Legitify Technical Guide

Document Information

Document Title	**Legitify Technical Guide**
Date	02/05/25
Authors	Christopher Dobey Padraig Mann
Supervisor	Sahraoui Dhelim

Table of Contents

Introduction

Project Overview

Legitify is a blockchain-based credential verification system that allows issuers to issue digital credentials, holders to manage and share them, and verifiers to verify their authenticity instantly. Our platform eliminates credential fraud and streamlines the verification process using secure, tamper-proof blockchain technology.
Today the recruiting process for many jobs has become increasingly digital. This can lead recruiters to have a necessary skepticism as people can often exaggerate their online persona compared to the reality of their real lives. Platforms such as LinkedIn and Indeed have done very little in the way of verifying what is claimed on potential candidates' profiles, Legitify aims to tackle this problem by providing an easy-to-use and secure platform to allow recruiters to verify these claims.
Our platform is aimed at institutions or businesses that would issue certificates, diplomas, or employee references, graduates or job seekers who are looking to showcase their verified credentials, and businesses that are recruiting for positions within their company that may require certain qualifications.

Project Background

Legitify aims to provide a secure and trusted way of distributing credentials among parties who have vested interest in ensuring those credentials are legitimate. The possibility of fraudulent credentials is a real threat and ensuring their legitimacy is paramount in this day and age. When we started developing ideas for this project we identified that there was a market for providing this security from fraudsters. We identified that blockchain technology could be leveraged to create a solution and began researching if there had been any solutions already made available.

By searching online we found that there were a couple of solutions that were similar. Sertifier and Accredible had solutions that created Digital Credentials and then secured their distribution using blockchain technology. However, the entire process of creating and the credential is encapsulated within their website. Our Legitify system differs from them as our solution does not force you to create the credential on our website and allows you to upload any PDF file and issue and verify that. Also our solution allowed the credential to be verified on our website, whereas others allowed the recipient to share the credential externally, which we believed sacrificed the security of the credential and left it at risk of being susceptible to fraud

---

System Design

Design Philosophy

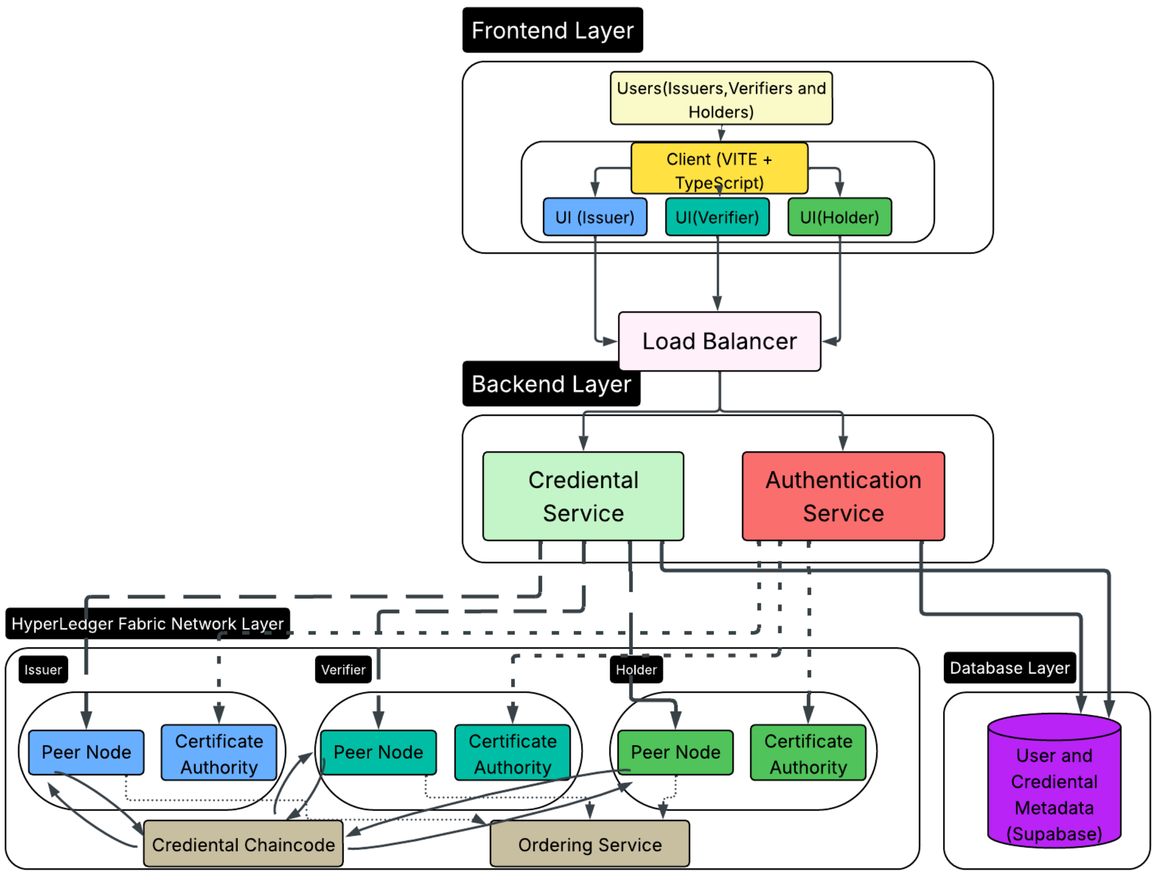
We wanted the system to be able to create trust between different stakeholders, ensuring that our design allowed for this trust at the forefront of our minds throughout the design process. We aimed for every credential action—issuance, revocation, verification—to leave an auditable trail, which created built-in accountability. This built confidence in the system itself while respecting privacy boundaries, we believe this is needed for the wide adoption of digital credentials.

