auditing of these operations, ensuring a secure and transparent consent management system.

The recent **enhancements** to the consent contract implementation introduces a notable improvement to access controls for managing consents and user identities, moving beyond the previous implementations of `isOwner` and `hasRole`. The refined access control mechanisms now incorporate a more comprehensive set of controls and permissions. Specifically, the smart contract introduces explicit methods for granting and revoking permissions, allowing for a more granular and flexible management of access rights. The **refined implementation** of the smart contract aligns closely with the **design** principles outlined in Section 5 of deliverable D6.4. The introduced access controls play a pivotal role in improving the security of the system by preventing unauthorized access and manipulation of patient consents. Furthermore, the enhanced access control mechanisms provide the flexibility to explicitly grant specific users (such as a unique doctor

identifier) or roles (for instance, all administrators of a specific hospital) the authority to perform actions on behalf of the patient.

The class diagram of Figure 13 illustrates the key components of the consent management process. The **ConsentContract** class serves as the central entity, providing essential functionalities for managing consents and facilitating the registration of identities.

ConsentContract Class: The **ConsentContract** embodies the core features necessary for consent management. Beyond its primary role in managing consents, it also accommodates the registration of identities, necessary for the integration of new participants (e.g. also client apps or users) into the FeatureCloud Blockchain ecosystem. The class acts as a central hub orchestrating the secure and efficient management of consent-related operations, and responsible for facilitating interactions between the client applications and the blockchain . It includes all functions of type **transaction**.

PermissionManager Class: The **PermissionManager** class oversees access permissions to the ConsentContract. It manages user, patient, and consent permissions, employing distinct data structures for each category. With a focus on security, the PermissionManager combines Role-Based Access Control (RBAC), Identity-Based Access Control (IBAC), and Attribute-Based Access Control (ABAC). This multi-layered approach ensures a fine-grained and flexible system for managing permissions within the ConsentContract.

Key Functionalities of the permission management components of our smart contract include:

1. **Granular Access Control:** the chaincode implements attribute-based access controls in specific functions, allowing for the verification of critical user attributes, such as the organizational affiliation (org) extracted from the x509 certificate. This attribute-based approach enhances the precision of access permissions.
2. **Permission Granting and Revocation:** PermissionManager orchestrates the granting and revocation of permissions, offering a dynamic and adaptable system. This feature ensures that access privileges can be flexibly adjusted based on evolving requirements or changing organizational structures, e.g. granting auditors access to specific patient consent history.
3. **Role-Based Access Control (RBAC):** RBAC is a fundamental aspect of the access control model implemented by the **PermissionManager**. In RBAC, users are assigned roles, and permissions are associated with these roles. The key technical components include:
 - a. **Role Assignment:** Users are assigned specific roles based on their organizational responsibilities or functional roles within FeatureCloud. Such roles are encoded within the identities issued to users (e.g., x509 certificates). Roles are predefined and associated with certain permissions related to consent management.
 - b. **Role Verification:** The class includes mechanisms for verifying user roles, a fundamental aspect of access control. This capability ensures that only users with designated roles can execute specific operations, reinforcing the principle of least privilege. Before executing certain operations, the PermissionManager verifies that the calling user has the requisite role to perform the action. This involves checking the user's identity against the roles assigned to determine if the user has the necessary privileges.

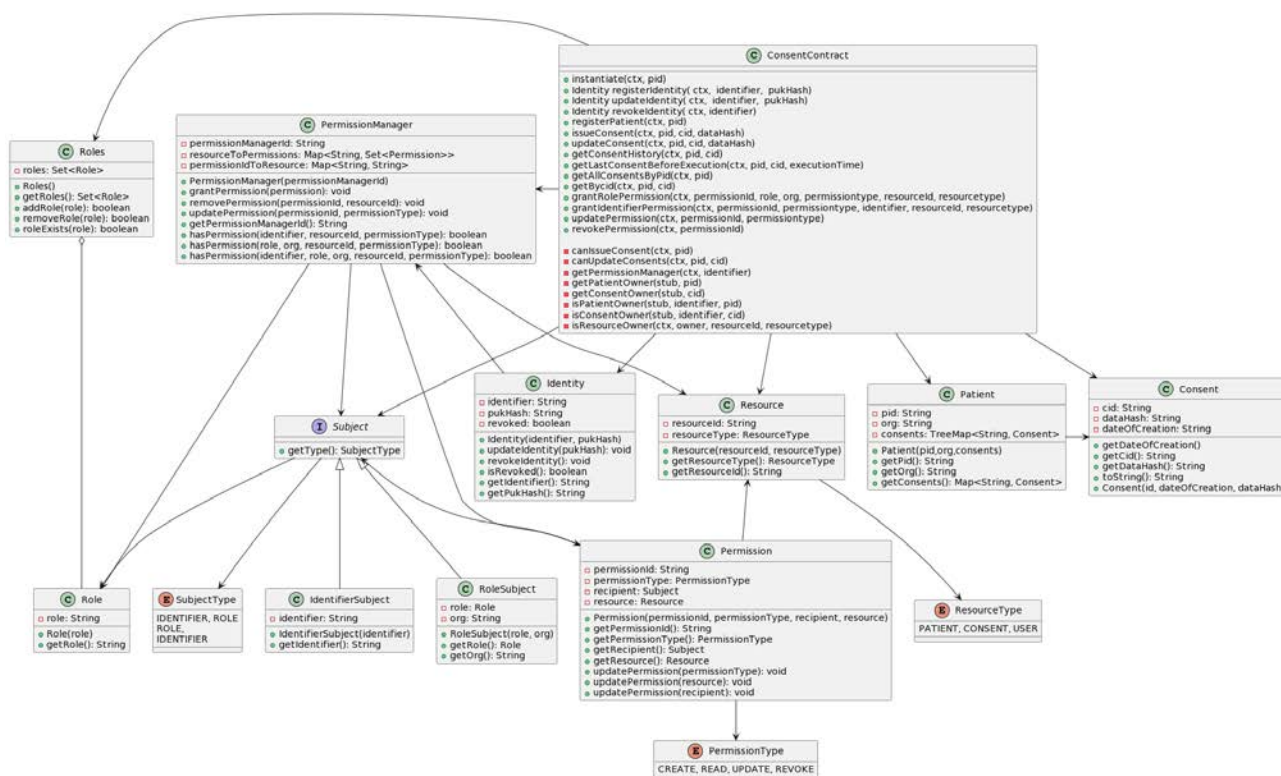


Figure 13. Class diagram for the consent management process²

4. **Identity-Based Access Control (IBAC):** IBAC complements RBAC by introducing identity-based controls. In this model, access permissions are directly associated with individual user identities. The technical aspects include:

a. **Identity-based Permissions:**

- Users are granted specific permissions based on their unique identity.
- This allows for more fine-granular control, especially in scenarios where individual users need specific privileges outside their assigned role.

b. **Permission Granting and Revocation:**

- The **PermissionManager** facilitates the dynamic granting and revocation of permissions on an individual basis.
- This ensures that changes in user responsibilities or status can be immediately reflected in the access control system.

5. **Attribute-Based Access Control (ABAC):** ABAC introduces an additional layer of control by considering attributes associated with users. In the context of consent management, a crucial attribute is the organizational affiliation (org) extracted from the x509 certificate. The technical aspects include:

a. **Attribute Verification:**

- The chaincode checks specific attributes, such as the organizational affiliation, to make nuanced access decisions.

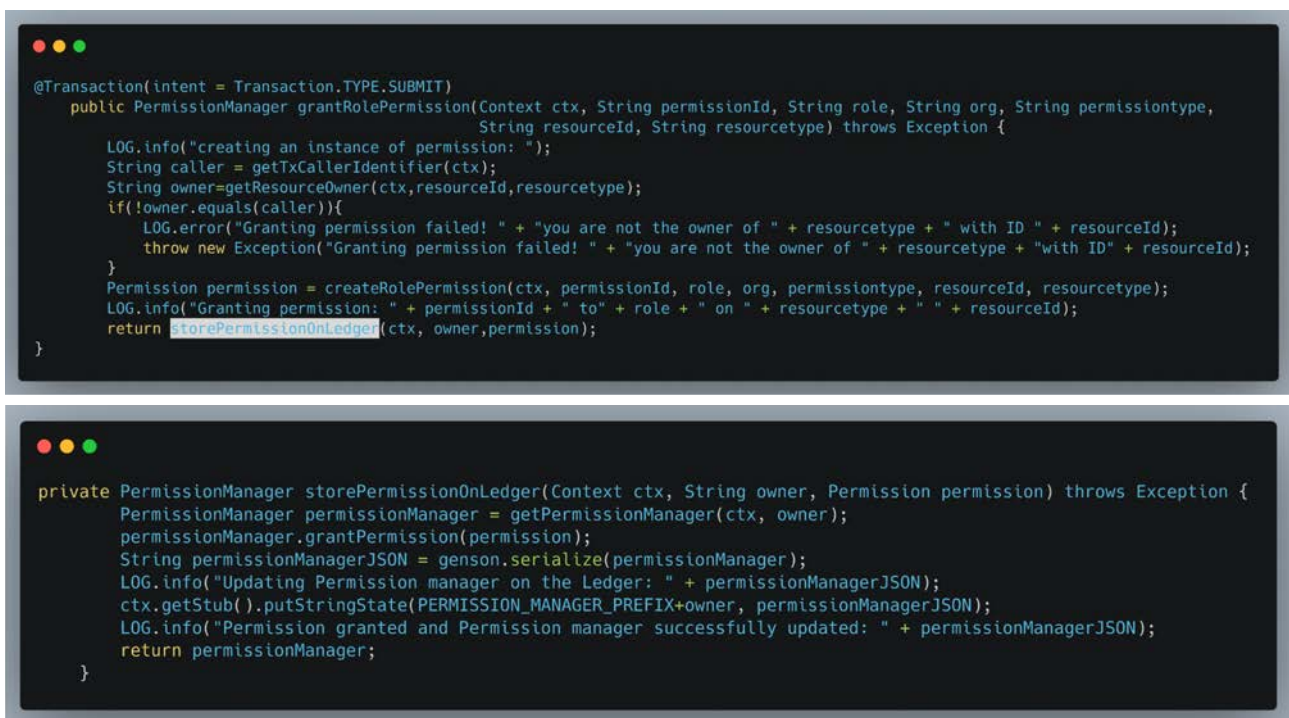
² *Note: The diagram has been simplified for readability, and certain details, including utility methods and additional functions, have been omitted. This simplification is intended to highlight the main system architecture and functionality.

- ii. For example, certain operations might be restricted to users affiliated with a particular organization.
- b. **Dynamic Attribute Policies:**
- i. Policies based on attributes can be dynamically adjusted to accommodate changes in organizational structures or requirements.
 - ii. This ensures adaptability and responsiveness to evolving access control needs.
6. **Ownership Validation:** Beyond the access control models mentioned, ownership validation is a critical technical aspect. The `PermissionManager` rigorously validates whether a user has ownership rights over specific resources (e.g., consents). This involves cross-referencing the caller's identity with the ownership information associated with the resource.

Detailed Design Process: For an in-depth understanding of the design rationale, refer to Section 5 of Deliverable D6.4, which provides a comprehensive walkthrough of the design process, outlining decisions, considerations, and the underlying architecture that governs the consent management system.

In summary, the access controls implemented by both the `PermissionManager` and the `ConsentContract` classes combine RBAC, IBAC, and ABAC, providing a robust and adaptable framework for managing permissions in the context of consent management.

Figure 14 depicts a snippet of the Java method `grantRolePermission` within the smart contract. This method implements a transaction, marked with `@Transaction(intent = Transaction.TYPE.SUBMIT)`, grants role-based permissions on a specified resource after verifying the caller's ownership. The logic includes, ownership checks, permission creation, and ledger storage.



```

@Transaction(intent = Transaction.TYPE.SUBMIT)
public PermissionManager grantRolePermission(Context ctx, String permissionId, String role, String org, String permissiontype,
                                             String resourceId, String resourcetype) throws Exception {
    LOG.info("creating an instance of permission: ");
    String caller = getTxCallerIdentifier(ctx);
    String owner = getResourceOwner(ctx, resourceId, resourcetype);
    if(!owner.equals(caller)){
        LOG.error("Granting permission failed! " + "you are not the owner of " + resourcetype + " with ID " + resourceId);
        throw new Exception("Granting permission failed! " + "you are not the owner of " + resourcetype + " with ID " + resourceId);
    }
    Permission permission = createRolePermission(ctx, permissionId, role, org, permissiontype, resourceId, resourcetype);
    LOG.info("Granting permission: " + permissionId + " to" + role + " on " + resourcetype + " " + resourceId);
    return storePermissionOnLedger(ctx, owner, permission);
}

private PermissionManager storePermissionOnLedger(Context ctx, String owner, Permission permission) throws Exception {
    PermissionManager permissionManager = getPermissionManager(ctx, owner);
    permissionManager.grantPermission(permission);
    String permissionManagerJSON = gson.toJson(permissionManager);
    LOG.info("Updating Permission manager on the Ledger: " + permissionManagerJSON);
    ctx.getStub().putStringState(PERMISSION_MANAGER_PREFIX+owner, permissionManagerJSON);
    LOG.info("Permission granted and Permission manager successfully updated: " + permissionManagerJSON);
    return permissionManager;
}

```

Figure 14. Snippet of the methods for granting a permission to a role, e.g., admin.

Figure 15 showcases the transaction method `updateConsent` responsible for updating a patient's consent, performing various checks, including input validity, patient and consent existence, and permission verification. The logic involves serialization of patient data, updating the consent, and storing the modified patient information on the ledger. Figure 16 represents the method, `canUpdateConsents`, accompanying the `updateConsent` method. This method checks whether the caller has the necessary permissions to update consents for a given patient and consent ID. It involves retrieving the caller's identity and attributes, checking against the permission manager, and evaluating permission types such as `UPDATE`, `CREATE`, and `UPDATE` for the specified patient and consent. The method utilizes the method of Figure 17 of the `PermissionManager` class for checking the permissions.

```
@Transaction(intent = Transaction.TYPE.SUBMIT)
public Patient updateConsent(Context ctx, String pid, String cid, String dataHash) throws ChaincodeException, Exception {
    LOG.info("Update a Consent: " + ctx);
    // input parameters validity check
    checkNotNullOrEmpty(pid, " Patient ID ");
    checkNotNullOrEmpty(cid, " Consent ID ");
    checkNotNullOrEmpty(dataHash, "Data Hash ");
    // checks patient and consent existence and permissions
    ChaincodeStub stub = ctx.getStub();
    if (!consentExists(ctx, pid, cid)) {
        LOG.info("Consent "+ cid +" does not exist or does not belong to patient "+pid);
        throw new ChaincodeException("Consent "+ cid +" does not exist or does not belong to patient "+pid);
    }
    if (!canUpdateConsents(ctx, pid, cid)) {
        LOG.info("Access to patient information denied.");
        throw new ChaincodeException("Access denied. Cannot update consent for patient " + pid);
    }
    Patient patient = getPatient(ctx, pid);
    String formattedDate = ctx.getStub().getTxTimestamp().atOffset(ZoneOffset.UTC).toString();
    Consent consent = new Consent(cid, formattedDate, dataHash);
    LOG.info("Create an update instance of the consent: " + consent);
    LOG.info("Update consent for given patient: " + patient);
    patient.getConsents().put(cid, consent);
    String patientSON = gson.serialize(patient);
    LOG.info("Update patient's consent: " + patientSON);
    stub.putStringState(pid, patientSON);
    LOG.info("Consent successfully updated: " + patientSON);
    return patient;
}
```

Figure 15. Snippets of the transaction method issue consent including permission checks.

```
private boolean canUpdateConsents(Context ctx, String pid, String cid) throws Exception {
    ChaincodeStub stub = ctx.getStub();
    PermissionManager permissionManager = getPermissionManager(ctx, getValueOfKey(stub, PATIENT_TO_OWNER_PREFIX, pid));
    String callerId = getTxCallerIdentifier(ctx);
    String callerOrg = ctx.getClientIdentity().getAttributeValue("org");
    String callerRole = ctx.getClientIdentity().getAttributeValue("role");
    return isConsentOwner(stub, callerId, cid)
        || permissionManager.hasPermission(callerId, callerRole, callerOrg, cid, PermissionType.UPDATE)
        || permissionManager.hasPermission(callerId, callerRole, callerOrg, pid, PermissionType.CREATE)
        || permissionManager.hasPermission(callerId, callerRole, callerOrg, pid, PermissionType.UPDATE)
}
```

Figure 16. Snippet of permission checks on a consent update

```

public boolean hasPermission(String identifier, String role, String org, String resourceId, PermissionType permissionType)
{
    if (!resourceToPermissions.containsKey(resourceId)) {
        return false;
    }
    Set<Permission> existingPermissions = resourceToPermissions.get(resourceId);
    Permission matchingPermission = existingPermissions
        .stream()
        .filter(permission -> matchesCallerIdentity(permission.getRecipient(), identifier, role, org))
        .filter(permission -> permission.getPermissionType().equals(permissionType))
        .findFirst()
        .orElse(null);
    return matchingPermission != null;
}

```

Figure 17. Snippet of the method `hasPermission` of the `PermissionManager`

6.4.2 MLStudy Contract

The `MLStudyContract` in FeatureCloud helps manage the machine learning studies, ensuring that study details, input data, and consents maintained off-chain for privacy are committed on-chain, allowing for transparent traceability and secure auditability [4].

In FeatureCloud, ML studies, for which the workflow and algorithms are provided by a coordinator, undergo local execution by participants using their locally managed healthcare data. The coordinator can invite and confirm participant additions to the study. Once the study state transitions to "executing," participants commence executing their ML applications and push commitments to the blockchain of updates to their local model results. This process continues until all participant models are aggregated. Aggregation can either be performed by the coordinator, or decentralized via Secure Multi-Party Computation (SMPC). For a comprehensive understanding of the federated machine learning process and ML study design, please refer to [5].

The class diagram of Figure 18 outlines the essential components of the ML study management process within FeatureCloud. The ML study permissions employ a similar approach to consent management, combining Identity-Based Access Control (IBAC), Role-Based Access Control (RBAC), and Attribute-Based Access Control (ABAC). In contrast to consent management, ML studies involve a singular asset/resource – the ML study itself – resulting in a simpler permission structure. The majority of permissions are implicitly based upon the study owner (typically the coordinator) and, eventually, the administrators of the owner organization (derived from the 'org' attribute in the owner's identity, i.e., the x509 certificate) or local project managers (lpm) assigned to the study. Noteworthy exceptions include the ability to submit intermediary results (commitments of local model results), a permission that can be selectively granted to study participants. This distinction accounts for the relatively simplified architecture of permission management depicted in this diagram.

MLStudyContract Class: The `MLStudyContract` class serves as the core smart contract for managing machine learning studies. It incorporates functionalities for announcing studies, adding and removing participants, changing study states, setting final results, and submitting intermediary results. The class interfaces with the Fabric blockchain through methods that interact with the ledger.

PermissionManager Class: The `PermissionManager` class helps enforce access controls within the `MLStudyContract`. It handles permissions related to various operations, ensuring that only authorized entities can perform specific actions. The class implements role-based access control (RBAC) and identity-based access control (IBAC), allowing for definition and verification of permissions. Table 1 summarizes a part of the permissions on ML studies and required conditions to get them granted.

