KYCID

An operational oauth2 integration of eKYC

Course of study Bachelor of Computer Sciences
Author M. Yann Mickael DOY
Advisor Prof. Emmanuel BENOIST
Expert Daniel VOISARD

June 13, 2024

► Technical and Information Technologie

Abstract

This bachelor's thesis, carried out by Mr Yann Mickael DOY and advised by Mr Emanuel BENOIST with the expertise of Mr Daniel VOISARD, explores the creation of an identity verification service platform (Know your customer, eKYC) called KYCID for "Know your customer's ID".

The service enables third-party applications (client apps), such as GNU Taler, a payment platform, to perform eKYC procedures, which verify either the telephone number via a code sent by SMS, or by checking identity papers, or both.

ID papers verification is carried out by taking a photograph of the ID card or passport and other images of the person in different positions using his camera or webcam. This enables an administrator to verify that the documents in question belong to the individual in question and to validate their account.

In light of the aforementioned considerations, it is clear that security is of paramount importance. This is why the integration between the client app and KYCID is done with OAuth2. OAuth2 is a protocol and a set of specialised practices for delegating authorisation over HTTPS. In its version 2, it is technically mature and widely used in the industry.

OAuth2 enables third parties (client applications) to request access to a protected resource on a service. In this case, the resource is the user's identity, and the service is KYCID. OAuth2 is not merely a protocol; it is also a framework that provides the technical knowledge to enable its implementation in a secure manner.

Furthermore, KYCID incorporates a comprehensive array of security measures, including password protection, an anti-brute force system, and filters to prevent SMS plumping, which involves the use of premium rate numbers to extort money from the service.

The KYCID functionality enables customers to register with an email address and verify it (to prevent the use of fake emails), verify a phone number and verify identity documents. Furthermore, KYCID allows customers to carry out an eKYC procedure without first creating an account. This account will be created automatically at the end of the eKYC procedure.

The code has been developed in accordance with the principles of clean architecture, which facilitates scalability and testability. This has been achieved by implementing a comprehensive suite of automated unit, acceptance, and integration tests.

Contents

Abstract iii

Acknowledgement vii

1. Introduction 1
   1.1. Problematics 1
   1.2. Roles 2
   1.3. OAuth2 2
   1.4. SMS Challenge for eKYC 2
   1.5. Document and Face challenge for eKYC 3
   1.6. Product vision 4

2. Architecture 7
   2.1. Top-level overview 7
   2.2. System architecture 9

3. Security 11
   3.1. Cryptography 11
   3.2. OAuth2 Framework 11
   3.3. Cross-site request forgery 13
   3.4. Client authentification 14

4. Design 15
   4.1. Approach 15
   4.2. Technologies 15
   4.3. Clean architecture 16
   4.4. Domain layer 17
   4.5. Application layer 17
   4.6. Infrastructure and presentation layer 18

5. Testing 19
   5.1. Strategy 20
   5.2. Unit tests 20
   5.3. Acceptance tests 20
   5.4. Integration tests 21
   5.5. End-to-end tests 21

6. Results 23

7. Conclusion 25

Bibliography 27

List of Figures 29

List of Tables 31

Listings 33

Glossary 35

A. User Manual 39
A.1. Requirement 39
A.2. Nix setup 39
A.3. Environment setup 40
A.4. Configuration 41
A.5. Postgres 42
A.6. SMTP 43
A.7. Usage 43

Acknowledgement

Prior to commencing this thesis, it is necessary to acknowledge the individuals and resources that have contributed to the completion of this work.

Firstly, the thesis draws upon a previous thesis project conducted by Mr Loïc Fauchère in 2023, under the supervision of Mr Emanuel BENOIST as advisor and Mr Daniel VOISARD as expert [11].

This work concerns the drafting of a service platform allowing third parties to carry out KYC procedures by manually verifying identity documents taken in photographs as well as different selfies of users in different positions. Thus, this thesis, which follows this work, will be supervised by the same people in the same roles (M. BENOIST and M. VOISARD).

Initially, the report and source code were used directly, but then they were progressively replaced. The technology influence (initially Adonis JS and Typescript) was retained and subsequently replaced with Deno and Typescript.

Secondly, as a tangible application of this service platform developed at work, I integrated the service into GNU Taler, a privacy-friendly payment platform [7]. As such, I was able to benefit from the assistance of Mr Christian GROTHOFF, who is a core developer on this project as well as a lecturer at the BFH. In particular, he provided a presentation on the KYC process in Taler via OAuth2 (see section 1.3).

Finally, the work is also based on a set of tools derived from artificial intelligence. These include ChatGPT, which was initially employed in the thesis but was subsequently superseded by other AI, including the two models from DeepL (Translate and Write) [2, 3].

1. Introduction

In order to comply with legal requirements, certain industries must verify the identity of their users. For instance, the banking industry is subject to anti-money laundering/terrorist financing laws. Similarly, casinos must ensure that their customers are of an appropriate age, as do shops selling alcohol.

All these practices and mechanisms put in place by these industries are collectively known as **KYC**, an acronym for *Know Your Customer* This work will focus more specifically on the IT version of KYC, known as **eKYC** for *electronic KYC*.

To successfully complete an eKYC, three key challenges must be addressed: the first is user authentication, the second is the authentication of identity information, and the third is non-usurpation of identity, which ensures that the identity in question belongs to the user.

In order to facilitate the provision of the eKYC procedure by third parties and to avoid the repetition of the same process in each project, this work introduces the creation of an eKYC-as-a-Service platform.

1.1. Problematics

In recent years, the development of remote tools has made it necessary to use eKYC on a larger scale than was previously necessary for face-to-face identity verification.

The emergence of Twint [14], a financial intermediary subject to Swiss anti-money laundering laws [9], is a case in point. Twint offers its users the possibility of opening an account without tying it to a bank, which means that anyone in Switzerland can open an account anywhere.

The same can be said of telephone operators, which are subject to regulation [10], and which also allow users to open an account themselves without going anywhere, thanks to eKYC.

The market is developing, but there is no open-source service using a standard protocol, such as OAuth2 (see section 1.3), to simplify its use with the ecosystem of tools needed for interoperability.

1.2. Roles

The project encompasses a number of user/machine roles, which are defined below.

Role	Type	Description
KYCID	Machine	Authorization and Resource Server developed in this work performing eKYC procedure
Client	Machine	Third party application delegating its Customer's eKYC procedure to KYCID
Customers	Human	Any user who needs to be authenticated during an eKYC procedure
Operator	Human	Person responsible for installing/maintaining the KYCID application (see section)
Admin	Human	Person responsible for validating customer profiles

Table 1.1.: Project Roles

1.3. OAuth2

OAuth2 is a network communication protocol based on HTTP (Web) that allows resources (scopes) to be authorised for access to a third-party client application.