Each layer of our system has its distinct function. The user experience, business logic, data management, and trust verification are all separated into different layers within the system architecture. This separation allows components to evolve independently throughout the design process while maintaining system integrity.

Not all the data was going to be processed the same way. Our approach of having both a Hyperledger Fabric blockchain and a more traditional PostgreSQL database meant that we could store credential verification status on the immutable blockchain while keeping frequently changing metadata and user information in a database. This meant we could have blockchain's trust advantages without its performance limitations.

We wanted our design to be scalable to allow for the organic growth of the application. The system is designed to start with three organization types but can expand to include multiple institutions in each role. New credential types and verifying entities are able to be added without having to restructure the core architecture.

Architecture Overview



Frontend Layer

This is where the User interacts with the system, it features three distinct user interfaces for each organisation after you log into the system, these are the Issuer, Verifier and Holder. Each UI has specific capabilities matching their responsibilities. Issuers can create and manage credentials, Holders can view and share their credentials, and Verifiers can check credential authenticity.

Backend Layer

The business logic layer that processes requests from the client interfaces. Includes authentication services for secure access and credential services that implement the core verification functionality. This layer translates user actions into blockchain transactions and database operations.

Database Layer

Stores user information and credential metadata that doesn't need to be on the blockchain. The database complements the blockchain by handling high-volume, frequently accessed data and personally identifiable information that requires controlled access.

Hyperledger Fabric Network Layer

The layer that provides immutable, verifiable credentials. Each organisation contains organisation-specific components (peer nodes and Certificate Authorities) and shared network infrastructure (ordering service, credential chaincode). This layer ensures that credentials can be verified without requiring complete trust between parties.