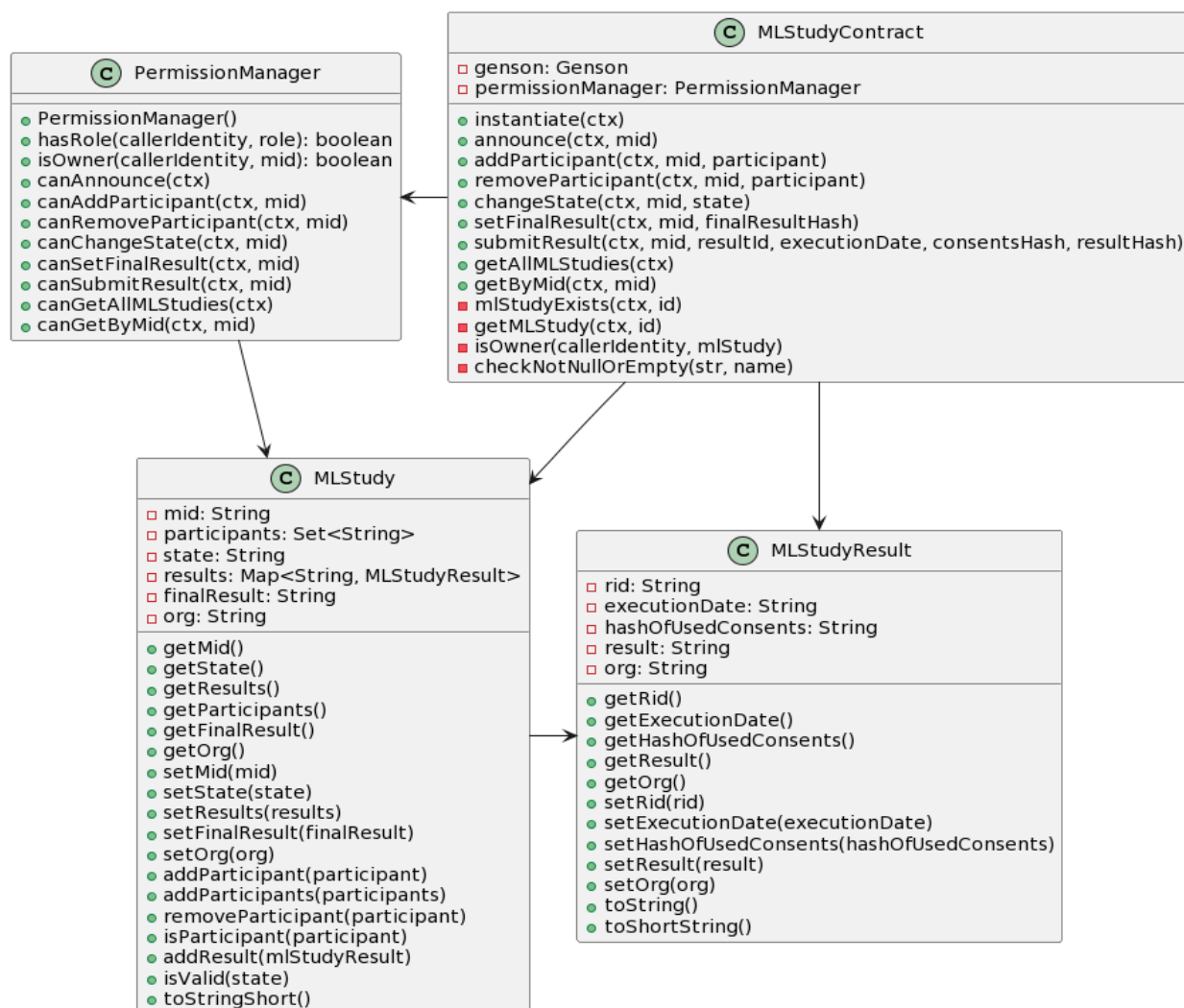


Figure 18. Class diagram for the ML study management process

Table 1. Permissions on ML studies. (lpm: local project manager, mid: machine learning study ID)

Permission/Condition	role=admin	owner	role=admin && org=owner.org	role=lpm && mid =studyId org=owner.org	isParticipant && role=admin	isParticipant && role=lpm && mid=studyId	role=auditor
canAnnounce	x						
canAddParticipant		x	x	x			
canRemoveParticipant		x	x	x			
canChangeState		x	x	x			
canSubmitResult					x	x	
canSetFinalResult		x	x	x			
canGetbyMid		x	x	x	x	x	x
canGetAllStudies		x	x	x	x	x	x

6.4.3 Chaincode Deployment

The Fabric chaincode lifecycle involves deploying and managing chaincode on a channel (cf. Figure 19). The process begins with **packaging the chaincode** into a tar file, including a metadata file specifying language, code path, and package label. Each organization independently packages the chaincode. Subsequently, the **chaincode package is installed** on every peer that will execute and endorse transactions, generating a package identifier. Channel members must then collectively **approve a chaincode definition**, reaching consensus on parameters such as name, version, and endorsement policy.

This approval is submitted to the ordering service by each participating organization. Once a sufficient number of organizations has approved, one organization **commits the approved chaincode definition** to the channel. This involves a commit transaction proposal sent to channel peers, who endorse it based on their approved definitions. The transaction is submitted to the ordering service, which finalizes the chaincode definition commitment to the channel. With the committed definition, **the chaincode container launches** on installed peers, making the chaincode available for channel members to use, subject to the specified endorsement policy. An initiation function may be invoked if required by the chaincode definition. The lifecycle process also accommodates chaincode upgrades. To upgrade, the chaincode can be repackaged, and the new package installed on peers.

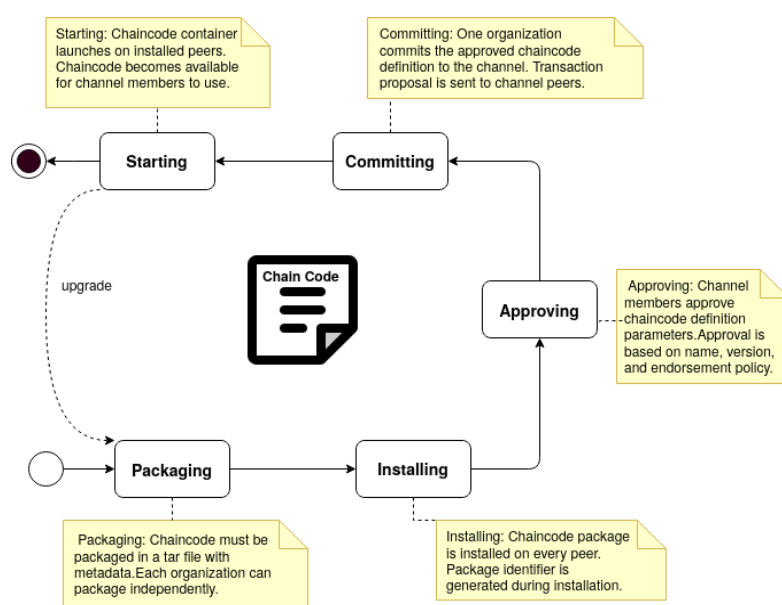


Figure 19. Chaincode lifecycle

In FeatureCloud, the deployment of each smart contract, such as the **ConsentContract** and **MLStudyContract**, is carried out on every peer, adhering to the outlined process. Figure 20 presents relevant code snippets, showcasing a portion of the script responsible for deploying the chaincode, providing an illustration of the steps involved in this process. Additionally, the deployment of one of the smart contracts on a specific peer, "uni-wien.featureCloud.net," serves as a practical illustration of how the deployment process is executed in a real-world scenario.

Once all participant identities are defined, and the peer and orderer nodes are deployed, the FeatureCloud channel is joined, and all chaincodes are successfully deployed on every peer, the FeatureCloud blockchain network is considered operational. At this point, nodes within the network are capable of receiving transaction requests and engaging in secure communication facilitated by TLS.

```
install_approve_chaincode() {
...
# Chaincode packaging
peer lifecycle chaincode package $PACKAGE_PATH --lang java --path ./${CONTRACT_NAME} --label $LABEL
peer lifecycle chaincode install $PACKAGE_PATH
#extract package ID
export PACKAGE_ID=$(peer lifecycle chaincode calculatepackageid $PACKAGE_PATH)
# Chaincode approval
peer lifecycle chaincode approveformyorg --orderer localhost:7050 --ordererTLSHostnameOverride
orderer.featurecloud.net --channelID fcchannel --name $CONTRACT_NAME -v 0 --package-id
$PACKAGE_ID --sequence 1 --tls --cafile $ORDERER_CA
}

# Install and approve consentcontract chaincode for UniWien
install_approve_chaincode localhost:11051 UniWienMSP $PWD/../fc-network/organizations/peerOrganizations/uni-
wien.featurecloud.net/peers/peer0.uni-wien.featurecloud.net/tls/ca.crt $PWD/../fc-network/organizations/peerOrganizations
/uni-wien.featurecloud.net/users/Admin@uni-wien.featurecloud.net/msp cp0_uni-wien consentcontract cp_0
# Chaincode commit for consentcontract
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride orderer.featurecloud.net
--peerAddresses localhost:9051 --tlsRootCertFiles --peerAddresses localhost:11051 --tlsRootCertFiles $PEER0_UniWien_CA
--channelID fcchannel --name consentcontract -v 0 --sequence 1 --tls --cafile $ORDERER_CA --waitForEvent
```

Figure 20. Code snippets of chaincode deployment scripts

In the illustrated scenario (refer to Figure 21), a participant equipped with only one peer node would involve five Docker containers. These include two for the smart contracts, namely MIStudyContract and ConsentContract, one for the peer node, another for the ledger state, and one more for the Certificate Authority (CA). It's worth noting that this configuration may vary in organizations with a different number of nodes, multiple CAs, or additional smart contracts. For more complex setups, additional containers might be deployed to accommodate operational services or client applications (c.f. Section 6.5).

CONTAINER ID	IMAGE	STATUS	PORTS	NAMES
b92468ac5b9d	dev-peer0.consentcontractID	Up About a minute		dev.consentContract.peer0.ID
decf113f0cc8	dev-peer0.mlstudycontractID	Up About a minute		dev.mlstudyContract.peer0.ID
65a69395c869	hyperledger/fabric-peer:latest	Up About an hour	7051/tcp, 131.129.1.7:11051->11051/tcp, :::11051->11051/tcp	peer0.uni-wien.featurecloud.net
75580cb25f55	couchdb:3.3.2	Up About an hour	4369/tcp, 9100/tcp, 131.129.1.7:9984->5984/tcp, :::9984->5984/tcp	couchdb4
61b0d1033f5	hyperledger/fabric-ca:latest	Up About an hour	7054/tcp, 131.129.1.7:11054->11054/tcp, :::11054->11054/tcp	ca_uni-wien

Figure 21. Docker containers for the deployed chaincodes

6.5 Integration and Client Application of the FeatureCloud Blockchain Network

In the previous sections, we covered deploying the FeatureCloud blockchain network and implementing chaincodes governing patient consents and ML studies. In this section, our focus shifts to the client applications, to facilitate smooth interactions with smart contracts and seamless integration with the FeatureCloud platform.

6.5.1 Architecture Overview

The architectural framework of the FeatureCloud blockchain is comprehensively illustrated in Figure 22. FeatureCloud participants have the flexibility to either operate their individual peer or orderer nodes or opt for shared deployment scenarios. The latter means that participants do not operate their dedicated peer or orderer nodes. Instead, they connect to peers belonging to other organizations without having nodes of their own. Moreover, a participant possesses the liberty to manage multiple peer and orderer instances, enhancing the security of the ledger.

Following an iterative approach, and in order to enhance the integration capabilities of the FeatureCloud blockchain within the broader FeatureCloud platform and improve the user experience, we've introduced **refinements** to the existing system described in **D7.5** and **D6.4**. While the initial design enables direct interaction between the FeatureCloud (FC) controller and the Blockchain CLI component within a local network (with the CLI deployed as a Docker container within the FeatureCloud controller), we recognize the need for a more versatile and scalable approach.

To address this, we have improved our integration strategy by incorporating RESTful services into the FeatureCloud blockchain architecture. This involves exposing all chaincode functions as REST services, leading to improved interoperability and streamlined chaincode invocations by the FeatureCloud controller. Participants now have the flexibility to choose between invoking functions directly through local communication channels or leveraging RESTful APIs for broader network scenarios.

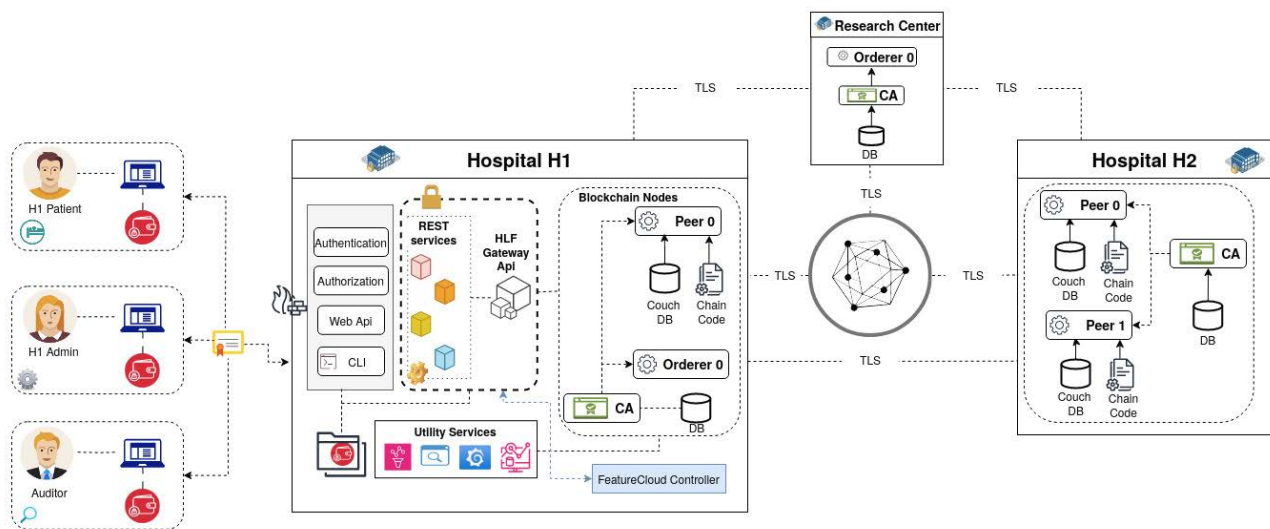


Figure 22. FeatureCloud Blockchain Architecture Overview

In Figure 22, the Hyperledger Fabric Gateway serves as an intermediary, simplifying the interaction between blockchain client applications and the network. It grants access to blockchain networks and Smart Contracts through a single endpoint, thereby facilitating easy submission of transactions and ledger queries on behalf of client applications. To establish a session for a specific client identity, the application builds and connects to a Fabric Gateway using a gRPC connection to the Gateway endpoint.

Before Hyperledger Fabric version 2.4, interaction with the blockchain was facilitated through a legacy Gateway SDK. This has since been replaced by the Gateway Client API 1.4, designed for Hyperledger Fabric versions 2.4 and beyond. While the Fabric Gateway client API is now the recommended choice for developing applications for Fabric from version 2.4 onward, the legacy SDK continues to be supported. Since much of the FeatureCloud blockchain code predates version 2.4, the legacy SDK was initially utilized, but transitioning to the new client API is a straightforward process. Key differences can be found in the migration documentation [6].

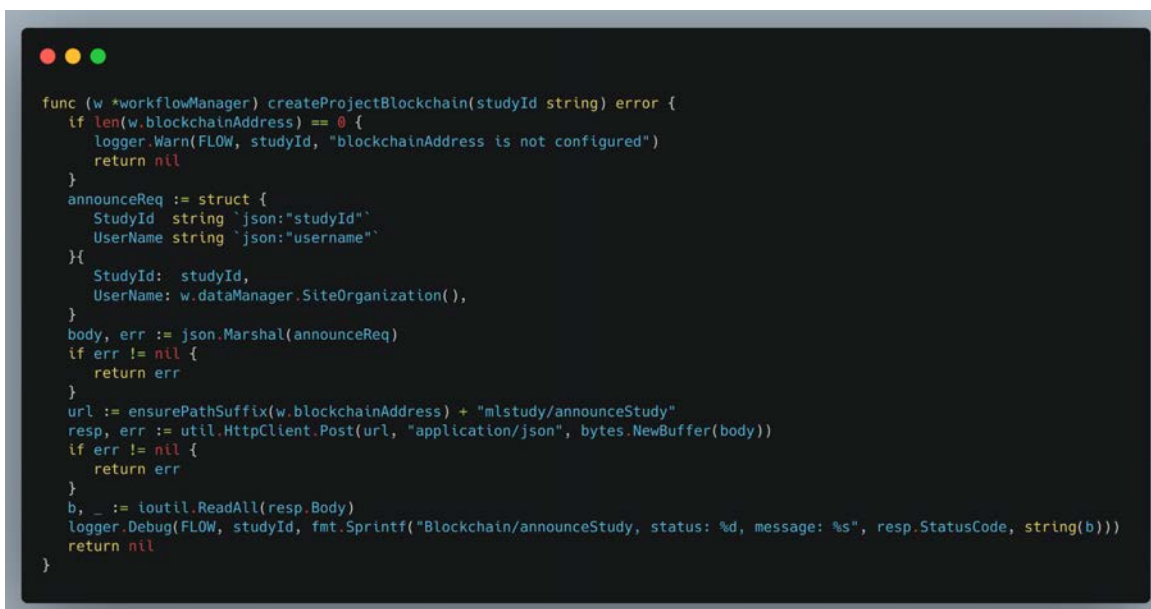
In conjunction with the aforementioned enhancements, our focus extended to refining the user interface for facilitating interaction with the FeatureCloud blockchain, considering users with diverse technical skills. The Command-Line Interface (CLI) caters to developers and advanced users, offering a direct and efficient way to interact with the FeatureCloud blockchain. Simultaneously, our user-friendly web interface serves as a gateway for users less familiar with technical details.

This user interface also incorporates new components for authentication and role-based authorization, ensuring that users have access only to functionalities and views aligned with their roles. These components dynamically shape the interface based on user permissions, offering a tailored experience aligned with specific roles within the FeatureCloud platform.

The approach of decentralized service hosting ensures that each participant has control over their specific services, autonomously hosting their own dedicated web API and REST services, along with an identity management system, all protected by a robust firewall, thereby safeguarding their services from potential external threats.

6.5.2 RESTful Integration with the FeatureCloud platform

As previously highlighted, the main FeatureCloud controller has now been extended to support communication and integration with REST services. This extension enables the invocation of smart contract functionalities, including the commitment and management of machine learning results and input data (i.e., commitments). In Figure 23, a code snippet in Go illustrates the invocation of the `mlstudy/announceStudy` endpoint from the main FeatureCloud controller.



```
func (w *workflowManager) createProjectBlockchain(studyId string) error {
    if len(w.blockchainAddress) == 0 {
        logger.Warn(FLow, studyId, "blockchainAddress is not configured")
        return nil
    }
    announceReq := struct {
        StudyId string `json:"studyId"`
        Username string `json:"username"`
    }{
        StudyId: studyId,
        Username: w.dataManager.SiteOrganization(),
    }
    body, err := json.Marshal(announceReq)
    if err != nil {
        return err
    }
    url := ensurePathSuffix(w.blockchainAddress) + "mlstudy/announceStudy"
    resp, err := util.HttpClient.Post(url, "application/json", bytes.NewBuffer(body))
    if err != nil {
        return err
    }
    b, _ := ioutil.ReadAll(resp.Body)
    logger.Debug(FLow, studyId, fmt.Sprintf("Blockchain/announceStudy, status: %d, message: %s", resp.StatusCode, string(b)))
    return nil
}
```

Figure 23. Go snippet of study announcement invocation

Furthermore, Figures 24 and 25 depict the FeatureCloud platform's capability to facilitate the **commitment** of federated learning results. Notably, an essential addition to the interface is the introduction of the "audit" checkbox. When selected, this checkbox triggers the local storage of data and commits corresponding hashes to the blockchain (cf. Figure 24). Additionally, if a workflow is initiated with the "audit" checkbox enabled, the interface dynamically reveals the "Add consent" button, providing participants with a straightforward means to upload consents for processing the data (e.g. for machine learning) and necessary for the audit process (cf. Figure 25). We refer to deliverable **D7.5** for more details about the integration of the blockchain prototype with the overall FeatureCloud platform.