OAuth2 is also a framework (see section 3.2) which defines a security model.

OAuth2 is the second iteration of OAuth, which has therefore been able to mature technically and become more robust thanks to this test of time because, since its creation, it has been particularly attacked.

1.4. SMS Challenge for eKYC

To perform an identity verification (eKYC), this work has proposed 2 methods:

Firstly, the indirect method, which consists in delegating the verification to a telecom operator and in verifying only 2 things: that the user is in control of the number and that the number is Swiss. Thanks to this, we can indirectly verify the identity of the user.

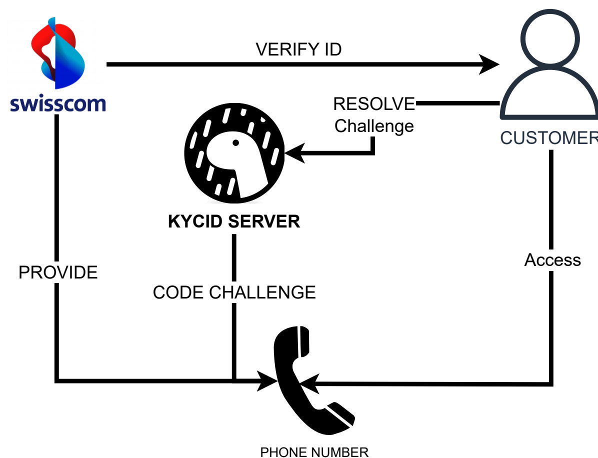


Figure 1.1.: eKYC by SMS challenge

The process is in 3 steps: The customer enters his telephone number; Then, a secret code will be sent by SMS to this number; Finally, the customer can enter the code received to complete the challenge.

Figure 1.2.: Process of SMS Challenge for eKYC

1.5. Document and Face challenge for eKYC

The second method is more direct. It consists of verifying the identity card or passport directly. To do this, we will use the user's webcam/camera to scan the ID card or passport.

On the back of the card or passport, there is a zone called MRZ for machine-readable zone. This is a standard used in particular in aviation to scan via OCR (optical character recognition) and thus extract all the information electronically.

```
PMCHESCHWEIZER<SAMPLE<<HELVETIA<<<<<<<<<<<<
S0A00A00<6CHE9508015F3210318<<<<<<<<<<<<<<08
```

Figure 1.3.: Specimen Machine Readable Zone (MRZ)

However, there is a potential issue: the images of cards or the cards themselves could be stolen. Therefore, it is necessary to implement measures to mitigate this risk of theft. To address this, we utilise a face challenge, which requires users to submit selfies in three different positions (head to the left, to the front, and to the right).

Figure 1.4.: Face challenge exemple

Consequently, an administrator can verify the photos to ascertain the legitimacy of the document and ascertain that all photos (document and face challenge) were taken with the same camera at the same time, among other criteria. If all criteria are met, the profile will be approved.

This method provides direct information on the identity of the customer, in contrast to the indirect method. However, it is a deferred method that necessitates human intervention.

1.6. Product vision

This work concerns the creation of a product designed to address the problem. The product is a web service, named KYCID, which stands for Know Your Customer's ID. It allows third-party applications (clients) to carry out their eKYC procedures by delegating the work to the service.

From the customer's perspective, using the service will be like a simple OAuth2 authorisation code flow connection. Once the Access Token has been granted, it will be possible to request an Endpoint with identity-related information.

From the customer's perspective, the process will be straightforward: they will simply click on button in client app to be redirected to web page on the platform's website, where they will carry out the eKYC procedure. Once completed, they will be redirected back to the customer and will have all the necessary information.

The eKYC procedure will be a linear process with optional steps listed below:

1. Obtain the user's consent for the client to access the requested Scopes.
2. Enter the email address.
3. Register if an account does not exist.
4. Verify the email address (a code will be sent by email) if the account is not verified.
5. Perform eKYC SMS Challenge procedure (see section 1.4) if it has been requested in the Scopes by the client.
6. Should the client request it, the eKYC document and face challenge procedure (see section 1.5) must be performed.

The registration of customers will be carried out by an operator with a technical profile (typically Mr Emanuel BENOIST) and does not necessarily require a graphical interface to perform this task.

In order to export a CSV file for the purpose of invoicing the service, the service provider must keep track of authorisation requests made by each client.

2. Architecture

2.1. Top-level overview

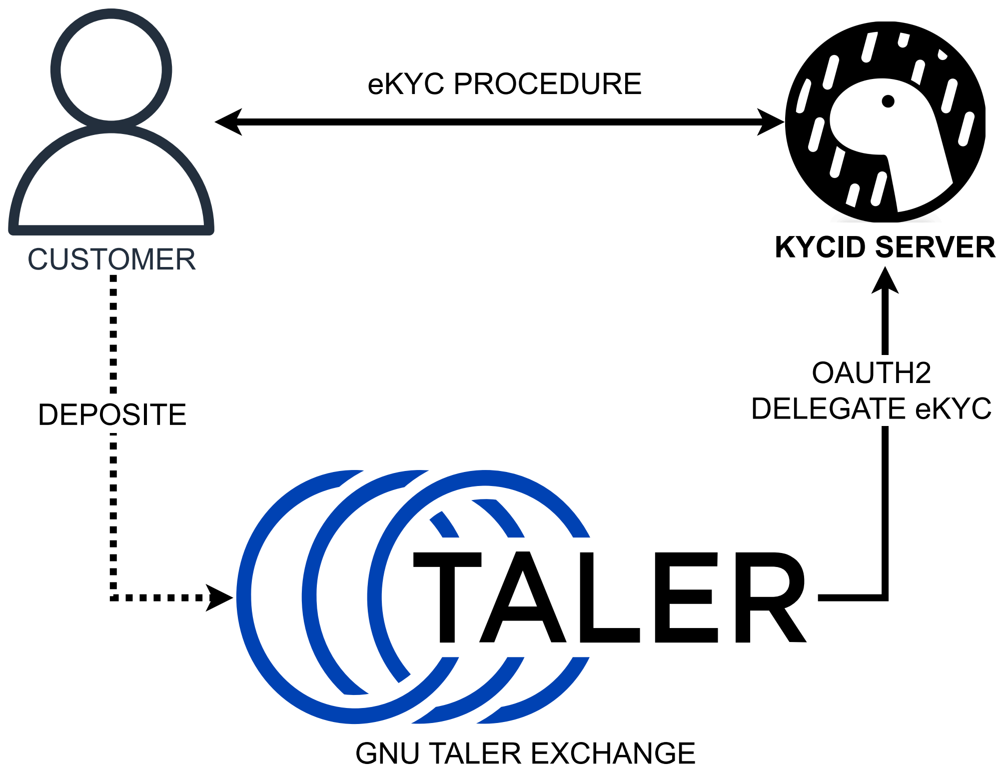


Figure 2.1.: Top-level project overview

The diagram above illustrates the three primary actors in the project:

1. **Customer**: the end user who wishes to deposit liquidity on the GNU Taler Exchange
2. **GNU Taler Exchange**: the payment service subject to AML, which delegates the eKYC process to the KYCID service
3. **KYCID**: the web service responsible for executing the eKYC process for GNU Taler Exchange

The following diagram is a model of the project's planned money deposit sequence.

