Implementacja Systemu Podpisywania Plików "File Cert"

Bartłomiej Adamiec

Abstrakt - **System podpisywania plików *File Cert* to rozwiązanie umożliwiające bezpieczne i wiarygodne składanie podpisów elektronicznych na dokumentach cyfrowych typu PDF. Produkt został zaprojektowany z myślą o zapewnieniu autentyczności, integralności oraz niezaprzeczalności danych w obiegu elektronicznym, zgodnie z założeniami infrastruktury klucza publicznego (PKI). Architektura rozwiązania opiera się na modelu klient-serwer, gdzie kluczową rolę pełni asynchroniczny serwer pełniący funkcję lokalnego Urzędu Certyfikacji (CA). Moduł ten odpowiada za generowanie i podpisywanie certyfikatów X.509, wiążąc tożsamość użytkownika z jego kluczem publicznym. Proces weryfikacji polega na ekstrakcji klucza publicznego z certyfikatu X.509 w celu potwierdzenia poprawności złożonego podpisu. Bezpieczeństwo systemu zapewnia uwierzytelnianie tokenami JWT, które uaktywniają sesję użytkownika na czasowo ograniczoną. W trakcie sesji użytkownik może wywołać komendy, które pozwalają: przesłać plik na serwer, podpisać go oraz zweryfikować plik PDF z podpisem. Po wywołaniu komendy weryfikującej serwer zwraca użytkownikowi raport w postaci pliku PDF zawierającym informację o tym czy każdy podpis zamieszczony w pliku przeszedł pełną walidację, nazwę użytkownika, który podpisał plik, datę podpisu, datę walidacji, czy zawartość pliku została zmieniona po podpisie oraz jakiego algorytmu użyto do podpisu. W zrealizowanej wersji MVP system oferuje interfejs CLI, obsługę rejestracji i logowania użytkownika, transfer plików, składanie podpisu w standardzie PAdES (PDF Advanced Electronic Signatures) z wykorzystaniem klucza prywatnego oraz weryfikację kryptograficzną dokumentów, które zostały podpisane w systemie File Cert. Implementacja bazuje na nowoczesnych bibliotekach języka Python: FastAPI, PyHanko, cryptography oraz Typer, co gwarantuje wydajność i zgodność z wymogami technicznymi. Docelowo platforma umożliwi obsługę szerszego spektrum formatów (XML, DOCX), stanowiąc kompleksowe narzędzie do zarządzania cyfrowym obiegiem dokumentów.**

Słowa kluczowe - **Podpis elektroniczny, PKI, PAdES, FastAPI, CA, bezpieczeństwo cyfrowe, Python.**

I. ZAKRES FUNKCJONALNY MVP (MINIMUM VIABLE PRODUCT)

Wersja "Minimum Viable Product" skupia się na realizacji ścieżki krytycznej: od bezpiecznej rejestracji i wydania tożsamości cyfrowej (CA) [1], poprzez autoryzację, aż po weryfikację podpisanego dokumentu PDF wraz z generowaniem raportu.

A. Wewnętrzny Urząd Certyfikacji (Internal CA)

System posiada wbudowany moduł PKI (Public Key Infrastructure) [1], pełniący rolę zaufanego urzędu. Odpowiada on za zarządzanie cyklem życia tożsamości cyfrowych: generowanie par kluczy asymetrycznych (prywatny/publiczny) oraz wystawianie i podpisywanie certyfikatów w standardzie X.509 [1] oraz ich bezpieczny eksport w kontenerach PKCS#12 [2].

B. Zarządzanie Tożsamością i Kontrola Dostępu

Fundament bezpieczeństwa systemu stanowią endpointy odpowiedzialne za uwierzytelnianie użytkowników:

1. Endpoint `/register`: Umożliwia utworzenie nowego konta w systemie. API [3] przyjmuje dane użytkownika, waliduje ich unikalność i bezpiecznie zapisuje w bazie (z wykorzystaniem haszowania).
2. Endpoint `/login`: Odpowiada za proces autoryzacji. Po weryfikacji poświadczeń wydaje token dostępu (JWT [4]), który jest niezbędny do komunikacji z chronionymi zasobami systemu.

C. Obsługa Dokumentów i Podpis Cyfrowy (Core)

Moduł odpowiedzialny za cykl życia dokumentu PDF:

1. Endpoint `/upload`: Służy do bezpiecznego transferu plików na serwer. Umożliwia przesłanie dokumentu do przestrzeni roboczej przed wykonaniem na nim operacji kryptograficznych.
2. Endpoint `/sign`: Inicjuje proces nałożenia podpisu elektronicznego (PAdES [5]) na przesłany dokument. Serwer wykorzystuje powiązane z użytkownikiem klucze kryptograficzne, aby zapewnić autentyczność i niezaprzeczalność pochodzenia pliku.

D. Weryfikacja i Raportowanie

Krytyczny element systemu służący do walidacji podpisów zewnętrznych. Endpoint `/verify` przyjmuje podpisany plik PDF i przeprowadza pełną analizę kryptograficzną. Endpoint sprawdza matematyczną poprawność podpisu, ścieżkę zaufania certyfikatu X.509

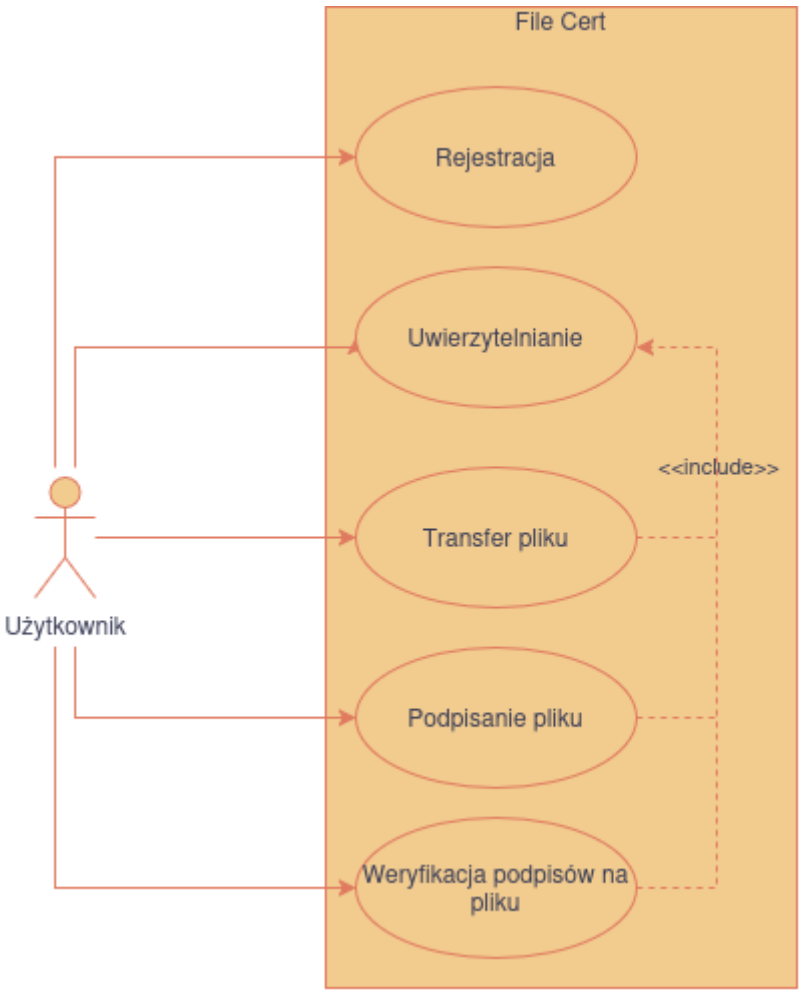
[1] oraz integralność pliku. W odpowiedzi zwraca szczegółowe metadane weryfikacji, które służą do wygenerowania końcowego raportu PDF.