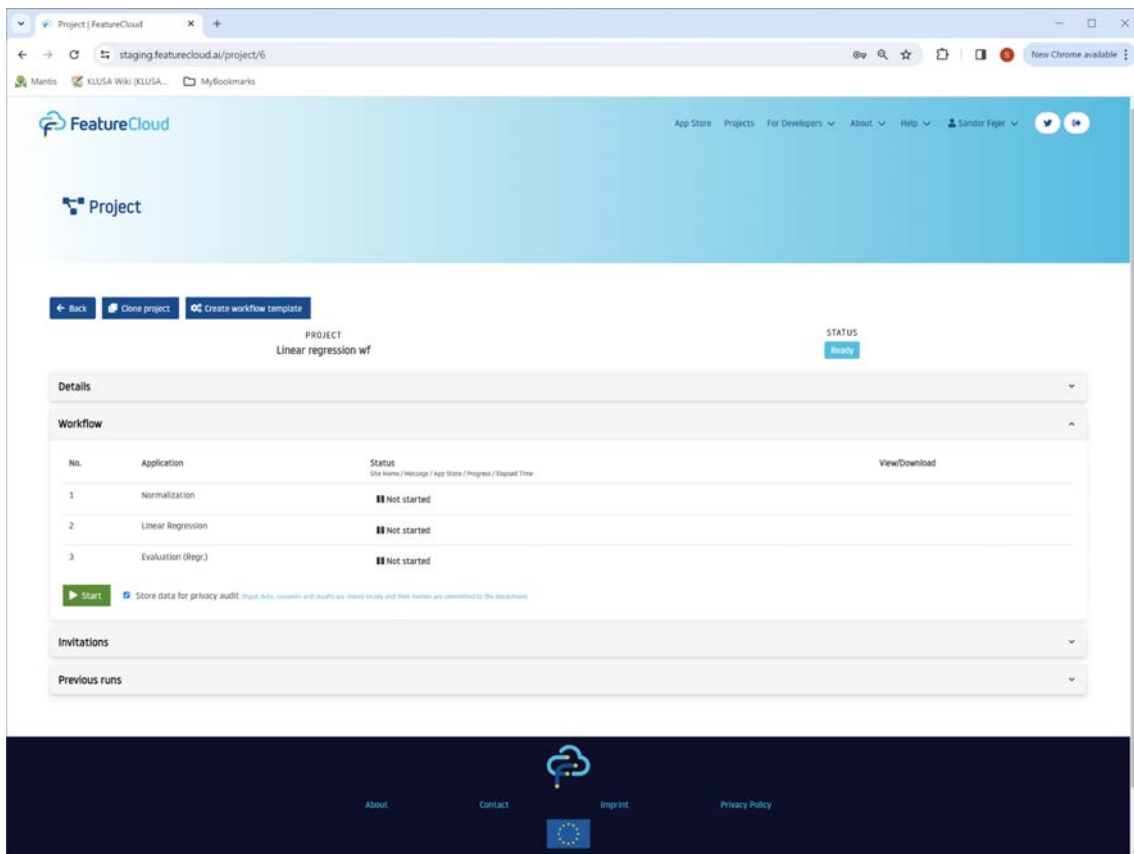


Figure 24. Blockchain integration with FeatureCloud platform

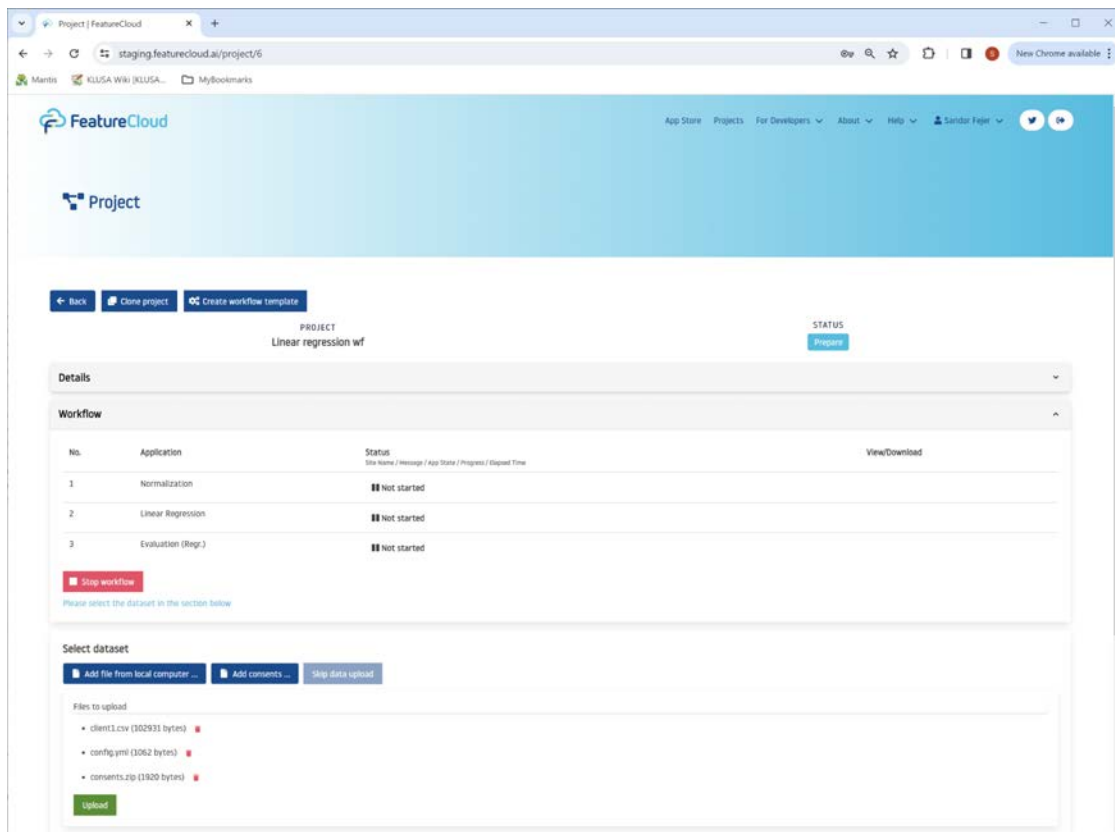


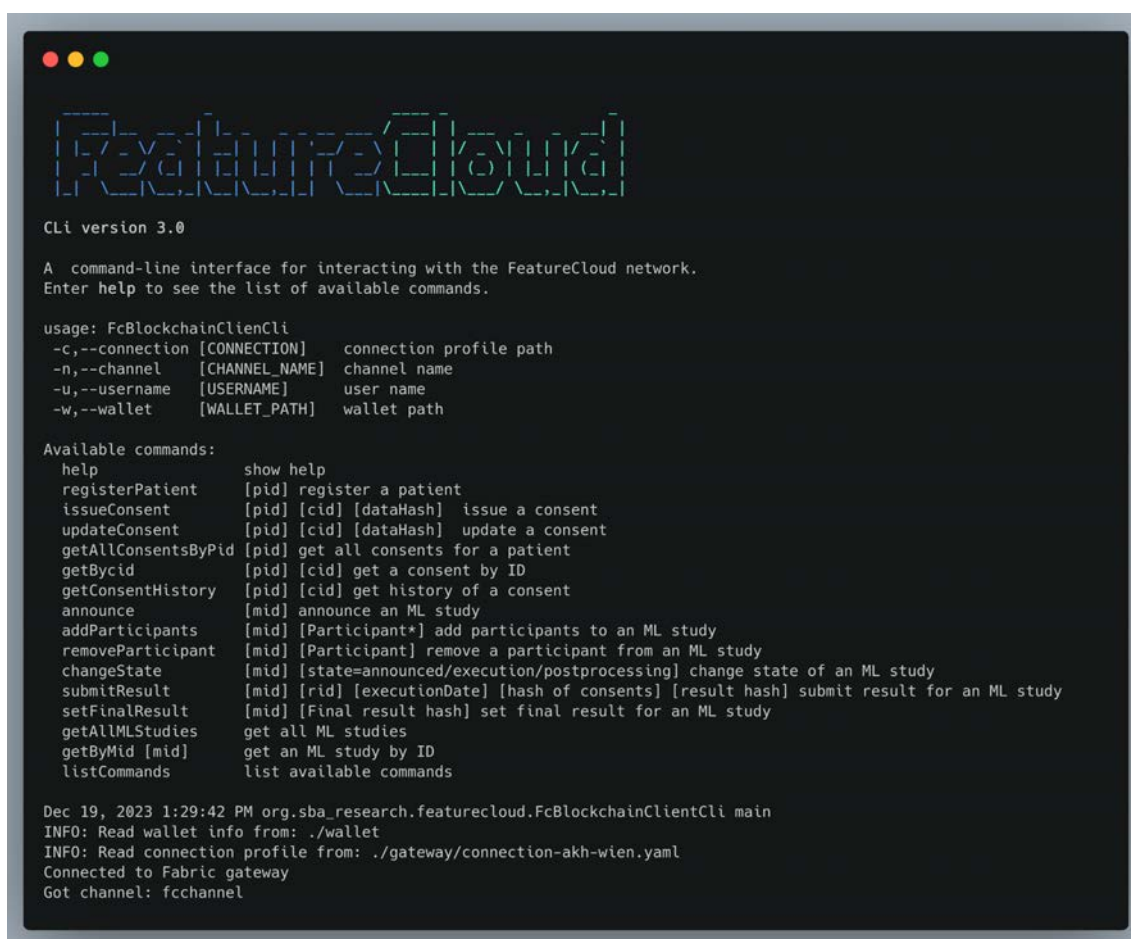
Figure 25. Blockchain integration with FeatureCloud platform

6.6 Command Line Interface (CLI)

In the following, we showcase the FeatureCloud blockchain CLI, and highlight various types of interaction and processes. Multiple CLI instances operate concurrently, with distinct participants running them using their individual credentials. In the following screenshots, the organization responsible for a specific CLI instance is identifiable through the connection profile used for blockchain connectivity (e.g., “INFO: Read connection profile from: `./gateway/connection-akh-wien.yaml`”).

Screenshots for Consents

Figure 26 shows the main command-line interface, executed by the participant `akh-wien`. It shows the available commands for interacting with the FeatureCloud blockchain, and specifically to the `fcchannel`.



```

[akh-wien@akh-wien ~]$ fccli
CLI version 3.0

A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

usage: FcBlockchainClientCli
  -c,--connection [CONNECTION]  connection profile path
  -n,--channel [CHANNEL_NAME]   channel name
  -u,--username [USERNAME]      user name
  -w,--wallet [WALLET_PATH]     wallet path

Available commands:
  help                show help
  registerPatient     [pid] register a patient
  issueConsent        [pid] [cid] [dataHash] issue a consent
  updateConsent       [pid] [cid] [dataHash] update a consent
  getAllConsentsByPid [pid] get all consents for a patient
  getBycid            [pid] [cid] get a consent by ID
  getConsentHistory   [pid] [cid] get history of a consent
  announce            [mid] announce an ML study
  addParticipants     [mid] [Participant*] add participants to an ML study
  removeParticipant   [mid] [Participant] remove a participant from an ML study
  changeState         [mid] [state=announced/execution/postprocessing] change state of an ML study
  submitResult        [mid] [rid] [executionDate] [hash of consents] [result hash] submit result for an ML study
  setFinalResult      [mid] [Final result hash] set final result for an ML study
  getAllMLStudies     get all ML studies
  getByMid [mid]      get an ML study by ID
  listCommands        list available commands

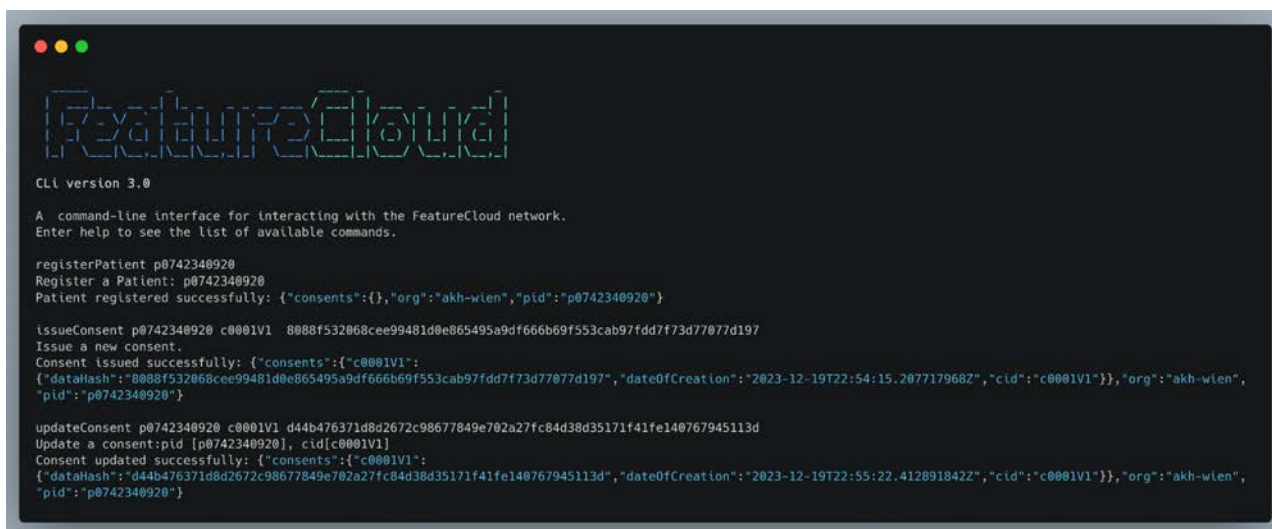
Dec 19, 2023 1:29:42 PM org.sba_research.featurecloud.FcBlockchainClientCli main
INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-akh-wien.yaml
Connected to Fabric gateway
Got channel: fcchannel

```

Figure 26. Featurecloud blockchain cli - main interface

Figure 27 shows the registration process for a patient `p0742340920`. The organization (`akh-wien`) is responsible for the registration. Once registered, consents can be issued or updated on behalf of the patient. This assumes that `akh-wien` has the necessary permissions to commit consent operations on behalf of the patient. In Figure 27, consent `c0001V1` is initially issued and subsequently updated. Updates may pertain to the scope of the consent or may involve

revocation. Notably, the revocation is processed as an update operation, aligning with the privacy and data protection considerations outlined in deliverable D6.5.



```

CLI version 3.0

A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

registerPatient p0742340920
Register a Patient: p0742340920
Patient registered successfully: {"consents": {}, "org": "akh-wien", "pid": "p0742340920"}

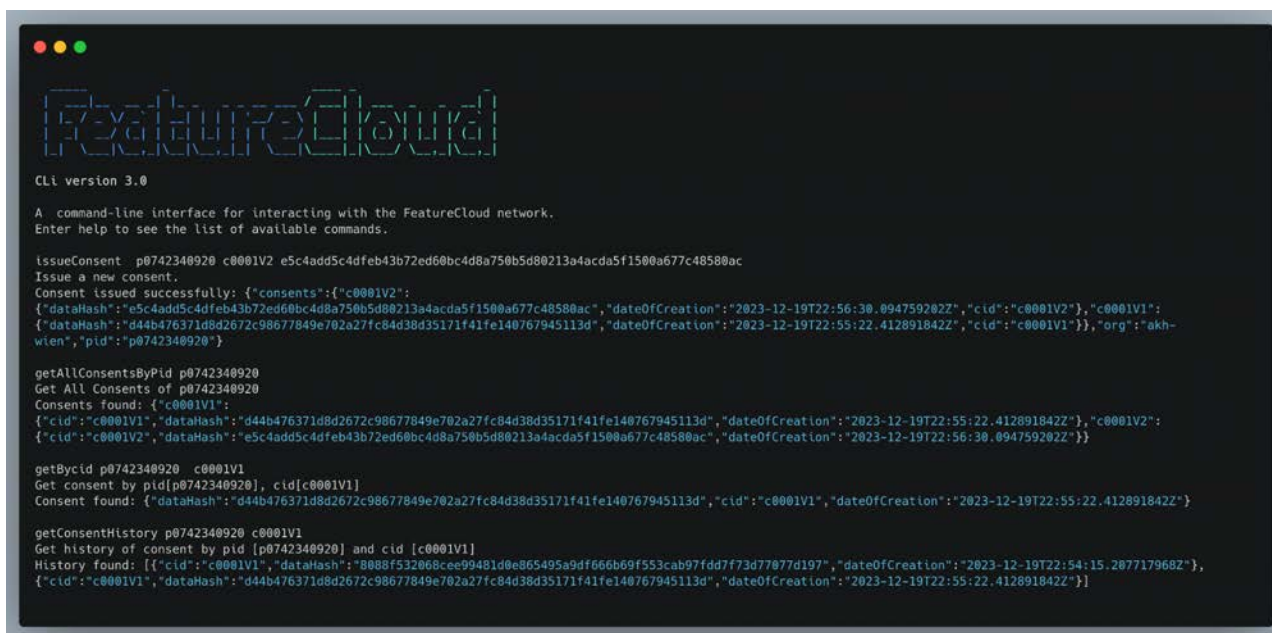
issueConsent p0742340920 c0001V1 8088f532068cee99481d0e865495a9df666b69f553cab97fdd7f73d77077d197
Issue a new consent.
Consent issued successfully: {"consents": {"c0001V1": {"dataHash": "8088f532068cee99481d0e865495a9df666b69f553cab97fdd7f73d77077d197", "dateOfCreation": "2023-12-19T22:54:15.207717968Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}}, "org": "akh-wien", "pid": "p0742340920"}

updateConsent p0742340920 c0001V1 d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d
Update a consent: pid [p0742340920], cid [c0001V1]
Consent updated successfully: {"consents": {"c0001V1": {"dataHash": "d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d", "dateOfCreation": "2023-12-19T22:55:22.412891842Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}}, "org": "akh-wien", "pid": "p0742340920"}