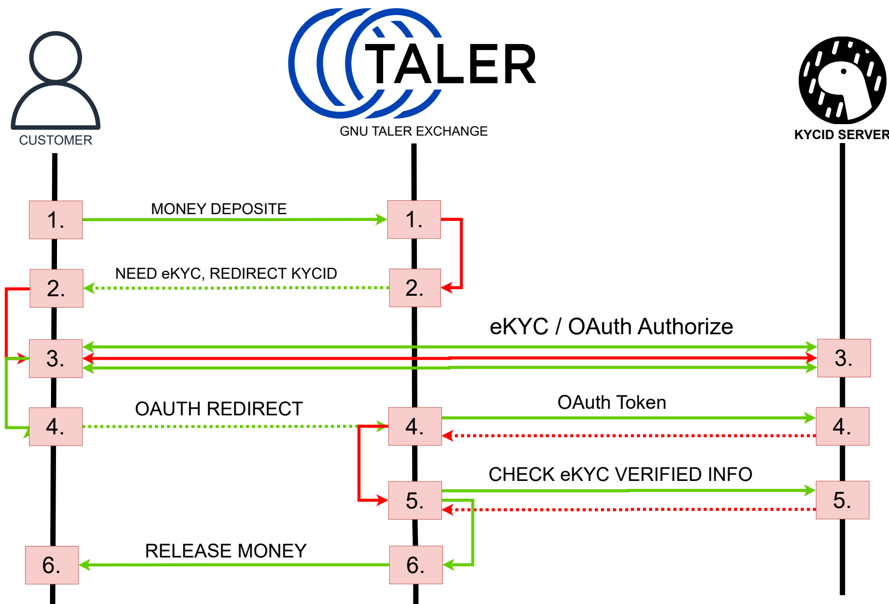


Figure 2.2.: Top-level project sequence flow

The following steps are involved in the process:

1. **Deposits**: The customer deposits liquidity on a GNU Taler exchange.
2. **Initiation of eKYC process**: As the exchange is subject to the AML, it initiates a KYC process using the KYCID service (delegation via OAuth2 authorisation flow). The customer's browser is redirected to the KYCID.
3. **OAuth front channel, eKYC**: comprises a series of round trips between the customer's browser and the KYCID, during which the KYC process is performed. This process requires interaction with the customer, as illustrated in figure.
4. **OAuth back channel**: Once the KYC process has been completed, the user's browser is redirected to the exchange with an authorisation code that allows it to retrieve an access token from the KYCID. This is the OAuth back channel.
5. **Retrieve eKYC information**: the exchange can retrieve the information from the eKYC process thanks to the access token previously granted.
6. **Release**: once verified, if the exchange criteria are satisfied. It can release the deposits.

The process described above is a case study of an OAuth authorisation code flow application for GNU Taler that performs an eKYC procedure to release money. you can find more details on how OAuth2 works in section 3.2.

2.2. System architecture

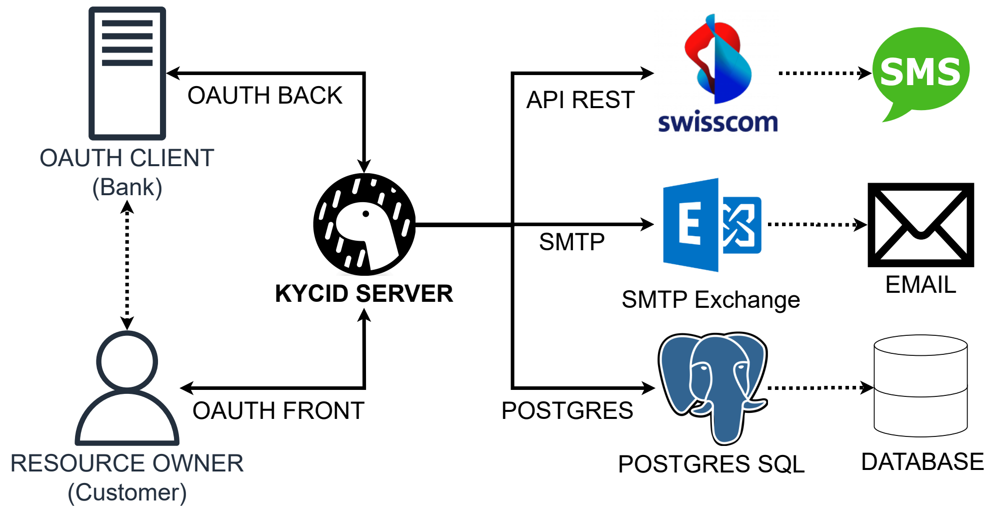


Figure 2.3.: System architecture

The figure above on the left shows the **primary actors** in the system (listed below).

- **Client**: The client is the service that delegates the KYC process to the system. An example of this is the GNU Taler exchange.
- **Customer**: The user whose identity is being verified.

And on the right shows the **secondary actors** in the system (listed below).

- **SMS Provider**: The system must send SMS messages to verify the phone number. Swisscom is the SMS provider via the text messaging product, which allows SMS to be sent via a REST API.
- **Mail sending server**: The system must also send an email to verify the address and notify the user. An SMTP server (such as Microsoft Exchange) is required.
- **Persistence**: A postgres sql database to store system status.

3. Security

3.1. Cryptography

The security of this application is contingent upon the implementation of cryptographic primitives. In this instance, we will utilise the robust **libsodium** library.

The first primitive employed is the hashing of passwords utilising Argon2id, which is particularly slow and therefore mitigates brute-force offline attacks. The result of this process, along with the salt and algorithm used, is encoded according to the standard PHC string format.

The slowness of Argon2id provides protection against offline attacks, but it is necessary to implement additional protection against online brute force attacks. The number of attempts over a period of time must therefore be limited.

The second primitive used is an AEAD (authenticated encryption with additional data), which is used to seal contextual data injected into an HTML form in order to implement workflow (succession of forms + navigation).