E. Interfejs Kliencki (CLI)

Interakcja z systemem odbywa się za pomocą terminala. Użytkownik nie wywołuje zapytań HTTP ręcznie - robi to za niego aplikacja konsolowa, tłumacząc komendy (np. `python3 client.py sign my_file.pdf`) na odpowiednie zapytania do API i prezentując odpowiedzi serwera w czytelnej formie lub w przypadku weryfikacji podpisu, zwracając plik.

F. Diagram Przypadków Użycia

Funkcjonalności opisane powyżej oraz interakcje użytkownika z systemem zostały zilustrowane na diagramie przypadków użycia (Rys. 1).



Rys. 1. Diagram przypadków użycia systemu File Cert.

II. NARZĘDZIA I TECHNOLOGIE

Implementacja systemu File Cert została zrealizowana w języku Python 3.12.3. Wybór ten podyktowany był szeroką dostępnością nowoczesnych bibliotek kryptograficznych oraz wsparciem dla asynchroniczności. Poniżej przedstawiono charakterystykę kluczowych komponentów systemu

A. Warstwa Backendowa i API

Rdzeń aplikacji serwerowej oparłem na frameworku `FastAPI` [3]. Framework oparty na standardzie ASGI (Asynchronous Server Gateway Interface), który umożliwia asynchroniczne przetwarzanie wielu zapytań jednocześnie (non-blocking I/O). Stanowi to przewagę nad starszymi rozwiązaniami (jak klasyczne Django czy Flask) opartymi na synchronicznym standardzie WSGI (Web Server Gateway Interface), w którym obsługa żądania blokuje wątek do momentu jego zakończenia (blocking I/O).

Do walidacji danych wykorzystałem bibliotekę `pydantic` [6], która używając natywnych podpowiedzi typów (Type Hints), stanowi fundament walidacyjny frameworka. Uzupełnieniem jest klasa `Annotated` z modułu `typing`, odgrywająca kluczową rolę w mechanizmie wstrzykiwania zależności (Dependency Injection), pozwalając na deklaratywne łączenie typów zmiennych z ich konfiguracją.

B. Warstwa Kryptograficzna i PKI

Za operacje na plikach PDF odpowiada biblioteka `PyHanko` [7], wyspecjalizowana w obsłudze struktur podpisu cyfrowego. Obsługuje ona standard PAdES oraz mechanizm LTV (Long Term Validation). Biblioteka zarządza niskopoziomową strukturą pliku poprzez tzw. przyrostowe aktualizacje (incremental updates), co gwarantuje integralność dokumentu i umożliwia wielokrotne podpisywanie bez naruszania wcześniejszych sygnatur.

Fundamentem kryptograficznym projektu jest biblioteka `cryptography` [8]. Odpowiada ona za pełną obsługę warstwy PKI: generowanie par kluczy asymetrycznych (RSA), tworzenie i podpisywanie certyfikatów X.509 oraz ich serializację do formatu PEM. Biblioteka ta dostarcza bezpieczne prymitywy kryptograficzne.

> *Definicja:* Prymityw kryptograficzny to niskopoziomowy algorytm wykonujący jedno konkretne zadanie (np. funkcja skrótu SHA-256 zamieniająca dane w ciąg o stałej długości), stanowiący podstawowy blok budulcowy złożonych protokołów bezpieczeństwa [9].

Decyzja o wyborze biblioteki `cryptography` (zamiast alternatywnej `PyCryptodome` [10]) podyktowana była faktem, iż stanowi ona bazę dla biblioteki `PyHanko`. Użycie dwóch silników kryptograficznych mogłoby sprawić, iż będą one ze sobą kolidować oraz zajmować dodatkową pamięć na serwerze.

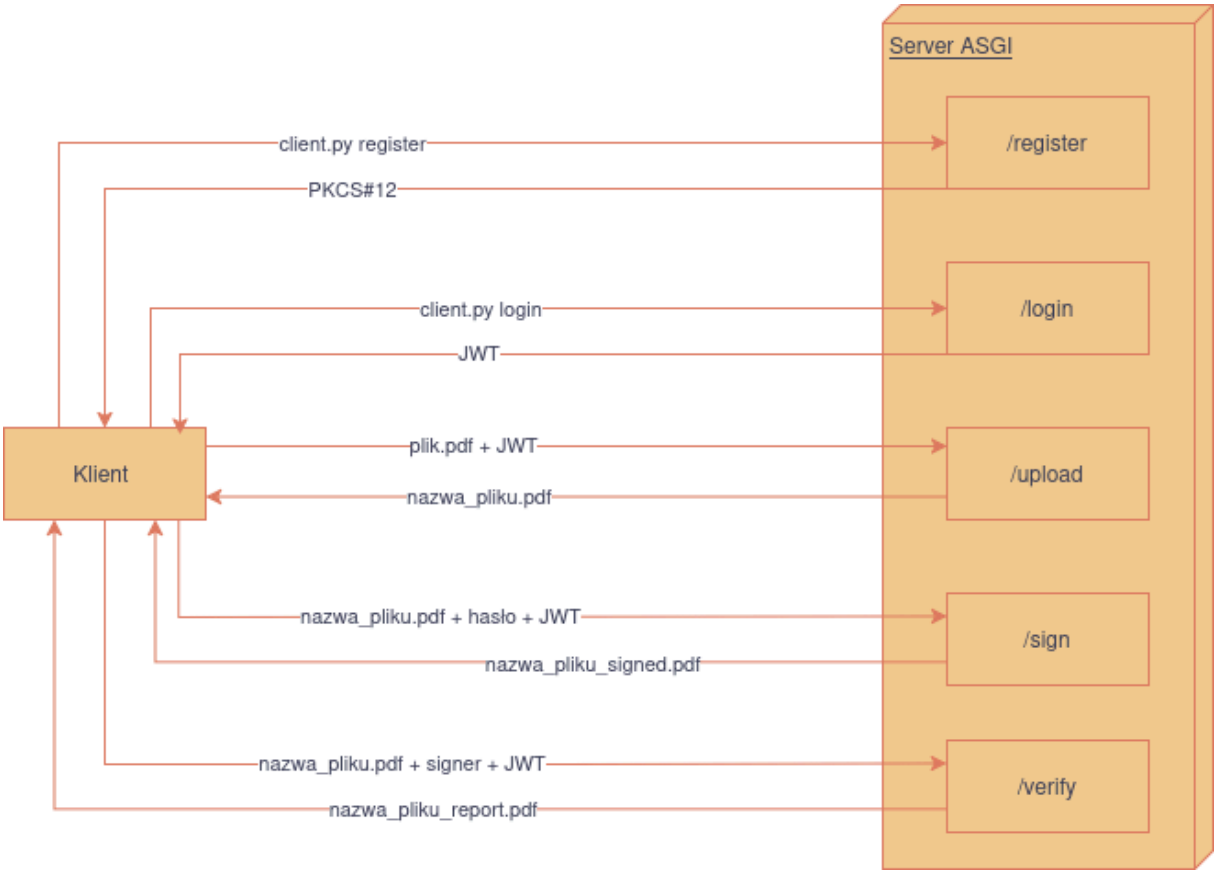
D. Interfejs Kliencki (CLI) i Narzędzia Pomocnicze

