Population Records Department

AUTHORS:

Marcin Sitarz

Łukasz Wdowiak

SUBSTANTIVE SUPPORT:

Roman Ptak, Ph.D.

Table of contents

1. Introduction

The Population Records Department is a public administration body responsible for collecting, updating and storing personal data of citizens and recording significant events such as births, marriages and deaths. Its main purpose is to create and manage a population register, which is used to determine the identification and status of citizens in a municipality.

1.1. Purpose of the project

The purpose of the project is to build a database for the Population Records Department and a website to automate the work of the clerk.

1.2. Scope of the project

Our task is to analyze the requirements, design the database structure, choose the appropriate technologies, create the database schema, implement it as well as the web application.

2. Requirements analysis

2.1. Description of operation and logical diagram of the system

An authorized employee of the office approves applications for: entry of a new person in the register, excerpt of a person from the register, change of data (name, surname, gender, place of residence, marital status), assigning or changing a PESEL number, declaration of a person as deceased. Each person from the register has a set of basic data enabling identification and finding of this person, e.g. name, surname, date of birth, place of birth, PESEL number, etc. Each registered person is assigned an account, from which it is possible to check their personal data and submit applications. Requests go through an initial validation of the data by the system for correctness and are then approved by an employee. Access to your personal data is limited to authorized personnel only.

2.2. Functional requirements

- A user residing in the municipality submits an application for entry of his/her data in the register of residents
- A user who is the legal guardian of a newborn child submits an application for the child's entry in the register of residents
- The user has the option of submitting an application for a PESEL number
- The user has the option to submit a request for a change of the PESEL number in the event of rectification of the date of birth, gender change or incorrect data
- The user has the option to request a change of residence
- The user has the option to submit a request for a name change
- The user has the option to apply for a gender change
- The user has the option to submit an application for a change of marital status

- The user has access to his personal data through the website
- The user has access to his applications through the website
- The user has the option of submitting an application for an excerpt from the register of residents due to moving outside the municipality
- The user has the option to inform about the death of a family member, which results in the death certificate number being added to the deceased person

2.3. Non-functional requirements

2.3.1. Technologies and tools used

Employees of the office work on computers connected to the local network, which connects to the Internet. They do their work with the help of a website set up on a server, from which an external port is issued, through which they and users can connect. Employees have administrator accounts with the ability to manage requests, as long as they are not submitted by themselves. The database does not store the certificates directly, only their identification numbers. The database service is written using MySQL. Access to it is provided by a backend service written using Python FastAPI. On the other hand, JavaScript using the React library is responsible for the appearance of the website.

2.3.2. Database size requirements

It is assumed that applications will be the most numerous entity in the database, which will be counted in hundreds of thousands. The rest will be counted in the tens of thousands. There are 10 employees working in the Population Registration Department of the municipality. Each of them examines about several dozen applications a day. It is assumed that the number of people browsing the site at any one time will not be greater than 1000.

2.3.3. System security requirements

Users have the ability to log in to their accounts using their login and password generated by the system and delivered to their e-mail address. In addition, with each application you are required to pass Captcha. Only users with a generated JWT have access to the API.

2.3.4. Verification requirements

The data provided in the application by the user goes through a preliminary system verification consisting of: checking whether all fields are filled in, whether each field requiring a specific number of characters has them, whether only allowed characters have been entered. Then, the data approved by the system is verified by an employee of the office. On the basis of whether the application has been filled out correctly, the employee may reject or approve the application.

3. System design

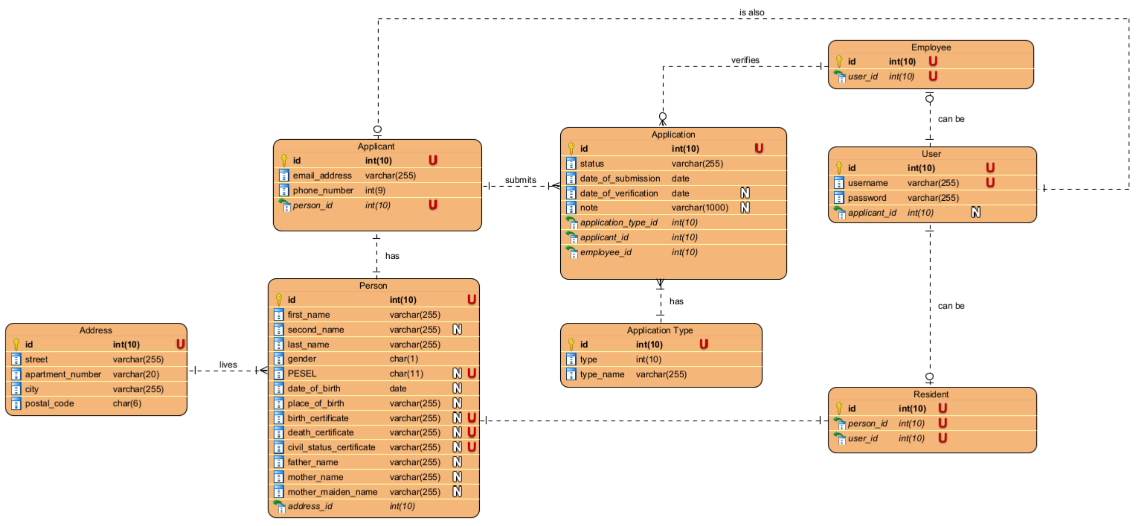
Database design and structure, mechanisms for ensuring the correctness of stored information, and data access control.

3.1. Database design

The database we create is a relational, non-temporal database based on the MySQL management system.

3.1.1. Physical model and data integrity constraints

Figure 1: Entity relationship diagram



3.1.2. Other elements of the schema — data processing mechanisms

1) Indexes:
   - Index 1: Table of "Persons" after "PESEL"
   - Index 2: Table of "Persons" after "last_name" + "first_name"
   - Index 3: Table "Application" after "applicant_id"
   - Index 4: Table "Application" after "application_type_id"
   - Index 5: "Application" table after "date_of_submission" descending
2) Example views:
   - View with information about requests and employees
     View Name – ApplicationEmployeeView
     Fields of view: application_id, status, date_of_submission, date_of_verification, note, application_type_name, applicant_id, employee_id, employee_first_name, employee_last_name
   - View with information about users and their requests
     View name – UserApplicationView
     Fields of view: user_first_name, user_last_name, user_PESEL, status, date_of_submission, date_of_verification, note, application_type_name

3) Component procedures:
    - Procedure for validating the PESEL number
    - Email Validation Procedure
    - Phone Number Validation Procedure
    - Postal Code Validation Procedure
    - Procedure for validating text data (whether the text data does not contain special characters and numbers)
    - Procedure for adding a new employee
4) Triggers:
    - Trigger that automatically creates a user after the petitioner submits their first application.
      Name: assign_employee_before_insert_application
      Trigger: Adding a new request to the Application table.
      Description of operation: When adding a new request, the trigger assigns the employee with the fewest requests to it. If this is the first request of the petitioner, the trigger creates a user account for him in the User table, generating a random password and using his PESEL number as the username.
    - A trigger that registers the petitioner as a resident after the application is accepted.
      Name: after_application_update
      Trigger: Update the request in the Application table.
      Description of activity: After changing the status of the application to "approved" and when the type of application corresponds to an entry in the register of residents, the trigger adds the petitioner to the Resident table, registering him as a resident.
    - A trigger that validates personal data before being added to the Person table.
      Name: before_insert_person
      Trigger: Adding a new record to the Person table.
      Description of activity: Before adding a person, the trigger checks the correctness of data such as PESEL, names, surnames, place of birth, using validation procedures. This prevents incorrect or incomplete personal information from being added.
    - A trigger that validates the applicant's data before adding it to the Applicant table.
      Name: before_insert_applicant
      Trigger: Adding a new record to the Applicant table.
      Functional description: This trigger validates the applicant's email address and phone number before adding them to the database. The use of validation procedures ensures that contact information is correct and up-to-date.
    - A trigger that validates the address before adding it to the Address table.
      Name: before_insert_address
      Trigger: Adding a new record to the Address table.
      Description: Before adding a new address, the trigger validates the postal code, street and city. This is important for maintaining correct and consistent address data in the system.

3.1.3. Design of database-level security mechanisms

At the very beginning, the data is validated by the component procedures. The data in the database is encrypted so that even in the event of a data leak, access to it is difficult. In addition, event logs are created to monitor the activity in the database - a system for logging operations and tracking application activities. A backup is made every day, so that in the event of a failure of the disk/server storing the collected data, the system can be quickly brought back to its feet.

3.1.4. Access to individual tables

We distinguish roles such as: administrator, manager, employee

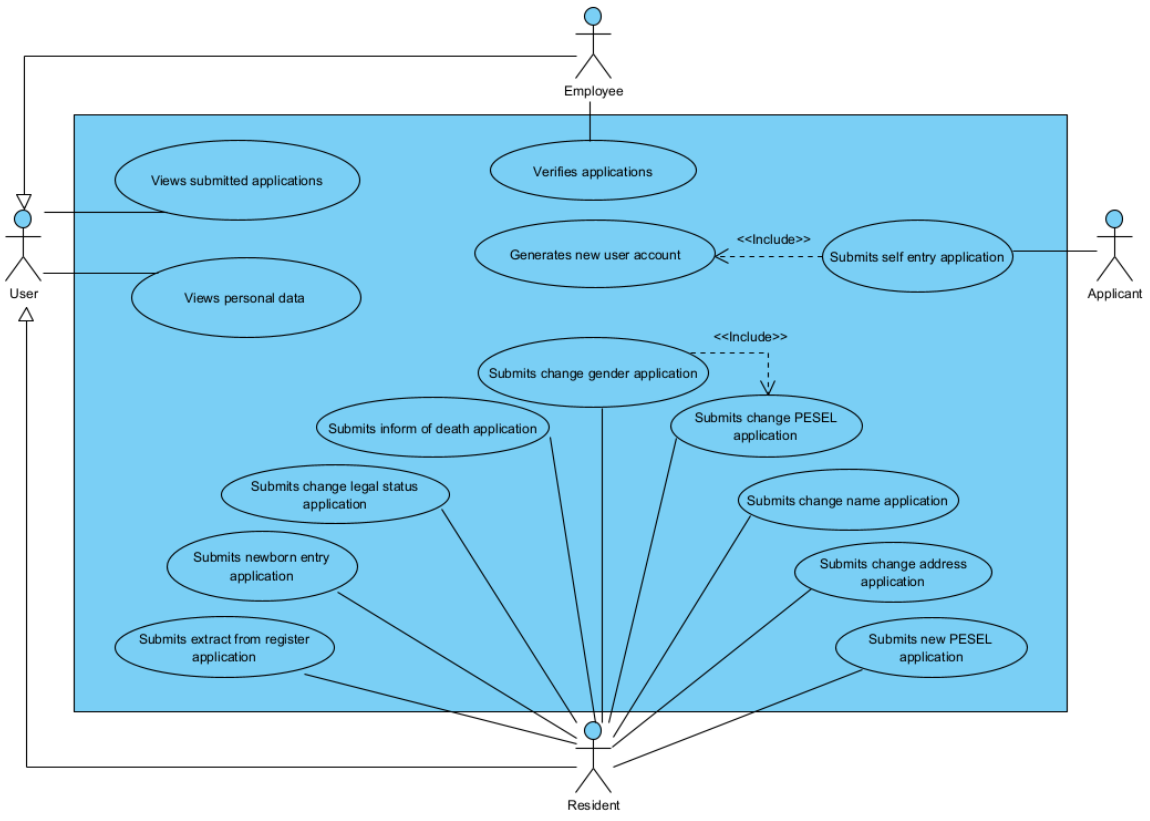
The administrator has access to all actions in all tables.

1) "Application" table:
    - The manager and employee have access to view, add, and update requests.
2) "Application Type" table:
    - The manager has access to view, add, update, and delete request types.
    - The employee has access to view request types.
3) "User" table:
    - The manager and employee have access to view, add, update, and delete users.
4) "Applicant" table:
    - The manager and employee have access to view, add, update, and delete petitioners.
5) Table "Resident":
    - The manager and employee have access to view, add, update, and delete residents.
6) "Person" table:
    - The manager and employee have access to view, add, update, and delete personal data.
7) "Address" table:
    - The manager and employee have access to view, add, update, and delete addresses.
8) Table "Employee":
    - The manager has access to view, add, update, and delete employees.