The AEAD used to create a crypto token utility to convey information between forms. The implementation is analogous to that of Branca [1].

The last primitive employed is the CSPRNG generator, which is used to generate non-guessable random code.

3.2. OAuth2 Framework

This project takes place in the context of the OAuth2 framework, which defines a model (notably for security) as well as a set of protections required to comply with the standard [8]. OAuth2 allows a third-party application (client) to access a set of resources (scope) belonging to a resource owner (RO) on a remote server.

OAuth2 defines 4 roles:

Role	Type	Description
Resource	Data	The resource in question is to be accessed. In KY-CID its an human identity
Resource owner (RO)	Person	Customer in KYCID. He own his ID
Client	Machine	Third party should perform and delegate to KYCID an eKYC process.
Authorization server	Machine	OAuth2 server. it's KYCID.
Resource server	Machine	Server provide resource access. it's KYCID.

Table 3.1.: Roles in OAuth2 Framework

To describe the different security features, we first need to describe the different steps in an authorisation flow sequence, as shown in the figure below:

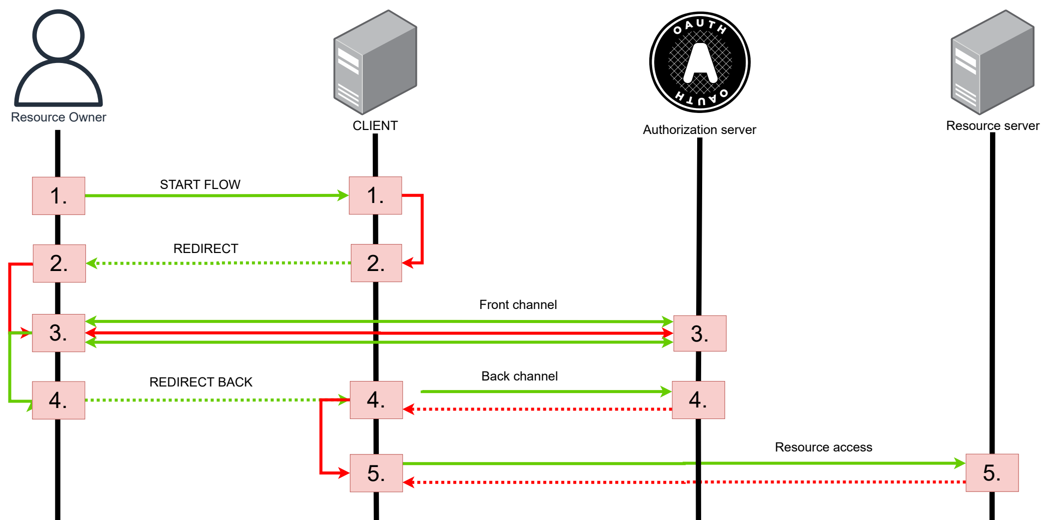


Figure 3.1.: OAuth2 authorization flow sequence

The figure above shows a sequence of steps in an authorisation code flow. The steps are explained below:

1. the resource owner initiates the flow.
2. the client redirects the user via the HTTP redirection mechanism. However, the client is susceptible to a CSRF attack due to the fact that initialising the flow is merely a URL to which the user is redirected. This URL is therefore susceptible to guessing, and therefore the attack can be carried out (see section 3.3).

3. The RO interacts with the authorization server in a front channel call phase due to the presence of a web app that allows interaction.

   In this case, it is the authorization server that is vulnerable to the RO attempting to circumvent the established procedures. Consequently, it is imperative to never trust any input originating from the user (systematic validation) and to implement measures to protect against CSRF (see section 3.3).

   The purpose of server authorisation is to determine whether to grant access to the requested resource (scopes). To achieve this, the server can implement any application logic (within the web app) to perform this action.

4. If the previous step has been authorised, the RO will be redirected to the client with an authorisation code and the csrf protection token from step 2 (this information is conveyed by query parameters called *code* and *state*).

   With this code we can make a so-called back channel request (as opposed to the front channel of step 3), because it is an http request from server to server and therefore there is no WebUI. This request is sent from the client to the authentication server with the received authentication code and something that authenticates the client to the authentication server (see section 3.4).

   This request retrieves an access token. The reason for having 2 tokens (authorisation code and access token) instead of a direct access token is related to the fact that the token returned by the front channel can only be passed in url (query parameter), which is not a secure means of transport for information as sensitive as an access token, as url can be logged in several places (proxy server, cache, history, intercepted by the application on Android, log server). Therefore, the authorisation code can only be used once to obtain the access token.

3.3. Cross-site request forgery

A CSRF attack may occur when an application contains actions that are linked to one another. It is possible for another site to forge a request for a given action, thus bypassing the intended steps and processes. This could result in the hijacking of the originally planned process by a hacker. Additionally, the request could be made on another site, leading to the recovery of the cookie at the time of the request. To mitigate this issue, two methods can be employed:

The first method involves setting the "Same-Site" attribute to "Lax" on the cookie. This prevents the cookie from being added if the request is made on another site or domain. This avoids recovering the session.

The second method is to add a CSRF token (which must not be forgeable anywhere other than on the server) in a hidden field. When the request is processed, the token is checked to ensure its validity. In KYCID, the CSRF token is implemented by an AEAD

(see section A), which encrypts the contextual data of the action, thus securing it. In addition, the action and session are used as additional data to ensure that the AEAD signature is unique for each action and session, thus avoiding reusing a token twice. Furthermore, the token has a defined lifetime.

3.4. Client authentification

Two methods exist for authenticating the client. The first and simplest method involves recording a secret that is known only to the client and the authorisation server. By sending this secret, the authorisation server can verify the client's authenticity.

However, this method can cause problems in the event of a leak or if the client is not executed on a server but directly in JavaScript in the browser or in an application on a smartphone. This is referred to as a non-confidential client. This client is unable to safeguard this secret.

In response to this problem, OAuth2 has introduced a second method of client authentication called PKCE (Proof Key for Code Exchange) which does not authenticate the client but rather authenticates that the Back Channel request was made by the same instance as the Front Channel request. To achieve this, the client generates a secret, named *code_verifier*, at random and hashes it with SHA256. The result of this process is called *code_challenge*. The authentication server is able to verify the authenticity of the client by repeating the process of hashing the *code_verifier* and comparing the result with the *code_challenge* sent in the front-channel request.

In OAuth 2.1, it is imperative to utilise PKCE, regardless of whether the client possesses a secret. This is to mitigate the risk of an attack in the event of a leak of this secret.

In the field of cryptography, PKCE represents a commitment scheme.