```

Figure 27. CLI - Registration process

Figure 28 illustrates the retrieval of consent information. The depicted commands include various actions such as issuing additional consent `c0001V2` (e.g., for distinct data or scope), retrieving all consents associated with a specific patient `p0742340920`, obtaining the latest value of a particular consent identified by its ID `c0001V1`, and accessing a detailed history of a specific consent.



```

CLI version 3.0

A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

issueConsent p0742340920 c0001V2 e5c4add5c4df6b43b72ed60bc4d8a750b5d80213a4acda5f1500a677c48580ac
Issue a new consent.
Consent issued successfully: {"consents": {"c0001V2": {"dataHash": "e5c4add5c4df6b43b72ed60bc4d8a750b5d80213a4acda5f1500a677c48580ac", "dateOfCreation": "2023-12-19T22:56:30.094759202Z", "cid": "c0001V2", "org": "akh-wien", "pid": "p0742340920"}, "c0001V1": {"dataHash": "d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d", "dateOfCreation": "2023-12-19T22:55:22.412891842Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}}, "org": "akh-wien", "pid": "p0742340920"}

getAllConsentsByPid p0742340920
Get All Consents of p0742340920
Consents found: {"c0001V1": {"cid": "c0001V1", "dataHash": "d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d", "dateOfCreation": "2023-12-19T22:55:22.412891842Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}, "c0001V2": {"cid": "c0001V2", "dataHash": "e5c4add5c4df6b43b72ed60bc4d8a750b5d80213a4acda5f1500a677c48580ac", "dateOfCreation": "2023-12-19T22:56:30.094759202Z", "cid": "c0001V2", "org": "akh-wien", "pid": "p0742340920"}}, "org": "akh-wien", "pid": "p0742340920"}

getBycid p0742340920 c0001V1
Get consent by pid [p0742340920], cid [c0001V1]
Consent found: {"dataHash": "d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d", "cid": "c0001V1", "dateOfCreation": "2023-12-19T22:55:22.412891842Z", "org": "akh-wien", "pid": "p0742340920"}

getConsentHistory p0742340920 c0001V1
Get history of consent by pid [p0742340920] and cid [c0001V1]
History found: [{"cid": "c0001V1", "dataHash": "8088f532068cee99481d0e865495a9df666b69f553cab97fdd7f73d77077d197", "dateOfCreation": "2023-12-19T22:54:15.207717968Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}, {"cid": "c0001V1", "dataHash": "d44b476371d8d2672c98677849e702a27fc84d38d35171f41fe140767945113d", "dateOfCreation": "2023-12-19T22:55:22.412891842Z", "cid": "c0001V1", "org": "akh-wien", "pid": "p0742340920"}], "org": "akh-wien", "pid": "p0742340920"}

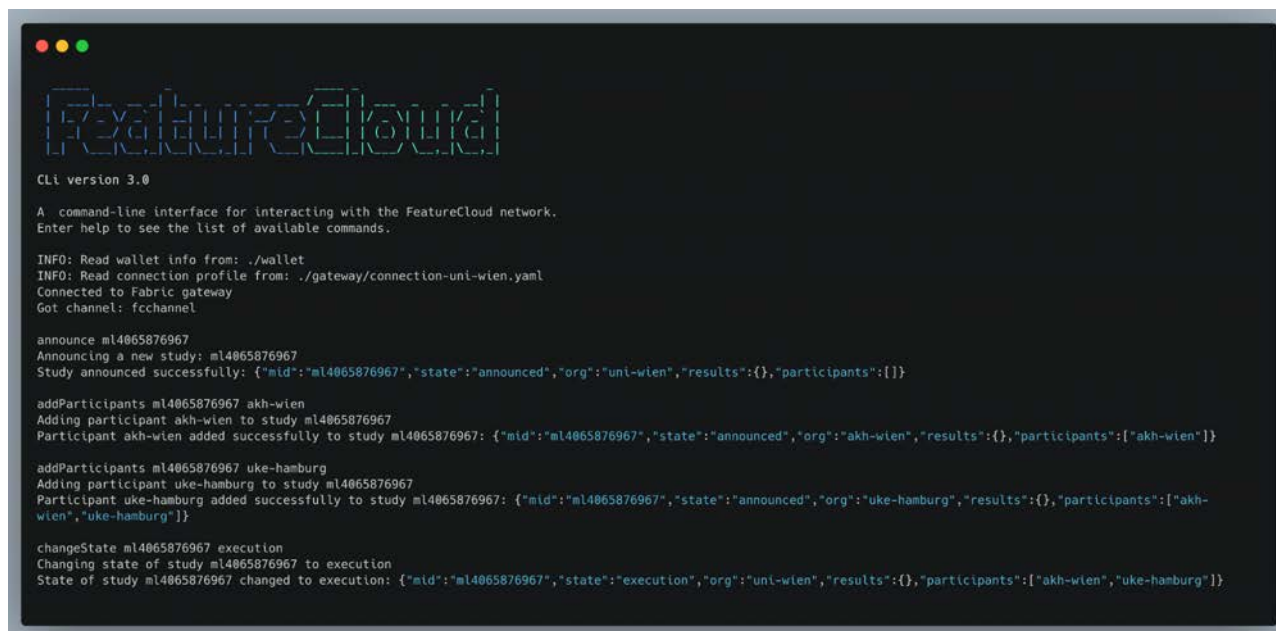
```

Figure 28. CLI - Consent management

Screenshots of ML studies

The interaction shown in Figure 29 announces a new Machine Learning (ML) study identified as `m14065876967`. This announcement is initiated by the organization `uni-wien`, signaling the commencement of the ML study. Participants, namely `akh-wien` and `uke-hamburg`, are included

in the ML study `ml4065876967`, with the organization `uni-wien` overseeing the participant addition process. To enable participants to submit interim results, `uni-wien` is required to transition the study's state to `executing`.



```

FeatureCloud
CLI version 3.0

A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-uni-wien.yaml
Connected to Fabric gateway
Got channel: fcchannel

announce ml4065876967
Announcing a new study: ml4065876967
Study announced successfully: {"mid":"ml4065876967","state":"announced","org":"uni-wien","results":{},"participants":{}}

addParticipants ml4065876967 akh-wien
Adding participant akh-wien to study ml4065876967
Participant akh-wien added successfully to study ml4065876967: {"mid":"ml4065876967","state":"announced","org":"akh-wien","results":{},"participants":["akh-wien"]}

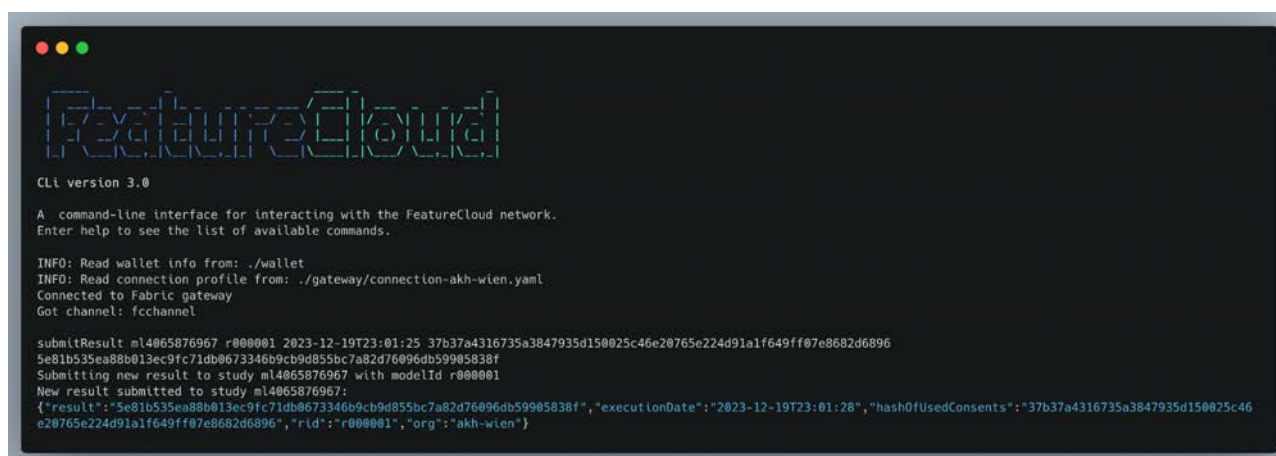
addParticipants ml4065876967 uke-hamburg
Adding participant uke-hamburg to study ml4065876967
Participant uke-hamburg added successfully to study ml4065876967: {"mid":"ml4065876967","state":"announced","org":"uke-hamburg","results":{},"participants":["akh-wien","uke-hamburg"]}

changeState ml4065876967 execution
Changing state of study ml4065876967 to execution
State of study ml4065876967 changed to execution: {"mid":"ml4065876967","state":"execution","org":"uni-wien","results":{},"participants":["akh-wien","uke-hamburg"]}

```

Figure 29. CLI - Announcing a new ML study

In Figures 30 and 31, interim results labeled `r000001` and `r000002` are submitted for the Machine Learning (ML) study `ml4065876967`. These submissions are independently carried out by distinct organizations, namely `akh-wien` and `uke-hamburg`, each utilizing their own cli instance and credentials for the submission process.



```

FeatureCloud
CLI version 3.0

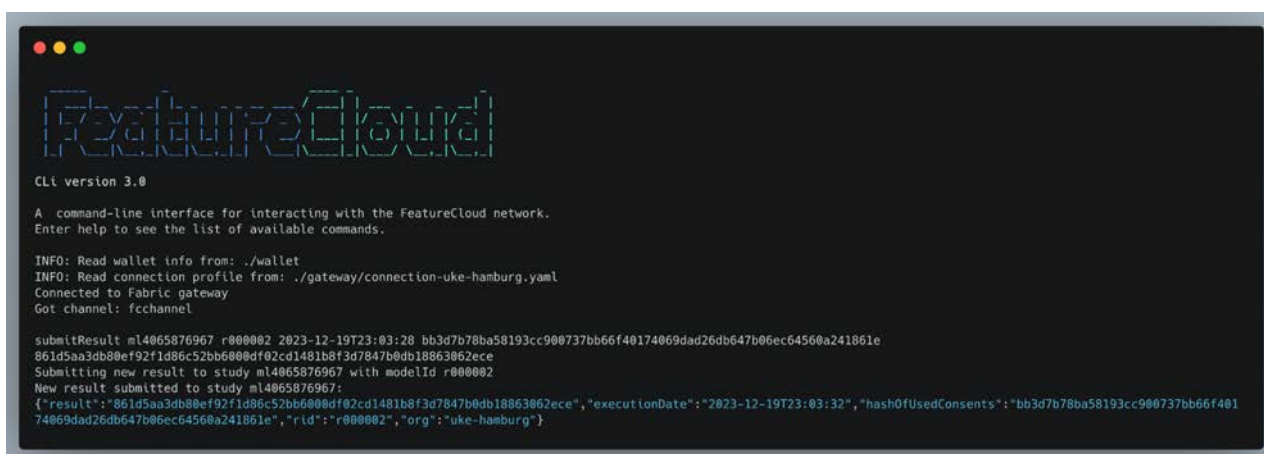
A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-akh-wien.yaml
Connected to Fabric gateway
Got channel: fcchannel

submitResult ml4065876967 r000001 2023-12-19T23:01:25 37b37a4316735a3847935d150025c46e20765e224d91a1f649ff07e8082d6896
5e81b535ea8b013ec9fc71db0673346b9cb9d855bc7a82d76006db59905838f
Submitting new result to study ml4065876967 with modelId r000001
New result submitted to study ml4065876967:
{"result":"5e81b535ea8b013ec9fc71db0673346b9cb9d855bc7a82d76006db59905838f","executionDate":"2023-12-19T23:01:28","hashOfUsedConsents":"37b37a4316735a3847935d150025c46e20765e224d91a1f649ff07e8082d6896","rid":"r000001","org":"akh-wien"}

```

Figure 30. CLI - Submitting results by participant akh-wien



```

CLI version 3.0

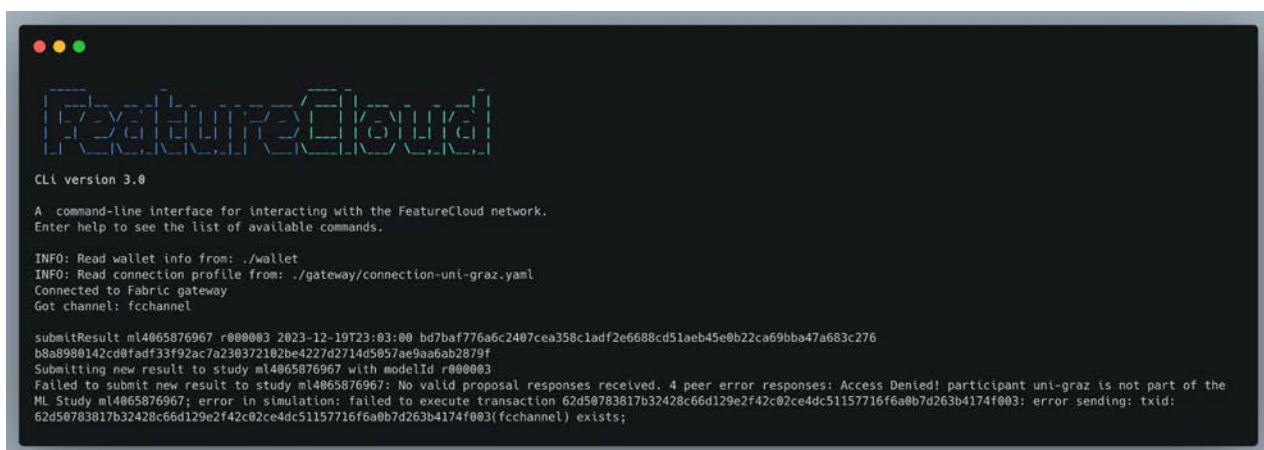
A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-uke-hamburg.yaml
Connected to Fabric gateway
Got channel: fcchannel

submitResult m14065876967 r000002 2023-12-19T23:03:28 bb3d7b78ba58193cc900737bb66f40174069dad26db647b06ec64560a241861e
861d5aa3db80ef92f1d86c52bb6000df02cd1481b8f3d7847b0db18863062ece
Submitting new result to study m14065876967 with modelId r000002
New result submitted to study m14065876967:
{"result":{"861d5aa3db80ef92f1d86c52bb6000df02cd1481b8f3d7847b0db18863062ece","executionDate":"2023-12-19T23:03:32","hashOfUsedConsents":"bb3d7b78ba58193cc900737bb66f40174069dad26db647b06ec64560a241861e","rid":"r000002","org":"uke-hamburg"}}
```

Figure 31. CLI -Submitting results by participant uke-hamburg

In Figure 32, interim results labeled **r000003** are attempted to be submitted for the Machine Learning (ML) study identified as **m14065876967** by the organization **akh-graz**. However, the transaction is rejected due to the organization lacking the necessary permissions to submit results and not being listed among the participants for that particular study.



```

CLI version 3.0

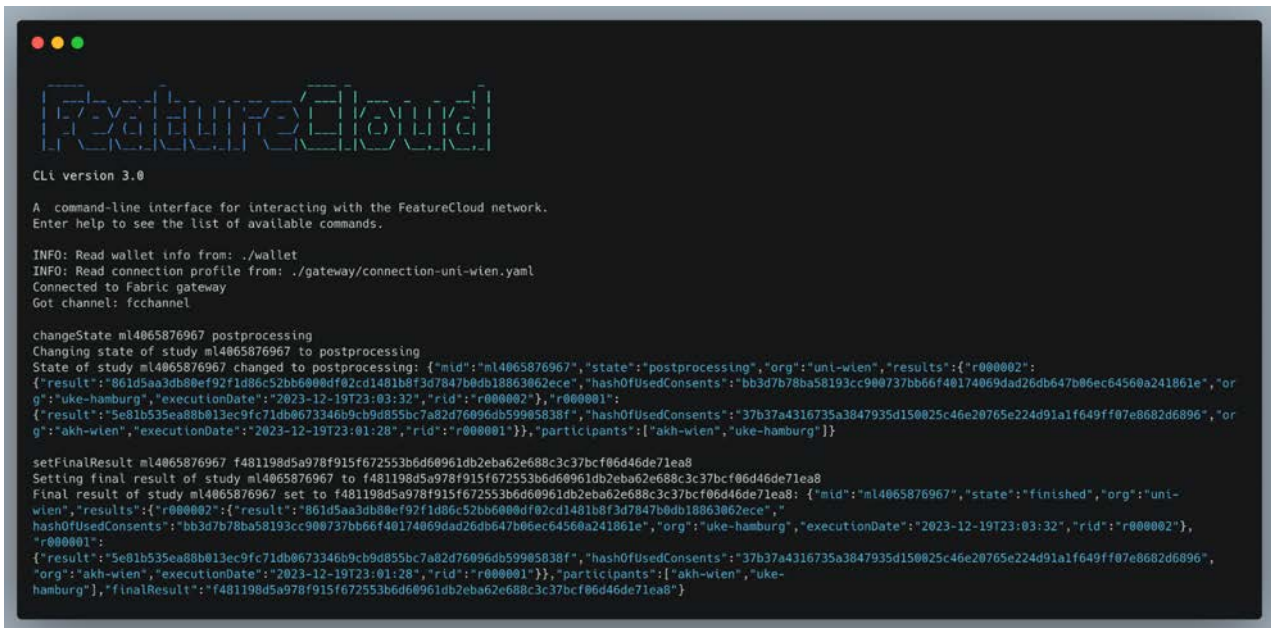
A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-uni-graz.yaml
Connected to Fabric gateway
Got channel: fcchannel

submitResult m14065876967 r000003 2023-12-19T23:03:00 bd7baf776a6c2407cea358c1adf2e6688cd51aeb45e0b22ca69bba47a683c276
b8a8980142cd0fadf33f92ac7a230372102be4227d2714d5057ae9aa6ab2879f
Submitting new result to study m14065876967 with modelId r000003
Failed to submit new result to study m14065876967: No valid proposal responses received. 4 peer error responses: Access Denied! participant uni-graz is not part of the
ML Study m14065876967; error in simulation: failed to execute transaction 62d50783817b32428c66d129e2f42c02ce4dc51157716f6a0b7d263b4174f003: error sending: txid:
62d50783817b32428c66d129e2f42c02ce4dc51157716f6a0b7d263b4174f003(fcchannel) exists;
```

Figure 32. CLI -Submitting results by a non-participant (uni-graz)

In Figure 33, the Machine Learning (ML) study, identified as **m14065876967**, transitions to the post-processing state under the coordination of the organization **uni-wien**. Following this progression, the final results (an aggregation of all local models) can be submitted by the coordinator **uni-wien**. This submission marks the successful completion of the study.



```

CLI version 3.0

A command-line interface for interacting with the FeatureCloud network.
Enter help to see the list of available commands.

INFO: Read wallet info from: ./wallet
INFO: Read connection profile from: ./gateway/connection-uni-wien.yaml
Connected to Fabric gateway
Got channel: fcchannel

changeState m14065876967 postprocessing
Changing state of study m14065876967 to postprocessing
State of study m14065876967 changed to postprocessing: {"mid": "m14065876967", "state": "postprocessing", "org": "uni-wien", "results": {"r000002": {"result": "861d5aa3db80ef92f1d86c52bb6000df02cd1481b8f3d7847b0db18863862ece", "hashOfUsedConsents": "bb3d7b78ba58193cc900737bb66f40174069dad26db647b06ec64560a241861e", "org": "uke-hamburg", "executionDate": "2023-12-19T23:03:32", "rid": "r000002"}, "r000001": {"result": "5e01b535ea8b013ec9fc71db0673346b9cb9d855bc7a82d76096db59905838f", "hashOfUsedConsents": "37b37a4316735a3847935d150025c46e20765e224d91af649ff07e8682d6896", "org": "akh-wien", "executionDate": "2023-12-19T23:01:28", "rid": "r000001"}}, "participants": ["akh-wien", "uke-hamburg"]}}

setFinalResult m14065876967 f481198d5a978f915f672553b6d60961db2eba62e688c3c37bcf06d46de71ea8
Setting final result of study m14065876967 to f481198d5a978f915f672553b6d60961db2eba62e688c3c37bcf06d46de71ea8
Final result of study m14065876967 set to f481198d5a978f915f672553b6d60961db2eba62e688c3c37bcf06d46de71ea8: {"mid": "m14065876967", "state": "finished", "org": "uni-wien", "results": {"r000002": {"result": "861d5aa3db80ef92f1d86c52bb6000df02cd1481b8f3d7847b0db18863862ece", "hashOfUsedConsents": "bb3d7b78ba58193cc900737bb66f40174069dad26db647b06ec64560a241861e", "org": "uke-hamburg", "executionDate": "2023-12-19T23:03:32", "rid": "r000002"}, "r000001": {"result": "5e01b535ea8b013ec9fc71db0673346b9cb9d855bc7a82d76096db59905838f", "hashOfUsedConsents": "37b37a4316735a3847935d150025c46e20765e224d91af649ff07e8682d6896", "org": "akh-wien", "executionDate": "2023-12-19T23:01:28", "rid": "r000001"}}, "participants": ["akh-wien", "uke-hamburg"]}, "finalResult": "f481198d5a978f915f672553b6d60961db2eba62e688c3c37bcf06d46de71ea8"}

```

Figure 33. CLI -Submitting Final results

6.7 Web Application

While the invocation of smart contract functions is feasible through both the CLI and REST services (leveraging an implemented REST connector directly exposed to the FeatureCloud platform), it is essential to accommodate non-technical users with an accessible user interface to easily manage their consents. To fulfill this requirement, we developed a user-friendly web application that aligns with FeatureCloud's commitment to user accessibility. The REST services and the web application are shielded by a firewall, adding an extra layer of protection to the entire system. To keep data integrity and privacy, the smart contracts themselves are equipped with access controls (cf. section 6.4) that restrict access to on-chain data and smart contract functions, ensuring that only authorized users can interact with specific elements of the blockchain. These security measures help safeguard user data and facilitate a user-friendly experience for patients.