Aplikacja kliencka została zbudowana w oparciu o bibliotekę `Typer` [11]. Podobnie jak backend, wykorzystuje ona adnotacje typów do automatycznej walidacji argumentów z poziomu wiersza poleceń, co w porównaniu do standardowych modułów (`sys, argparse`) pozwoliło na znaczną redukcję kodu. Warstwę wizualną CLI (formatowanie tekstu, panele) obsłużono przy użyciu biblioteki `Rich` [12].

Do generowania raportów weryfikacyjnych w formacie PDF wykorzystano bibliotekę `fpdf2` [13], umożliwiającą precyzyjne rozmieszczanie elementów graficznych przy użyciu systemu współrzędnych. Manipulacja czasem (znaczniki czasowe podpisów) realizowana jest przez wbudowany moduł `datetime`.

E. Diagram blokowy architektury systemu

Schemat logiczny komunikacji oraz przepływ danych pomiędzy aplikacją kliencką a serwerem ASGI zilustrowano na Rysunku 2.



Rys. 2. Diagram blokowy architektury systemu File Cert.

III. STRUKTURA KATALOGÓW I ORGANIZACJA KODU

Projekt `file_cert` charakteryzuje się modułową architekturą, oddzielającą warstwę logiki biznesowej, interfejsów API oraz konfiguracji. Poniższy schemat (Listing 1) prezentuje hierarchię plików w repozytorium.

Listing 1. Drzewo katalogów projektu File Cert.

```
file_cert/
│
├── app/
│   ├── core/
│   │   ├── config.py
│   │   └── security.py
│   ├── db/
│   │   └── database.py
```

```
│   ├── models/
│   │   └── schemas.py
│   ├── routers/
│   │   ├── auth.py
│   │   └── documents.py
│   ├── services/
│   │   ├── ca_service.py
│   │   ├── report_generator.py
│   │   ├── signer.py
│   │   └── validator.py
│   ├── __init__.py
│   └── main.py
│
├── certs/
├── client/
│   └── client.py
├── storage/
├── .env
├── .gitignore
├── init_ca.py
├── README.md
└── requirements.txt
```

A. Moduł Aplikacji Serwerowej (app/)

Jest to główny katalog zawierający kod źródłowy backendu. Jego struktura wewnętrzna została podzielona według odpowiedzialności:

Rdzeń (`core/`): Plik config.py centralizuje konfigurację globalną (zmienne środowiskowe, ścieżki), natomiast `security.py` implementuje mechanizmy bezpieczeństwa, w tym haszowanie haseł (bcrypt) oraz obsługę tokenów JWT.

Baza Danych (`db/`): Moduł `database.py` odpowiada za inicjalizację połączenia z nierelacyjną bazą danych MongoDB.

Modele Danych (`models/`): Plik `schemas.py` definiuje obiekty transferu danych (DTO) przy użyciu biblioteki pydantic, zapewniając walidację typów w żądaniach HTTP.

Routery (`routers/`): Katalog ten grupuje punkty końcowe API. Moduł `auth.py` obsługuje procesy rejestracji i logowania, a `documents.py` zarządza operacjami na plikach (przesyłanie, podpisywanie, weryfikacja).

Logika Biznesowa (`services/`): Warstwa serwisowa izoluje logikę aplikacji od warstwy HTTP:

`ca_service.py`: Implementuje funkcje wewnętrznego CA (generowanie kluczy, certyfikatów, obsługa PKCS#12).

`signer.py`: Odpowiada za nakładanie podpisu PAdES na pliki PDF.

`validator.py`: Realizuje kryptograficzną weryfikację podpisów i łańcucha zaufania.

`report_generator.py`: Generuje raporty poweryfikacyjne w formacie PDF.

Punkt Wejścia: Plik `main.py` inicjalizuje instancję FastAPI i scala wszystkie routery.

B. Komponenty Zewnętrzne i Skrypty

Poza główną aplikacją, projekt zawiera kluczowe elementy infrastruktury:

Klient CLI (`client/`): Katalog zawiera skrypt `client.py`, będący konsolowym interfejsem użytkownika, integrującym wszystkie funkcjonalności systemu.

Inicjalizacja CA (`init_ca.py`): Skrypt narzędziowy służący do jednorazowego wygenerowania "Kotwicy Zaufania" (pary kluczy i certyfikatu Root CA).

Zasoby (`certs/`, `storage/`): Katalogi przeznaczone na przechowywanie kluczy kryptograficznych urzędu certyfikacji oraz plików przetwarzanych przez użytkowników.

Konfiguracja: Plik `.env` przechowuje wrażliwe zmienne środowiskowe, a `requirements.txt` definiuje listę zależności projektowych.

IV. SZCZEGÓŁY IMPLEMENTACYJNE APLIKACJI SERWEROWEJ

Architektura aplikacji serwerowej (`app/`) została zaprojektowana w sposób modułowy, separując warstwę konfiguracyjną, modele danych oraz logikę biznesową od interfejsów API.

A. Konfiguracja i Bezpieczeństwo (core/)

Moduł ten stanowi fundament działania aplikacji, centralizując zarządzanie stałymi oraz mechanizmami ochronnymi

`config.py` - Przechowuje wszystkie stałe, które przechowują: zmienne środowiskowe odczytywane z plik `.env` (tajny klucz do generowania JWT i hasło do certyfikatu Root CA), ścieżki do bazy danych, klucza Root CA, certyfikatu Root CA, folderu do operacji na plikach, nazwę algorytmu do podpisywania JWT, czas ważności JWT wyrażony w minutach, schemat `OAuth2PasswordBearer` służący do automatycznego ekstrahowania tokenów JWT z nagłówka HTTP `Authorization: Bearer`

`security.py` - Moduł obsługujący aspekt bezpieczeństwa projektu. Znajdują tu się funkcje pozwalające na hashowanie hasła algorytmem brypt. Wykorzystałem do tego klasę `passlib.context.CryptContext` oraz weryfikację zahashowanego hasła. Zawiera również fukcję tworzącą tokeny JWT ważne domyślną, bądź podaną ilość czasu oraz funkcje zwracającą aktualnego użytkownika na podstawie zależności (funkcja `get_current_user`) dekoduje token oraz weryfikuje jego poprawność umożliwiając autoryzację dostępu do chronionych endpointów.