4. Design

4.1. Approach

In this project, the approach used is Domain Driven Design (here after DDD). This approach, originally introduced by Eric EVANS in an over-rated book called BlueBook, considers that the domain/business of the application is far more valuable than the technique and should therefore be put first.

There are 2 aspects to this theory: strategy and tactics.

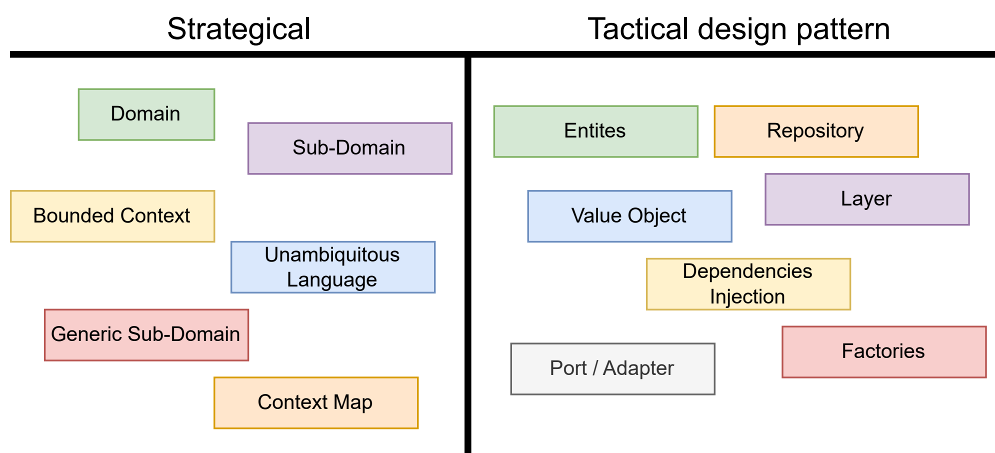


Figure 4.1.: DDD Strategical concept vs Tactical concept

The figure above lists the main patterns used in the project, both strategic and tactical. The strategic patterns are more conceptual and mainly used for modelling, whereas the tactical patterns are standard design patterns.

4.2. Technologies

The technologies used are listed below:

- **Typescript**: superset of ecmascript (JS) allowing to devéloppé in JS with a powerful system of verification of type.

- **Libsodium**: serious library written in C that allows to perform various cryptographic tasks. Used for encryption / token authentication (see Token Security section) as well as password hashing (see Password Security section). Has a JS/TS port/module for use in Typescript.
- **PostgreSQL**: database engine. Chosen because already used by M. Emanuelle BENOIST to be used after the thesis.
- **Deno fresh**: Small HTTP/HTTPS framework developed in Typescript (modification compared to Adonis previously used by Mr Loïc Fauchière).
- **Valita**: Small library allowing to make verification / validation of data and allowing to make correspond the Typescript types defined at compilation, but with a runtime verification (validation).
- **Tesseract**: ORC Engine to scan MRZ on ID Document
- **Deno**: Secure runtime (change from node used before).

4.3. Clean architecture

The KYCID server software is designed on a "clean architecture" model, as described in reference [12]. This model is a layered model, but where the domain is the base layer rather than the persistence layer.

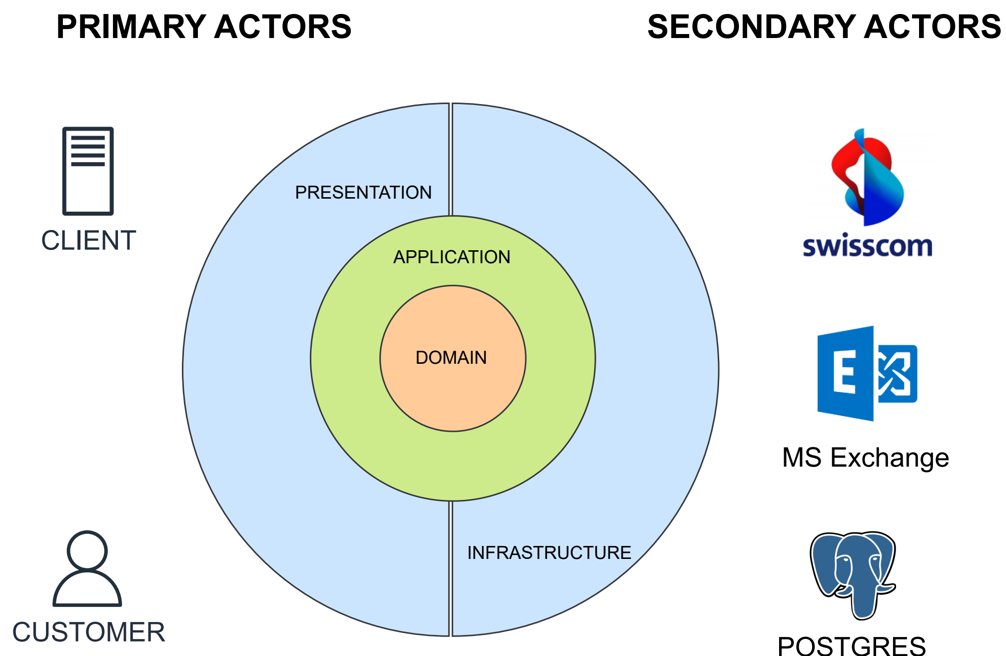


Figure 4.2.: Software Layer architecture

The design comprises three layers: the domain layer, the application layer and the infrastructure and presentation layer. The domain and application layers constitute the core layer.

4.4. Domain layer

This is the most elementary layer of the system, so it depends only on itself. There will only be simple classes that model the domain (hence the name) and raise exceptions if invariants are not respected.

It is important to distinguish between the domain model and the database persistence model. For instance, in a database, the objective is to avoid duplication. In the domain, it is possible to have two distinct classes representing a user in two different contexts. However, in the database, the aim is to merge these into a single table.

4.5. Application layer

This layer constitutes the core of the domain and is dependent on both itself and the domain layer situated directly below. The role of the application layer is to connect the domain with the external environment. In order to fulfil this function, it is necessary to invert control using the port/adapter pattern. This involves creating an interface (port) in the application layer which can be used by the layer but placing the implementation (adapter) in the layer above. It is preferable to place as little business code as possible in this layer and to instead place it in the domain layer.

It is also necessary for the application layer to prevent the infrastructure layer from depending indirectly on the application layer. For example, exceptions raised in the domain layer must be handled by the application layer. Similarly, it is important to avoid raising exceptions in the application layer. However, exceptions that are raised by adapters should not be handled by the application layer.

It is not advisable for the application layer to raise exceptions, as this should be the domain layer's responsibility. However, any exception raised in an adapter in the infrastructure layer that is not a domain exception (for example, a connection failure exception) should not be handled by the application layer.