3.2. User Application Design

3.2.1. Application architecture and design diagrams

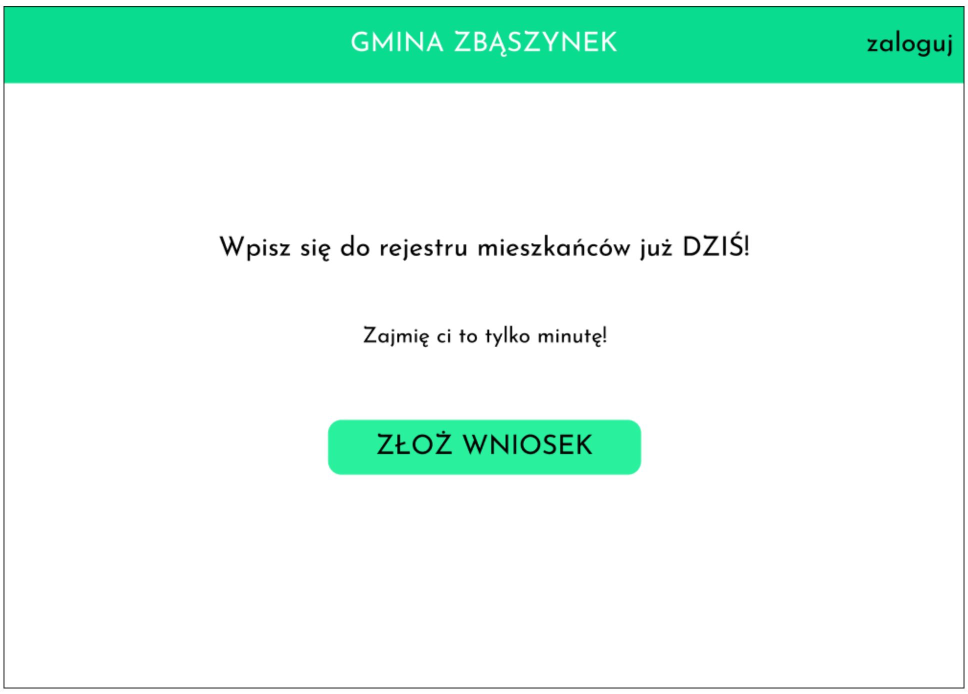
Image 2: Use case diagram


Employee
User
Applicant
Resident
Views submitted applications
Verifies applications
Generates new user account
<<Include>>
Submits self entry application
Views personal data
<<Include>>
Submits change gender application
Submits inform of death application
Submits change PESEL application
Submits change legal status application
Submits change name application
Submits newborn entry application
Submits change address application
Submits extract from register application
Submits new PESEL application

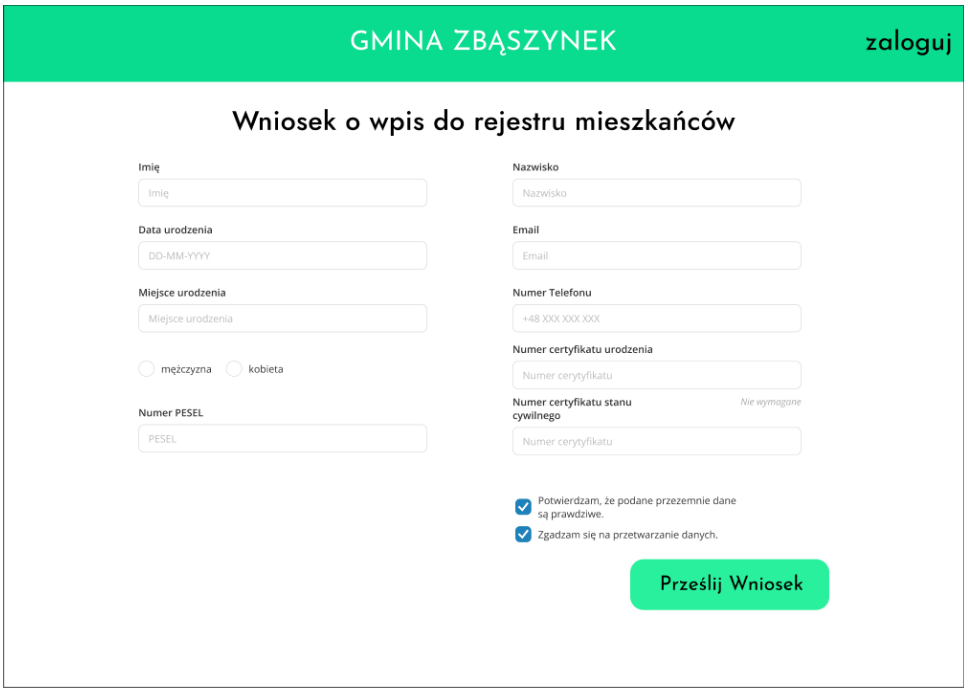
3.2.2. Graphical interface and menu structure

The graphic design of the application was created using the "figma.com" website.

Picture 3: Home page


GMINA ZBĄSZYNEK
zaloguj
Wpisz się do rejestru mieszkańców już DZIŚ!
Zajmię ci to tylko minutę!
ZŁOŻ WNIOSEK

Picture 4: Application form for registration


GMINA ZBĄSZYNEK zaloguj

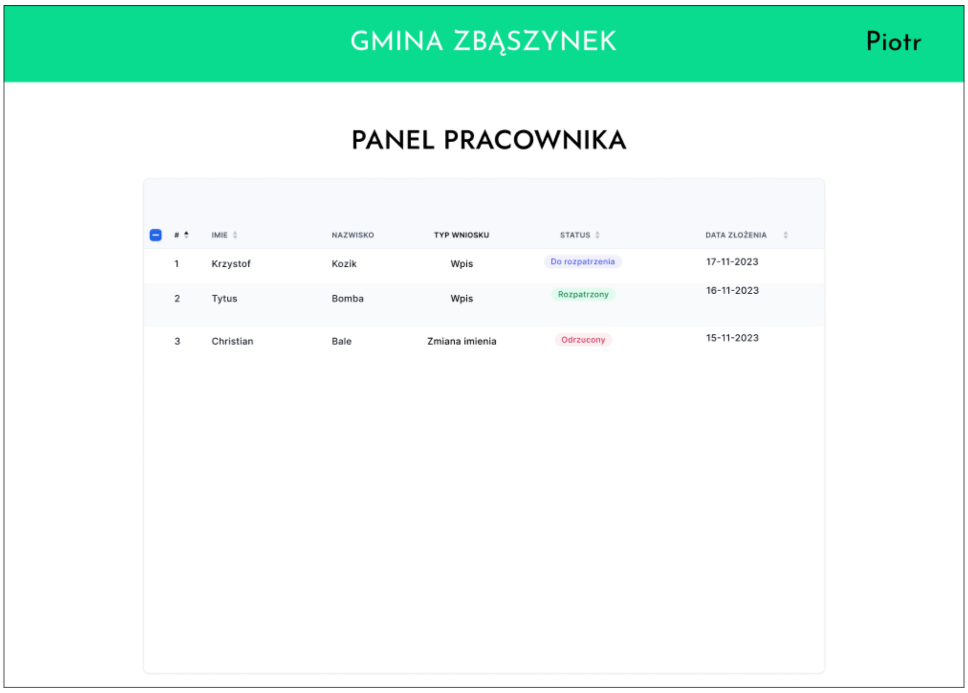
Wniosek o wpis do rejestru mieszkańców

Imię: Imię	Nazwisko: Nazwisko
Data urodzenia: DD-MM-YYYY	Email: Email
Miejsce urodzenia: Miejsce urodzenia	Numer Telefonu: +48 XXX XXX XXX
○ mężczyzna ○ kobieta	Numer certyfikatu urodzenia: Numer cerytyfikatu
Numer PESEL: PESEL	Numer certyfikatu stanu cywilnego (Nie wymagane): Numer cerytyfikatu

✓ Potwierdzam, że podane przezemnie dane są prawdziwe.

✓ Zgadzam się na przetwarzanie danych.

Picture 5: Employee panel


GMINA ZBĄSZYNEK Piotr

PANEL PRACOWNIKA

#	IMIE	NAZWISKO	TYP WNIOSKU	STATUS	DATA ZŁOŻENIA
1	Krzystof	Kozik	Wpis	Do rozpatrzenia	17-11-2023
2	Tytus	Bomba	Wpis	Rozpatrzony	16-11-2023
3	Christian	Bale	Zmiana imienia	Odrzucony	15-11-2023

3.2.3. Database connection method — database integration

Connection to the database will be done using the mysql-connector-library (native driver), thanks to which, after providing the correct login data to our database, we will be able to call SQL queries directly from our backend application. Then, our backend application will expose the API on a specific port, so that our frontend application will have access to the data from the database.

3.2.4. Application-level security design

To defend against attacks such as SQL injection, backend queries are parameterized. This is due to the fact that the user of our application does not have direct access to the database, so there is no way to inject SQL.

The forms that users fill out automatically check the correctness of the data they have entered. For example, it will display an error if you enter too many characters in your PESEL number or phone number, check if each field has been filled in and if the filled field contains allowed characters.

To defend against attacks that would be aimed at sending a large number of false requests, a captcha will be displayed at the time of an attempt to submit such a request, which must be properly passed in order for the application to be sent for further verification by the employee.

Logging in to your account generates a JWT that allows you to continue browsing our website. This token creates a so-called login session that will expire after a certain period of time.

4. Database System Implementation

4.1. Creating Tables and Defining Constraints

The tables and constraints were generated by Visual Paradigm.

4.1.1. Creating tables

- Creating the "User" table:

```
CREATE TABLE 'User' (id int(10) NOT NULL AUTO_INCREMENT, username varchar(255) NOT NULL UNIQUE,
password varchar(255) NOT NULL, applicant_id int(10), CONSTRAINT id PRIMARY KEY ( id));
```

- Creating the "Employee" table:

```
CREATE TABLE Employee (id int(10) NOT NULL AUTO_INCREMENT, user_id int(10) NOT NULL UNIQUE,
CONSTRAINT id PRIMARY KEY (id));
```

- Creating the "Application" table:

```
CREATE TABLE Application (id int(10) NOT NULL AUTO_INCREMENT, status varchar(255) DEFAULT "in
review" NOT NULL, date_of_submission DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL,
date_of_verification DATETIME ON UPDATE CURRENT_TIMESTAMP, note varchar(1000),
application_type_id int(10) NOT NULL, applicant_id int(10) NOT NULL, employee_id int(10) NOT
NULL, CONSTRAINT id PRIMARY KEY (id));
```

- Creating a "Person" table:

```
CREATE TABLE Person (id int(10) NOT NULL AUTO_INCREMENT, first_name varchar(255) NOT NULL,
second_name varchar(255), last_name varchar(255) NOT NULL, gender char(1) NOT NULL, PESEL
char(11) UNIQUE, date_of_birth date, place_of_birth varchar(255), birth_certificate varchar(255)
UNIQUE, death_certificate varchar(255) UNIQUE, civil_status_certificate varchar(255) UNIQUE,
father_name varchar(255), mother_name varchar(255), mother_maiden_name varchar(255), address_id
int(10) NOT NULL, CONSTRAINT id PRIMARY KEY (id));
```

- Creation of the "Application Type" table:

```
CREATE TABLE 'Application Type' (id int(10) NOT NULL AUTO_INCREMENT, type int(10) NOT NULL,
type_name varchar(255) NOT NULL, CONSTRAINT id PRIMARY KEY (id));
```

- Creating the "Applicant" table:

```
CREATE TABLE Applicant (id int(10) NOT NULL AUTO_INCREMENT, email_address varchar(255) NOT NULL,
phone_number int(9) NOT NULL, person_id int(10) NOT NULL UNIQUE,  CONSTRAINT id PRIMARY KEY
(id));
```

- Creating a "Resident" table:

```
CREATE TABLE Resident (id int(10) NOT NULL AUTO_INCREMENT, person_id int(10) NOT NULL UNIQUE,
user_id int(10) NOT NULL UNIQUE, CONSTRAINT id PRIMARY KEY (id));
```

- Creating the "Address" table:

```
CREATE TABLE Address (id int(10) NOT NULL AUTO_INCREMENT, street varchar(255) NOT NULL,
apartment_number varchar(20) NOT NULL, city varchar(255) NOT NULL, postal_code char(6) NOT NULL,
CONSTRAINT id PRIMARY KEY (id));
```

4.1.2. Creating Table Constraints

- Creating a constraint for the "Application" table on the basis of the "Application Type" table:

```
ALTER TABLE Application ADD CONSTRAINT FKApplicatio410864 FOREIGN KEY (application_type_id)
REFERENCES 'Application Type' (id);
```

- Creating a constraint for the "Applicant" table based on the "Person" table:

```
ALTER TABLE Applicant ADD CONSTRAINT FKApplicant692669 FOREIGN KEY (person_id) REFERENCES Person
(id);
```

- Creating a constraint for the "Application" table based on the "Applicant" table:

```
ALTER TABLE Application ADD CONSTRAINT FKApplicatio896957 FOREIGN KEY (applicant_id) REFERENCES
Applicant (id);
```

- Creating a restriction for the "Employee" table based on the "User" table:

```
ALTER TABLE Employee ADD CONSTRAINT FKEmployee631292 FOREIGN KEY (user_id) REFERENCES 'User'
(id);
```

- Creating a constraint for the "Resident" table on the basis of the "Person" table:

```
ALTER TABLE Resident ADD CONSTRAINT FKResident140468 FOREIGN KEY (person_id) REFERENCES Person
(id);
```

- Creating a constraint for the "Resident" table based on the "User" table:

```
ALTER TABLE  Resident ADD CONSTRAINT FKResident964261 FOREIGN KEY (user_id) REFERENCES 'User'
(id);
```

- Creating a constraint for the "Application" table based on the "Employee" table:

```
ALTER TABLE Application ADD CONSTRAINT FKApplicatio361712 FOREIGN KEY (employee_id) REFERENCES
Employee (id);
```

- Creating a restriction for the "Person" table based on the "Address" table:

```
ALTER TABLE Person ADD CONSTRAINT FKPerson829511 FOREIGN KEY (address_id) REFERENCES Address
(id);
```

- Creating a constraint for the "User" table based on the "Applicant" table:

```
ALTER TABLE 'User' ADD CONSTRAINT FKUser595632 FOREIGN KEY (applicant_id) REFERENCES Applicant
(id);
```

4.2 Implementation of data processing mechanisms

The mechanisms were written in SQL without the use of code generators.