B. Warstwa Danych i Modele (`db/`, `models/`)

Warstwa ta odpowiada za definicję kontraktów danych oraz komunikację z systemem składowania.

`database.py`: Realizuje interfejs połączeniowy z nierelacyjną bazą danych MongoDB przy użyciu `pymongo`.

`schemas.py`: Zawiera definicje struktur danych (DTO – Data Transfer Objects), zaimplementowane jako klasy dziedziczące po `pydantic.BaseModel`. Modele te odpowiadają za walidację typów danych przesyłanych w ciele żądań HTTP (Request Body) oraz formatowanie odpowiedzi serwera.

C. Logika Aplikacyjna i Routing (`routers/`)

Katalog ten grupuje punkty końcowe (endpointy) REST API, podzielone tematycznie na moduły uwierzytelniania i obsługi dokumentów.

1) Moduł Uwierzytelniania (`auth.py`) Realizuje procesy zarządzania tożsamością i sesją użytkwownika.

Endpoint `/register`: Przeprowadza walidację unikalności nazwy użytkownika i zapisuje jego dane w bazie. Następnie wywołuje usługę `ca_service`, która generuje osobisty kontener kryptograficzny PKCS#12 (zawierający klucz prywatny i certyfikat). Plik ten jest zapisywany w magazynie serwera (`/storage`) i zwracany użytkownikowi jako wynik rejestracji.

Endpoint `/login`: Odpowiada za uwierzytelnianie. Weryfikuje zgodność przesłanego hasła z zapisanym w bazie skrótem (wykorzystując funkcję `core.security.verify_password`). W przypadku sukcesu, serwer generuje i zwraca token dostępu JWT.

2) Moduł Dokumentów (`documents.py`) Obsługuje operacje na plikach PDF. Dostęp do wszystkich zasobów w tym module jest chroniony - każde wywołanie wymaga wstrzyknięcia zależności `core.security.get_current_user` w celu potwierdzenia tożsamości.

Endpoint `/upload`: Waliduje rozszerzenie przesyłanego pliku i zapisuje go w katalogu `/storage`, z prefiksem nazwy użytkownika `<current_user>_<filename>.pdf`, co zapobiega konfliktom nazw między użytkownikami.

Endpoint `/sign`: Weryfikuje fizyczną obecność pliku PDF oraz certyfikatu użytkownika na serwerze. Następnie deleguje zadanie do serwisu `services.signer.sign_pdf_service`, który nakłada podpis zgodny z PAdES i zwraca podpisany cyfrowo dokument według formatu `<filename>_signed.pdf`.

Endpoint `/verify`: Na podstawie parametrów wejściowych (dla `<filename>.pdf` szuka `<current_user>_<filename>_signed.pdf`) lokalizuje odpowiedni podpisany plik. Przeprowadza analizę kryptograficzną przy użyciu funkcji `services.validator.verify_pdf_service`, a następnie wyzwala

`services.report_generator` w celu dynamicznego wygenerowania i zwrócenia do klienta szczegółowego raportu weryfikacji w formacie PDF.

D. Logika Biznesowa i Serwisy (`services/`)

Warstwa serwisowa izoluje złożoną logikę biznesową od interfejsów HTTP, realizując kluczowe operacje kryptograficzne oraz procesy przetwarzania dokumentów.

`ca_service.py` – Moduł realizujący logikę wewnętrznego Urzędu Certyfikacji (CA). Funkcja `ca_service` automatyzuje proces wydawania tożsamości cyfrowej użytkownikowi. W pierwszej kolejności sprawdza istnienie zmiennej środowiskowej `ROOT_CA_PASSWORD`, następnie generuje parę kluczy RSA (2048-bit) oraz tworzy żądanie podpisania certyfikatu (CSR), uzupełniając je danymi podmiotu (`Common Name` = nazwa użytkownika). Następnie, wykorzystując klucz prywatny Root CA podpisuje certyfikat użytkownika, nadając mu roczny okres ważności oraz odpowiednie rozszerzenia (`KeyUsage: digitalSignature`, `BasicConstraints: CA=False`). Certyfikat wzbogacono o rozszerzenie Subject Key Identifier (SKI), które poprzez unikalny identyfikator (skrót) klucza publicznego optymalizuje proces budowania ścieżki zaufania przez oprogramowanie weryfikujące. Końcowy produkt, czyli klucz prywatny użytkownika wraz z łańcuchem certyfikatów, jest pakowany do bezpiecznego kontenera PKCS#12, zaszyfrowanego hasłem użytkownika.

`signer.py` – Moduł implementujący logikę nakładania podpisu elektronicznego w standardzie PAdES. Funkcja `sign_pdf_service` wykorzystuje bibliotekę `pyHanko` do obsługi operacji na plikach PDF. Wczytuje ona tożsamość użytkownika (klucz prywatny i certyfikat) z kontenera PKCS#12, a następnie tworzy strukturę podpisu z włączoną opcją LTA (Long Term Validation), co przygotowuje strukturę dokumentu do przyszłej rozbudowy o znaczniki czasu (LTV - Long Term Validation). Kluczowym elementem jest użycie klasy `IncrementalPdfFileWriter`, która dopisuje podpis na końcu pliku (incremental update), nie naruszając oryginalnej zawartości dokumentu, co jest krytyczne dla zachowania jego integralności.

`validator.py` – Moduł służący do kryptograficznej weryfikacji podpisów. Funkcja `verify_pdf_service` tworzy kontekst zaufania (instancja klasy `ValidationContext`), wskazując certyfikat Root CA jako jedyny zaufany punkt odniesienia (Trust Anchor). Następnie iteruje po wszystkich podpisach osadzonych w dokumencie PDF, sprawdzając ich poprawność matematyczną, ważność certyfikatu oraz integralność pliku (intact). Moduł zwraca zbiorczy status walidacji (Boolean) oraz szczegółową listę metadanych dla każdego podpisu, która służy następnie do wygenerowania raportu.

`report_generator.py` - Moduł odpowiedzialny za wizualizację wyników weryfikacji. Funkcja `report_generator_service` przetwarza listę słowników zawierających dane

walidacyjne (otrzymanych z `verify_pdf_service`) i przy użyciu biblioteki `fpdf2` generuje czytelny raport w formacie PDF. Dokument ten zawiera szczegółowe informacje o każdym podpisie znalezionym w pliku, w tym tożsamość podpisującego, datę złożenia podpisu, użyty algorytm haszujący oraz status integralności dokumentu (informację, czy plik nie był modyfikowany po podpisaniu).

E. Inicjalizacja i Punkt Wejścia Aplikacji

`__init__.py` - Plik techniczny inicjalizujący pakiet języka Python, który umożliwia traktowanie katalogu jako modułu oraz zarządza przestrzenią nazw, pozwalając na poprawne rozwiązywanie zależności i importowanie komponentów wewnątrz struktury aplikacji. Powinien zostać pusty.