6.7.1 Web Authentication

The web application incorporates authentication mechanisms to ensure secure access and exposes RESTful endpoints, to facilitate interactions with the blockchain network, offering an intuitive experience for users. Authentication is achieved through a standard login/password mechanism. Upon successful authentication, a Json Web Token (JWT) is issued to the user. This JWT serves as a secure credential, encapsulating the user's authentication details such as roles or other associated information. Figure 34 illustrates the flow of this process. The use of JWTs offers a stateless and scalable authentication solution. This JWT acts as a bearer token, enabling the user to make subsequent authenticated requests without the need to re-enter their login credentials. The token is automatically stored in the cookies of the user's browser, although it can alternatively be saved locally and is included in the Authorization header for subsequent requests. The server can then validate the token without the need for storing a user session state on the server. This approach not only simplifies the user experience (e.g., better performance) but also ensures secure and efficient interactions.

Furthermore, the system allows users, particularly patients, to delegate the management of their cryptographic keying material and certificates to trusted entities (following the process described in Section 6.4), such as hospitals. This delegation of complexity enhances user convenience while maintaining the security of their interactions with the blockchain network.

Overall Interaction flow:

1. **User Login:** The user provides credentials (e.g., username and password) to the server, and upon successful authentication, the server responds with a JWT. The server also selects the credentials associated with the user that subsequent requests to the blockchain are correctly authenticated by the peers and the right permissions are given on-chain .
2. **Token Storage:** The client (typically a web browser) needs to store the JWT securely. Common storage options include browser storage mechanisms like localStorage or sessionStorage. The token is then used for subsequent requests.
3. **Token Inclusion in Requests:** For each subsequent request to the server, the client includes the JWT in the HTTP headers. This is often done automatically by client-side libraries or frameworks. The most common way is to include the token in the **Authorization** header using the "Bearer" scheme.
4. **Handling of the Requests:** The requests are handled by the server and the transactions to the FeatureCloud Blockchain are submitted through the gateway. Request results are then forwarded to the user.

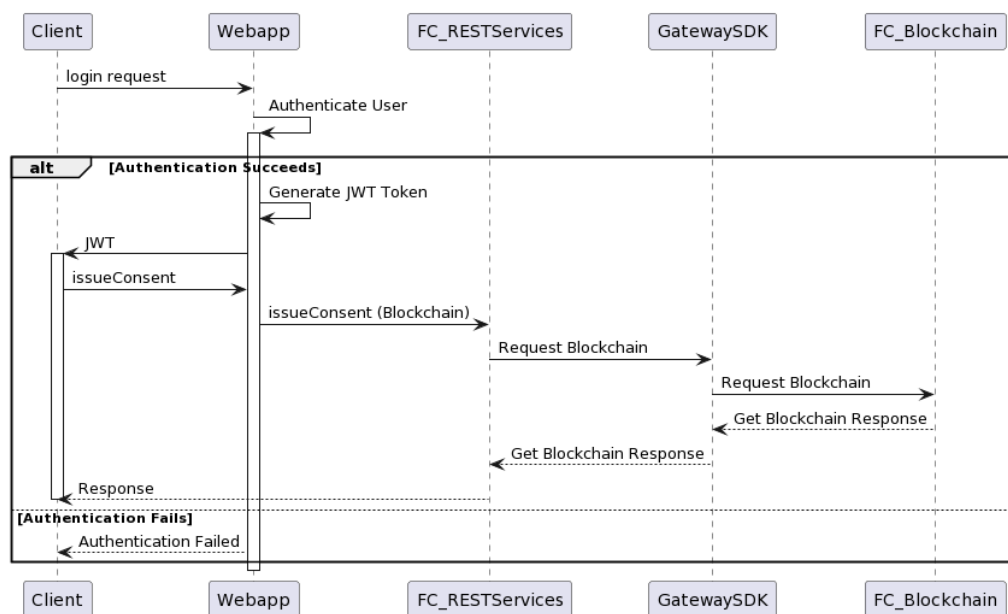


Figure 34. Interaction Flow

Implementation in Spring Boot:

The authentication was implemented as part of the overall Spring Boot project for the web interface and REST services. The user triggers authentication via the web application, initiating a process through the Filter Chain (cf. Figure 35). The Authentication Filter manages the authentication flow, interfacing with the Authentication Manager, which, in turn, delegates tasks to the Authentication

Provider. User details are retrieved from a User Repository, and successful authentication leads to JWT token generation by a JWT Token Provider. The Authentication Filter processes the token, and in successful cases, the Success Handler manages the response, providing the user with a JWT token for local storage or cookie saving. Failure cases are handled by the Failure Handler. Throughout this process, the Security Context Holder maintains the security context, ensuring the integrity of the authentication flow.

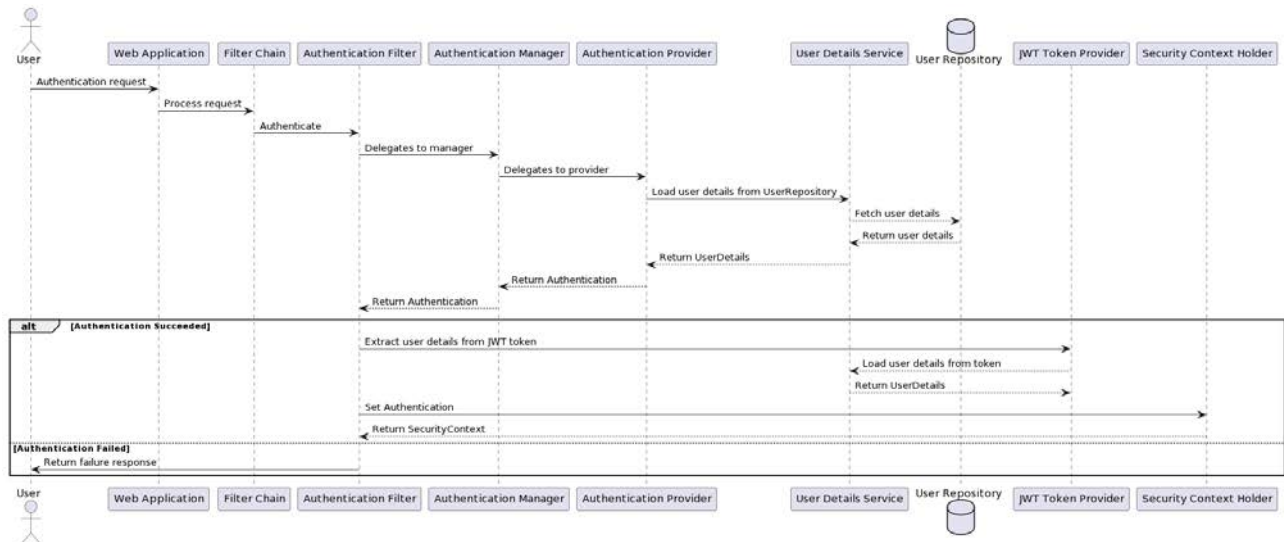


Figure 35. Interaction flow of the authentication components

Figure 36 shows the login interface for the FeatureCloud blockchain platform. To gain access, users are required to input their username and password. Based on their role, they will be redirected to the respective home view (e.g., admin, patient, auditor).

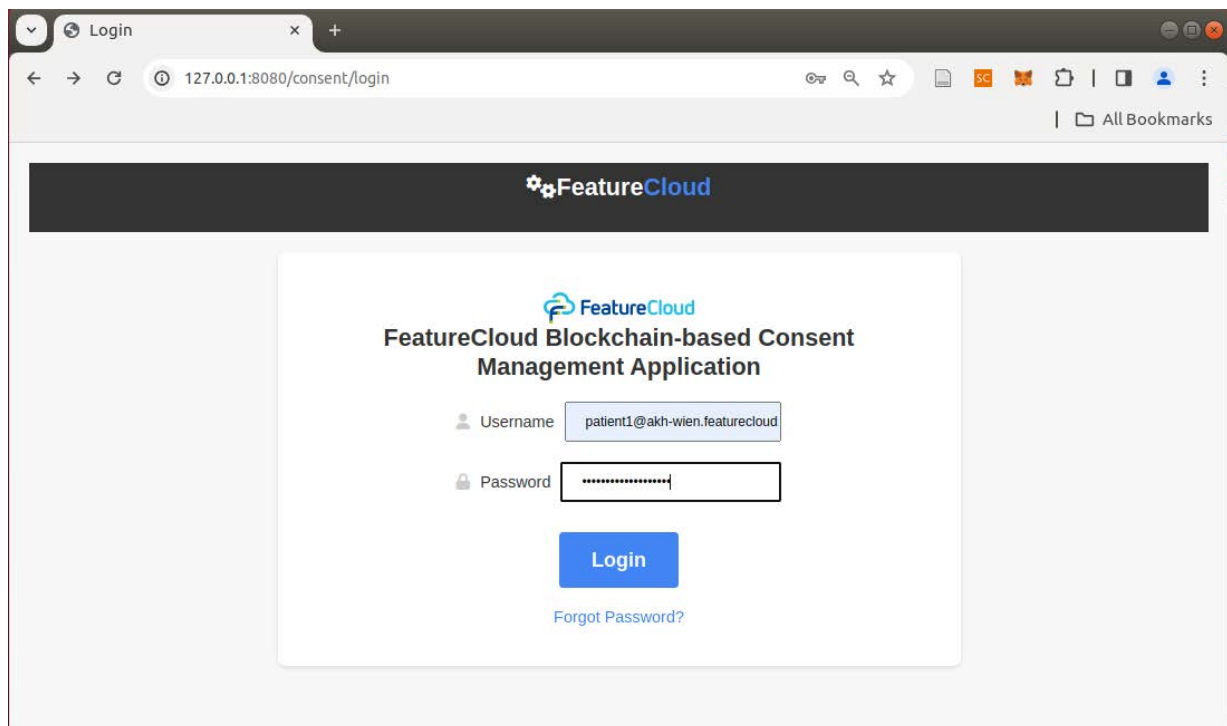


Figure 36. Screenshot of the login interface

Figure 37 shows the correspondent snippet of the AJAX call to the login REST service. This snippet captures the process of setting the JWT in the cookie and subsequently redirecting to the corresponding home view. The appropriate home view is selected and displayed to the user based on the role used in login page (cf. Figure 38). Depending on the role, different functions are available.

```
$.ajax({
  url: '/consent/login',
  method: 'POST',
  contentType: 'application/json',
  data: JSON.stringify({
    username: username,
    password: password
  }),
  success: function(response) {
    console.log(response); var roles = response.roles;
    var token = response.accessToken;
    setCookie("jwt_token", token, 1);
    var homeUrl = '/consent/home';
    homeUrl += '?role=' + encodeURIComponent(roles.join(','));
    $.ajax({
      url: homeUrl,
      method: 'GET',
      headers: {
        'Authorization': 'Bearer ' + token
      },
      success: function(redirectResponse) {
        console.log('Redirect Success:', redirectResponse);
        window.location.href = homeUrl
      },
      error: function(redirectError) {
        console.error('Redirect Error:', redirectError);
      }
    });
  }
});
...
```

Figure 37. Ajax code snippet for login redirect

```
@GetMapping("/home")
public String showHomePage(Model model) {
  try {
    // Retrieve the authentication object from the SecurityContextHolder
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    if (authentication.getPrincipal() instanceof UserDetailsImpl) {
      // Get the user's authorities/roles from the UserDetailsImpl object
      UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
      String role = userDetails.getAuthorities().iterator().next().getAuthority();
      // Check if the role is in the roleView enum
      roleView role_view = roleView.valueOf(role.toUpperCase());
      // If the role is found, return the corresponding view
      model.addAttribute("viewName", role_view.getViewName());
      return role_view.getViewName();
    }
  } catch (IllegalArgumentException e) {
    return "redirect:/errorPage";
  }
  return "redirect:/errorPage";
}
```

Figure 38. Java code snippet for Role-based home view redirect

6.7.2 Home view and consent management Features

Figure 39 provides a snapshot of the admin role's home screen. Within this interface, administrators have the capability to register patients, access information related to issued consents or consent states, and execute actions such as issuing and updating consents on behalf of the patients (given they have the right permissions).

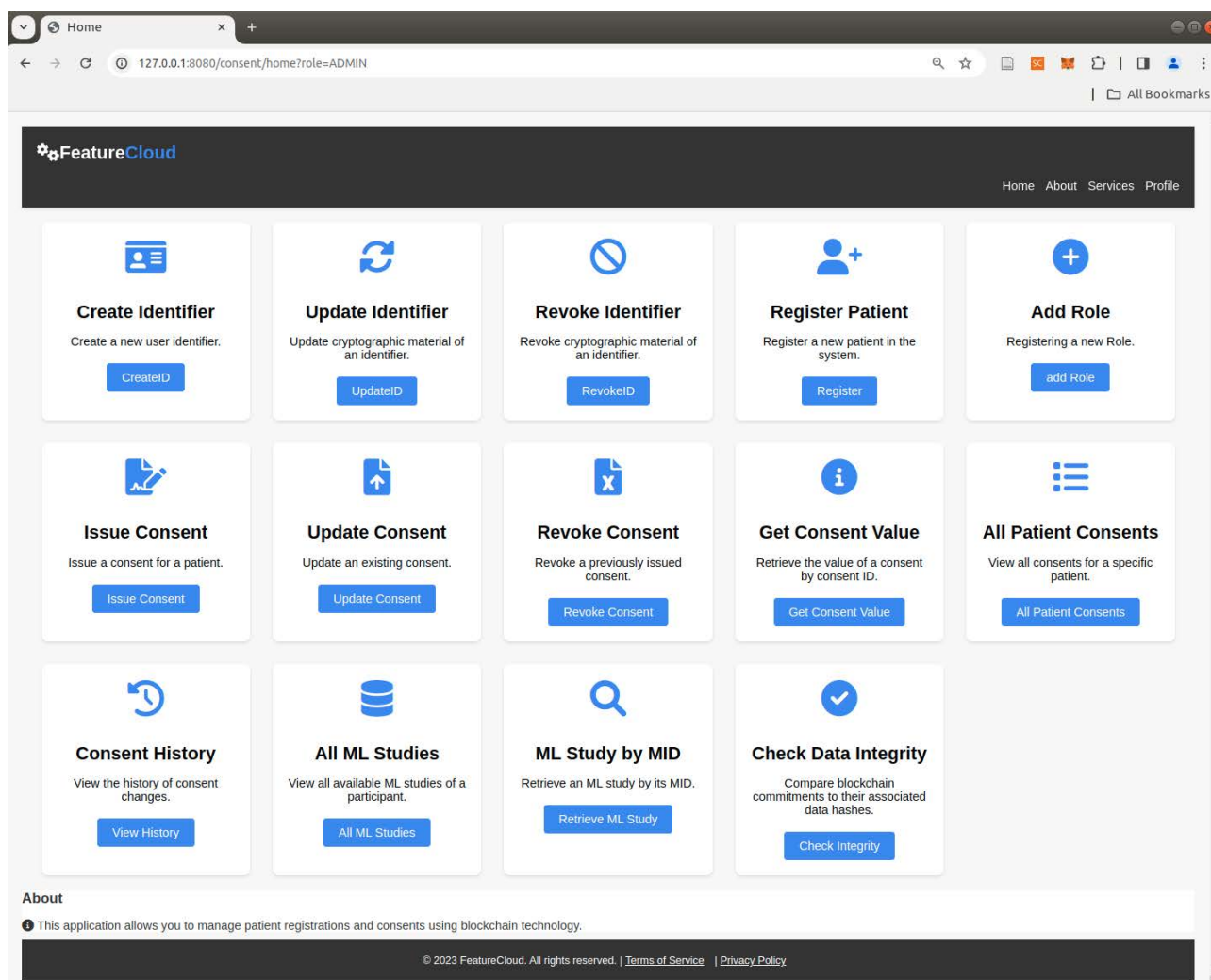


Figure 39. Home View for an Admin role

Figure 40 shows the home screen of the auditor role. Auditors can retrieve committed consents and ML studies, providing them with a comprehensive overview of the blockchain's recorded data. A pivotal feature of the auditor interface is the capability to perform integrity checks. This involves a comparison of on-chain hashes with the corresponding off-chain data hashes, ensuring the consistency and reliability of the stored information. Moreover, auditors possess the authority to execute updates and revocations concerning the cryptographic material linked to their identifiers. This includes operations such as updating the hash of their public key.

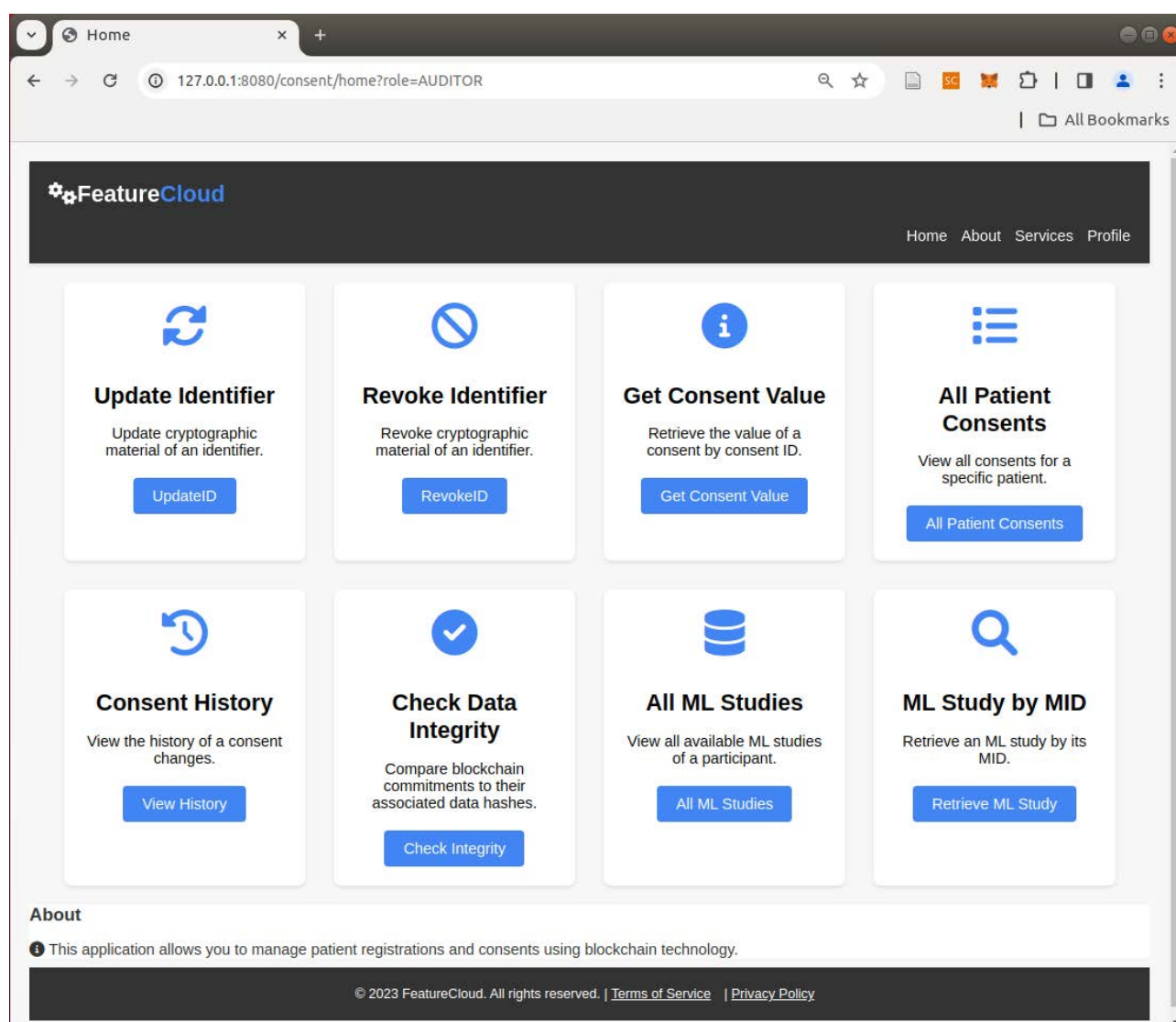


Figure 40. Home View for an Auditor role

Figure 41 presents the enhanced home screen for the patient role. In this interface, patients gain increased control and functionality for managing their consents and personal information. Users can issue, update, and revoke consents, in addition to retrieving detailed information about their committed data, such as the history of consent changes. Alongside the ability to manage their identifiers (e.g., `p0742340920` and associated cryptographic keys; see deliverable D6.4) by updating or revoking them, patients can now easily grant or revoke permissions for third parties to handle their consents on their behalf, as elaborated in Section 6.4.