4.2.1. Creating Indexes

- Index 1: Table of "Persons" after "PESEL"

```
CREATE INDEX idx_pesel ON Person (PESEL);
```

- Index 2: Table of "Persons" after "last_name" + "first_name"

```
CREATE INDEX idx_last_name_first_name ON Person (last_name, first_name);
```

- Index 3: Table "Application" after "applicant_id"

```
CREATE INDEX idx_applicant_id ON Application  (applicant_id);
```

- Index 4: Table "Application" after "application_type_id"

```
CREATE INDEX idx_application_type_id ON Application  (application_type_id);
```

- Index 5: "Application" table after "date_of_submission" descending

```
CREATE INDEX idx_date_of_submission_desc ON Application  (date_of_submission DESC);
```

4.2.2. Creating Views

- View 1: Information about requests and employees

```
CREATE VIEW ApplicationEmployeeInfo AS
SELECT
 a.id AS ApplicationID,
    a.AS ApplicationStatus,
    a.date_of_submission AS SubmissionDate,
    a.date_of_verification AS VerificationDate,
    a.note AS ApplicationNote,
 e.id AS EmployeeID,
    u.username AS EmployeeUsername,
    p.first_name AS EmployeeFirstName,
    p.last_name AS EmployeeLastName
FROM Application a
JOIN Employee e ON a.employee_id = e.id
JOIN 'User' u ON e.user_id = u.id
JOIN Person p ON u.applicant_id = p.id;
```

- View 2: Information about users and their requests

```
CREATE VIEW UserApplicationInfo AS
SELECT
 u.id AS UserID,
    u.username AS Username,
 a.id AS ApplicationID,
    a.AS ApplicationStatus,
    a.date_of_submission AS SubmissionDate,
    a.date_of_verification AS VerificationDate,
    a.note AS ApplicationNote,
    at.type_name AS ApplicationTypeName
FROM 'User' u
JOIN Applicant app ON u.applicant_id = app.id
JOIN Application a ON app.id = a.applicant_id
JOIN 'Application Type' at ON a.application_type_id = at.id;
```

- View 3: Number of apps for each app type

```
CREATE VIEW ApplicationTypeSummary AS
SELECT
    at.type_name,
    COUNT(a.id) AS NumberOfApplications
FROM 'Application Type' at
JOIN Application a ON at.id = a.application_type_id
GROUP BY at.type_name;
```

- View 4: Residents' contact information

```
CREATE VIEW ResidentContactInfo AS
SELECT
 r.id AS ResidentID,
    p.first_name,
    p.last_name,
    p.PESEL (Personal Identification Number),
    a.street,
    a.apartment_number,
    a.city,
    a.postal_code,
    app.email_address,
    app.phone_number
FROM Resident r
JOIN Person p ON r.person_id = p.id
JOIN Address a ON p.address_id = a.id
JOIN Applicant app ON p.id = app.person_id;
```

- View 5: Roles and access levels of each user

```
CREATE VIEW UserRoleAccess AS
SELECT
 u.id AS UserID,
    u.username,
    CASE
        WHEN e.id IS NOT NULL THEN 'Employee'
        WHEN app.id IS NOT NULL THEN 'Applicant'
        WHEN r.id IS NOT NULL THEN 'Resident'
        ELSE 'Unknown'
    END AS UserRole
FROM 'User' u
LEFT JOIN Employee e ON u.id = e.user_id
LEFT JOIN Applicant app ON u.applicant_id = app.id
LEFT JOIN Resident r ON u.id = r.user_id;
```

- View 6: Employees and their app assignments

```
CREATE VIEW EmployeeTaskAssignment AS
SELECT
 e.id AS EmployeeID,
    p.first_name AS EmployeeFirstName,
    p.last_name AS EmployeeLastName,
 a.id AS AssignedApplicationID,
    a.AS ApplicationStatus
FROM Employee e
JOIN Application a ON e.id = a.employee_id
JOIN 'User' u ON e.user_id = u.id
JOIN Person p ON u.applicant_id = p.id;
```

- View 7: Employees and the number of applications assigned to them

```
CREATE VIEW EmployeeApplicationCount AS
SELECT
    E.id AS employee_id,
    U.username AS employee_username,
    COUNT(A.(id) AS application_count
FROM
    Employee E
JOIN
    'user' u ON E.user_id = U.id
LEFT JOIN
    Application A ON E.id = A.employee_id
GROUP BY
    E.id, U.username;
```

4.2.3. Creation of procedures

- Procedure 1: Validation of the PESEL number

```
CREATE PROCEDURE ValidatePesel(IN p_PESEL CHAR(11))
BEGIN
 DECLARE error_message VARCHAR(255);

 IF LENGTH(p_PESEL) <> 11 OR NOT p_PESEL REGEXP '^[0-9]+$' THEN
        SET error_message = 'PESEL is incorrect';
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
    END IF;
END ;
```

- Procedure 2: Validate your email address

```
CREATE PROCEDURE ValidateEmail(IN p_email_address VARCHAR(255))
BEGIN
 DECLARE error_message VARCHAR(255);

    IF NOT p_email_address REGEXP '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[ A-Za-z]{2,}$' THEN
        SET error_message = 'Email address is incorrect';
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
    END IF;
END;
```

- Procedure 3: Validate the phone number

```
 CREATE PROCEDURE ValidatePhone(IN p_phone_number INT)
 BEGIN
 DECLARE error_message VARCHAR(255);

 IF LENGTH(p_phone_number) <> 9 OR NOT p_phone_number REGEXP '^[0-9]+$' THEN
         SET error_message = 'Phone number is incorrect';
         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
     END IF;
 END ;
```

- Procedure 4: Validate Text Data

```
CREATE PROCEDURE ValidateTextData(IN p_text_data VARCHAR(255), IN fieldName VARCHAR(255))
BEGIN
 DECLARE error_message VARCHAR(255);

    -- Validate that text data does not contain special characters and numbers
    IF p_text_data REGEXP '[^A-Za-z ]' THEN
        SET error_message = CONCAT('Field ', fieldName, ' is incorrect. It should contain only
letters.');
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
    END IF;
END;
```

- Procedure 5: Postal Code Validation

```
CREATE PROCEDURE ValidatePostalCode(IN p_postal_code VARCHAR(6))
BEGIN
 DECLARE error_message VARCHAR(255);

    IF NOT p_postal_code REGEXP '^[0-9]{2}-[0-9]{3}$' THEN
        SET error_message = 'Postal code is incorrect. It should be in the format xx-xxx where x
is a digit.';
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
    END IF;
END;
```

- Procedure 6: Adding a new employee

```
CREATE PROCEDURE AddEmployee(IN p_user_id INT)
BEGIN
 DECLARE error_message VARCHAR(255);

     -- Checking if there is a user
     IF NOT EXISTS (SELECT 1 FROM 'User' WHERE id = p_user_id) THEN
         SET error_message = 'User with that id doesn't exists';
         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_message;
     END IF;

     -- Adding a new employee
     INSERT INTO Employee (user_id) VALUES (p_user_id);
END;
```

4.2.4. Trigger creation

- Trigger 1: Automatic assignment of an employee to a request. Trigger selects the employee who has the fewest requests on their account. In addition, if the applicant is an employee himself, he or she chooses another employee. Trigger also checks if this is the first application of a given applicant - if so, it creates a new user account for them in the User table, generating a random password.

```
CREATE TRIGGER assign_employee_before_insert_application
BEFORE INSERT ON Application
FOR EACH ROW
BEGIN
    DECLARE available_employee_id INT;
 DECLARE applicant_pesel CHAR(11);
 DECLARE random_password VARCHAR(255);

    -- Finds the employee with the fewest requests
    SELECT e.id INTO available_employee_id
    FROM Employee e
    LEFT JOIN (SELECT employee_id, COUNT(*) AS application_count
               FROM Application
               GROUP BY employee_id) AS app_count ON e.id = app_count.employee_id
    ORDER BY app_count.application_count ASC, RAND()
    LIMIT 1;

    -- Checks that the employee is not an applicant for this application
    IF NEW.applicant_id = available_employee_id THEN
        SELECT e.id INTO available_employee_id
        FROM Employee e
 WHERE e.id <> NEW.applicant_id
        ORDER BY RAND()
        LIMIT 1;
    END IF;

    -- Assigns an employee to a request
 SET NEW.employee_id = available_employee_id;

    -- Checks if this is the applicant's first application
    SELECT COUNT(*) INTO @existing_applications
    FROM Application
 WHERE applicant_id = NEW.applicant_id;

    IF @existing_applications  = 0 THEN
        -- Password generation
        SELECT SUBSTRING(MD5(RAND()), 1, 10) INTO random_password;

        -- Retrieving the Social Security number
        SELECT p.PESEL INTO applicant_pesel
        FROM Applicant
        JOIN Person p ON Applicant.person_id = p.id
        WHERE Applicant.id = NEW.applicant_id;

        -- Creating a new user
        INSERT INTO 'User' (username, password, applicant_id)
        VALUES (applicant_pesel, random_password, NEW.applicant_id);

    END IF;
END;
```

- Trigger 2: Adding a new resident to the registry. If the status of the application has been changed to "approved" and it is an application for entry in the register, then a new resident is created.

```
 CREATE TRIGGER after_application_update
 AFTER UPDATE ON Application
 FOR EACH ROW
 BEGIN
     DECLARE application_type INT;
     DECLARE person_id_for_resident INT;
     DECLARE user_id_for_resident INT;

     SELECT type INTO application_type FROM 'Application Type' WHERE id =
NEW.application_type_id;

     -- Checks that the status of the application is approved and that it is an application for
entry in the register
     IF OLD.status <> 'approved' AND NEW.status = 'approved' AND application_type = 1 THEN

         -- Retrieves person_id from  the Applicant  table associated with the application
         SELECT Applicant.person_id INTO person_id_for_resident
         FROM Applicant
  WHERE Applicant.id = NEW.applicant_id;

         -- Retrieve user_id from  the User  table associated with the request
         SELECT id INTO user_id_for_resident
         FROM 'User'
  WHERE applicant_id = NEW.applicant_id;

         -- Adds a new resident
         INSERT INTO Resident (person_id, user_id)
  VALUES (person_id_for_resident, user_id_for_resident);
     END IF;
 END;
```

- Trigger 3: Validation of data by the system before adding personal data.

```
 CREATE TRIGGER before_insert_person
 BEFORE INSERT ON Person
 FOR EACH ROW
 BEGIN
  IF NEW.PESEL IS NOT NULL THEN
  CALL ValidatePesel(NEW.PESEL number);
     END IF;
  CALL ValidateTextData(NEW.first_name, "first_name");

  IF NEW.second_name IS NOT NULL THEN
  CALL ValidateTextData(NEW.second_name, "second_name");
     END IF;
  CALL ValidateTextData(NEW.last_name, "last_name");

  IF NEW.place_of_birth IS NOT NULL THEN
  CALL ValidateTextData(NEW.place_of_birth, "place_of_birth");
     END IF;

  IF NEW.father_name IS NOT NULL THEN
  CALL ValidateTextData(NEW.father_name, "father_name");
     END IF;

  IF NEW.mother_name IS NOT NULL THEN
  CALL ValidateTextData(NEW.mother_name, "mother_name");
     END IF;

  IF NEW.mother_maiden_name IS NOT NULL THEN
  CALL ValidateTextData(NEW.mother_maiden_name, "mother_maiden_name");
     END IF;
 END;
```

- Trigger 4: Validation of data by the system before adding an applicator.

```
CREATE TRIGGER before_insert_applicant
BEFORE INSERT ON Applicant
FOR EACH ROW
BEGIN
    -- Email validation
 CALL ValidateEmail(NEW.email_address);

    -- Phone number validation
 CALL ValidatePhone(NEW.phone_number);
END;
```

- Trigger 5: Validation of data by the system before adding address data.

```
CREATE TRIGGER before_insert_address
BEFORE INSERT ON Address
FOR EACH ROW
BEGIN
    CALL ValidatePostalCode(NEW.postal_code);
 CALL ValidateTextData(NEW.street, "street");
 CALL ValidateTextData(NEW.city, "city");
END;
```

4.3. Implementation of permissions and other safeguards

4.3.1. Creation of roles

```
CREATE ROLE 'administrator_role';
CREATE ROLE 'manager_role';
CREATE ROLE 'employee_role';
```

4.3.2. Granting of rights

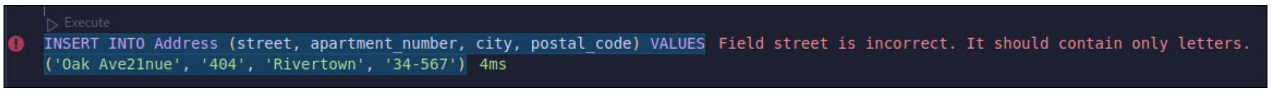
```
GRANT ALL PRIVILEGES ON del.* TO administrator_role;

GRANT SELECT, INSERT, UPDATE ON del.Application TO manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.'Application Type' TO manager_role;

GRANT SELECT ON del.'Application Type' TO employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.'User' IS manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.Applicant TO manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.Resident TO manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.Person TO manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.Address TO manager_role, employee_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON del.Employee TO manager_role;
```