`main.py` - Punkt wejściowy serwera, tworzący instancję `FastAPI` i scalający wszystkie routery w jednym pliku.

F. Skrypt Inicjalizacji Infrastruktury PKI (`init_ca.py`)

Skrypt odpowiedzialny za inicjalizację infrastruktury klucza publicznego (PKI) poprzez wygenerowanie głównego urzędu certyfikacji (Root CA), stanowiącego "Kotwicę Zaufania" dla całego systemu. Funkcja `generate_root_ca` została zaprojektowana jako mechanizm jednorazowego użytku – przed rozpoczęciem działania weryfikuje fizyczną obecność plików w katalogu certyfikatów, aby zapobiec przypadkowemu nadpisaniu istniejącej tożsamości CA. Skrypt generuje parę kluczy RSA o zwiększonej długości (4096 bitów) niż standardowe dla użytkowników (2048 bitów) dla zapewnienia najwyższego poziomu bezpieczeństwa. Następnie tworzy certyfikat typu Self-Signed (podpisany własnym kluczem prywatnym) z 10-letnim okresem ważności. Konfiguracja certyfikatu obejmuje krytyczne rozszerzenia X.509 definiujące jego rolę: `BasicConstraints: ca=True` (nadające techniczne uprawnienia urzędu certyfikacji) oraz `KeyUsage` z flagami `key_cert_sign` i `crl_sign` (umożliwiające podpisywanie certyfikatów użytkowników i list unieważnień). Certyfikat wzbogacono również o `SubjectKeyIdentifier` dla optymalizacji weryfikacji. Wygenerowany klucz prywatny jest serializowany i szyfrowany symetrycznie algorytmem AES przy użyciu hasła pobranego ze zmiennej środowiskowej `ROOT_CA_PASSWORD`, a następnie zapisywany wraz z certyfikatem publicznym w ścieżkach zdefiniowanych w `core.config`

V. HARMONOGRAM REALIZACJI PROJEKTU

Proces wytwórczy systemu został podzielony na osiem etapów (kamieni milowych), realizowanych w dwutygodniowych iteracjach (sprintach).. Szczegółowy plan przedstawiono w Tabeli I.

TABELA I

PLAN WYDAWNICZY I HARMONOGRAM PRAC (RELEASE PLAN)

Etap	Okres Realizacji	Zakres Realizowanych Zadań	Kryterium Akceptacji (Definition of Done)
M1 - Koncept i Research	13.10.2025 – 26.10.2025	Wstępne zdefiniowanie założeń projektowych. Analiza literatury oraz dobór bibliotek i stosu technologicznego.	Zdefiniowany koncept systemu. Zgromadzona literatura źródłowa.
M2 - Projektowanie i Konfiguracja	27.10.2025 – 09.11.2025	Opracowanie diagramu Use Case (UML), definicja zakresu MVP. Inicjalizacja repozytorium Git, konfiguracja środowiska oraz analiza infrastruktury PKI i standardu X.509.	Utworzone repozytorium. Gotowy diagram przypadków użycia. Zrozumienie mechanizmów PKI.
M3 - Analiza Technologiczna	10.11.2025 – 23.11.2025	Badanie struktury plików PDF (zakres bajtów, polityka modyfikacji). Nauka bibliotek `PyHanko` i `FastAPI` w kontekście podpisu cyfrowego.	Zrozumienie struktury logicznej PDF oraz metod implementacji podpisu w Pythonie.
M4 - Fundament Backendu	24.11.2025 – 07.12.2025	Implementacja serwera `FastAPI`. Opracowanie endpointów do transferu plików (`POST`). Generowanie	Działający serwer przyjmujący pliki PDF. Wygenerowany lokalnie certyfikat testowy.

		testowych certyfikatów `.p12` na potrzeby deweloperskie.	
M5 - Moduł Kryptograficzny (Podpis)	09.12.2025 – 21.12.2025	Implementacja endpointu `/sign`. Logika nakładania podpisu PAdES z wykorzystaniem klucza prywatnego.	Plik PDF podpisany cyfrowo, poprawnie walidowany przez zewnętrzne oprogramowanie (np. Adobe Reader, Okular).
M6 - Weryfikacja i Autoryzacja	22.12.2025 – 04.01.2026	Implementacja endpointu `/verify` (walidacja kryptograficzna i integralności). Generowanie raportów PDF. Zabezpieczenie API tokenami JWT (`/auth, /login`).	API zwraca poprawny status walidacji (True/False) oraz generuje raport. Endpointy chronione autoryzacją.
M7 - Klient CLI i Baza Danych	05.01.2026 – 18.01.2026	Integracja z bazą danych MongoDB. Implementacja interfejsu wiersza poleceń (`client.py`) obsługującego pełny proces (wysłanie -> podpis -> odbiór).	Działający scenariusz *end-to-end* wywoływany z poziomu terminala.
M8 - Finalizacja	19.01.2026 – 26.01.2026	Testy całościowe systemu. Opracowanie instrukcji wdrożeniowej (`README.md`) oraz redakcja ostatecznej dokumentacji technicznej.	Brak błędów krytycznych. Dokumentacja zgodna ze standardami IEEE.

VI. PROCES REALIZACJI PROJEKTU

M1: Koncept i Research (13.10.2025 – 26.10.2025)

Pierwotna geneza projektu wywodziła się z chęci stworzenia narzędzia integrującego się z API platformy Discord. Wstępnym założeniem była realizacja bota o charakterze finansowym, który na żądanie użytkownika dostarczałby dane giełdowe. Wizja funkcjonalności zakładała mechanizm, w którym po wpisaniu komendy (np. nazwy spółki), bot zwracałby wygenerowany wykres. Od strony technologicznej planowałem oprzeć to rozwiązanie na bibliotece `discord.py` oraz narzędziach do wizualizacji danych, takich jak `matplotlib, seaborn` czy `plotly`.

Jednakże tydzień między 20 a 26 października 2025 roku okazał się kluczowy dla ukształtowania ostatecznego charakteru projektu. Bezpośrednią inspiracją do zmiany kierunku była sugestia prowadzącego, wskazująca kryptografię jako potencjalny obszar projektowy. Zaintrygowany tą tematyką oraz chęcią poszerzenia swojej wiedzy w dziedzinie, która dotychczas była dla mnie enigmatyczna, zdecydowałem się na rezygnację z tworzenia bota. Odwołując się do wiedzy z zajęć z cyberbezpieczeństwa, zdefiniowałem nowy cel: stworzenie bezpiecznego systemu do cyfrowego podpisywania i weryfikacji plików, koncentrując się w pierwszej fazie na dokumentach PDF.