Technology Stack

Frontend

- Vite
- TypeScript
- Mantine Components Library

Backend

- Node
- Typescript
- React

Blockchain Network

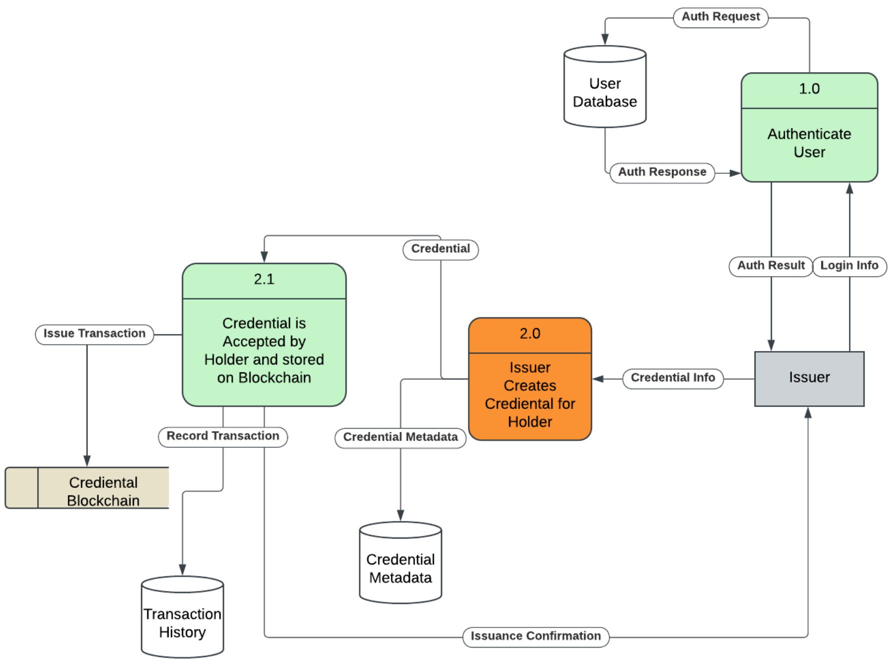
- Hyperledger Fabric
- Go (chaincode implementation)

Database

- PostgreSQL
- Supabase

Data Flow

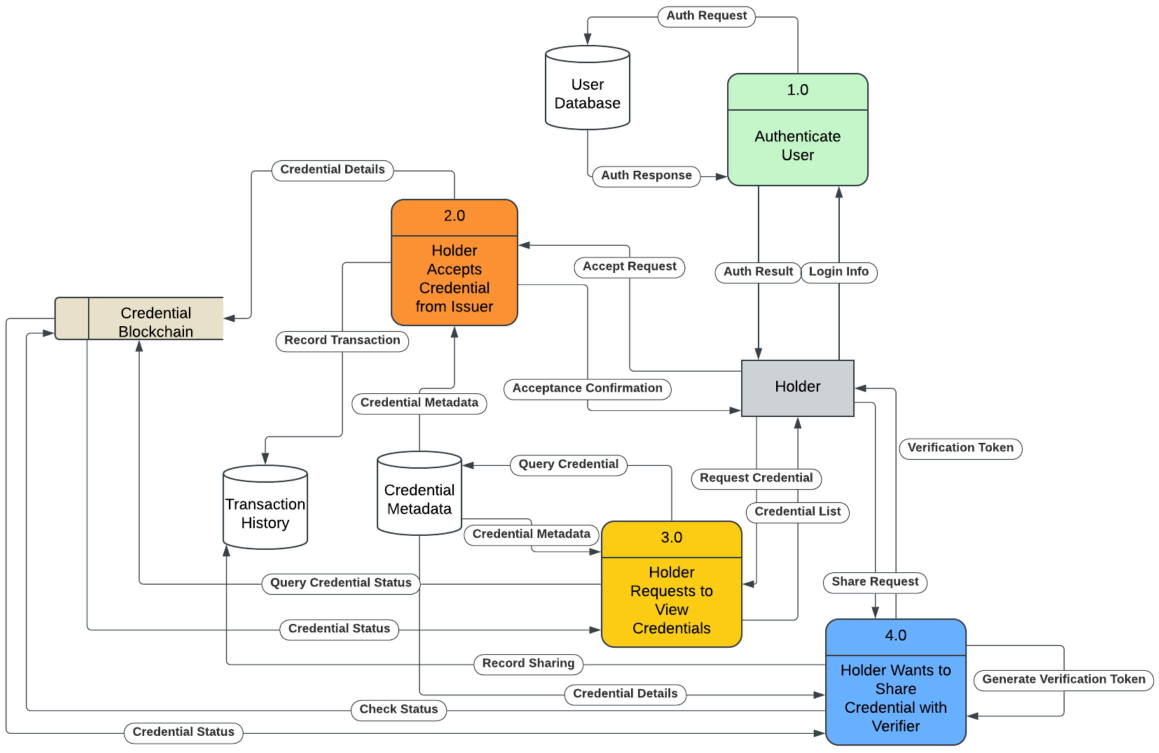
Issuer Data Flow Diagram



This Data flow diagram represents the Data Flow of an Issuer User, it shows the processes of:

- Authenticating an Issuer User
- An Issuer creating a Credential for a Holder

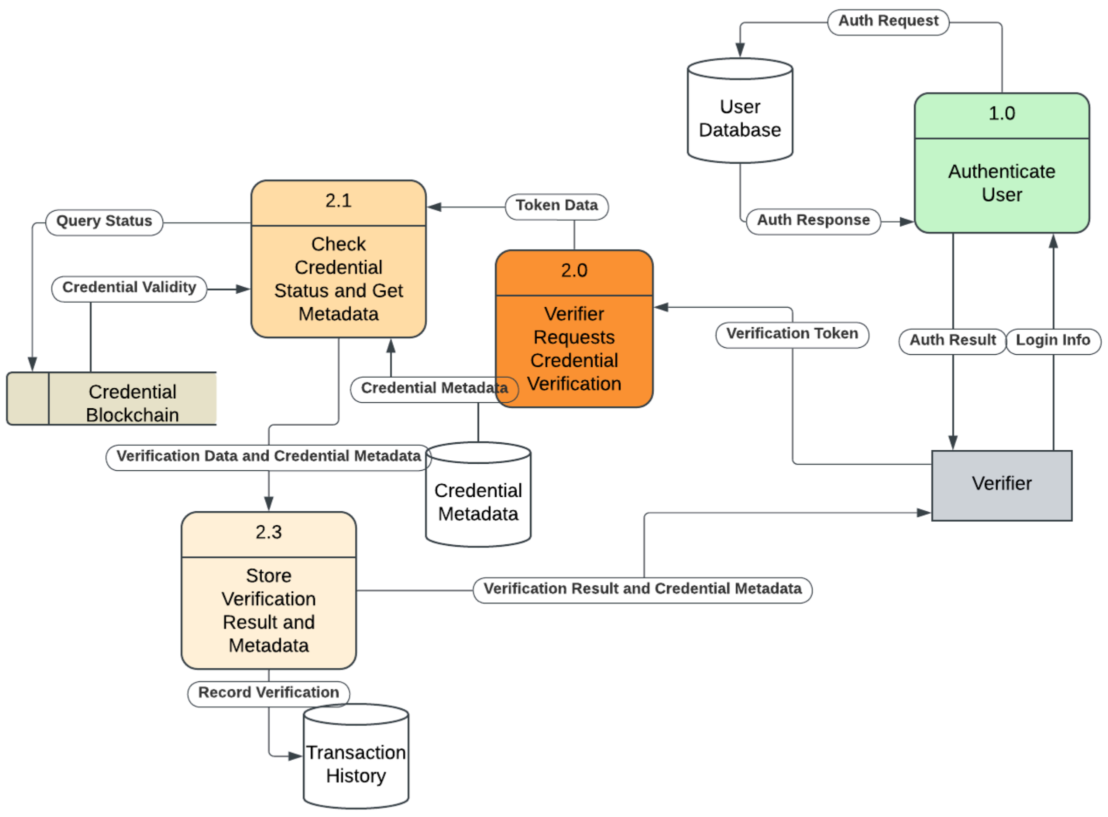
Holder Data Flow Diagram



This Data Flow Diagram shows the Data Flow for a Holder User, it shows the processes of:

- Authenticating a Holder User
- A Holder user accepting a Credential from an Issuer
- A Holder requesting to view their Credentials
- A Holder sharing a Credential with a Verifier

Verifier Data Flow Diagram



This Data Flow Diagram shows the Data Flow for a Verifier User, it shows the processes of:

- Authenticating a Verifier User
- A Verifier requesting to verify a Holder User's credential, checking the validity of the Credential and returning the information to the Verifier

---

Implementation Details

Hyperledger Fabric Network

Our platform is built on top of HYperledger Fabric, version 2.5.10. We chose hyperledger fabric for its highly customizable and permission based blockchain design that allowed us to build a blockchain network tailored to our specific needs.

Our implementation uses a single channel called “legitifychannel”, which maintains a unified ledger, clearly divided by organization based boundaries. All blockchain based actions are recorded on this immutable blockchain ledger, which ensures an immutable audit trail.

Our network contains three “organizations”, each with distinct roles:

1. OrgIssuer - Represents credential issuing bodies such as educational institutions or certifying bodies. Users that are a part of this umrella org have the ability to create and issue verifiable credentials on the channel.

2. OrgVerifier - Represents entities such as employers who seek to validate the authenticity of credentials on the network.