4.4. Testing the Database on Sample Data

Tests were conducted in Visual Studio Code with the help of MySQL

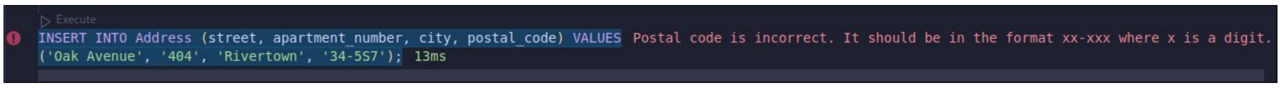
4.4.1. Trigger Tests

Picture 5: Trigger test 'before_insert_address'



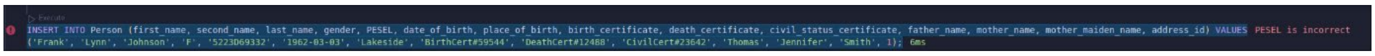
```
Execute
INSERT INTO Address (street, apartment_number, city, postal_code) VALUES Field street is incorrect. It should contain only letters.
('Oak Ave21nue', '404', 'Rivertown', '34-567') 4ms
```

Picture 6: Trigger test 'before_insert_address'



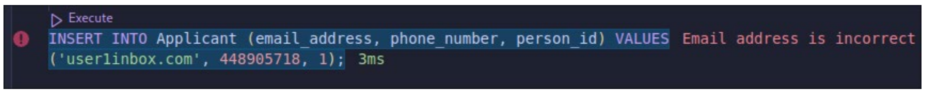
```
Execute
INSERT INTO Address (street, apartment_number, city, postal_code) VALUES Postal code is incorrect. It should be in the format xx-xxx where x is a digit.
('Oak Avenue', '404', 'Rivertown', '34-5S7'); 13ms
```

Picture 7: Trigerr test 'before_insert_person'



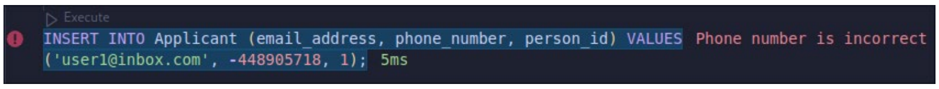
```
Execute
INSERT INTO Person (first_name, second_name, last_name, gender, PESEL, date_of_birth, place_of_birth, birth_certificate, death_certificate, civil_status_certificate, father_name, mother_name, mother_maiden_name, address_id) VALUES PESEL is incorrect
('Frank', 'Lynn', 'Johnson', 'F', '5223D69332', '1962-03-03', 'Lakeside', 'BirthCert#59544', 'DeathCert#12488', 'CivilCert#23642', 'Thomas', 'Jennifer', 'Smith', 1); 6ms
```

Picture 8: Trigger test 'before_insert_applicant'



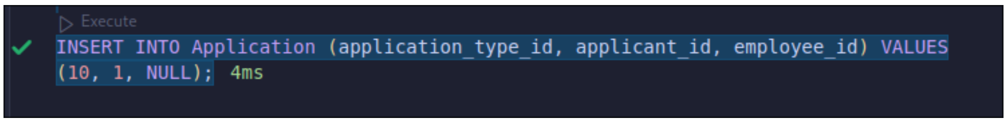
```
Execute
INSERT INTO Applicant (email_address, phone_number, person_id) VALUES Email address is incorrect
('user1inbox.com', 448905718, 1); 3ms
```

Picture 9: Trigerr test 'before_insert_applicant'

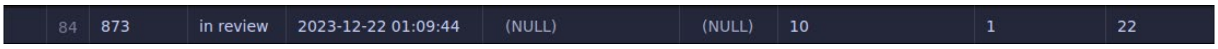


```
Execute
INSERT INTO Applicant (email_address, phone_number, person_id) VALUES Phone number is incorrect
('user1@inbox.com', -448905718, 1); 5ms
```

Photos 10 & 11: Trigger test "assign_employee_before_insert_application"

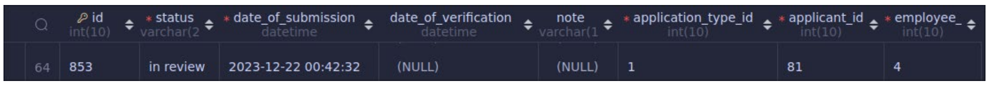


```
Execute
INSERT INTO Application (application_type_id, applicant_id, employee_id) VALUES
(10, 1, NULL); 4ms
```

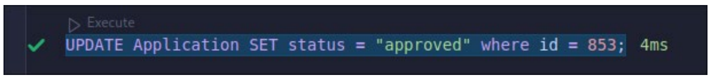


84	873	in review	2023-12-22 01:09:44	(NULL)	(NULL)	10	1	22

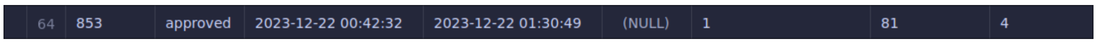
Photos 11, 12, 13 and 14: Trigger test "after_application_update"



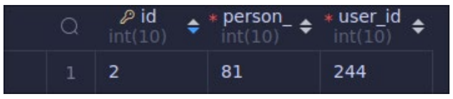
	id int(10)	status varchar(2	date_of_submission datetime	date_of_verification datetime	note varchar(1	application_type_id int(10)	applicant_id int(10)	employee_ int(10)
64	853	in review	2023-12-22 00:42:32	(NULL)	(NULL)	1	81	4



```
Execute
UPDATE Application SET status = "approved" where id = 853; 4ms
```



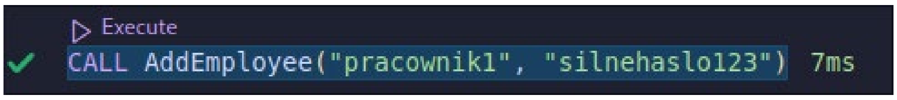
64	853	approved	2023-12-22 00:42:32	2023-12-22 01:30:49	(NULL)	1	81	4



	id int(10)	person_ int(10)	user_id int(10)
1	2	81	244

4.4.2. Testing of procedures

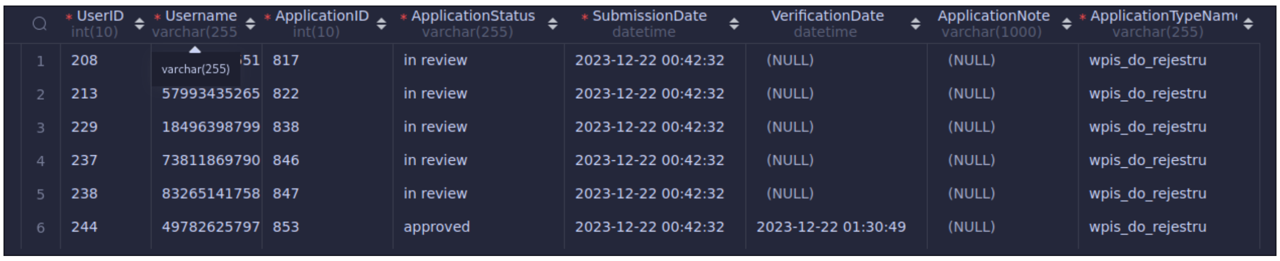
Picture 15: Test of the "AddEmployee" procedure



```
Execute
CALL AddEmployee("pracownik1", "silnehaslo123")  7ms
```

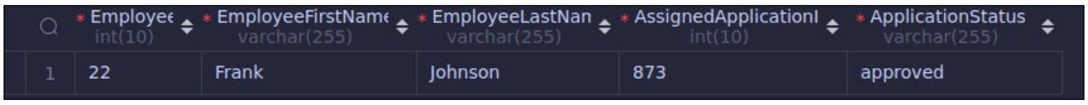
4.4.3. View tests

Figure 16: Test of the "UserApplicationInfo" view



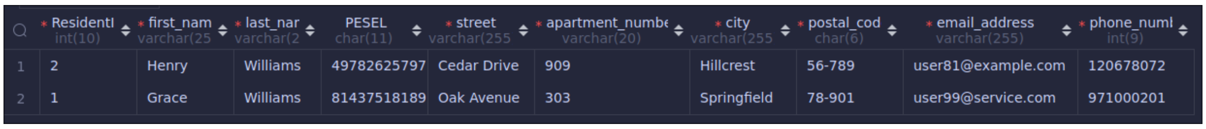
	UserID int(10)	Username varchar(255	ApplicationID int(10)	ApplicationStatus varchar(255)	SubmissionDate datetime	VerificationDate datetime	ApplicationNote varchar(1000)	ApplicationTypeNam varchar(255)
1	208	varchar(255) 51	817	in review	2023-12-22 00:42:32	(NULL)	(NULL)	wpis_do_rejestru
2	213	57993435265	822	in review	2023-12-22 00:42:32	(NULL)	(NULL)	wpis_do_rejestru
3	229	18496398799	838	in review	2023-12-22 00:42:32	(NULL)	(NULL)	wpis_do_rejestru
4	237	73811869790	846	in review	2023-12-22 00:42:32	(NULL)	(NULL)	wpis_do_rejestru
5	238	83265141758	847	in review	2023-12-22 00:42:32	(NULL)	(NULL)	wpis_do_rejestru
6	244	49782625797	853	approved	2023-12-22 00:42:32	2023-12-22 01:30:49	(NULL)	wpis_do_rejestru

Figure 17: Test of the "EmployeeTaskAssignment" view



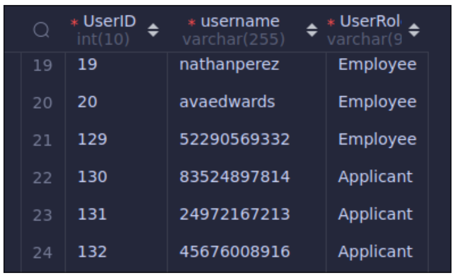
	Employee int(10)	EmployeeFirstName varchar(255)	EmployeeLastNan varchar(255)	AssignedApplicationI int(10)	ApplicationStatus varchar(255)
1	22	Frank	Johnson	873	approved

Figure 18: Test of the "ResidentContactInfo" view



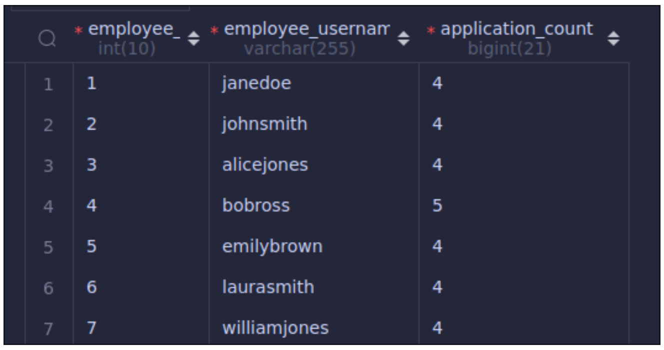
	ResidentI int(10)	first_nam varchar(25	last_nar varchar(2	PESEL char(11)	street varchar(255	apartment_numbe varchar(20)	city varchar(255	postal_cod char(6)	email_address varchar(255)	phone_numl int(9)
1	2	Henry	Williams	49782625797	Cedar Drive	909	Hillcrest	56-789	user81@example.com	120678072
2	1	Grace	Williams	81437518189	Oak Avenue	303	Springfield	78-901	user99@service.com	971000201

Figure 19: Test of the "UserRoleAccess" view



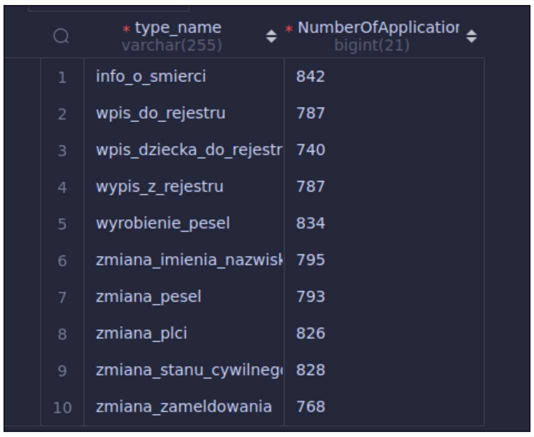
	UserID int(10)	username varchar(255)	UserRol varchar(9
19	19	nathanperez	Employee
20	20	avaedwards	Employee
21	129	52290569332	Employee
22	130	83524897814	Applicant
23	131	24972167213	Applicant
24	132	45676008916	Applicant

Image 20: Test of the "EmployeeApplicationCount" view



	employee_ int(10)	employee_usernam varchar(255)	application_count bigint(21)
1	1	janedoe	4
2	2	johnsmith	4
3	3	alicejones	4
4	4	bobross	5
5	5	emilybrown	4
6	6	laurasmith	4
7	7	williamjones	4