Realizację nowego założenia rozpocząłem od zgłębienia teoretycznych podstaw podpisu elektronicznego, w tym zasad działania kryptografii asymetrycznej (klucze prywatne i publiczne, algorytm RSA). Następnym krokiem był dobór stacku technologicznego, co stanowiło wyzwanie ze względu na specyfikę tematu. Do obsługi warstwy kryptograficznej wybrałem bibliotekę `cryptography` (generowanie kluczy i certyfikatów) oraz specjalistyczne narzędzie `pyHanko`, które obsługuje standard PAdES, certyfikaty X.509 oraz weryfikację podpisów. Do manipulacji samymi plikami PDF wyselekcjonowałem biblioteki `FPDF2` lub `ReportLab`.

W kwestii technologii backendowej przeprowadziłem analizę dostępnych frameworków webowych, rozważając `Django`, `Flask` oraz `FastAPI`. Ostatecznie, zdecydowałem się na `Django`. Wybór ten podyktowany był moim wcześniejszym doświadczeniem z przedmiotu Projektowanie Aplikacji Webowych oraz gotową strukturą frameworka, co pozwoli sprawniej zarządzać złożonością nowego projektu.

M2: Projektowanie i Konfiguracja (27.10.2025 – 09.11.2025)

Okres od 27 października do 2 listopada 2025 roku poświęciłem na sformalizowanie założeń projektu, któremu nadałem nazwę "File Cert", oraz na zdobycie niezbędnej wiedzy teoretycznej. Kluczowym krokiem było zdefiniowanie MVP (Minimum Viable Product), czyli minimalnego zestawu funkcjonalności niezbędnego do działania

aplikacji. Ustaliłem, że system opierać się będzie na trzech głównych endpointach: przesyłaniu pliku na serwer (`/upload`), kryptograficznym podpisywaniu dokumentu certyfikatem X.509 (`/sign`) oraz weryfikacji poprawności podpisu (/`verify`).

Równolegle przeprowadziłem pogłębioną analizę standardu PKI (Public Key Infrastructure). Skupiłem się na zrozumieniu mechanizmów kryptografii asymetrycznej, w której bezpieczeństwo gwarantuje użycie klucza prywatnego do podpisu oraz klucza publicznego do jego weryfikacji. Zapoznałem się również ze strukturą certyfikatu X.509, analizując przechowywane w nim dane, takie jak informacje o wystawcy czy okres ważności. Wiedza ta posłużyła mi do zredagowania oficjalnego abstraktu dokumentacji, streszczającego cel projektowy. Dodatkowo, za sugestią prowadzącego, rozszerzyłem planowany stack technologiczny o bibliotekę `pyCryptodome`, która ma wspierać aspekty kryptograficzne projektu.

Przeszedłem od teorii do przygotowania środowiska pracy. Skonfigurowałem środowisko deweloperskie, tworząc lokalne repozytorium Git połączone ze zdalnym repozytorium na GitHubie oraz wirtualne środowisko Python (f`ile_cert_venv`) w celu izolacji zależności. W pliku requirements.txt uwzględniłem wstępnie dobrane biblioteki: `pyCryptodome`, `cryptography`, `pyHanko` oraz `Django`, a także przygotowałem strukturę katalogów z wydzielonym miejscem na dokumentację techniczną.

Aby uporządkować wizję systemu, wykorzystałem program Draw.io do stworzenia diagramu w notacji UML. Wizualizacja ta pozwoliła mi upewnić się co do spójności założonego MVP. Zdefiniowałem aktora (Użytkownika) oraz konkretne przypadki użycia: “Rejestracja”, “Uwierzytelnianie”, “Transfer pliku”, “Podpisanie pliku” i “Weryfikacja podpisów na pliku”. Diagram ten stał się bezpośrednią mapą dla planowanych endpointów. Całość prac zwieńczyło opracowanie planu wydania (Release Plan), w którym projekt został podzielony na 8 dwutygodniowych sprintów, uwzględniających czas zarówno na rozwój kodu, jak i na niezbędną naukę.

M3: Analiza technologiczna (10.11.2025 – 23.11.2025)

Okres od 10 do 23 listopada 2025 roku poświęciłem na intensywne pogłębianie wiedzy technicznej oraz podjęcie kluczowej decyzji architektonicznej. Pierwszy tydzień skupiał się na niskopoziomowej analizie formatu PDF w celu zrozumienia technicznej realizacji podpisu cyfrowego. Zgłębiając specyfikację standardu PAdES (*PDF Advanced Electronic Signatures*), dowiedziałem się, że proces ten polega na dodaniu nowej struktury (*Signature Dictionary*) poprzez mechanizm tzw. przyrostowej aktualizacji (*Incremental Update*). Kluczowym elementem, który zrozumiałem, jest koncepcja *ByteRange* - tablicy definiującej zakresy bajtów objęte podpisem, która

chroni treść dokumentu, wyłączając jednocześnie samą wartość podpisu, aby uniknąć problemu rekurencji. Przeanalizowałem również rolę funkcji skrótu, w szczególności algorytmu SHA-256, który poprzez generowanie nieodwracalnego, 256-bitowego ciągu znaków zapewnia integralność i autentyczność przetwarzanych danych.

W drugim tygodniu, motywowany chęcią poznania nowocześniejszych rozwiązań oraz specyfiką projektu, zdecydowałem się na zmianę technologii backendowej. Po przeprowadzeniu researchu porównawczego między `Django`, `Flaskiem` a `FastAPI`, postanowiłem zmienić framework z `Django` na `FastAPI`. Decyzja ta podyktowana była przede wszystkim natywnym wsparciem dla asynchroniczności (przy użyciu serwera ASGI Uvicorn), co jest krytyczne dla wydajności systemu opartego na operacjach wejścia/wyjścia, takich jak przesyłanie i przetwarzanie plików PDF. Dodatkowym argumentam za wyborem `FastAPI` było automatyczne generowanie dokumentacji w Swagger UI, co znacząco przyspieszy testowanie API. Równolegle rozpocząłem praktyczną naukę biblioteki `pyHanko`, koncentrując się na analizie modułu `pyhanko.sign`, odpowiedzialnego za techniczne osadzanie pól formularzy z podpisem.

M4: Fundament Backendu (24.11.2025 – 07.12.2025)

W okresie od 24 listopada do 7 grudnia 2025 roku przystąpiłem do realizacji kamienia milowego M4, którego głównym celem było uruchomienie środowiska deweloperskiego oraz przygotowanie fundamentów pod kod aplikacji. Pierwszy tydzień poświęciłem na naukę i konfigurację frameworka FastAPI. Zaktualizowałem plik requirements.txt, zastępując zależności `Django` zależnościami `FastAPI`. Po utworzeniu pliku startowego main.py, pomyślnie uruchomiłem lokalny serwer i zweryfikowałem działanie automatycznej dokumentacji w `Swagger UI`.

Równocześnie zaprojektowałem modułową strukturę katalogów, aby zapewnić skalowalność projektu. Kod źródłowy umieściłem w katalogu `app/`, wydzielając podkatalogi: `routers/` (definicje endpointów z wykorzystaniem klasy `APIRouter`), /core (konfiguracja globalna), `models/` (klasy walidacyjne `Pydantic`) oraz `services/` (logika biznesowa, w tym obsługa podpisów). Przygotowałem również infrastrukturę dla danych: folder `storage/` na przesyłane pliki PDF oraz `certs/` na klucze kryptograficzne.