3. OrgHolder - Represents individuals who can manage and receive credentials that have been accredited to them, sharing with verifiers at will.

To ensure high availability, and fault tolerance, while ensuring the integrity of transactions on the network, we opted for a RAFT based consensus protocol. In our network, there were four ordering nodes that seamlessly handled leader election among themselves and automatic failover, ensuring continuous safe operation of our network, even in the event of isolated node failures. The use of a RAFT based consensus meant that transactions on the network could still be validated when at least 50% of the ordering nodes are online, only failing when the majority of ordering nodes have failed.

Each of the three fabric organizations on the network has their own certificate authority (CA) for identity management. The orderers also have a ca node. These CA’s manage and issue cryptographic materials to the nodes on the network, ensuring communication is secure and transactions are valid.

Each organization also runs a single peer node, with all communication protected by end-to-end TLS encryption, with all certificates managed by each organization's respective CA. Each transaction made on the network is signed with these certificates to ensure a verifiable audit trail on the ledger.

A unique aspect of our blockchain implementation is our use of composite keys. While we initially explored using fabric’s built in organization specific identity management features, we had a desire for new members to quickly and easily be able to join our platform and begin making blockchain transactions. Because of the complex setup involved in adding new peers and identities to the existing network, we opted for the use of composite keys as a more dynamic solution. This allowed a number of real world issuers to coexist inside the same fabric organization, each with a unique namespace to prevent ID conflicts. Composite keys combined the issuer's id with the credential id to ensure secure and manageable access to credentials within this network structure.

We are using NodeOU’s in the Membership Service Provider (MSP) to enforce role specific access within our network, ensuring only org specific actions can be performed on the network.

Finally, we have also integrated hyperledger explorer into our network configuration, which allows us the ability to view blockchain transactions in a graphical GUI, allowing us to assess individual blocks and transactions inside the network.

Smart Contract (Chaincode)

Hyperledger fabric allows users to define smart contracts which encapsulate business logic and data structures for operations performed on the platform. Fabric is largely agnostic when it comes to smart contract implementation, allowing developers to choose their preferred implementation language. In our case we chose to implement our smart contracts using the Golang programming language.

Core Data Structures:

Credential

DocID	Unique credential identifier
DocHash	Cryptographic hash of the issued doc
Type	Type of credential (degree, certificate, etc.)
HolderID	Identity of holder receiving credential
IssuerID	Identity of issuer issuing identity
IssuerOrgID	Organization the issuing user belongs to
Accepted	Boolean value indicating acceptance status of doc
Title	Human readable doc title
Description	Human readable doc description
LedgerTimestamp	Exact unix timestamp of doc issuance on ledger
AchievementDate	Specified date the credential was earned
ExpirationDate	Optional unix timestamp for doc expiration date
ProgramLength	Length of program undertaken to earn credential
Domain	Area of credential i.e. Computer Science
Attributes	Dynamic map of optional key-value pairs associated with credential

IssuerHolderRelationship

HolderID	Identity of holder receiving credential
IssuerID	Identity of issuer issuing identity

Status	Current relationship status (pending, active, rejected)

AccessGrant

DocID	Unique credential identifier
RequestedBy	Identity of user requesting access to the credential
GrantedAt	Unix timestamp of when the access request was sent

Key Functions:

1. IssueCredential: Allows an authorized user to issue a credential on the ledger.
2. AcceptCredential: Allows credential holders to accept or reject credentials issued to them.
3. ReadCredential: Retrieves credential details by credential ID
4. VerifyHash: Validates document authenticity by comparing a provided doc hash with the hash stored on the ledger.
5. GetIssuerCredentials/GetHolderCredentials: Gets all credentials associated with the provided issuer/holder id.
6. GrantAccess: Grants access to a user's credential to a verifying user.
7. AddIssuerHolderRelationship/CheckIssuerHolderRelationship: Manages and verifies relationships between issuers and holders.

Composite Key Strategy:

The use of composite keys in our chaincode implementation allows us to query the ledger much more efficiently. The chaincode creates three types of composite keys; 1. Issuer~doc, holder~doc, and issuerOrg-doc. This greatly improves the efficiency of our application when querying and fetching credentials associated with specific users.