Figure 21: Test of the "ApplicationTypeSummary" view



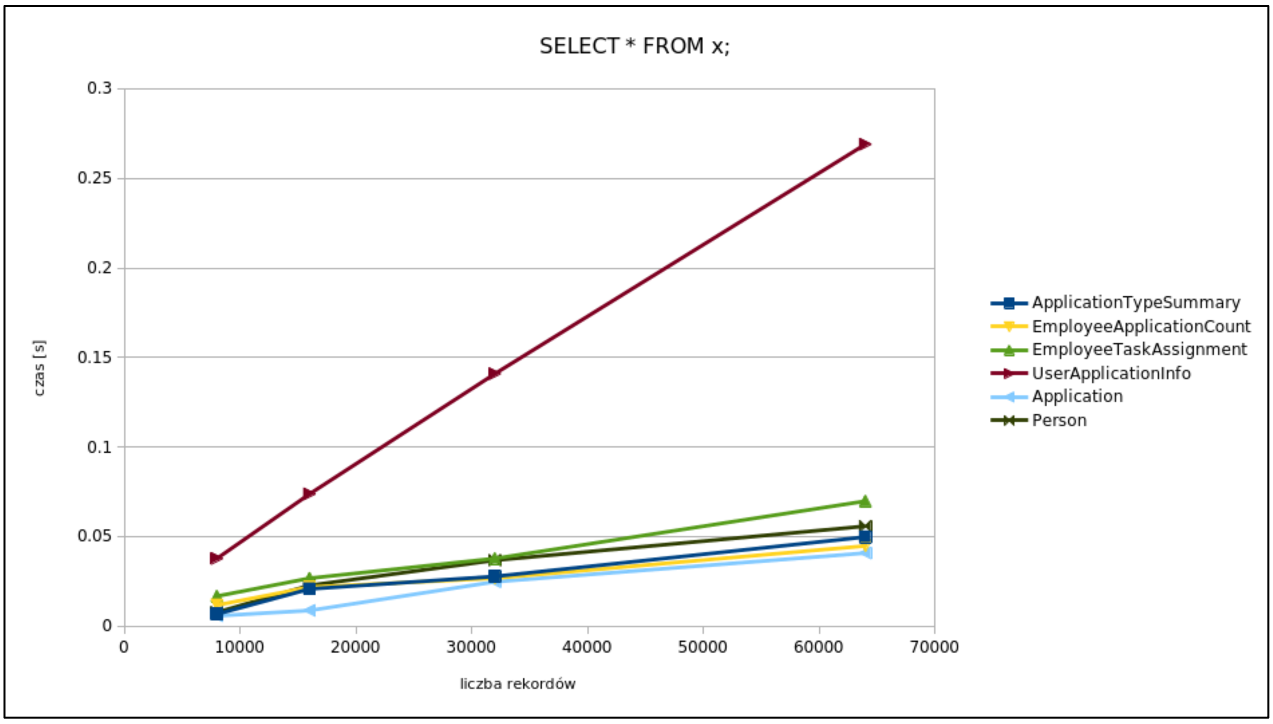
	type_name varchar(255)	NumberOfApplicatior bigint(21)
1	info_o_smierci	842
2	wpis_do_rejestru	787
3	wpis_dziecka_do_rejestr	740
4	wypis_z_rejestru	787
5	wyrobienie_pesel	834
6	zmiana_imienia_nazwisk	795
7	zmiana_pesel	793
8	zmiana_plci	826
9	zmiana_stanu_cywilneg	828
10	zmiana_zameldowania	768

4.4.3. Query performance tests

Table 1: SELECT * FROM x query performance test;

	liczba rekordów			
	8000	**16000**	**32000**	**64000**
ApplicationTypeSummary [s]	0.007	0.021	0.028	0.05
EmployeeApplicationCount [s]	0.012	0.022	0.027	0.045
EmployeeTaskAssignment [s]	0.017	0.027	0.038	0.07
UserApplicationInfo [s]	0.038	0.074	0.141	0.269
Application [s]	0.006	0.009	0.025	0.041
Person [s]	0.008	0.023	0.037	0.056

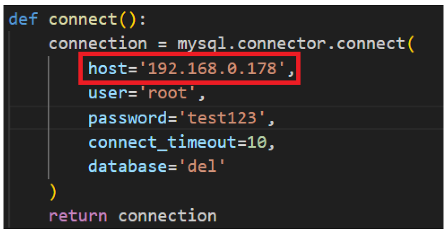
Figure 1: Query performance test



5. Application implementation and testing

5.1. System Installation and Configuration

- Clone the repository from github
- In the "PRD\backend>" directory, modify the "database.py" file by entering your ip address in the "host" field:



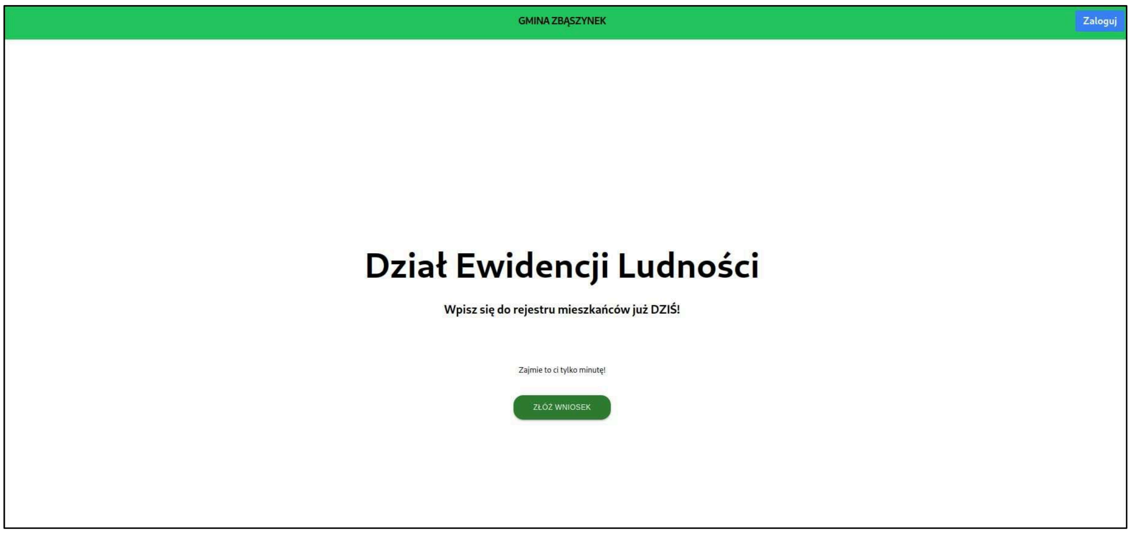
```python
def connect():
    connection = mysql.connector.connect(
        host='192.168.0.178',
        user='root',
        password='test123',
        connect_timeout=10,
        database='del'
    )
    return connection
```

Picture 22: Editing the host ip

- Being in the "PRD\frontend>" directory, run the command: npm install
- In the "PRD\backend>" directory, create a ".env" file containing the following key environment variables:
    1) secret - Secret key used to encode JWT tokens
    2) algorithm - An algorithm used to encode JWT tokens, e.g. "HS256"
- Once in the "PRD>" directory, run the following commands:
  docker-compose -f docker-compose.yml build
  docker-compose -f docker-compose.yml up
- In your browser, go to: http://localhost:5173/

5.2. Instructions for use of the app

- View after entering the main page:



Dział Ewidencji Ludności

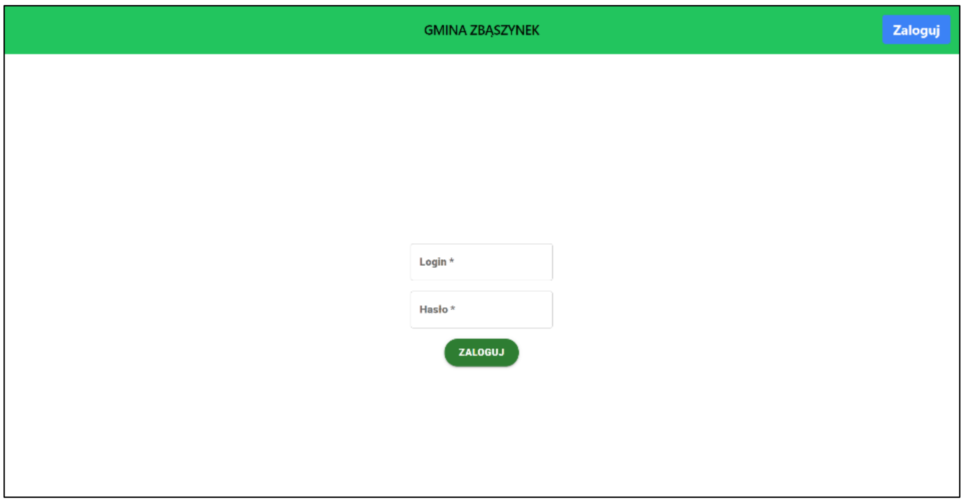
Wpisz się do rejestru mieszkańców już DZIŚ!

Zajmie to ci tylko minutę!

Picture 22: Home page

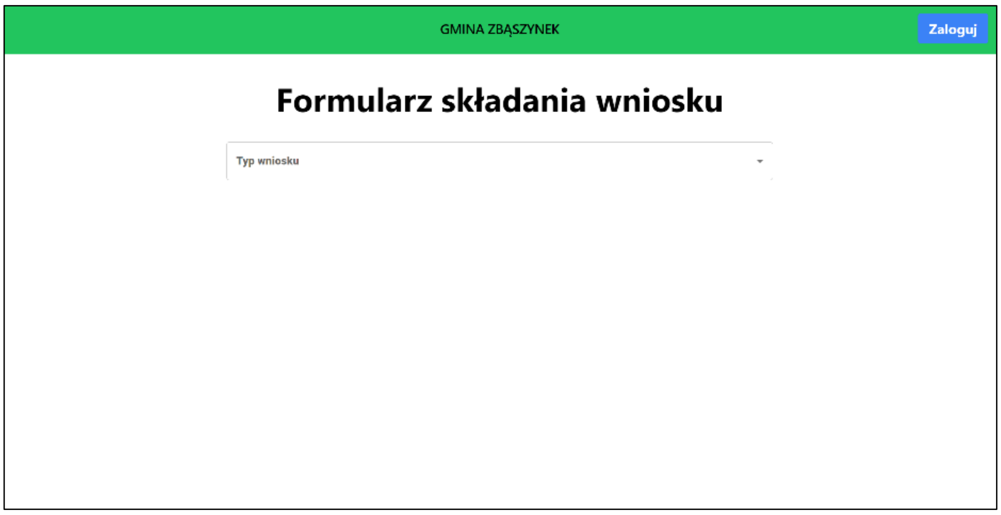
- Being on the home page there are two possibilities:

1. If you are already registered in the Register of Residents or have previously applied for registration, you can log in by clicking on the "Login" button in the upper right corner, which will take you to the login page:



Picture 23: Login page

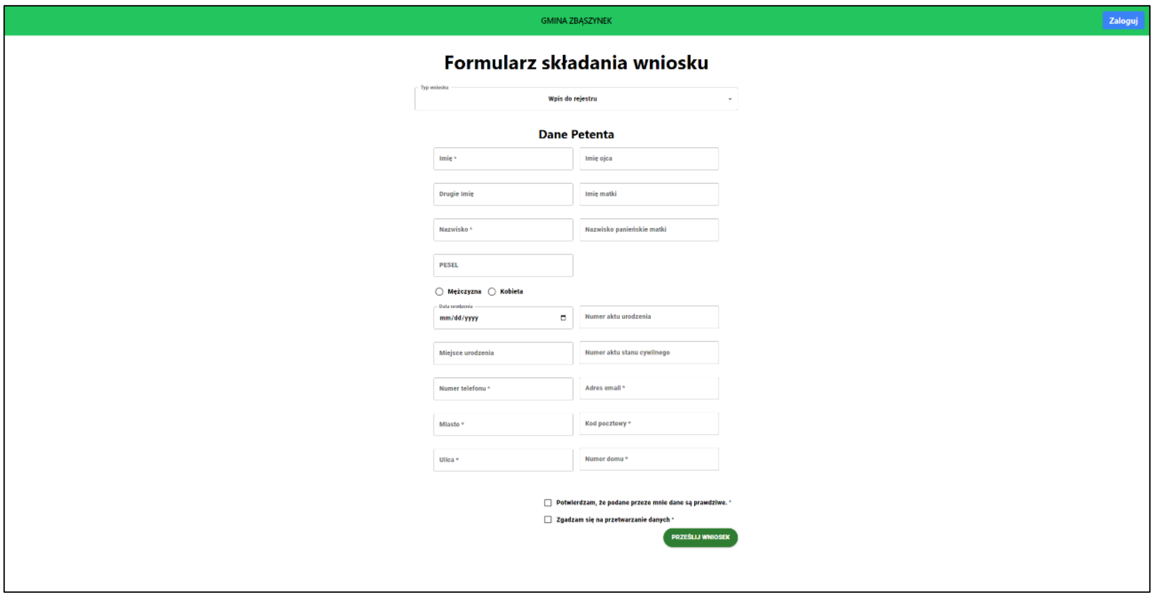
2. A person wants to apply for registration by clicking on the "Apply" button, which will take them to the form page:



Formularz składania wniosku

Picture 24: Application form page

In the "Type of application" field, he/she must select the only "Entry in the register" option available at the moment and fill in the required field with the relevant data:



Formularz składania wniosku

Typ wniosku: Wpis do rejestru

Dane Petenta

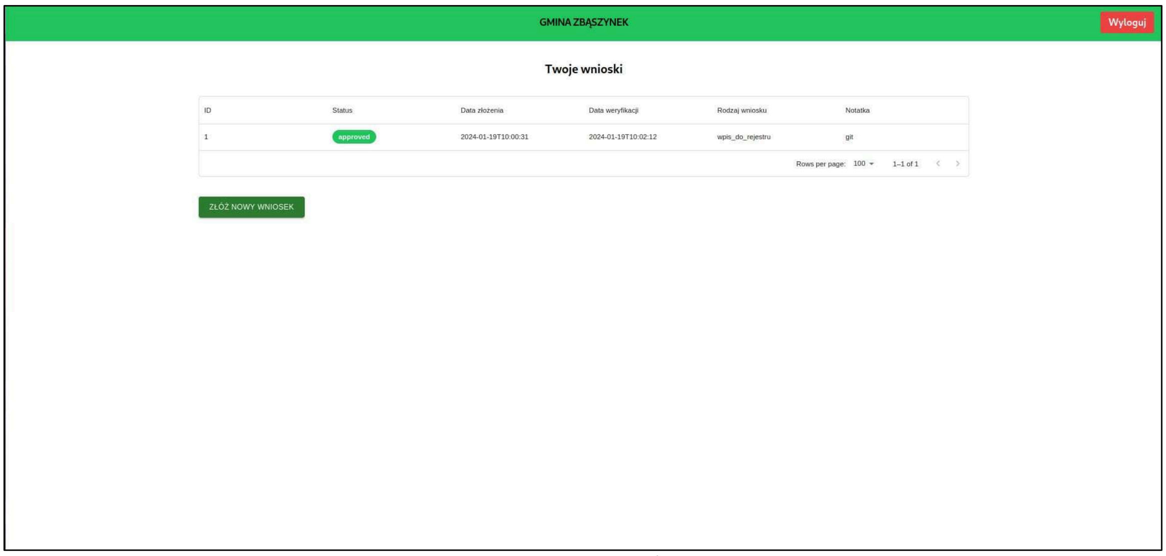
Imię *	Imię ojca
Drugie imię	Imię matki
Nazwisko *	Nazwisko panieńskie matki
PESEL	
Mężczyzna / Kobieta	
Data urodzenia: mm/dd/yyyy	Numer aktu urodzenia
Miejsce urodzenia	Numer aktu stanu cywilnego
Numer telefonu *	Adres email *
Miasto *	Kod pocztowy *
Ulica *	Numer domu *

☐ Potwierdzam, że podane przeze mnie dane są prawdziwe. *

☐ Zgadzam się na przetwarzanie danych *

Picture 24: Application form page

- After submitting their first application, the person can log in to the system by clicking the "Login" button in the upper right corner, entering their PESEL number as a login and entering a string of characters generated by the system as a password, which was delivered to the e-mail address provided in the form by that person. After logging in, the person can view their applications in the user panel as well as submit new ones:



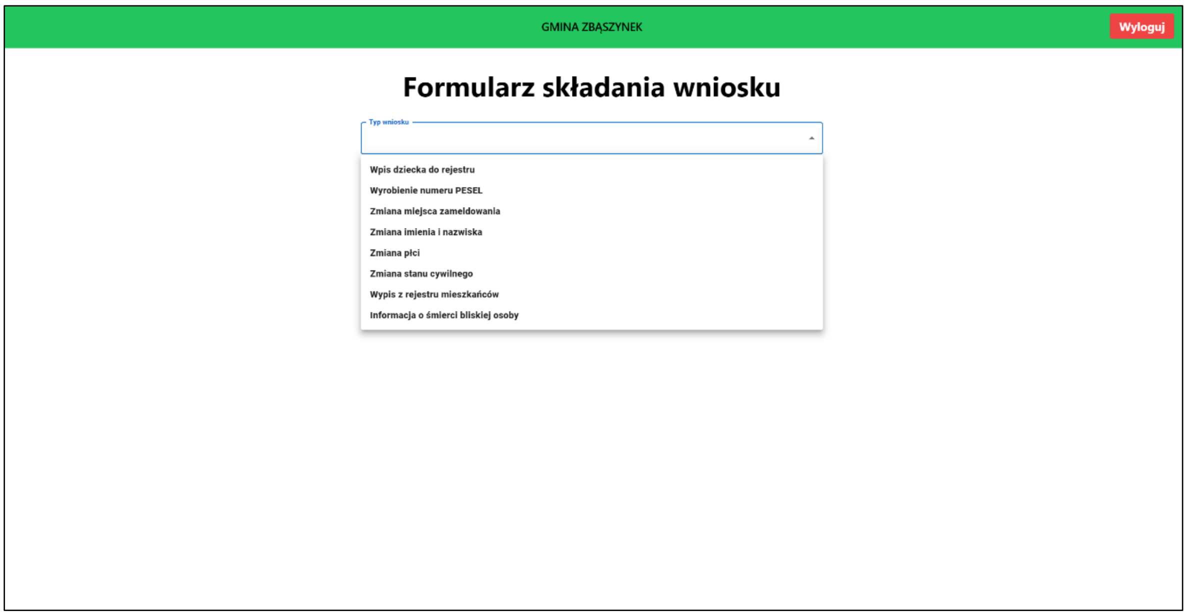
Twoje wnioski

ID	Status	Data złożenia	Data weryfikacji	Rodzaj wniosku	Notatka
1	approved	2024-01-19T10:00:31	2024-01-19T10:02:12	wpis_do_rejestru	git

Rows per page: 100 1–1 of 1

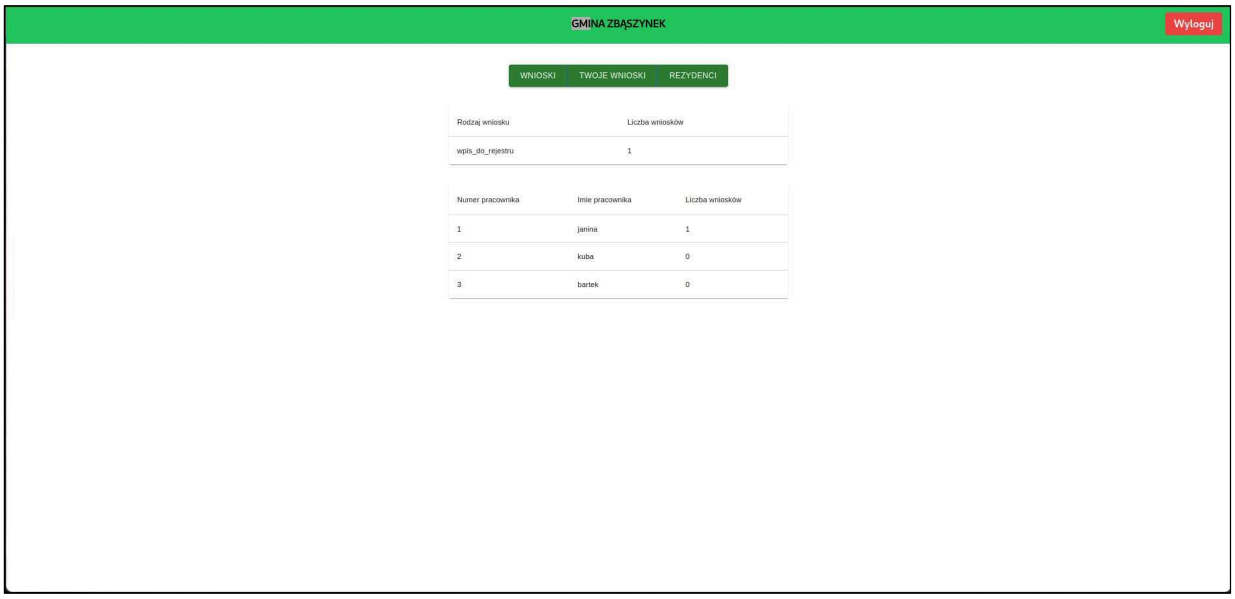
Picture 25: User panel page

A logged-in user, after clicking the "Submit a new application" button, already has the option to submit an application of any type:


GMINA ZBĄSZYNEK
Wyloguj
Formularz składania wniosku
Typ wniosku
Wpis dziecka do rejestru
Wyrobienie numeru PESEL
Zmiana miejsca zameldowania
Zmiana imienia i nazwiska
Zmiana płci
Zmiana stanu cywilnego
Wypis z rejestru mieszkańców
Informacja o śmierci bliskiej osoby

Picture 26: Application forms page

- If the person logging in to our website is an employee, they will go to the employee panel, where you will see a list of all employees, the type and number of applications processed by this employee, and the ability to check their own applications, the applications they have processed and the list of residents of the municipality:

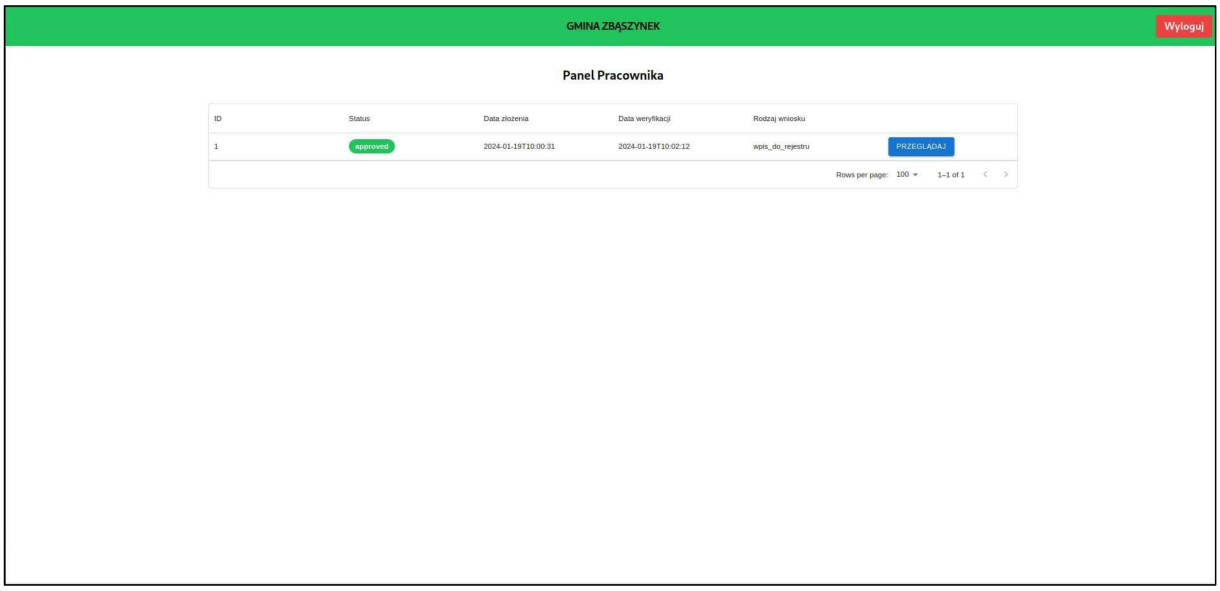


Rodzaj wniosku	Liczba wniosków
wpis_do_rejestru	1

Numer pracownika	Imie pracownika	Liczba wniosków
1	janina	1
2	kuba	0
3	bartek	0

Picture 27: Employee dashboard page

After clicking the "Requests" button, a logged-in employee will see a list of applications that have been processed:



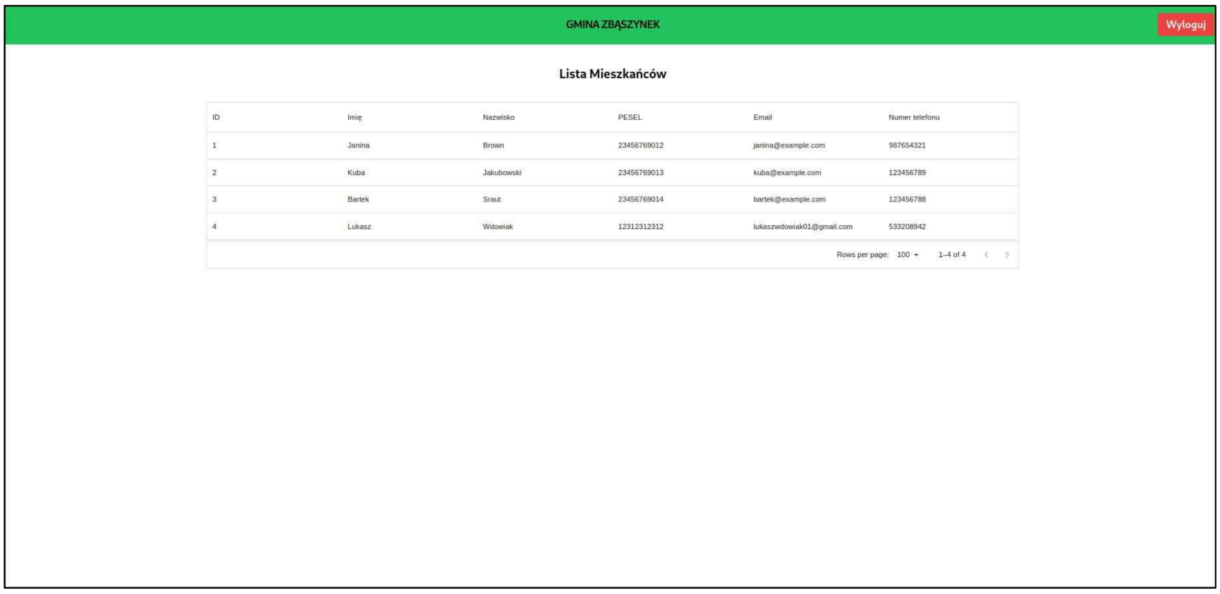
Panel Pracownika

ID	Status	Data złożenia	Data weryfikacji	Rodzaj wniosku	
1	approved	2024-01-19T10:00:31	2024-01-19T10:02:12	wpis_do_rejestru	PRZEGLĄDAJ

Rows per page: 100 1–1 of 1

Picture 28: Employee's processed applications page

After clicking the "Residents" button, a logged-in employee will see a list of residents of the municipality listed in the register:



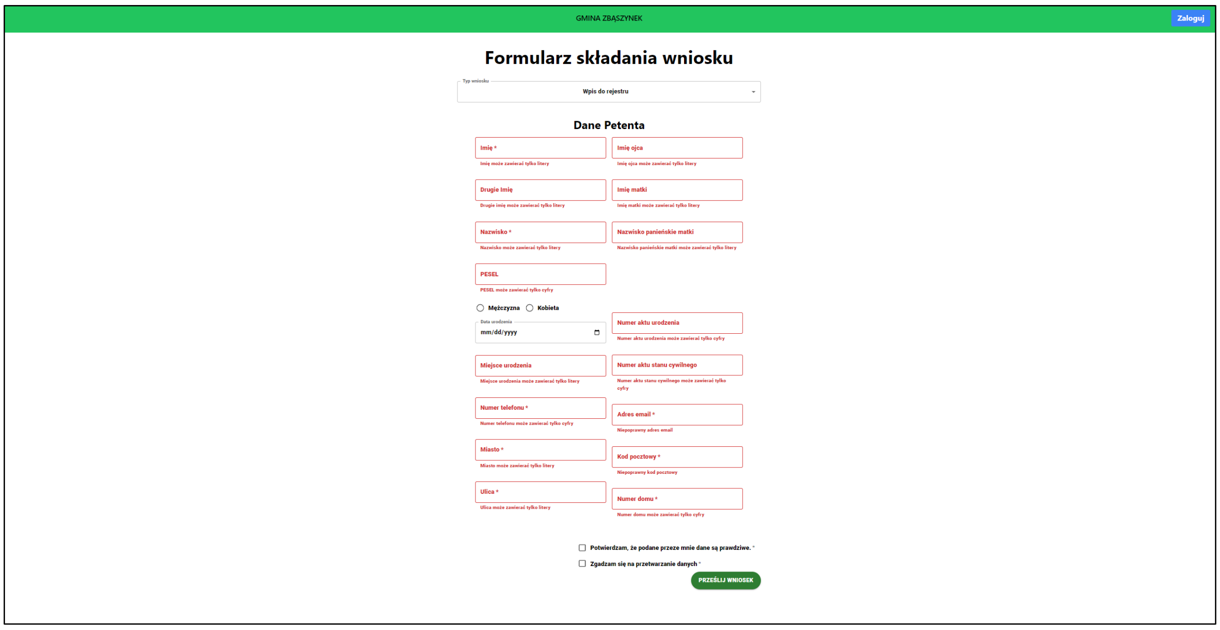
Lista Mieszkańców

ID	Imię	Nazwisko	PESEL	Email	Numer telefonu
1	Janina	Brown	23456769012	janina@example.com	987654321
2	Kuba	Jakubowski	23456769013	kuba@example.com	123456789
3	Bartek	Sraut	23456769014	bartek@example.com	123456788
4	Lukasz	Wdowiak	12312312312	lukaszwdowiak01@gmail.com	533208942

Rows per page: 100 1–4 of 4

5.3. Testing of developed system functions

- Incorrect entry of data when applying for entry in the register:


GMINA ZBĄSZYNEK

Zaloguj

Formularz składania wniosku

Typ wniosku: Wpis do rejestru

Dane Petenta

Imię *
Imię może zawierać tylko litery

Imię ojca
Imię ojca może zawierać tylko litery

Drugie Imię
Drugie imię może zawierać tylko litery

Imię matki
Imię matki może zawierać tylko litery

Nazwisko *
Nazwisko może zawierać tylko litery

Nazwisko panieńskie matki
Nazwisko panieńskie matki może zawierać tylko litery

PESEL
PESEL może zawierać tylko cyfry

Mężczyzna Kobieta

Data urodzenia: mm/dd/yyyy

Numer aktu urodzenia
Numer aktu urodzenia może zawierać tylko cyfry

Miejsce urodzenia
Miejsce urodzenia może zawierać tylko litery

Numer aktu stanu cywilnego
Numer aktu stanu cywilnego może zawierać tylko cyfry

Numer telefonu *
Numer telefonu może zawierać tylko cyfry

Adres email *
Niepoprawny adres email

Miasto *
Miasto może zawierać tylko litery

Kod pocztowy *
Niepoprawny kod pocztowy

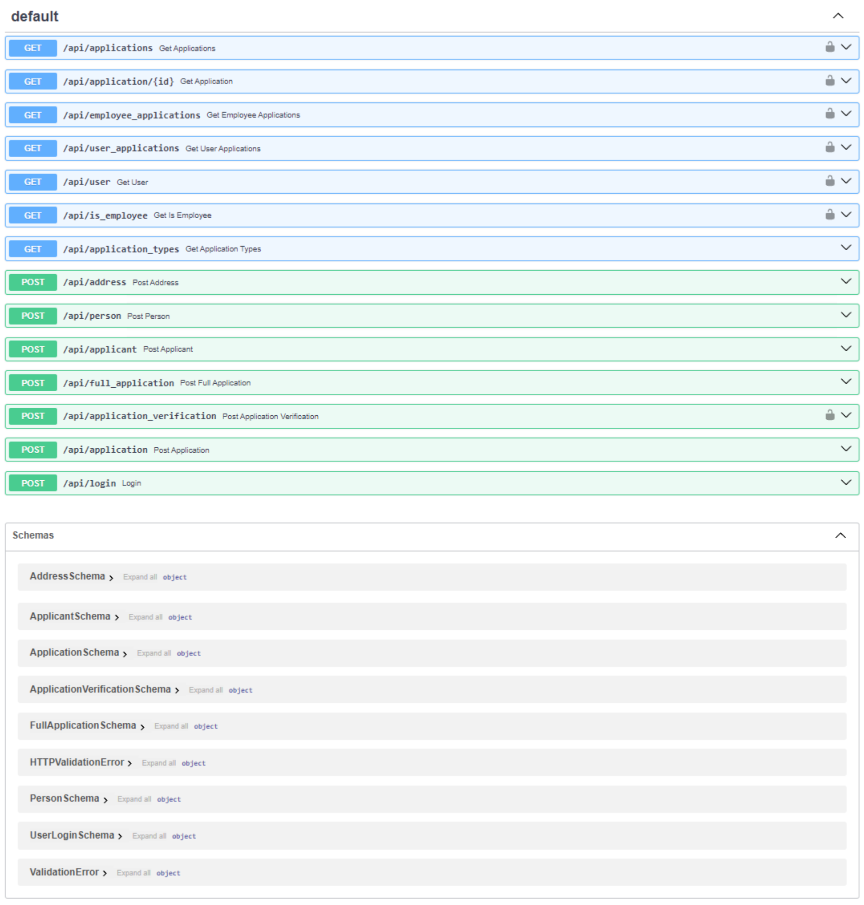
Ulica *
Ulica może zawierać tylko litery

Numer domu *
Numer domu może zawierać tylko cyfry

☐ Potwierdzam, że podane przeze mnie dane są prawdziwe. *
☐ Zgadzam się na przetwarzanie danych *

PRZEŚLIJ WNIOSEK

- Testing the API directly from the browser:
  One of the main advantages of the FastAPI used in this project is that it automatically generates documentation for the API using Swagger UI. Thanks to this, users can easily understand how to use it, what are the available endpoints, what parameters they take, what are the expected formats of input and output data, as well as test the API directly in the browser.
  To get there, enter the address: http://localhost:8000/docs

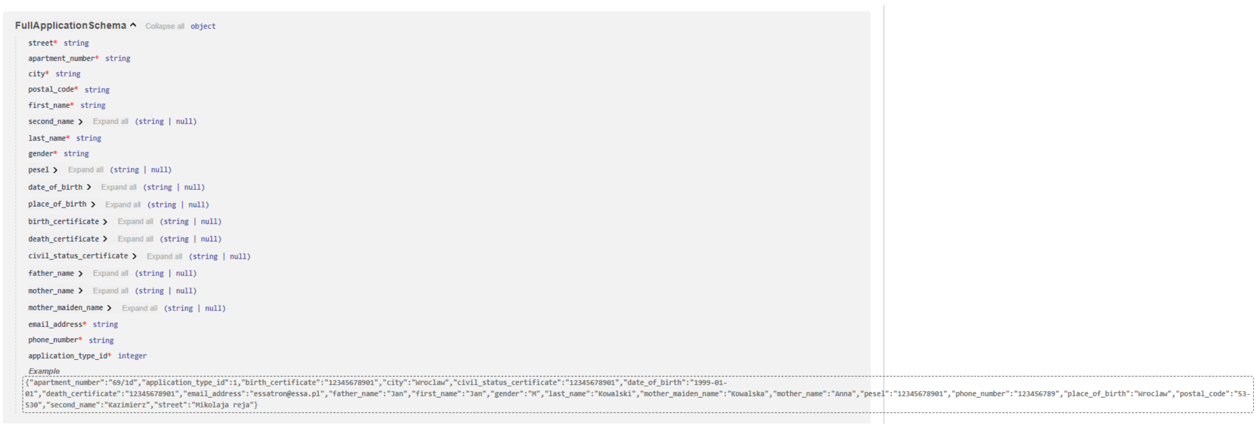

default

GET /api/applications Get Applications
GET /api/application/{id} Get Application
GET /api/employee_applications Get Employee Applications
GET /api/user_applications Get User Applications
GET /api/user Get User
GET /api/is_employee Get Is Employee
GET /api/application_types Get Application Types
POST /api/address Post Address
POST /api/person Post Person
POST /api/applicant Post Applicant
POST /api/full_application Post Full Application
POST /api/application_verification Post Application Verification
POST /api/application Post Application
POST /api/login Login

Schemas

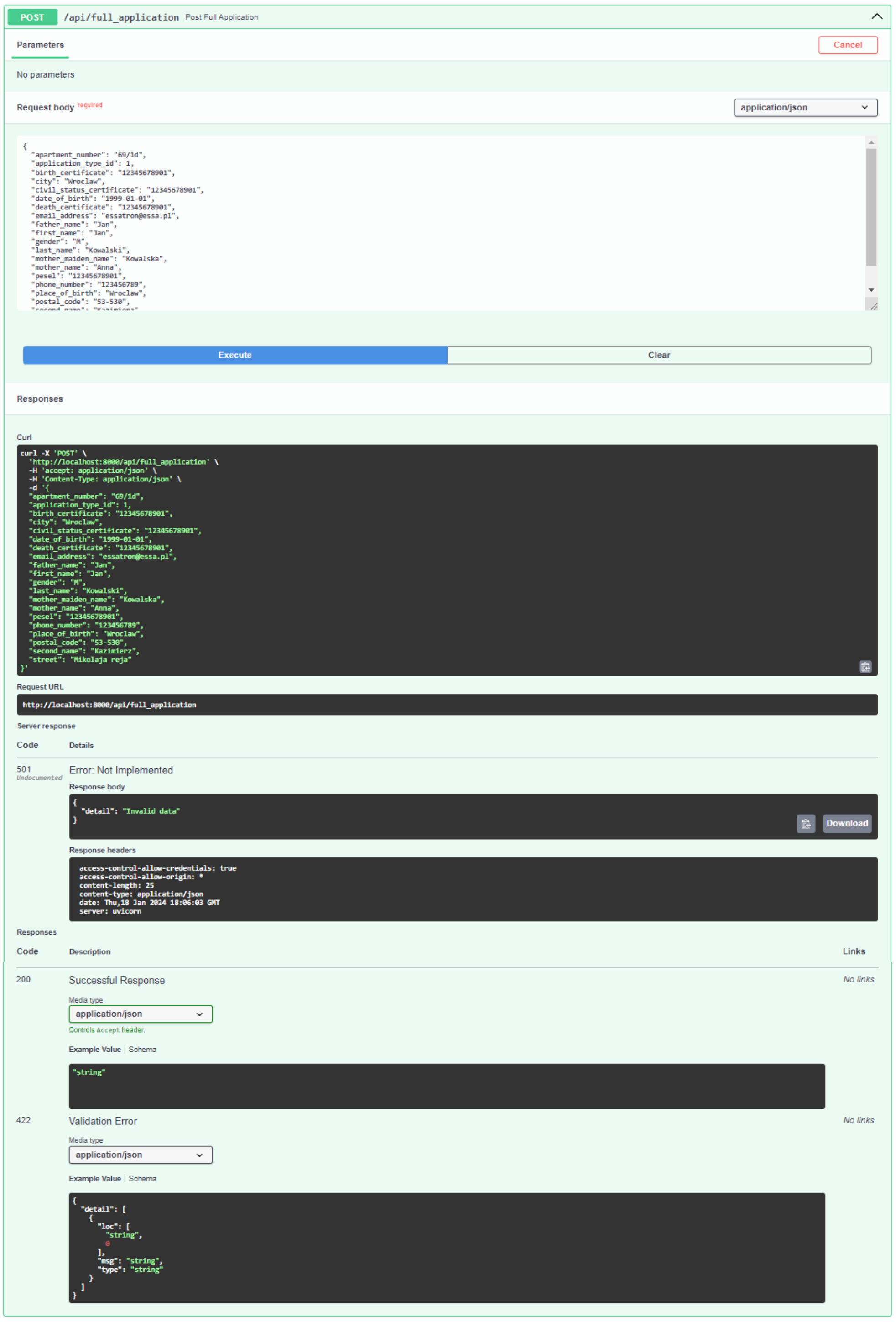
AddressSchema > Expand all object
ApplicantSchema > Expand all object
ApplicationSchema > Expand all object
ApplicationVerificationSchema > Expand all object
FullApplicationSchema > Expand all object
HTTPValidationError > Expand all object
PersonSchema > Expand all object
UserLoginSchema > Expand all object
ValidationError > Expand all object