W drugim tygodniu przeszedłem do prac programistycznych i konfiguracji bezpieczeństwa. Zaimplementowałem pierwszy endpoint `upload/`, obsługujący żądania metodą `POST`. Wyposażyłem go w walidację akceptującą wyłącznie pliki PDF oraz mechanizm zapisu do katalogu storage/ przy użyciu biblioteki `shutil`.

Poprawność działania endpointu potwierdziłem testem manualnym. Kluczowym elementem tego etapu było również wdrożenie narzędzia OpenSSL. Wygenerowałem za jego pomocą parę kluczy RSA (2048 bit) oraz certyfikat typu Self-Signed (X.509). Następnie, zgodnie z wymogami biblioteki `pyHanko`, skonwertowałem klucz prywatny i certyfikat do bezpiecznego formatu PKCS#12 (.p12) - zaszyfrowanego kontenera binarnego, chroniącego tożsamość cyfrową systemu.

M5: Moduł Kryptograficzny (Podpis) (09.12.2025 – 21.12.2025)

Realizację kamienia milowego M5 rozpocząłem od izolacji środowiska pracy poprzez utworzenie dedykowanej gałęzi w systemie kontroli wersji. W pierwszej fazie prac skoncentrowałem się na logice biznesowej, tworząc funkcję `sign_pdf_service` w pliku `app/services/signer.py`. Wykorzystując bibliotekę `pyHanko`, oparłem mechanizm podpisu na klasie `IncrementalPdfFileWriter`. Pozwoliło to na zastosowanie techniki "przyrostowej aktualizacji" (*Incremental Update*), dzięki której struktura podpisu jest dopisywana na końcu pliku PDF, nie naruszając jego pierwotnej zawartości binarnej - co jest kluczowe dla zachowania integralności dokumentu. Skonfigurowałem również metadane zgodne ze specyfikacją PAdES, definiując parametry takie jak cel podpisu (reason), lokalizację oraz flagę `use_pades_lta=True` (Long Term Archival), wspierającą długoterminową ważność podpisu.

W drugim tygodniu prac udostępniłem logikę poprzez endpoint `API /sign`. Endpoint ten, zamiast przesyłać plik, przyjmuje w formacie JSON nazwę dokumentu (znajdującego się już na serwerze) oraz dane uwierzytelniające do certyfikatu. Etap ten okazał się wyzwaniem ze względu na złożone problemy z kompatybilnością bibliotek kryptograficznych, które objawiły się podczas testów integracyjnych.

Napotkałem dwa krytyczne błędy. Pierwszy, związany z obsługą kontenerów PKCS#12 (`AttributeError: 'NoneType' object...`), wynikał z problemów biblioteki `pyHanko` z ekstrakcją klucza prywatnego z certyfikatu Self-Signed. Drugi problem, tzw. "piekło zależności" (*Dependency Hell*), objawił się błędem `AttributeError: 'builtin_function_or_method' object has no attribute 'algorithm'`. Zdiagnozowałem konflikt między `pyHanko` a nowszą wersją biblioteki cryptography, która zmieniła API dostępu do algorytmów haszujących. Problem rozwiązałem poprzez przymusową aktualizację obu bibliotek do najnowszych, kompatybilnych wersji.

Po ustabilizowaniu środowiska system poprawnie wygenerował plik `<filename>_signed.pdf`. Weryfikacja w zewnętrznym oprogramowaniu Adobe Acrobat Reader potwierdziła sukces - program wyświetlił panel informujący o poprawnym podpisie cyfrowym (pochodzącym od nieznanego urzędu certyfikacji, co

jest zachowaniem oczekiwanym dla certyfikatów Self-Signed). Prace zakończyłem scaleniem gałęzi M5 z główną gałęzią master.

M6: Weryfikacja i Autoryzacja (22.12.2025 – 04.01.2026)

Realizację kamienia milowego M6 rozpocząłem od utworzenia dedykowanej gałęzi, koncentrując się początkowo na stworzeniu modułu weryfikacyjnego (`validator.py`) oraz logiki funkcji `verify_pdf_service`. Fundamentem mechanizmu weryfikacji stała się koncepcja "Kotwicy Zaufania" (*Trust Anchor*). Skonfigurowałem system tak, aby autoryzował podpisy w oparciu o certyfikat `root_ca.crt`, co umożliwia kryptograficzne potwierdzenie tożsamości poprzez sprawdzenie całego łańcucha certyfikacji. Istotnym wyzwaniem technicznym było obsłużenie dokumentów wielokrotnie podpisanych; wdrożyłem algorytm iterujący przez wszystkie osadzone podpisy, gdzie flaga walidacyjna `is_all_valid` przyjmuje wartość negatywną, jeśli choć jeden podpis w historii dokumentu jest nieprawidłowy.

W drugim tygodniu prac przeprowadziłem kluczową refaktoryzację endpointu `/verify`, zmieniając metodę `HTTP` z `GET` na `POST`. Zmiana ta była podyktowana ograniczeniami metody `GET` w zakresie przesyłania danych oraz koniecznością zwracania, zamiast prostych wartości logicznych, uporządkowanych metadanych w formacie JSON (tożsamość, status, data). Dane te stały się wsadem dla nowo utworzonego serwisu `report_generator_service`. Wykorzystując bibliotekę `fpdf2`, zaimplementowałem mechanizm mapowania surowych wyników JSON na czytelne raporty PDF (`_report.pdf`), zawierające szczegóły dotyczące integralności i użytych algorytmów.

Równolegle uporządkowałem architekturę projektu. W pliku core/config.py scentralizowałem stałe konfiguracyjne (np. `BASE_URL`), co ułatwi przyszłe wdrożenie produkcyjne. Zbudowałem również kompletny moduł bezpieczeństwa w `core/security.py`, wykorzystując biblioteki `passlib` (haszowanie haseł algorytmem bcrypt) oraz jose (generowanie tokenów JWT). Zaimplementowałem funkcje `get_password_hash` oraz `verify_password`, a także mechanizm generowania tokenów JWT (`create_access_token`) podpisanych kluczem HS256. Całość zwieńczyło uruchomienie routera routers/auth.py z endpointami `/register` i `/login` (zgodnymi ze standardem OAuth2), co pozwoliło na bezpieczną rejestrację i autoryzację użytkowników. Prace zakończyłem scaleniem gałęzi M6 z główną gałęzią master.

M7: Klient CLI i Baza Danych (05.01.2026 – 18.01.2026)

Realizację kamienia milowego M7 rozpocząłem od gruntownych porządków w strukturze projektu i refaktoryzacji kodu na nowej gałęzi repozytorium. W celu poprawy czytelności przeniosłem definicje modeli danych (`Pydantic`) z plików routerów do dedykowanego modułu schemas.py. Istotnym usprawnieniem w logice biznesowej była modyfikacja serwisu raportującego - system został zabezpieczony przed awarią w przypadku weryfikacji niepodpisanych plików, generując teraz dokument PDF z czytelnym komunikatem "`No signatures found`".