Access Control and Privacy:

Privacy was an important consideration when designing our application, and as such, we have enforced access control at the chaincode level, ensuring the following:

- Credentials can only be issued by authorised issuers.
- Holders can only accept/deny credentials specifically associated with them.
- Verifiers can only gain access to credentials that have been explicitly shared with them.

Backend Implementation

Server Architecture:

- **Express.js Framework** - Provided core HTTP server functionality with support for middleware.

- **Prisma ORM** - Type safe database access which made schema design and db migrations easier.
- **Supabase Auth** - Managed our user authentication using an open source platform called supabase, helping us to manage sessions using JWT tokens.
- **Fabric SDK** - Provided custom gateway and wallet integrations for connecting to our hyperledger fabric network

Authentication & Authorization:

As mentioned above, we leveraged supabase for identity management across our application which provided the following:

- **JWT Authentication** - Secure tokens used for user identification.
- **Multi-factor Authentication (2FA)** - Optional TOTP-based 2FA enhances security which is a focus of our application.
- **Role-based Access Control** - Three distinct roles, each with specific permissions (Issuer, Verifier, Holder).
- **Database-backed Gateway Authentication** - Custom wallet implementation that stored fabric identities in our postgres database.

API Design:

Our API is built using REST principles and was documented using OpenAPI / Swagger to create dev friendly documentation, with the following sets of routes:

- **Authentication routes** - User registration, login, and session management
- **Credential Management routes** - Endpoints for issuing, accepting, denying, and verifying credentials
- **Access Control routes** - Manages access requests and grants between parties.
- **User Management routes** - For user profile and related user settings
- **Organization routes** - to support issuers and organizational based access control structures on the network

Integration Points:

The backend of our application serves as a central hub, facilitating communication between our various system components:

- **Fabric Network** - Custom gateway with dynamic connection profile management.
- **Supabase** - For authentication as well as storage buckets facilitating profile pictures and issuing org logos.
- **PostgreSQL** - Stored user data, credential metadata, blockchain identities and any other data unsuitable for storage on ledger.
- **Frontend Client** - Provided RESTful endpoints consumed by our React based frontend.

Database Design

Legitify uses PostgreSQL, managed with Prisma ORM to provide type safe access and database migrations. Careful consideration went into the definition of our database schema as we tried to balance a hybrid approach that allowed us to store information not suitable for our ledger, while still leveraging the core benefits provided by our use of an immutable ledger. Our database acts as a bridge between the blockchain and our application layers, storing info on specific user / organization profiles and metadata associated with credentials stored on the ledger.

Core Data Models:

The **User model** represents individuals with any possible role (issuer, verifier, or holder), and includes login credentials, role based access, personal profile info and associated settings such as 2fa settings.
The **Issuer model** defines organizations capable of issuing credentials and stores their related profile information such as verification data, logo, address etc.
The **Credential model** stores metadata related to credentials that have been issued on the ledger. While document verification occurs on the ledger layer, this model provides useful human readable descriptions of the document that offer additional context to what is stored on the ledger itself.
We are also defining several relationship models, which allow us to map users to other organizations, such as the **IssuerMember model** which maps issuing users with their associated issuing organization, the **IssuerAffiliation model**, which maps holders to the organizations issuing the credentials (issuers can only issue documents to users that have an affiliation with the issuing organization), and the **Request model** and **IssuerJoinRequest model**, which map join requests from issuing users to an issuing body and holders to an issuing body respectively.
Finally, we are using a **WalletIdentity model** which maps users to their respective X.509 certificates which are used to identify them during ledger transactions made through the fabric SDK.

Implementation Highlights + Technical Challenges

End-to-end TLS

```
- CORE_PEER_TLS_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
```

Every node on our network was bootstrapped with TLS on by default. Certificate paths were injected with environment variables so that we could use the same docker compose files locally or in production. This worked well locally, but presented some challenges when deploying. Because we were using a custom domain at network.legitifyapp.com, we faced two main issues:

1. Mixed SAN's - We faced some difficulty establishing TLS handshakes as clients were presenting with localhost in production. To get around this, we had to regenerate connection profiles in deployment with the domain in the subject alt name set.
2. Dynamic IP's - When deploying in AWS, we faced an issue with our environments ip address dynamically changing outside of our control, resulting in TLS failures. To get around this, we created a startup lambda script that would intercept our EC2 container startups and map their new dynamic IP's to our custom domain name, ensuring that our application was always configured with up to date ip addresses, mitigating TLS issues in deployment.