An example of a scheme for submitting an application:



```
FullApplicationSchema ^  Collapse all  object
  street*  string
  apartment_number*  string
  city*  string
  postal_code*  string
  first_name*  string
  second_name >  Expand all  (string | null)
  last_name*  string
  gender*  string
  pesel >  Expand all  (string | null)
  date_of_birth >  Expand all  (string | null)
  place_of_birth >  Expand all  (string | null)
  birth_certificate >  Expand all  (string | null)
  death_certificate >  Expand all  (string | null)
  civil_status_certificate >  Expand all  (string | null)
  father_name >  Expand all  (string | null)
  mother_name >  Expand all  (string | null)
  mother_maiden_name >  Expand all  (string | null)
  email_address*  string
  phone_number*  string
  application_type_id*  integer
  Example
  {"apartment_number":"69/1d","application_type_id":1,"birth_certificate":"12345678901","city":"Wroclaw","civil_status_certificate":"12345678901","date_of_birth":"1999-01-
  01","death_certificate":"12345678901","email_address":"essatron@essa.pl","father_name":"Jan","first_name":"Jan","gender":"M","last_name":"Kowalski","mother_maiden_name":"Kowalska","mother_name":"Anna","pesel":"12345678901","phone_number":"123456789","place_of_birth":"Wroclaw","postal_code":"53-
  530","second_name":"Kazimierz","street":"Mikolaja reja"}
```

An example of a test for submitting an application:



POST /api/full_application Post Full Application

Parameters — Cancel

No parameters

Request body required — application/json

```
{
  "apartment_number": "69/1d",
  "application_type_id": 1,
  "birth_certificate": "12345678901",
  "city": "Wroclaw",
  "civil_status_certificate": "12345678901",
  "date_of_birth": "1999-01-01",
  "death_certificate": "12345678901",
  "email_address": "essatron@essa.pl",
  "father_name": "Jan",
  "first_name": "Jan",
  "gender": "M",
  "last_name": "Kowalski",
  "mother_maiden_name": "Kowalska",
  "mother_name": "Anna",
  "pesel": "12345678901",
  "phone_number": "123456789",
  "place_of_birth": "Wroclaw",
  "postal_code": "53-530",
```

Execute | Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/api/full_application' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "apartment_number": "69/1d",
  "application_type_id": 1,
  "birth_certificate": "12345678901",
  "city": "Wroclaw",
  "civil_status_certificate": "12345678901",
  "date_of_birth": "1999-01-01",
  "death_certificate": "12345678901",
  "email_address": "essatron@essa.pl",
  "father_name": "Jan",
  "first_name": "Jan",
  "gender": "M",
  "last_name": "Kowalski",
  "mother_maiden_name": "Kowalska",
  "mother_name": "Anna",
  "pesel": "12345678901",
  "phone_number": "123456789",
  "place_of_birth": "Wroclaw",
  "postal_code": "53-530",
  "second_name": "Kazimierz",
  "street": "Mikolaja reja"
}'
```

Request URL

```
http://localhost:8000/api/full_application
```

Server response

Code	Details
501 *Undocumented*	Error: Not Implemented

Response body

```
{
  "detail": "Invalid data"
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
content-length: 25
content-type: application/json
date: Thu,18 Jan 2024 18:06:03 GMT
server: uvicorn
```

Responses

Code	Description	Links
200	Successful Response	*No links*
422	Validation Error	*No links*

200 Successful Response — Media type: application/json — Controls Accept header. — Example Value | Schema

```
"string"
```

422 Validation Error — Media type: application/json — Example Value | Schema

```
{
  "detail": [
    {
      "loc": [
        "string",
        0
      ],
      "msg": "string",
      "type": "string"
    }
  ]
}
```

5.4. Discussion of selected software solutions

5.4.1. Implementation of the database access interface

The "mysql.connector" library was used to connect to the MySQL database. The connection is configured with the host's IP address, username, password, and database name in the "database.py" file:

```
MySQL import.Connector

def connect():
    connection = mysql.connector.connect(
        host='192.168.0.178',
        user='root',
        password='test123',
        connect_timeout=10,
        database='del'
   )
   return connection
```

The configuration of the database service ("db") is contained in the "docker-compose.yml" file. The "mysql:8" image is used for the database container, and environment variables such as "MYSQL_USER", "MYSQL_ROOT_PASSWORD", "MYSQL_DATABASE" are used to configure the database. Port "3306" is mapped, allowing access to the database from the host. The "web" and "api" services are configured as dependent on the "db" service, which ensures that the database is available before these services are launched. In contrast, "my-db:/var/lib/mysql" is used to store database data, which ensures data persistence beyond the lifecycle of the container. This pattern allows you to flexibly manage database connections because all connection strings are concentrated in one place, making them easy to modify and manage.

```
 version: "3.8"
 services:
  db:
    image: mysql:8
    environment:
      MYSQL_USER: user
      MYSQL_ROOT_PASSWORD: test123
      MYSQL_DATABASE: del
    ports:
      - "3306:3306"
    volumes:
      - my-db:/var/lib/mysql
  Web:
    build: ./frontend
    ports:
      - "5173:5173"
    volumes:
      - ./frontend:/app
  API:
    build: ./backend
    ports:
      - "8000:8000"
    depends_on:
      - db
    volumes:
      - ./backend:/app
    restart: on-failure
volumes:
 my-db:
```

5.4.2. Implementation of selected system functionalities

- Router in FastAPI:

  A mechanism that allows you to organize groups of endpoints. It allows for better code management and project structure, especially when the application includes many different paths and functions.

  Examples of implemented features using the router:

```
@router.get("/application/{id}", dependencies=[Depends(JWTBearer())])
async def get_application(id: int):
    query = "SELECT * FROM FullApplicationInfo WHERE application_id=%s"
    cursor.execute(query, (id,))
    result = cursor.fetchone()

    if result is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail="Application not found")
    return result
```

The "get_application" function is an API endpoint for retrieving the details of a single application based on its ID, and requires a JWT for access (uses "Depends(JWTBearer())"). Supports the "HTTP GET" request. At the very beginning, it executes a SQL query to the database to retrieve all the information about the application with the specified ID ("application_id=%s"). It then uses "cursor.execute" to execute the query with the given ID. It stores the result in the "result" variable, and retrieves it from the database using "cursor.fetchone()". If the application is found to return data, otherwise (the result is "None"), it returns a 404 error with an appropriate message.

```
@router.post("/address")
async def post_address(address: AddressSchema):
    query = "INSERT INTO Address (street, apartment_number, city, postal_code) VALUES (%s, %s,
        %s, %s)"
    cursor.execute(query, (address.street, address.apartment_number,
                   address.city, address.postal_code))
    conn.commit()

    item_id = cursor.LastRidId

    cursor.execute("SELECT * FROM Address WHERE id = %s", (item_id,))
    result = cursor.fetchone()

    if result is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail="Address not found after insertion")
return result
```

The "post_address" function is an API endpoint for adding a new address to the database. Supports "HTTP POST" request. It takes an object "address", which is an instance of "AddressSchema". It executes a SQL query to add a new address to the "Address" table in the database. After adding it, the function retrieves it with "cursor.lastrowid" and returns it as a response. If the address is not found after being added, the function returns an HTTP 404 error.

- Defining model classes in Pydantic ("model.py" file):
  They are used to structure and validate input data in a FastAPI application. Each class represents a data schema for different types of objects, such as users, addresses, people, applicants, applications, full applications, and application validation. The class uses "BaseModel" from Pydantic, which provides automatic input validation and facilitates instant serialization/deserialization of JSON data.
  Example implementation:

```python
class AddressSchema(BaseModel):
    street: str = Field(...)
    apartment_number: str = Field(...)
    city: str = Field(...)
    postal_code: str = Field(...)
    class Config:
        json_schema_extra = {
            "example": {
                "street": "Mikołaj reja",
                "apartment_number": "69/1d",
                "city": "Wroclaw",
                "postal_code": "53-530"
            }
        }
```

  "AddressSchema" inherits from "BaseModel" from Pydantic, which means that it uses the functions of this library to validate and serialize data. Defines text fields that correspond to address components, such as "street", "apartment_number", "city", "postal_code". The use of "Field(...)" in the definition of each field indicates that the field is required. Pydantic automatically checks whether the data passed to the "AddressSchema" model is compatible with the specified data types and that the required fields are not empty. In the "Config" class inside the "AddressSchema", a "json_schema_extra" is defined, containing sample data used for documentation and testing, helping other developers or tools understand what data is expected by the model.

- Private routes ("PrivateRoutes.jsx" file):
  In web applications, they are a mechanism that controls access to specific sections or pages of the application. Their main purpose is to ensure that only authenticated and authorized users can access certain resources or functionality.

```jsx
import { Outlet, Navigate } from "react-router-dom";

const PrivateRoutes = () = > {
  function getToken() {
    const tokenString = sessionStorage.getItem("token");
    const userToken = JSON.parse(tokenString);
    return userToken?.access_token;
  }
  return getToken() ? <Outlet /> : <Navigate to="/login" />;
};

export default PrivateRoutes;
```

  The "getToken" function attempts to retrieve an authentication token from the browser's "sessionStorage". If the token is present (indicating that the user is logged in), then "PrivateRoutes" renders an "Outlet" component that allows further rendering of nested routes defined in the application. If the token does not exist (the user is not logged in), the "Navigate" component redirects the user to the login page ("/login").

5.4.3. Implementation of security mechanisms

- CORS implementation ("main.py" file):
  CORS (Cross-Origin Resource Sharing) is a mechanism that allows or denies resource requests on a website from domains other than the domain from which the first resource originated. This is a key aspect of security and access control in web applications.

```
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

origins = ["*"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

The "origins" list specifies which origins are allowed to access the API. In this case, "*" means acceptance of requests from all sources.

- Data validation on the frontend ("ApplicationForm.jsx" file):
  It is a key aspect of every form in a web application. It ensures that the data entered by users is in the correct format and meets certain criteria before it is uploaded to the server. This is important for both the usability of the app and security, as it helps prevent bugs and potential attacks.

  The "useState" hook was used to track errors in the form (it stores validation errors for individual fields):

```
const [errors, setErrors] = useState({});
```

An example of a function that validates data is a function that validates an entered email address:

```
const validateEmail = () => {
    const regex = /^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[ A-Za-z]+$/;
    validateInput(email_address.current, regex);
};
```

It uses the regular expression "regex", which checks whether an e-mail address consists of: one or more letters of the Latin alphabet or numbers before the "@" sign, after the "@" sign, another string of letters and numbers, a period "." followed by one or more letters of the alphabet.

On the other hand, error messages are displayed in the "TextField" fields, thanks to which the user receives a message about the incorrectness of the data entered by the user immediately after submitting the request:

```
          <TextField
                      id="phone_number"
                      required
                      label="Phone number"
                      inputRef={phone_number}
                      error={errors.phone_number}
                      helperText={
                        errors.phone_number &&
                        "A phone number can only contain numbers"
                      }
            />
```

- JSON Web Tokens (JWTs):
  They are widely used in web applications to securely transfer information between a client and a server. They are particularly useful for authenticating and managing user sessions.
  Token generation in the project ("auth.py" file):

```
Import JWT
from decouple import config

JWT_SECRET = config("secret")
JWT_ALGORITHM = config("algorithm")

def sign_jwt(date: str):
    payload = {"data": data, "expires": time.time() + 300}
    token = jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALGORITHM)
    return token_response(token)
```

The JWT token is generated using the "jwt" library. The token contains a "payload" containing the user's data and information about the token's expiration, which is encrypted using a secret key (JWT_SECRET) and algorithm (JWT_ALGORITHM).

Use of JWT Tokens in the project:

1) User Authentication: Upon successful login, the user receives a JWT token, which is then used to access protected resources.
2) Session Management: JWTs can contain expiration time information, allowing you to control the user's session activity.
3) Security: JWT offers a secure way to store and verify a user's identity, as it is difficult to forge due to the use of an encryption algorithm and a secret key.

6. Summary and Conclusions

The completed project based on the creation of a database for the Population Records Department and a website shows that the implementation of automation can significantly increase the work efficiency of officials, reducing the time spent on routine tasks and allowing for better use of human resources. In addition, it provides convenience of use for both employees and residents (or potentially people applying for registration in the municipality), which may translate into increased attractiveness of employment in the office and increased migration of new people to the municipality. The user and employee interface is simple and clear, so a person of any age should be able to use it. A lot of sensitive information is processed in the Population Registration Department, hence great emphasis has been placed on data security, implementing mechanisms for encryption, authentication and protection against unauthorized access. The use of modern technologies such as Python FastAPI, JavaScript React and MySQL ensures efficiency, stability and the possibility of expanding the system in the future with new functionalities, as well as preparing it for a significant increase in the number of people and data processed for the municipality. The implemented system further ensures the quality and consistency of data, thanks to the use of validation procedures and data integrity constraints, which is necessary to provide reliable and accurate information. Thorough testing for a variety of usage scenarios confirmed the reliability of the system and its readiness for deployment.

7. Literature

1) Documentation of the technologies used:

https://dev.mysql.com/doc/

https://fastapi.tiangolo.com/learn/

https://react.dev/learn

2) Substantive support:

https://www.wroclaw.pl/urzad/zameldowanie

https://bip.um.wroc.pl/artykul/227/3174/wydzial-spraw-obywatelskich

https://www.gov.pl/web/gov/zalatwiaj-sprawy-urzedowe-przez-internet-na-epuap