Figure 42 showcases a patient registration form, illustrating the creation of a new patient object on-chain linked to the patient identifier `p0002`. The figure confirms the successful submission of the transaction. At this stage, the list of consents linked to the patient object is empty.

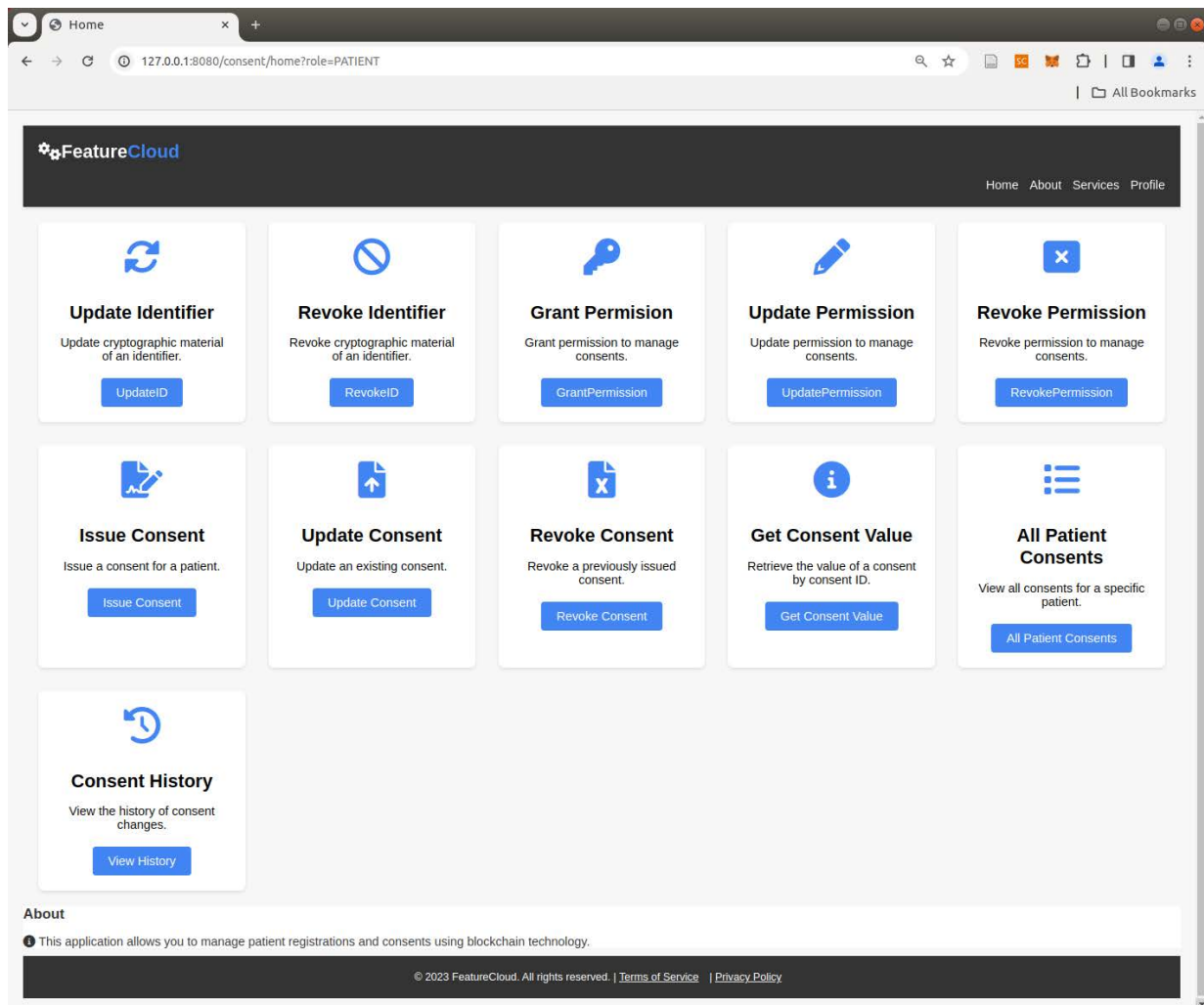


Figure 41. Home View for a Patient role

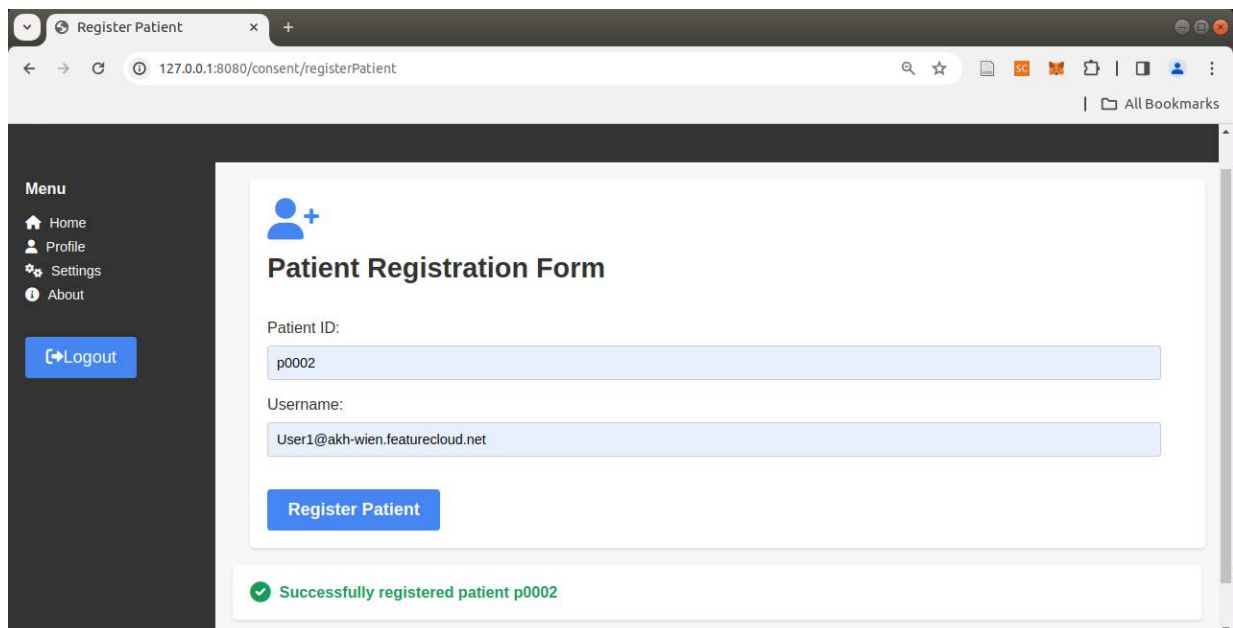
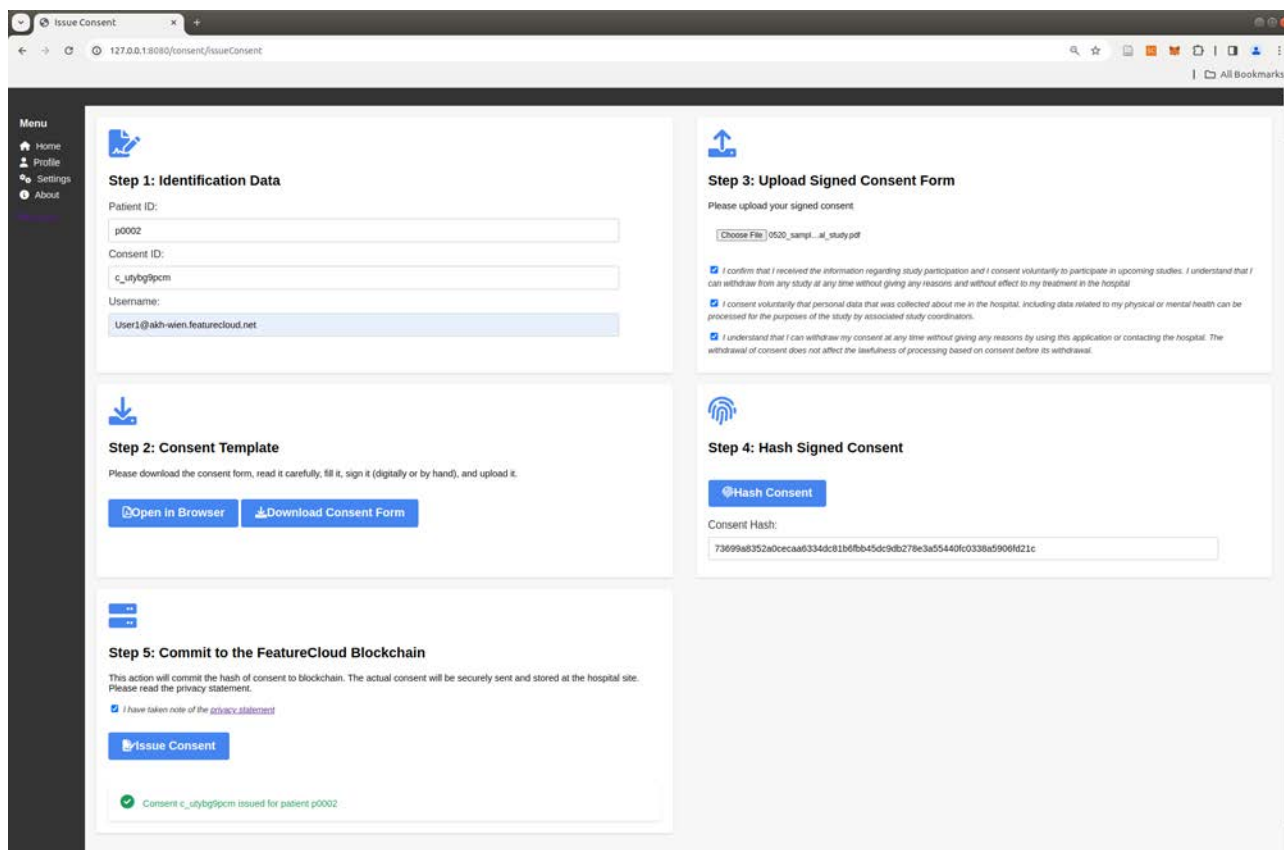


Figure 42. Admin Form for Patient Registration

Figure 43³ shows the user interface designed for consent issuance. Notably, certain fields are conveniently prefilled based on the user's login information, streamlining the process. The system also automatically generates a unique consent ID to ensure traceability. To initiate the consent process, patients download a preconfigured consent template containing all legally mandated details, such as the scope and purpose. Subsequently, patients fill in and sign the document, either manually or electronically, before reuploading it to the system. Upon agreeing to the terms, an automatic generation of the consent hash takes place, and users can then submit the consent. The consent document is stored locally, while the corresponding hash is securely recorded on the blockchain, ensuring immutability and transparency.



The screenshot displays the 'Issue Consent' web interface. It features a sidebar menu with links to Home, Profile, Settings, and About. The main content area is divided into five steps:

- Step 1: Identification Data**: Fields for Patient ID (p0002), Consent ID (c_u7ybg9pcm), and Username (User1@akh-wien.featurecloud.net).
- Step 2: Consent Template**: Instructions to download, fill, sign, and upload the template. Buttons for 'Open in Browser' and 'Download Consent Form' are present.
- Step 3: Upload Signed Consent Form**: A file upload section with a 'Choose File' button and a list of consent terms with checkboxes.
- Step 4: Hash Signed Consent**: A 'Hash Consent' button and a text field displaying the generated 'Consent Hash' (73699a8352a0ceca8334dc81b8fb45dc9db278e3a55440fc0338a5906f421c).
- Step 5: Commit to the FeatureCloud Blockchain**: A confirmation message and an 'Issue Consent' button. A green checkmark indicates the consent is issued for patient p0002.

Figure 43. Issue Consent Interface

Figure 44 shows the JavaScript function that hashes consent documents using SHA-256. This function automates the hashing and transaction submission process, ensuring the integrity and security of consent-related information (e.g., to avoid errors in copying the hash). Notably, the function integrates with the user interface, automatically populating the corresponding field with the generated hash.

³ The user interface screenshots presented in this section were captured at **various instances** using **diverse simulation data**. As a result, the utilization of identifiers across the screenshots may occasionally appear **inconsistent**. The identifiers shown are solely for the purpose of showcasing different scenarios.

```

// Function to hash the content of a consent file using SHA-256
function hashFile(file) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader()
    reader.onload = function (event) {
      const fileContent = event.target.result
      const hash = sha256.create().update(fileContent).hex()
      resolve(hash)
    };
    reader.onerror = function (event) {
      reject(event.target.error)
    };
    reader.readAsBinaryString(file)
  });
}

// Function to hash the uploaded consent document
function hashConsentDocument() {
  const consentHashField = document.getElementById('consentHash')
  const uploadedFile = document.getElementById('uploadConsentFile').files[0]
  hashFile(uploadedFile).then(function (hashedResult) {
    consentHashField.value = hashedResult
  });
}

```

Figure 44. Javascript function for hashing the consent file

Figure 45 shows the AJAX code snippet associated with the "Issue Consent" button, which invokes the corresponding REST service. This code snippet serves as the backend logic that orchestrates the communication between the user interface and the FeatureCloud blockchain platform.

```

// Form submission script
$(document).ready(function () {
  $('#issueConsentForm').submit(function (event) {
    event.preventDefault();
    var patientId = $('#patientId').val();
    var consentId = $('#consentId').val();
    var consentHash = $('#consentHash').val();
    var username = $('#username').val();
    var requestData = {
      patientId: patientId,
      consentId: consentId,
      consentHash: consentHash,
      username: username,
    };
    $.ajax({
      url: '/consent/issueConsent',
      type: 'POST',
      contentType: 'application/json',
      data: JSON.stringify(requestData),
      success: function (response) {
        displayResult('success', response)
      },
      error: function (xhr, status, error) {
        displayResult('error', xhr.responseText)
      },
    });
  });
});

```

Figure 45. Ajax call to the REST service for issuing the consent

Figures 46 and 47 depict the user interface for updating consents. The patient needs to upload the signed document for the new consent and check all relevant boxes. The interface resembles the issue consent interface, with the distinction that the consent identifier is already issued. If the ID does not exist on-chain or belongs to a different user without permission, the transaction will fail (Figure 47).

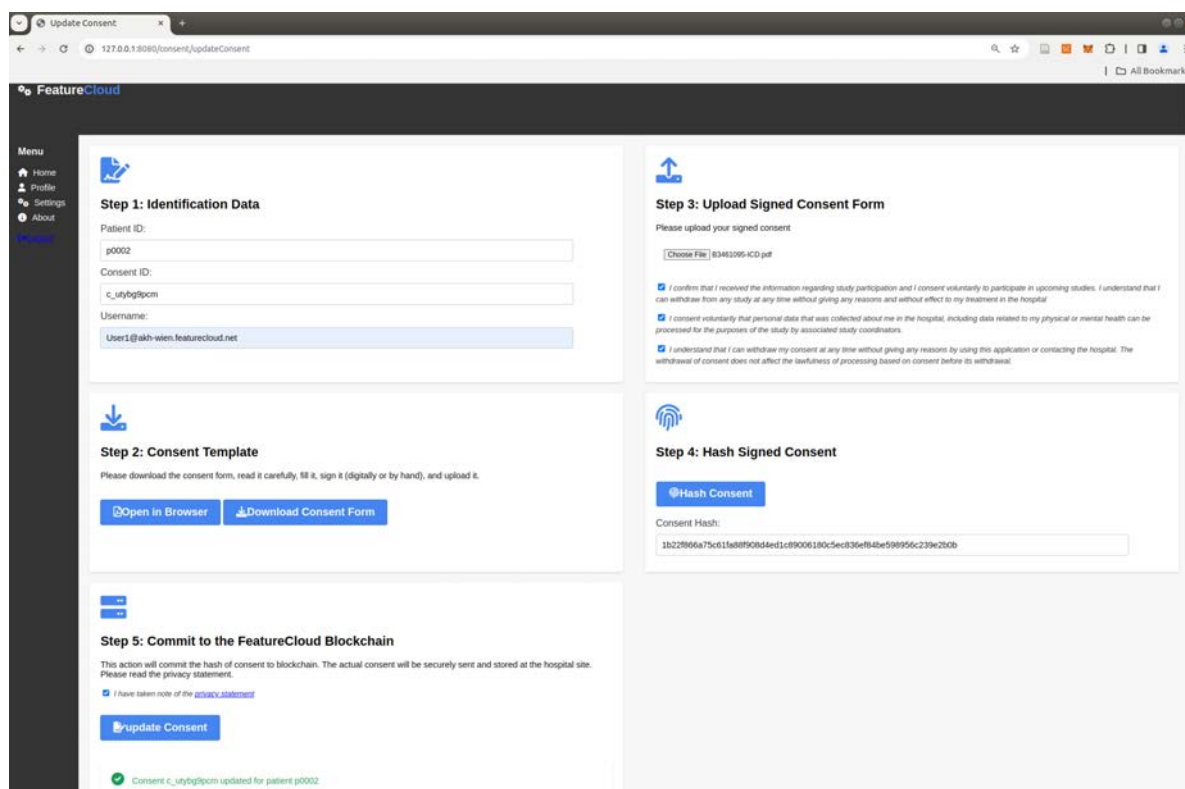
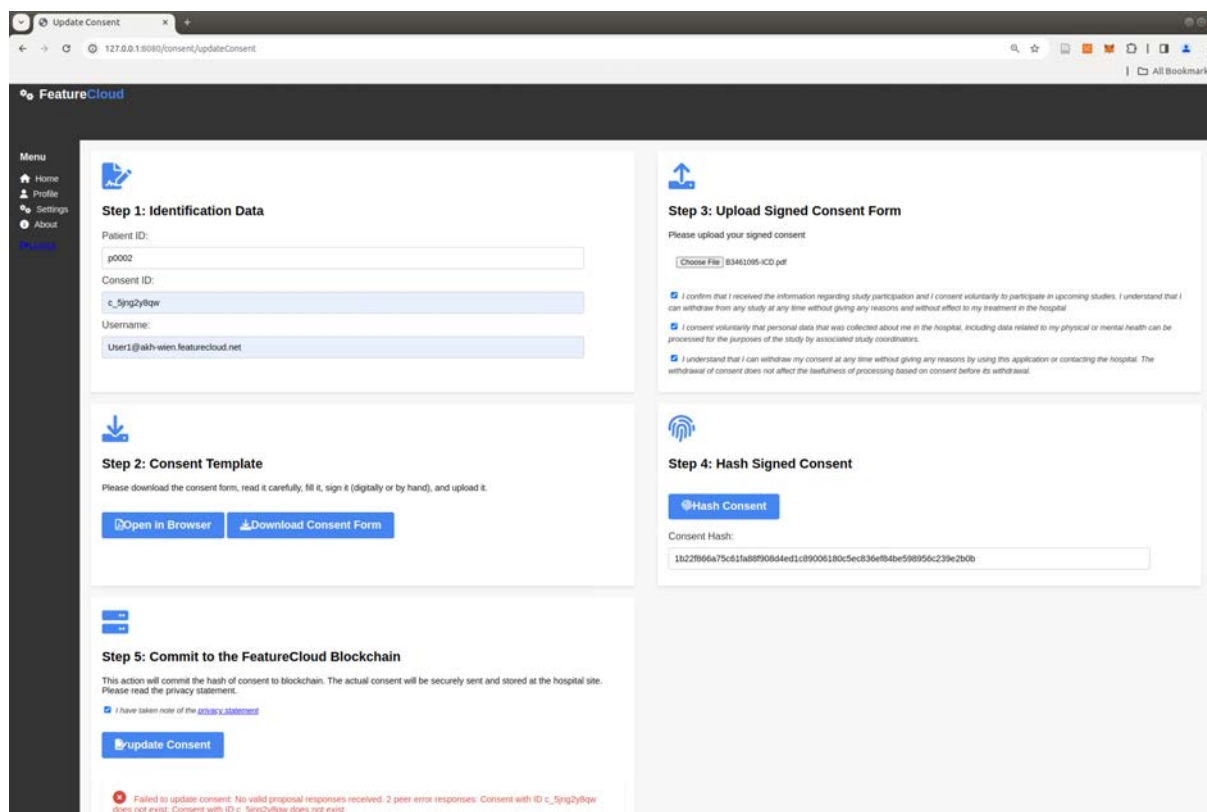
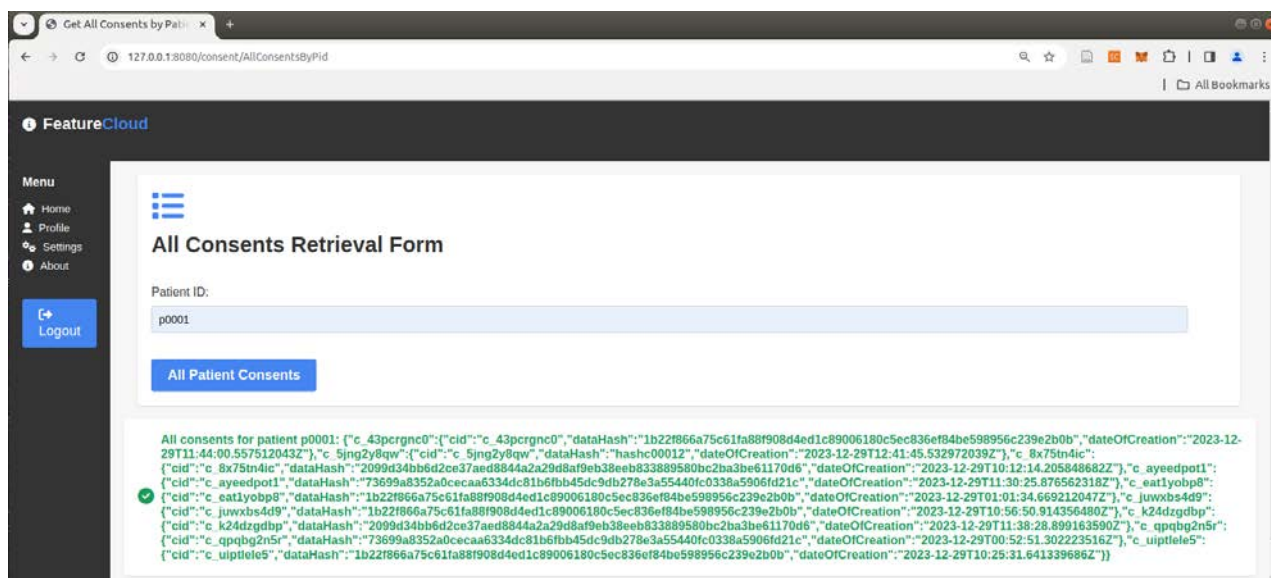