The application layer must not depend on infrastructure or technical code that you have not written yourself (avoid complex libraries).

4.6. Infrastructure and presentation layer

This layer is responsible for encapsulating all the technical complexity associated with infrastructure and user presentation. It will contain the most direct code to implement the application layer adapter. Not any business logic is included.

This layer should be independent of the domain, as it is not directly below it. This principle is linked to the fact that we do not want a change in the domain to imply a change in the infrastructure, nor vice versa.

5. Testing

KYCID uses a TDD [4] approach which consists of writing the tests before the code and following the cycle below.

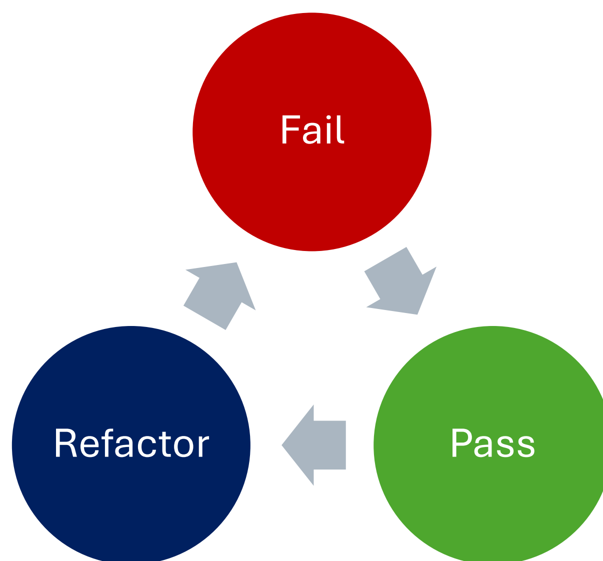


Figure 5.1.: TDD Developpment Cycle

Cycle steps:

1. write the test, run it and then fail it (to check the test's ability to detect)
2. have the tests taken as directly as possible
3. refactor the code
4. repeat the cycle as many times as necessary

The following narrative scheme [5] is particularly useful for writing tests:
1. **Given**: a certain situation (arrange)
2. **When**: trigger action (act)
3. **Then**: verify result of action (assert)

5.1. Strategy

It is important to note that not all tests are equally useful or even cost the same. In our case, we can list four types of test: unit tests, acceptance tests, integration tests and end-to-end tests. These can be hierarchised in a diamond below:

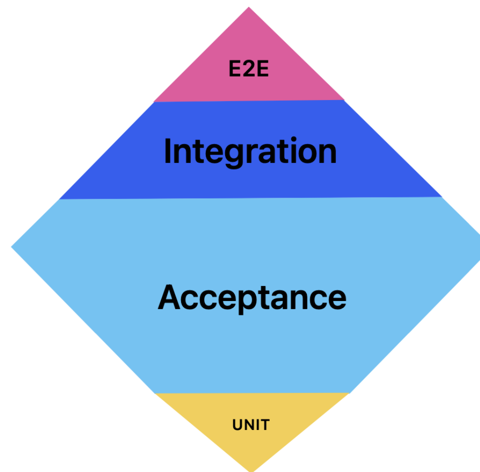


Figure 5.2.: Diamond testing strategy

The subsequent section will provide detailed information regarding the specific nature of each test.

5.2. Unit tests

These tests are designed to assess a single unit of code, hence the name. They are relatively simple and quick to write.

However, they tend to focus on minor technical details that may not be directly relevant to the final project. Consequently, we only use unit tests when necessary and not as a systematic approach. This is in contrast to pyramid testing [6], which relies on them as a fundamental basis.

5.3. Acceptance tests

The purpose of acceptance testing is to validate/accept the existence of a given use case.

They are more interesting because they verify functionality, which is the most valuable thing in software (at least in terms of agility).

These tests are more difficult to write. However, with a clean architecture (see section 4.3), these tests are greatly simplified because we are testing the core layer (domain and application layer). This layer does not depend on the infrastructure, but on ports that have been specially designed for this case. There is therefore no technical code, only business code, which means that the acceptance test becomes a kind of large unit acceptance test.

5.4. Integration tests

The objective of integration tests is not to be exhaustive but to verify the communication (integration) between components. These tests are necessary to ensure the overall functionality of the application, but they are complex to write. Consequently, the strategy in diamand is to write the minimum necessary and no more.

5.5. End-to-end tests

In end-to-end testing, the objective is to test from the perspective of the end user. This makes these tests the most challenging to write, as they require the control of a significant amount of code, which introduces a high degree of complexity and technical detail. Unlike integration tests (see section A), they must also simulate the entire application and its infrastructure, which contributes to the high cost.

However, these tests, which verify the user's point of view, are considered the most valuable according to the principles of agility. Consequently, there will be few end-to-end tests in the diamond strategy.

6. Results

Following a significant investment of time and effort, I was able to develop KYCID with a clean code structure (in accordance with the principles of clean architecture), which allows the code to evolve and be maintained.

In addition, the testing strategy has been followed more closely, with the implementation of unit, acceptance and integration tests that are relatively comprehensive. However, due to a lack of time and the necessary setup to get started, no end-to-end tests could be set up and were replaced by manual tests.

In terms of operational functionality, the system can be readily configured via the variable environment, as detailed in the user manual (see section A.4). The configuration options allow the user to select and configure various aspects of the system, including persistence, email and SMS sending, and HTTPS server configuration.

Persistence has two modes: the first is in-memory, where all data is stored in memory (a useful feature for testing and development purposes), and the second is postgres, where all data is stored in the database with the same name. In terms of email transmission, there are two modes: a "fake mode," which logs the email to the console (used for testing and development), and an SMTP mode, which sends an email using this protocol.

With regard to SMS transmission, there are also two modes: a "fake mode," which is similar to sending email, and a Swisscom mode, which uses the Text Messaging (SMS) service to send SMS.

In terms of functionality, the main features are present, namely a connection system with email verification, a brute force protection system for passwords and codes entered, and a procedure for sending emails and text messages. Additionally, a session system, as well as user verification by SMS, identity verification with ID card/passport scanning, document MRZ with validation by admin, and connection via OAuth2 authorisation flow are included.

Nevertheless, certain functionalities are absent, including PKCE security, password reset, request to forget, and a CSV export system for billing the service to the customer.

7. Conclusion

KYCID is a pre-production prototype. There are numerous avenues for further development. In particular, we may cite the non-implemented functionalities mentioned in the results (see results 6).

The potential enhancements may be found in the AI, which would permit the human to be assisted in his task of detecting fraud in the verification of identity documents. This would permit the process to be enhanced and industrialised, for instance, to pre-filter the profile for human validation.