Dynamic connection profile fetching

As opposed to baking profiles into the frontend, we chose to ship rhythm at runtime, allowing us to always ship with updated connection profiles and certificates.

```
   console.log(`Fetching ${certType} certificate for ${orgType} ${org}...`);
    const response = await makeRequest(`/certs/${orgType}/${org}/${certType}`);

    // Handle the case where response might be an object with data property
    const certData = response.data || response;

    const orgDir = path.join(CERTS_DIR, org);
    if (!fs.existsSync(orgDir)) {
      fs.mkdirSync(orgDir, { recursive: true });
    }
```

Thuis allowed us to always pull up to date connection info from the fabric network when required by the server.

Composite keys for multi-tenant isolation

Due to the nature of our ledger, storing substantial credential information was going to lead to significant performance drops over time. To get around this, we implemented a composite key pattern which effectively allowed us to isolate separate issuing organizations from each other and also ensured that we could make consistently fast ledger queries:

```
issuerDocKey, _ := ctx.GetStub().CreateCompositeKey("issuer~doc",
[]string{issuerID, docID})
holderDocKey, _ := ctx.GetStub().CreateCompositeKey("holder~doc",
[]string{holderID, docID})
issuerOrgDocKey, _ := ctx.GetStub().CreateCompositeKey("issuerOrg~doc",
[]string{issuerOrgID, docID})
```

Many of the challenges we faced involved moving our fabric network and platform from a locally functioning demo. To a real world production environment. Challenges involved TLS SAN's, mutable IP's, secret management, and gossiping between nodes on the network. Overcoming these challenges involved applying a range of different practices involving AWS configs, careful secret management, and environment configuration, presenting an opportunity to gain familiarity with a range of different systems involved in deploying our application in the real world.

Testing & Validation

Our testing strategy involved multiple layers of testing to ensure reliability within our application as well as reliable deployment.

End-to-End Testing:

We implemented a comprehensive CI/CD pipeline using GitLab CI that runs full end-to-end tests on every merge request and deployment. All merge requests must successfully deploy in our test environment and once a deployment is triggered, it must undergo a series of end-to-end tests to verify the stability and reliability of the deployment itself. Our E2E test flow (test-flow.sh) simulates real world interactions with our API including the following:

1. User registration for all roles
2. Organization setup and holder affiliation
3. Credential issuance and acceptance
4. Access request and grant workflows
5. Credential verification

Our pipeline validates the whole system by:

- Setting up a fresh test database
- Starting the hyperledger fabric network in an isolated environment, hosted on oour gitlab runner (an EC2 instance running on AWS)
- Deploying and testing the chaincode
- Running the backend server
- Executing a full test flow
- Validating the database and blockchain state

Chaincode Testing:

Our chaincode testing is done using a shell script called testCredentialChaincode.sh, which validates:

- Issuer-Holder relationship management
- Credential issuance and status changes
- Credential hash verification
- Access control mechanisms
- Query functionality for all roles

We are testing our chaincode with both valid and invalid inputs, ensuring proper error handling.

Frontend Testing:

Frontend unit testing was done using React Testing Library with the Vitest framework, which used a range of different testing techniques:

Component Testing:

- Tested graphical layout components such as AppHeader, AppNavigation, Breadcrumbs etc. to ensure graphical consistency.
- Form validation on a range of components.
- Protected routes and role based access control.
- Loading states and error handling.

Role Specific Testing:

- Issuer credential management
- Holder credential viewing and sharing
- Verifier access request workflows

Mocking Strategy:

We employed the use of a mocking library called MSW (Mock Service Worker), which allowed us to mock a range of different endpoints in order to isolate our frontend components with consistent API responses for:

- Authentication context
- File uploads
- Blockchain interactions

Backend Testing:

For backend testing, we were also able to use Vitest which allowed us to maintain testing consistency across our application, and cover:

- Authentication and authorization flows
- Database operations
- Blockchain interactions
- File handling
- Error cases and edge conditions

Using Vitest allowed us to mock our other dependencies to:

- Provide test database connections
- Simulate blockchain network interactions
- Mock other external dependencies such as Supabase storage buckets