W obszarze usług certyfikacji (`services/ca_service.py`) dokonałem całkowitego uniezależnienia aplikacji od zewnętrznych wywołań systemowych OpenSSL, przepisując logikę generowania Root CA w czystym Pythonie przy użyciu biblioteki cryptography. Wdrożyłem obsługę standardu SKI (*Subject Key Identifier*) w certyfikatach X.509, co optymalizuje proces weryfikacji poprzez jednoznaczną identyfikację klucza publicznego (analogicznie do indeksów w bazach danych). Zaimplementowałem również mechanizm CSR (*Certificate Signing Request*) oraz zmieniłem format eksportu tożsamości cyfrowej na standard PKCS#12 (.p12), pakujący klucz prywatny i certyfikat w jeden bezpieczny, zaszyfrowany kontener.

W drugim tygodniu prac stanąłem przed decyzją architektoniczną dotyczącą przetwarzania plików "w locie" (in-memory) w celu eliminacji zapisu na dysku. Ze względu na ograniczone ramy czasowe projektu, zdecydowałem się utrzymać katalog `storage/`, skupiając się na innych kluczowych aspektach. Zastąpiłem tymczasowe przechowywanie danych w pamięci integracją z nierelacyjną bazą danych MongoDB, co zapewniło trwałość danych użytkowników i elastyczność schematu. Rozbudowałem również logikę autoryzacji: naprawiłem błąd walidacji unikalności nazw użytkowników oraz zintegrowałem proces rejestracji (`/register`) z usługą CA. Dzięki temu automatycznie generuje i przypisuje użytkownikowi cyfrową tożsamość. Dostęp do chronionych zasobów zabezpieczyłem zależnością `get_current_user`, weryfikującą tokeny JWT.

Końcowymm etapem było ukończenie prac nad klientem konsolowym (`client.py`) opartym na bibliotece `Typer` i `Rich`. Narzędzie to integruje pełny proces biznesowy: od rejestracji i logowania (z lokalnym zapisem sesji), przez przesyłanie plików (`upload`), aż po ich podpisywanie i weryfikację. Zmodyfikowałem przy tym endpoint /sign, który teraz bezpośrednio zwraca strumień podpisanego pliku PDF, zamiast łącza do zasobu. Klient CLI posiada pełną obsługę błędów, co znacząco ułatwia interakcję z systemem i diagnostykę problemów.

M8: Finalizacja (19.01.2026 – 26.01.2026)

Ostatni tydzień prac projektowych poświęciłem na stabilizację rozwiązania, poprawę jakości kodu oraz przygotowanie materiałów na prezentację. Był to etap porządkowania i dokumentowania, mający na celu uczynienie projektu nie tylko funkcjonalnym, ale i czytelnym dla osób trzecich. Przeprowadziłem gruntowny przegląd kodu źródłowego (zarówno serwera `FastAPI`, jak i klienta CLI), uzupełniając go o szczegółowe komentarze oraz `docstrings` zgodne ze standardem `Sphinx`. Stworzyłem również kompleksowy plik `README.md`, zawierający instrukcję instalacji zależności, konfiguracji zmiennych środowiskowych (`.env`) oraz uruchomienia systemu, co czyni aplikację gotową do wdrożenia na dowolnej maszynie.

W ramach przygotowań do prezentacji opracowałem schemat architektury systemu. Diagram ten wizualizuje przepływ danych pomiędzy klientem a serwerem ASGI, uwzględniając kluczowe procesy: rejestrację (zwracającą kontener PKCS#12), uwierzytelnianie (wymiana nazwy użytkownika i hasła na token JWT) oraz zabezpieczone operacje na plikach (`/upload, /sign, /verify`). Materiał ten, wraz z opisem technologii, przypadków użycia oraz wyjaśnieniem procesu podpisu cyfrowego, stał się trzonem prezentacji multimedialnej. Równolegle przeprowadziłem manualne testy końcowe (End-to-End), weryfikując tzw. "happy path" - od założenia konta po wygenerowanie raportu weryfikacyjnego PDF. Przy okazji poprawiłem obsługę wyjątków w kliencie CLI, aby komunikaty błędów były bardziej przyjazne dla użytkownika.

Projekt został sfinalizowany 26 stycznia 2026 roku podczas prezentacji i demonstracji. Przedstawiłem pełną ścieżkę działania systemu w czasie rzeczywistym - od uruchomienia serwera po weryfikację dokumentu. Kluczowym elementem pokazu był dowód bezpieczeństwa: celowe usunięcie jednego znaku z tokenu JWT natychmiast zablokowało dostęp do operacji na plikach, potwierdzając skuteczność mechanizmu autoryzacji. Dodatkowo udowodniłem, że moje rozwiązanie jest rozpoznawane przez zewnętrzne programu, otwierając podpisany plik w niezależnym programie "Okular", który poprawnie rozpoznał matematyczną ważność złożonego podpisu cyfrowego oraz wyświetlił metadane podpisu.

LITERATURA

[1] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," IETF, RFC 5280, May 2008. [Online]. Dostępne: https://datatracker.ietf.org/doc/html/rfc5280.

[2] M. Moriarty, M. Nystrom, S. Parkinson, and A. Rusch, "PKCS #12: Personal Information Exchange Syntax v1.1," IETF, RFC 7292, July 2014. [Online]. Dostępne: https://datatracker.ietf.org/doc/html/rfc7292.

[3] S. Ramírez, "FastAPI Documentation," 2024. [Online]. Dostępne: https://fastapi.tiangolo.com/.

[4] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF, RFC 7519, May 2015. [Online]. Dostępne: https://datatracker.ietf.org/doc/html/rfc7519.

[5] ETSI, "Electronic Signatures and Infrastructures (ESI); PAdES digital signatures," European Telecommunications Standards Institute, ETSI EN 319 142-1 V1.1.1, Apr. 2016. [Online]. Dostępne: https://www.etsi.org/deliver/etsi_en/319100_319199/31914201/01.02.01_60/en_31914201v010201p.pdf.

[6] S. Colvin, "Pydantic Documentation," 2024. [Online]. Dostępne: https://docs.pydantic.dev/.

[7] M. Valvekens, "PyHanko: Digital signatures for PDF documents in Python," Documentation, 2024. [Online]. Dostępne: https://pyhanko.readthedocs.io/.

[8] The Python Cryptographic Authority, "Cryptography 42.0.5 Documentation," 2024. [Online]. Dostępne: https://cryptography.io/.

[9] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996, ch. 1, pp. 1-5.

[10] H. T. De la To, "PyCryptodome Documentation," 2024. [Online]. Dostępne: https://www.pycryptodome.org/.

[11] S. Ramírez, "Typer Documentation," 2024. [Online]. Dostępne: https://typer.tiangolo.com/.

[12] W. McGugan, "Rich Documentation," 2024. [Online]. Dostępne: https://rich.readthedocs.io/.

[13] L. Rebuffat, "fpdf2 Documentation," 2024. [Online]. Dostępne: https://py-pdf.github.io/fpdf2/.