Figure 46. Update Consent Form (transaction success)


Figure 47. Update Consent Form (transaction failure: consent **c_5jng2y8qw** does not belong to patient **p0002**)

Figures 48 to 50 illustrate various operations performed by an auditor, which can also be invoked by administrators or patients with the appropriate permissions. A patient may have granted multiple consents with different scopes, and updates may have been applied to each of these consents. The operation in Figure 48 enables the retrieval of the latest version of all consents for a specific patient `p0001`. As shown in Figure 49, the form allows for retrieving the historical versions of a specific consent `c_5jng2y8qw`, while the form in Figure 50 facilitates retrieving the current value of the same consent ID `c_5jng2y8qw`.



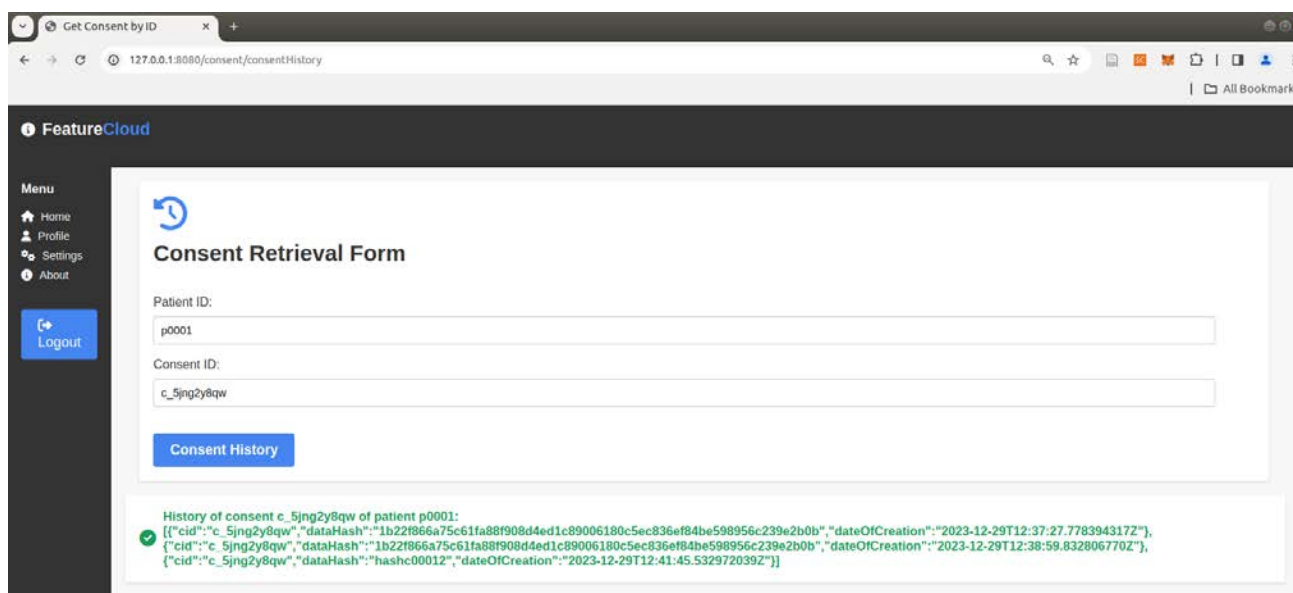
All Consents Retrieval Form

Patient ID:
p0001

All Patient Consents

All consents for patient p0001: [{"cid":"c_43pcrgnc0","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T11:44:00.557512043Z"}, {"cid":"c_5jng2y8qw","dataHash":"hashc00012","dateOfCreation":"2023-12-29T12:41:45.532972039Z"}, {"cid":"c_8x75tn4ic","dataHash":"2099d34bb6d2ce37aed8844a2a29d8af9eb38eeb833889580bc2ba3be61170d6","dateOfCreation":"2023-12-29T10:12:14.205848682Z"}, {"cid":"c_ayeedpot1","dataHash":"73699a8352a0cecaa6334dc81b6fbb45dc9db278e3a55440fc0338a5906fd21c","dateOfCreation":"2023-12-29T11:30:25.876562318Z"}, {"cid":"c_eatlyobp8","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T01:01:34.669212047Z"}, {"cid":"c_juwxbs4d9","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T10:56:50.914356480Z"}, {"cid":"c_k24dzgdbp","dataHash":"2099d34bb6d2ce37aed8844a2a29d8af9eb38eeb833889580bc2ba3be61170d6","dateOfCreation":"2023-12-29T11:38:28.899163590Z"}, {"cid":"c_qpqbqzn5r","dataHash":"73699a8352a0cecaa6334dc81b6fbb45dc9db278e3a55440fc0338a5906fd21c","dateOfCreation":"2023-12-29T00:52:51.302223516Z"}, {"cid":"c_uipptele5","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T10:25:31.641339686Z"}]

Figure 48. Auditor Role retrieving all consents of patient `p0001`



Consent Retrieval Form

Patient ID:
p0001

Consent ID:
c_5jng2y8qw

Consent History

History of consent c_5jng2y8qw of patient p0001: [{"cid":"c_5jng2y8qw","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T12:37:27.778394317Z"}, {"cid":"c_5jng2y8qw","dataHash":"1b22f866a75c61fa88f908d4ed1c89006180c5ec836ef84be598956c239e2b0b","dateOfCreation":"2023-12-29T12:38:59.832806770Z"}, {"cid":"c_5jng2y8qw","dataHash":"hashc00012","dateOfCreation":"2023-12-29T12:41:45.532972039Z"}]

Figure 49. Auditor retrieving the history of consent `c_5jng2y8qw` of patient `p0001`

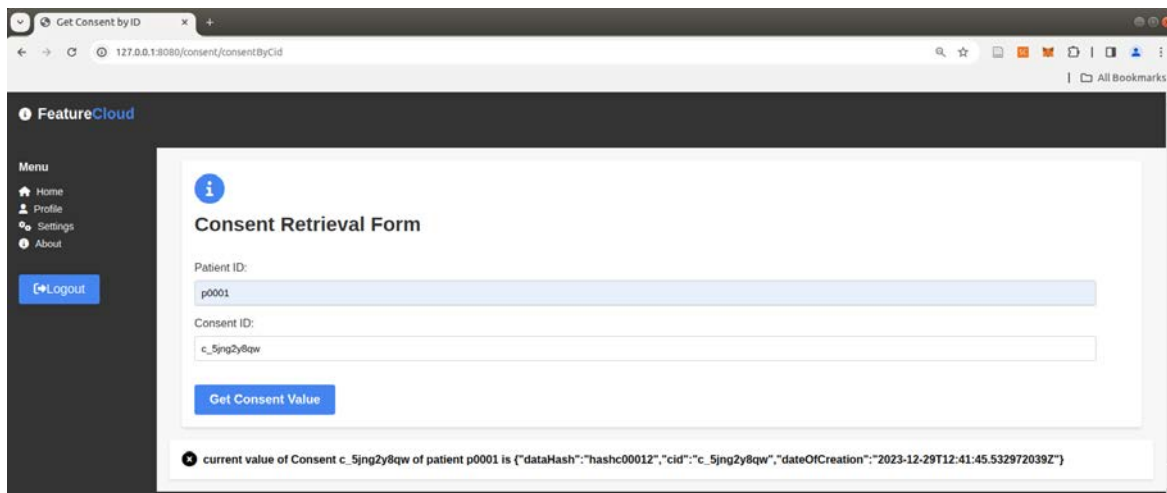


Figure 50. Auditor retrieving current value of consent `c_5jng2y8qw` of patient `p0001`

Figure 51 presents a snapshot of the auditor interface designed for validating the integrity of consents used in Machine Learning (ML) studies. The process begins with the auditor uploading the consent document, triggering an automated hashing mechanism to generate the corresponding hash. Subsequently, the auditor enters information, including the Consent ID and the date of the study. Concurrently, the system interacts with the blockchain, retrieving the hash of the consent value at the specified time. The retrieved hash, representing the state of the consent precisely during the study period, is then displayed for the auditor. Clicking the "Check" button initiates a comparison between the two values: the hash generated from the uploaded consent document and the hash retrieved from the blockchain. The outcome of this comparison verifies whether the most recent version of the consent was indeed utilized in the study. Auditors can also utilize the consent history form and manually cross-reference corresponding dates.

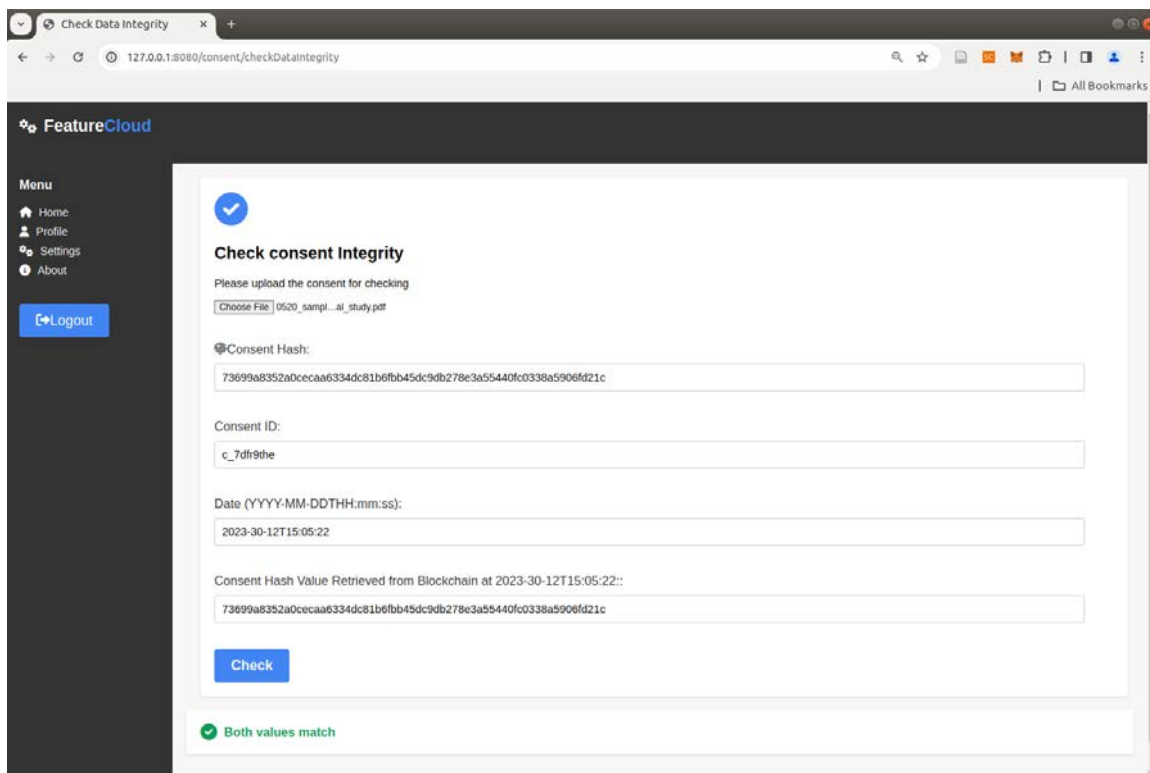
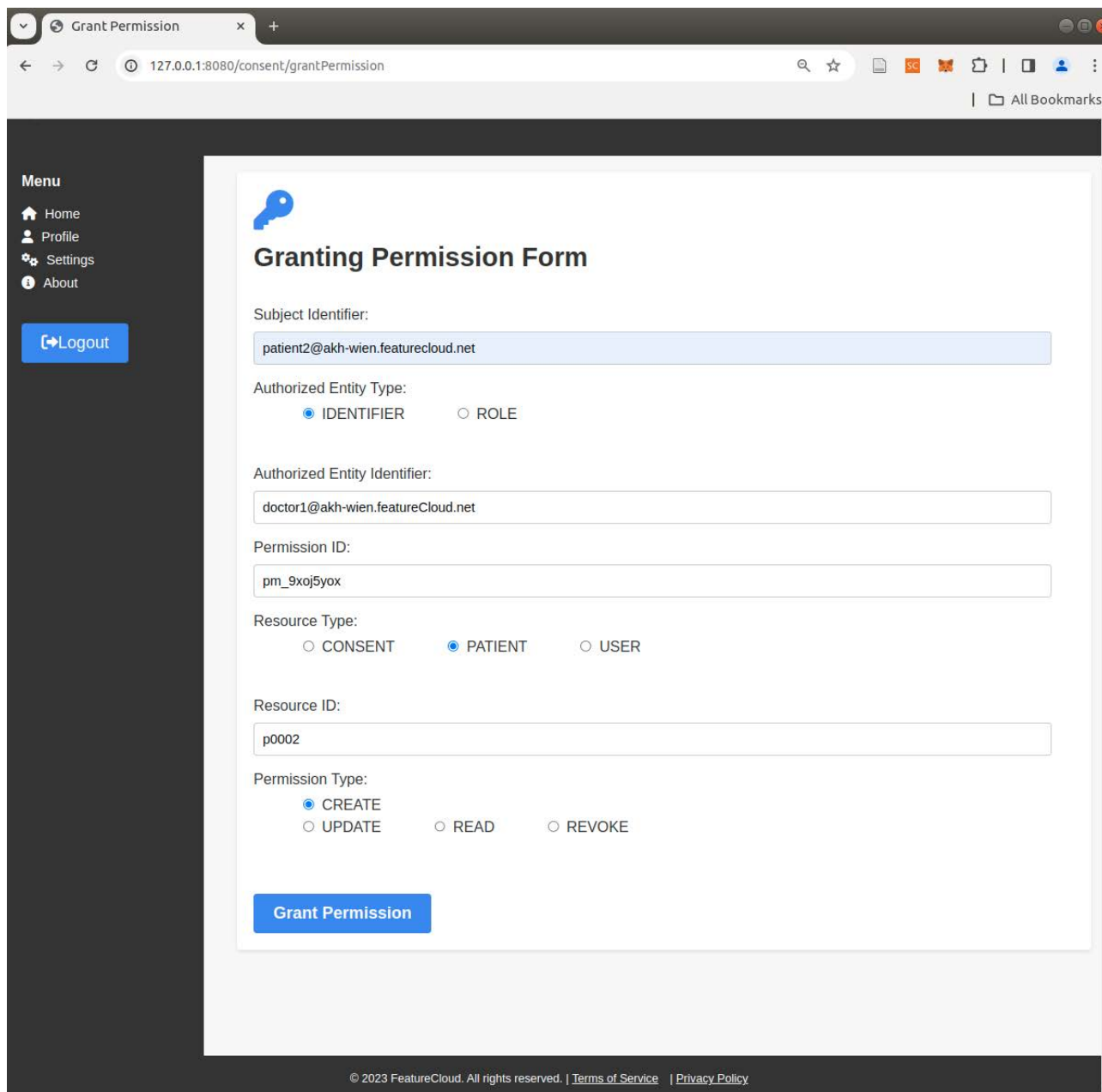


Figure 51. Consent Integrity Check by Auditor

In Figures 52 and 53, the granting permission form is illustrated, providing a visual representation of the interaction between patient `patient2@akh-wien.featurecloud.net` and the permission management features of the consent management smart contract as described in earlier sections.