Continuous Integration

Our GitLab CI pipeline ensured:

- All tests were required to pass before emerging.
- Code quality standards were maintained
- Database migrations were non breaking and database was operational
- Blockchain network deployed correctly
- Backend server builds without errors
- Frontend client builds without errors

Our pipeline contained three stages:

1. Test - Ensures stability and no breaking changes have been introduced
2. Deploy - Deploys changes to live server hosting our application in production
3. Post-deploy - Runs a series of test flow E2E tests on the new deployment to ensure correct operation

Additionally, we followed a number of standard development practices that ensure safety at all times on our main branch, including:

- Branch protections on main that ensured all changes must be made by raising a merge request.
- Merge requests were tied to specific issues in gitlab which defined the problem domain.
- All merge requests required 1 peer review before merging into main.
- All merge requests required to pass pipeline successfully before merging.

User Testing Results

Overview:

During our user testing phase, we gathered feedback from users about the usability, functionality and their overall satisfaction with the platform. We guided users through the supervised process, asking them to complete certain steps as they went along and noting their experiences as they did. Their response provided valuable feedback, some of which we were able to implement immediately within our platform.

Key Feedback:

- Overall, users were either satisfied or very satisfied with the usability of the platform, with many indicating that they would recommend the platform for real world usage.
- Users mentioned clear and easy navigation as being important, leading us to implement some additional features for easily navigating the site, such as breadcrumbs, allowing the user to easily move back to the previous page they were on.
- Users also indicated a desire to make the application more personal, leading us to implement the addition of user profile pictures, laying the groundwork for increased personalization on the platform.
- Onme user suggested making the credential issuance more flexible, which was a very important point, as it led us to implement an attributes map that could be stored on the ledger associated with a credential. This attribute map could be populated with any custom fields the user wanted, leading to much more flexibility in credential issuance.

On top of these points, any of which we were able to implement working solutions to already, some other suggestions we received from users are the ability to export credentials, some minor feedback about speed of performance on the platform, and a desire for notifications to be implemented on the system. These provide valuable features to add to our roadmap and provide useful next steps in the development of the platform.

Future Development

Scalable production-grade deployment:

- Kubernetes & Helm: Replace the use of docker compose to allow rolling upgrades, auto scaling or peers/orderers, and self-healing

- Observability: Implementation of production logging and tracing using tools like Prometheus and Grafana to observe our platform at load

Truly Distributed Storage:

- Use of a distributed file storage system such as IPFS (Inter Planetary File System), allowing storage of distributed credentials across nodes on the network.

Dynamic Consortium Growth:

- One click org onboarding: Create a setup wizard that generates CA, MSP and channel configs dynamically, so that every organization gets its own respective fabric organisation, policies, and identities.

Interoperability & Privacy:

- Introduce support for the import and export of industry standard credential shapes such as open badges 3.0, or W3C verifiable credentials.
- Introduce selective disclosure using zero knowledge proofs (included partially in current implementation, but could be expanded on).

Conclusion

Project Summary

Legitify successfully delivers on our vision for a permissioned, multi-tenant credential network, that allows issuers, verifiers, and credential holders to seamlessly interact with each other, leveraging the immutable records provided by the platform's hyperledger fabric network. By wrapping our platform in a sleek, user-friendly UI, we have abstracted the difficulties of working with complex blockchains, making the system usable for a large proportion of users without significant training or skills. Our automated CI/CD, dynamic DNS, and end to end TLS turned a proof of concept into a production ready service.

Key Takeaways

- Trust-by-design: Every issuance, acceptance, and verification made on the platform is cryptographically signed, auditable and respects privacy.
- Hybrid storage essential: Balancing storage on our ledger with off chain metadata stored on postgres was vital and was carefully considered to find the perfect solution.
- Composite keys & role-based access: Keeps the ledger fast and tenants isolated, using fabric channels to manage multiple organizations in one place.

Final Thoughts

Digital credentials are only as valuable as the entity that verifies their authenticity. Legitify shows that with a right mix of blockchain and thoughtful UX, verifiable credentials can be made frictionless and resistant to fraud. There remains a valuable roadmap to pursue with this project to make it more complete, but the core goal of delivering a way to inspire trust in digital credentials remains unchanged.