Similarly, to ascertain that it is indeed the holder's identity card, we utilise a face-challenge. We can envisage more complex face-challenges, such as live actions (using a video stream instead of photos).

Another area for improvement is the operational aspect, in particular the introduction of an observability/monitoring system with security audit logs. In addition, an administration system should be implemented to enable the client application to register, and a billing and payment system should be developed. This project has considerable potential for further development.

To conclude on a more personal note and to draw some experience from this project, it can be observed that even a project that seems simple because it's an idea that can be quickly explained can reveal unexpected complexities and workloads.

This is particularly evident in terms of planning, where the tasks may not seem complicated at first sight, but they are very numerous and, furthermore, the number of tasks is underestimated due to poor identification. It is therefore of great importance to utilise planning tools in order to ensure that the project remains on track. The project management issues encountered in this thesis can be categorised into two distinct categories.

The first of these is that when a project is behind schedule, the necessity to catch up tends to result in a reduction in the rigour of maintenance for documentation, schedules and tests. This reduction in quality will ultimately lead to a loss of work efficiency, which will accentuate the delay (negative cycle). The challenging aspect of this is that the loss of efficiency is only visible weeks later.

Secondly, experience plays a crucial role, particularly in terms of taking a step back and not wanting to go too fast on certain tasks while not dragging your feet. This is not a simple balance to achieve.

Bibliography

[1] *Authenticated and encrypted API tokens*. URL: https://branca.io/.

[2] *DeepL Translate: The world's most accurate translator*. URL: https://www.deepl.com/translator.

[3] *DeepL Write: AI-powered writing companion*. URL: https://www.deepl.com/write.

[4] Martin Fowler. *Bliki: Test driven development*. Dec. 11, 2023. URL: https://martinfowler.com/bliki/TestDrivenDevelopment.html.

[5] Martin Fowler. *Given When Then*. Aug. 21, 2013. URL: https://martinfowler.com/bliki/GivenWhenThen.html.

[6] Martin Fowler. *The Practical Test Pyramid*. Feb. 26, 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html.

[7] *GNU Taler*. URL: https://taler.net/fr/.

[8] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Tech. rep. 6749. Oct. 2012. 76 pp. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749.

[9] *loi sur les établissements financiers (LEFin)*. Jan. 1, 2020. URL: https://www.fedlex.admin.ch/eli/cc/2018/801/fr.

[10] *Loi sur les télécommunications (LTC)*. Jan. 1, 2021. URL: https://www.fedlex.admin.ch/eli/oc/2020/1019/fr.

[11] Fauchère Loïc. "KYC-procedures as a service". Bachelor's thesis. Berner Fachhochschule, June 2023.

[12] Robert C. Martin. *The Clean Architecture*. Aug. 13, 2012.

[13] *Nix and NixOS*. URL: https://nixos.org.

[14] *TWINT*. URL: https://twint.ch.

List of Figures

1.1. eKYC by SMS challenge 3
1.2. Process of SMS Challenge for eKYC 3
1.3. Specimen Machine Readable Zone (MRZ) 4
1.4. Face challenge exemple 4

2.1. Top-level project overview 7
2.2. Top-level project sequence flow 8
2.3. System architecture 9

3.1. OAuth2 authorization flow sequence 12

4.1. DDD Strategical concept vs Tactical concept 15
4.2. Software Layer architecture 16

5.1. TDD Developpment Cycle 19
5.2. Diamond testing strategy 20

List of Tables

1.1. Project Roles 2

3.1. Roles in OAuth2 Framework 12

A.1. KYCID Software Dependencies 39

Listings

A.1. Multi-user nix install on Linux 39
A.2. Nix install on MacOS 40
A.3. Single-user nix install on Linux 40
A.4. Enabled Nix Flake feature 40
A.5. Disposable dev environment shell 40
A.6. Production environment install 41
A.7. All Environment Variable and `.env.sample` 41
A.8. Setup local postgres server 42
A.9. Fake SMTP Server for developpment 43
A.10. Usage command cheatsheet 43

Glossary

Access Token An authorization token allowing access to a resource

CSPRNG Cryptographically-safe pseudo random generator

CSRF Cross-site request forgery is well-know HTTP vulnerability

eKYC electronic KYC

Endpoint URL on which we can access from machine to machine (API)

KYC acronym for Know your Customer is a set of practices aimed at verifying user identity

KYCID eKYC-as-a-Service web platform developed in this thesis and acronym of *Know your customer's ID*

MRZ Zone on an identity card or passport where all the information on the document is encoded and easily scannable

OAuth2 An HTTP-based communication protocol and framework for granting third-party access to resources

OCR Optical character recognition is an algorithm / process for extracting text from an image / scan

Scopes List of strings specifying the resources the client wants to access

Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

June 13, 2024

Y. Doy

A. User Manual

A.1. Requirement

The following software is required for the deployment of KYCID:

Dependencies	Version	Comment
Deno	>= 1.43	KYCID Runtime environement
Postgres	compatible with 15	database
Tesseract	compatible with 5.3	OCR Engine for Card or Passport Scanning
Nix	any with flake feature	Environement manager, optional
Mailcatcher	any	fake SMTP server with webUI, only for dev
TexLive	with BFH template	only for compiling documentation

Table A.1.: KYCID Software Dependencies

It is possible to install these dependencies independently, in accordance with the usual installation procedure. In this case, it is not necessary to install Nix. Alternatively, the guide below provides instructions on how to install Nix and use it to set up the environment.

A.2. Nix setup

Nix is a packet manager [13], which allows the user to create immutable and reproducible builds.

It is important to note that Nix, as a packet manager, is distinct from NixOS, a Linux distribution that employs Nix as a general packet manager.

To install nix you can just run install with following command:

Listing A.1: Multi-user nix install on Linux

```
student@ubuntu:~$ sh <(curl -L https://nixos.org/nix/install) --daemon
```

The installer will run interactively and pose a series of questions. Once the installation process is complete, it is necessary to restart the terminal. If the user is using a Mac, the following command should be executed:

Listing A.2: Nix install on MacOS

```
student@macintosh:~$ sh <(curl -L https://nixos.org/nix/install)
```

In addition, should you wish to avoid installing the Nix system for all users, you may opt to install it in single user mode via the following command:

Listing A.3: Single-user nix install on Linux

```
student@ubuntu:~$ sh <(curl -L https://nixos.org/nix/install) --no-daemon
```

It should be noted that, in order to utilise the Flake feature, which has been designated as experimental, it is necessary to perform the requisite activation procedure.

Despite the note on the flake functionality as experimental, this is not the case. It is a significant change, and the features are named as such for the sake of Nix retro-compatibility. The functionality is widely stable and used.