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8080/consent/grantPermission`. The page has a dark sidebar menu on the left with links for Home, Profile, Settings, and About, and a Logout button. The main content area is titled 'Granting Permission Form' and contains the following fields and options:

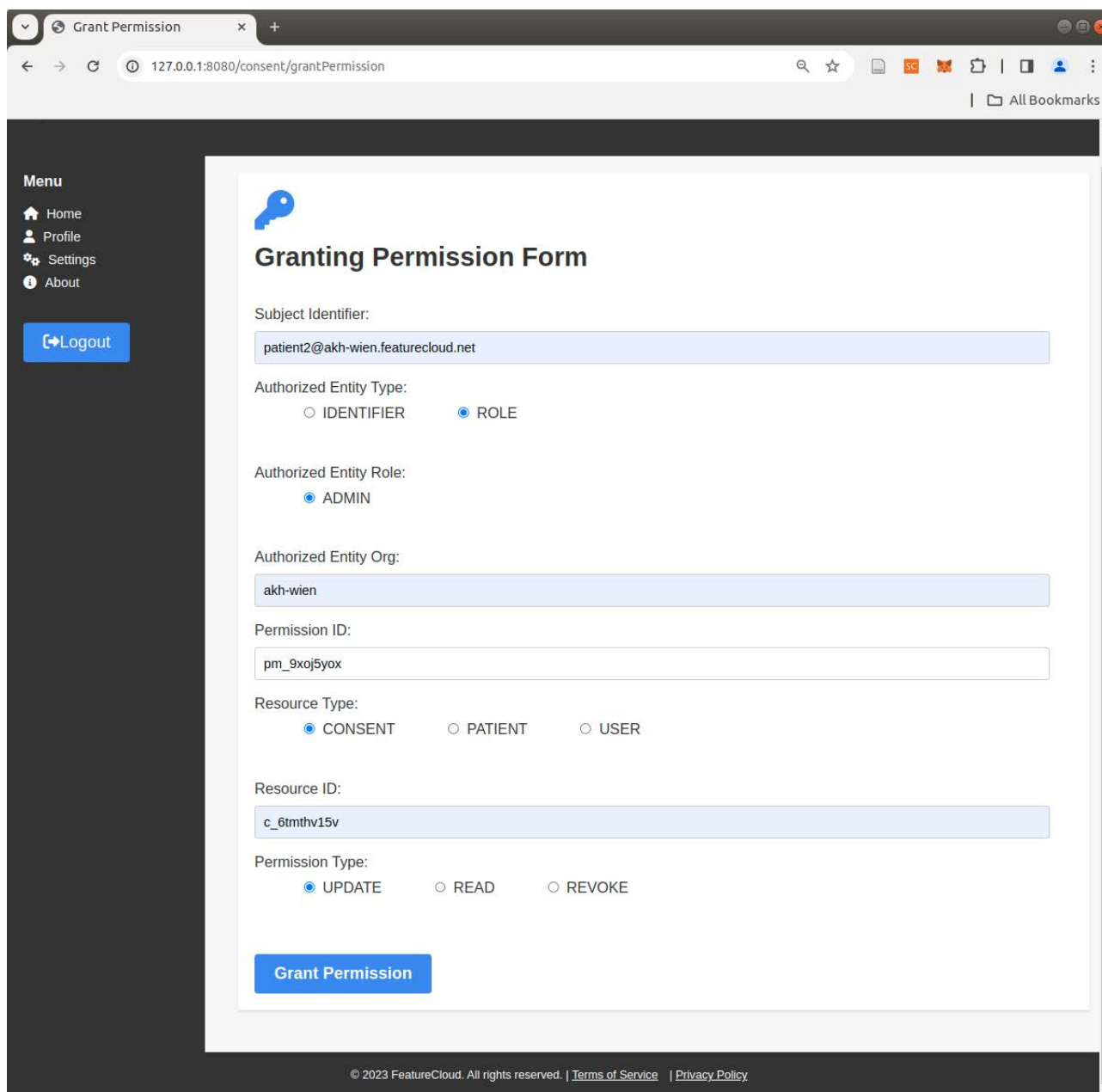
- Subject Identifier:** `patient2@akh-wien.featurecloud.net`
- Authorized Entity Type:** ☒ IDENTIFIER ☐ ROLE
- Authorized Entity Identifier:** `doctor1@akh-wien.featurecloud.net`
- Permission ID:** `pm_9xoj5yox`
- Resource Type:** ☐ CONSENT ☒ PATIENT ☐ USER
- Resource ID:** `p0002`
- Permission Type:** ☒ CREATE ☐ UPDATE ☐ READ ☐ REVOKE

A blue 'Grant Permission' button is located at the bottom of the form. The footer of the page reads: © 2023 FeatureCloud. All rights reserved. | [Terms of Service](#) | [Privacy Policy](#)

Figure 52. Patient form for granting **CREATE** Permission on a patient object `p0002`

In Figure 52, a patient with identifier `patient2@akh-wien.featurecloud.net` authorized user `doctor2@akh-wien.featurecloud.net`, possibly identified as a doctor or a trusted individual, to manage consents on their behalf. This expansive permission encompasses the ability to create new consents, as well as update or revoke existing ones, illustrating a use case tailored for patients with limited technological proficiency.

Figure 53 depicts a scenario where patient `patient2@akh-wien.featurecloud.net` narrows the authorization, granting permission to all administrators affiliated with `akh-wien.featurecloud.net`, to oversee a specific consent with the identifier `c_6tmthv15v` is restricted solely to update operations on the specified consent.



Grant Permission

127.0.0.1:8080/consent/grantPermission

Menu

- Home
- Profile
- Settings
- About

Logout

Granting Permission Form

Subject Identifier:

patient2@akh-wien.featurecloud.net

Authorized Entity Type:

☐ IDENTIFIER ☒ ROLE

Authorized Entity Role:

☒ ADMIN

Authorized Entity Org:

akh-wien

Permission ID:

pm_9xoj5yox

Resource Type:

☒ CONSENT ☐ PATIENT ☐ USER

Resource ID:

c_6tmthv15v

Permission Type:

☒ UPDATE ☐ READ ☐ REVOKE

Grant Permission

© 2023 FeatureCloud. All rights reserved. | [Terms of Service](#) | [Privacy Policy](#)

Figure 53. Patient form for granting UPDATE Permission on a consent

The screenshot of Figure 54 illustrates the user interface designed for updating the identifier of the user with the email address `patient2@akh-wien.featurecloud.net`. The purpose of this operation is to replace the old public key linked to the user's identifier with a new one. To ensure the legitimacy of this update and prove ownership of the new public key, a signature is generated using the SHA256 algorithm with the ECDSA (Elliptic Curve Digital Signature Algorithm) over both the hash of the old key and the new public key, utilizing the corresponding new private key. This

process is implemented to prevent the association of an unauthorized key with the user's identifier, thereby maintaining control over the security of the user's information.

The signing mechanism employed for updating user identifiers can be extended to sign all other user operations (e.g., consents), allowing them to retain control over the data they commit to the blockchain, while specifically entrusting the interaction with the blockchain, utilizing the users' X.509 certificates for authentication within the blockchain network, to the respective hospital. This delegation ensures that the hospital is solely involved in blockchain-related interactions, specifically relaying user transactions. This ensures the integrity of the data without granting unrestricted control. Note, that Hyperledger fabric also supports offline signing of transactions where users have the possibility to maintain their private keys external to the client application ⁴.

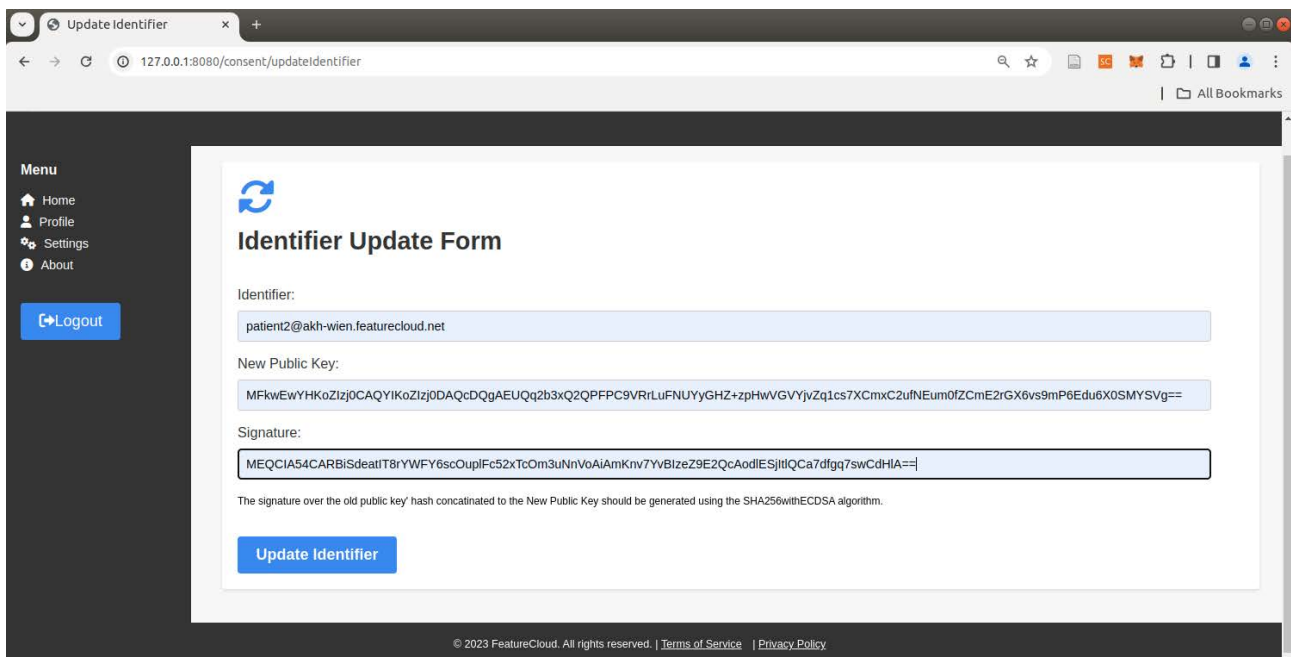


Figure 54. Update form for the user Identifier

6.8 FeatureCloud Blockchain operator

To facilitate metrics and monitoring capabilities within the FeatureCloud blockchain network, we utilize existing tools such as i) Grafana for visualizing and analyzing metrics [7], and ii) Prometheus for monitoring and alerting [8].

It's important to note that the accompanying screenshot in Figure 55 is provided solely for illustrative purposes. It does not reflect the current operational state of the network and was captured during the testing phase of the prototype."

⁴ <https://hyperledger.github.io/fabric-sdk-node/release-1.4/tutorial-sign-transaction-offline.html>



Figure 55. Screenshot of Grafana operator

6.9 Technology Stack used for FeatureCloud blockchain

The technology stack for the FeatureCloud blockchain prototype covers a range of tools for frontend and backend development, deployment through containerization, secure data handling with a blockchain framework, and diverse databases for various storage needs (ledger, identity management). Additional tools such as Postman, Grafana, and Prometheus were employed for testing, monitoring, and analytics within the FeatureCloud blockchain platform.

Table 2. Technology stack used for FeatureCloud Blockchain

Category	Technology	Description
Programming Languages	Java 17, JavaScript, HTML, Bash Script	Languages used for backend, frontend, markup, and scripting.
Web Technologies	JWT, Web Security, bcrypt	Secure authentication, web security, and password hashing.
Containerization and Orchestration	Docker 24.0.2	Platform for containerization and application deployment.
Blockchain Framework	Hyperledger Fabric 2.5	Framework for building distributed ledgers.
Backend Framework	Spring Boot 4.18.1	Framework for building Java-based web applications.
Integrated Development Environment	Eclipse 4.29.0	IDE for software development.
Database Technologies	MySQL 8.0.33, CouchDb3.3.2	Relational and NoSQL databases for data storage.
Blockchain Gateway	Gateway SDK	Software Development Kit for interacting with blockchain.
API Development and Testing	Postman	Tool for API testing.
Monitoring and Analytics	Grafana 2.3.4, Prometheus V2.32.1	Open-source analytics and monitoring platform.
Configuration and Data Formats	JSON, YAML	Formats for configuration and data serialization.

Category	Technology	Description
Communication Protocols	gRPC, TLS, x509	Protocols for secure communication.
Shell Scripting	Bash	Scripting language for command-line tasks.
JavaScript Library	AJAX	Library for asynchronous web requests.
Operating System	Ubuntu 18.04 and 22.04	Operating system used for development and deployment.

7 Deviations

In compliance with rigorous privacy and data protection rules (e.g., GDPR), the FeatureCloud blockchain platform refrains from storing actual consents directly on the blockchain, as outlined in deliverables D6.5 and D6.2. Instead, a decision has been made to commit all consent management operations on-chain. While sensitive consents are not stored directly on the blockchain, this method ensures that crucial consent-related actions are executed and documented on the immutable ledger. Through this approach, FeatureCloud enhances its audit processes, establishing a transparent and secure framework for monitoring and validating consent-related activities. This design aligns with the privacy considerations and the imperative for a verifiable and accountable consent management system.

8 Conclusion

Deliverable D6.7 builds upon the foundational work of D6.4, and introduces substantial enhancements to the FeatureCloud blockchain solution. This iteration refines the prototype for securing federated machine learning processes, with an emphasis on a user-friendly interface. D6.7 presents the steps of designing, developing, and deploying the FeatureCloud blockchain solution using Hyperledger Fabric. The refined prototype not only enhances existing features but also introduces new capabilities, and more complex access controls. Smart contract functions for patient consent management are made accessible through both a web application and REST services, prioritizing a user-centric approach.

9 References

- [1] "<https://www.hyperledger.org/projects/fabric>."
- [2] "<https://gdpr-info.eu/>."
- [3] "https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html#raft."
- [4] W. Fdhila, N. Stifter, and A. Judmayer, "Challenges and Opportunities of Blockchain for Auditable Processes in the Healthcare Sector," in *Business Process Management: Blockchain, Robotic Process Automation, and Central and Eastern Europe Forum*, 2022, pp. 68–83. doi: 10.1007/978-3-031-16168-1_5.
- [5] J. Matschinske *et al.*, "The FeatureCloud Platform for Federated Learning in Biomedicine: Unified Approach," *J. Med. Internet Res.*, vol. 25, p. e42621, Jul. 2023, doi: 10.2196/42621.
- [6] "<https://hyperledger.github.io/fabric-gateway/migration#event-reconnect>."
- [7] "<https://grafana.com/>."
- [8] "<https://prometheus.io/>."

10 Other supporting figures

The following figures are additional snippets of methods within the implemented chaincodes.

```
@Transaction(intent = Transaction.TYPE.SUBMIT)
public Patient issueConsent(Context ctx, String pid, String cid, String dataHash) throws Exception {
    LOG.info("Issue a Consent: " + ctx);
    // input parameters validity check
    checkNotNullOrEmpty(pid, " Patient ID ");
    checkNotNullOrEmpty(cid, " Consent ID ");
    checkNotNullOrEmpty(dataHash, "Data Hash ");
    //checks patient and consent existence and permissions
    ChaincodeStub stub = ctx.getStub();
    if (!patientExists(ctx, pid)) {
        LOG.info("patient " + pid + "does not exist !");
        throw new ChaincodeException("patient " + pid + "does not exist !");
    }
    if (!canIssueConsent(ctx, pid)) {
        LOG.info("Access to patient information denied.");
        throw new ChaincodeException("Only the patient or authorized users can issue a consent for patient " + pid);
    }
    if (consentExists(ctx, pid, cid)) {
        throw new ChaincodeException("Consent with ID " + cid + " already exists");
    }
    // create an instance of the consent
    Patient patient = getPatient(ctx, pid);
    String formattedDate = ctx.getStub().getTxTimestamp().atOffset(ZoneOffset.UTC).toString();
    Consent consent = new Consent(cid, formattedDate, dataHash);
    LOG.info("creating an instance of consent: " + consent);
    LOG.info("Issuing consent: " + cid + " to patient: " + patient);
    patient.getConsents().put(cid, consent);
    String patientSON = genson.serialize(patient);
    LOG.info("Adding patient's consent to the Ledger: " + patientSON);
    ctx.getStub().putStringState(pid, patientSON);
    LOG.info("Consent successfully issued: " + patientSON);
    createKeyPairMap(stub, cid, getPatientOwner(stub, pid), CONSENT_TO_OWNER_PREFIX);
    return patient;
}
```

Figure 56. Issue Consent Transaction Method

```
private boolean canIssueConsent(Context ctx, String pid) throws Exception {
    ChaincodeStub stub = ctx.getStub();
    PermissionManager permissionManager = getPermissionManager(ctx, getValueOfKey(stub, PATIENT_TO_OWNER_PREFIX, pid));
    LOG.info("Retrieving permissionManager: " + permissionManager);
    String callerId = getTxCallerIdentifier(ctx);
    LOG.info("Caller Identifier: " + callerId);
    String callerOrg = ctx.getClientIdentity().getAttributeValue("org");
    LOG.info("Caller org: " + callerOrg);
    String callerRole = ctx.getClientIdentity().getAttributeValue("role");
    LOG.info("Caller role: " + callerRole);
    return isPatientOwner(stub, callerId, pid) ||
        permissionManager.hasPermission(callerId, callerRole, callerOrg, pid, PermissionType.CREATE);
}
```

Figure 57. Issue Consent Transaction Method

```

@Transaction(intent = Transaction.TYPE.SUBMIT)
public Identity updateIdentity(Context ctx, String identifier, String publicKey, String signature) throws Exception {
    Identity identity = getIdentity(ctx, identifier);
    if(!identityExists(ctx, identifier)){
        throw new ChaincodeException("Identity with ID " + identifier + " does not exist in the ledger");
    }
    if(identity.isRevoked()) {
        throw new ChaincodeException("Update operation denied! Identity with ID " + identifier + " was revoked");
    }
    String callerRole = ctx.getClientIdentity().getAttributeValue("role");
    String callerOrg = ctx.getClientIdentity().getAttributeValue("org");
    String old_pukHash = identity.getPukHash();
    LOG.info("old public Key Hash of " + identifier + " = "+ old_pukHash);
    LOG.info("New Public Key Value " + identifier + " = "+ publicKey);
    LOG.info("submitted signature " + signature );
    String new_pukHash = hashPublicKey(publicKey);
    String callerPukHash = getTxCallerPublicKeyHash(ctx);
    if(!old_pukHash.equals(callerPukHash) &&
        !getPermissionManager(ctx, identifier).hasPermission(identifier, callerRole, callerOrg, identity.getIdentifer(),
            PermissionType.UPDATE)
    ){
        throw new ChaincodeException("Caller does not have permission to update Identity "+ identifier);
    }
    String data =old_pukHash+publicKey;
    LOG.info("unsigned Data for verification = " +data);
    boolean isSignatureValid = verifySignature(publicKey, data, signature);
    LOG.info("Is Signature valid? " + isSignatureValid);
    if (!isSignatureValid) {
        throw new ChaincodeException("Signature verification failed for identity update");
    }
    ChaincodeStub stub = ctx.getStub();
    LOG.info("Updating Identity: " + identifier);
    identity.updateIdentity(new_pukHash);
    LOG.info("Identity Updated: " + identity);
    // Add the consent to the list of all similar consent in the ledger
    String identityJSON = gson.serialize(identity);
    LOG.info("Storing Identity information in the ledger: " + identityJSON);
    stub.putStringState(identifier, identityJSON);
    LOG.info("Updated Identity successfully stored in the ledger: " + identityJSON);
    createKeyPairMap(stub, new_pukHash, identifier, PUKHASH_TO_IDENTITY_OWNER_PREFIX);
    deleteKeyPairMap(stub, old_pukHash, PUKHASH_TO_IDENTITY_OWNER_PREFIX);
    return identity;
}

```

Figure 58. Issue Consent Transaction Method

```

private boolean verifySignature(final String publicKey, final String data, final String signature) throws Exception {
    try {
        String dataToVerify = data;
        byte[] publicKeyBytes = Base64.getDecoder().decode(publicKey);
        byte[] signatureBytes = Base64.getDecoder().decode(signature);
        KeyFactory keyFactory = KeyFactory.getInstance("EC");
        PublicKey publicKeyObj = KeyFactory.generatePublic(new X509EncodedKeySpec(publicKeyBytes));
        Signature sig = Signature.getInstance("SHA256withECDSA");
        sig.initVerify(publicKeyObj);
        sig.update(dataToVerify.getBytes());
        return sig.verify(signatureBytes);
    } catch (Exception e) {
        throw new Exception(e);
    }
}

```

Figure 59. Issue Consent Transaction Method