Run the following command to enabled NIX Flake feature (the same on Linux and MacOS):

Listing A.4: Enabled Nix Flake feature

```
student@macintosh:~$ mkdir -p ~/.config/nix # to be sure that folder exists
student@macintosh:~$ echo 'experimental-features = nix-command flakes' >
    ~/.config/nix/nix.conf
```

A.3. Environment setup

To set up the environment for execution, simply install the various dependencies listed in the software requirement section. With NIX, the following command will suffice:

Listing A.5: Disposable dev environment shell

```
student@macintosh:~$ cd /path/to/projet
student@macintosh:~$ nix develop
student@macintosh:~$
student@macintosh:~$ # without TexLive
student@macintosh:~$ nix shell .#deno .#tesseract .#postgresql .#mailcatcher
```

This command will start a shell in which the dependencies will be available. This will ensure that they do not conflict with the rest of the system. Once the shell is closed, the applications will still be stored in the Nix cache, but they will not be accessible in the PATH.

Please note that this shell will have ALL the dependencies, including those used only in development such as TexLive (> 4GB), Mailcatcher or Postgres (as long as you already have a Postgres server elsewhere).

Please be aware that only the binaries and the library are installed, and no configuration file will be generated on your system or any daemon started.

In order to install only the dependencies in production, the following commands must be executed:

```
student@macintosh:~$ cd /path/to/projet
student@macintosh:~$ ## Global install on system
student@macintosh:~$ nix profile install ".#deno" ".#tesseract"
    ".#postgresql"
student@macintosh:~$ ## Local install on current shell
student@macintosh:~$ nix shell ".#deno" ".#tesseract" ".#postgresql"
```

Listing A.6: Production environment install

A.4. Configuration

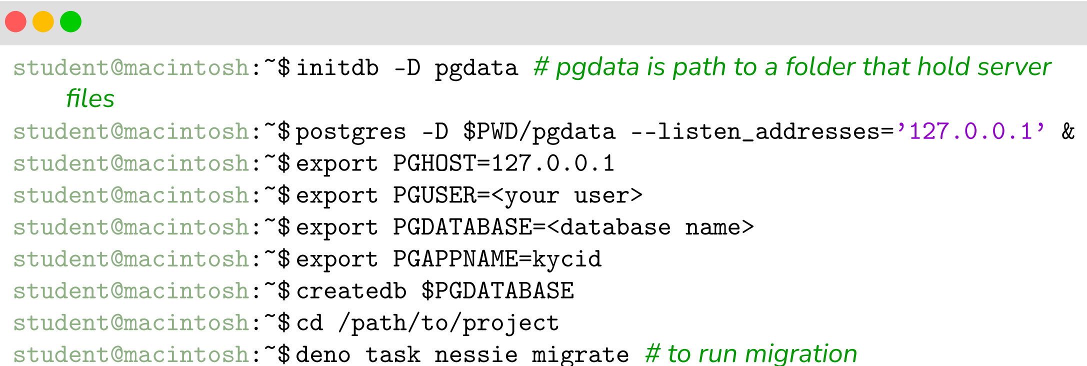
The configuration of all elements takes place via environment variables. It is possible to log these variables in a file with the extension `.env`.

```
#
# HTTPS
#
HTTPS_HOST="0.0.0.0" # hostname or ip
HTTPS_PORT="443"     # listening port
HTTPS_KEY="./src/http/ca-key.dev.pem" # path to TLS certificate key
HTTPS_CERT="./src/http/ca-cert.dev.pem" # path to TLS certificate

#
# MAILER
#
```

```
MAILER="smtp"            # mailer type either "smtp" or "fake" (for test or dev)
SMTP_HOST="127.0.0.1"
SMTP_TLS=1               # only if enabled TLS (remove variable to disable)
SMTP_PORT="1025"
#SMTP_USER="kycid@smtp.local"   # SMTP connection auth user
#SMTP_PASS="smtp_password"      # SMTP connection auth password
SMTP_FROM="kycid@smtp.local"    # email as sender

#
# SMS PROVIDER (https://digital.swisscom.com)
#
SMS_PROVIDER="swisscom" # sms provider type either "swisscom" or "fake" (for test or dev)
SWISSCOM_SMS_TOKEN_ENDPOINT=https://api.swisscom.com/oauth2/token
SWISSCOM_SMS_MESSAGE_ENDPOINT=https://api.swisscom.com/messaging/sms
SWISSCOM_SMS_CLIENT_ID=client-id
SWISSCOM_SMS_CLIENT_SECRET=client-secret


#
# PERSISTENCE
#
PERSISTANCE="postgres" # persistance type either "postgres" or "memory" (for test or dev)
PGHOST="127.0.0.1"
PGDATABASE="kycid"
PGUSER="ydo"
PGAPPNAME="kycid"

#
# TESSERACT
#
TESSERACT_PATH=/path/to/bin/tesseract
```

Listing A.7: All Environment Variable and `.env.sample`

A.5. Postgres



```
student@macintosh:~$ initdb -D pgdata # pgdata is path to a folder that hold server files
student@macintosh:~$ postgres -D $PWD/pgdata --listen_addresses='127.0.0.1' &
student@macintosh:~$ export PGHOST=127.0.0.1
student@macintosh:~$ export PGUSER=<your user>
student@macintosh:~$ export PGDATABASE=<database name>
student@macintosh:~$ export PGAPPNAME=kycid
student@macintosh:~$ createdb $PGDATABASE
student@macintosh:~$ cd /path/to/project
student@macintosh:~$ deno task nessie migrate # to run migration
```

Listing A.8: Setup local postgres server

A.6. SMTP

```
student@macintosh:~$ mailcatcher --ip 127.0.0.1 --smtp-port 1025 --http-port
    1080 --foreground
```

Listing A.9: Fake SMTP Server for developpment

Access to the email messages sent via this fake SMTP server is available via a web interface at the following address: `http://127.0.0.1:1080`.

A.7. Usage

```
student@macintosh:~$ # MIGRATE POSTGRES SCHEMA
student@macintosh:~$ deno task nessie migrate
student@macintosh:~$
student@macintosh:~$ # RUN DEV SERVER (AUTO RELOAD ON FILE CHANGE)
student@macintosh:~$ deno task dev
student@macintosh:~$
student@macintosh:~$ # RUN PRODUCTION SERVER
student@macintosh:~$ deno run --allow-all src/http/main.ts
student@macintosh:~$
student@macintosh:~$ # COMPILE LaTeX thesis (only on dev env)
student@macintosh:~$ cd docs
student@macintosh:~$ bfhlatex thesis
```

Listing A.10: Usage command cheatsheet