UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

DESIGN AND IMPLEMENTATION OF AN AUTOMATED FORM GENERATION AND PARSING TOOL

Absolvent
Theodor Pierre Moroianu

Coordonator Ştiinţific
Alin Ştefănescu

București, Iunie 2022

Abstract

Am gândit şi implementat serverul unei aplicaţii de creare, gestionare şi parsare automată a formularelor. Produsul complet este API-ul utilizat de o aplicaţie web modernă (web 2.0), care ofera urmatoarele functionalitaţi: autentificarea utilizatorilor folosind *OAuth2*, generarea formularelor şi descarcarea acestora în format PDF, completarea formularelor online, extragerea automată a raspunsurilor din pozele unui formular imprimat, cât şi control deplin (citire / creare / modificare / ştergere) asupra formularelor şi a răspunsurilor.

Împreună cu aplicaţia client (implementată in cadrul altui proiect), produsul nostru constituie o alternativă a serviciilor precum *Google Forms* sau *Microsoft Forms*. Aplicaţia este disponibilă pe internet la adresa *https://smartforms.ml*, dar poate fi şi găzduită local – codul este open-source şi distribuit sub o licenţă *MIT* permisivă. Prin capacitatea acesteia de-a citi şi parsa formulare imprimate şi completate de mână, aplicaţia este ideală pentru profesori, secretariate sau oricine având de digitalizat cantitaţi mari de date.

Abstract

I designed and implemented the backend of an app focused on the creation and management of forms and surveys. The final product is the API server used by a *Web 2.0* application, offering the following functionality: user authentication (using *OAuth 2*),creation of forms and the ability to download them as PDF documents, filling forms online, extracting answers from pictures or scans of filled forms, and CRUD control over forms and answers.

Together with the client app (which was implemented within another project), our software is an alternative to online websites such as *Google Forms* or *Microsoft Forms*. The application is accessible online at *https://smartforms.ml*, but can be also ran locally – the code is open-source and distributed under a permissive *MIT* licence. Thanks to its ability to digitize printed forms, our software is especially useful to to teachers, receptions or anywhere massive amounts of digitisation are required.

Contents

Chapter 1

Introduction

1.1 Functionality

Our project, called *SmartForms* is centered around creating and digitization of forms, for minimizing human intervention. By using the software we created, users can, between others:

- Create simple forms, in a PDF format, prompting readers to fill in multiple choice questions or manually writing answers.
- Upload scans of filled forms, which are then automatically parsed and added to the list of answers.
- Share the link to an online form, similar to modern survey websites.
- Fully control (View / Edit / Delete) forms and answers.

1.2 Use Case and Target Audience

The main use case of this software is situations where massive amounts of physical data need to be digitized to be further processed, or, due to legal or connectivity issues, alternatives like *Google Forms* or *Microsoft Forms* are not an option.

Due to the its high flexibility, our software aims to solve the given scenarios:

Scenario 1: Written Exams

A teacher wants to organize an exam for a large number of students.
The exam is comprised of questions which are either multiple choice, either with a restrained set of possible answers (e.g. for a geography course, an answer could be "*Paris*", or for a mathematics course a possible answer would be "3.1415").

The teacher doesn't want to give his/her students access to the internet, to avoid fraud, so an online form isn't feasible. On the other hand, a written exam means the teacher has to manually go throughout each exam paper.

Our software gives the teacher two options:

1. Host the software's server locally, letting students connect to the local network without them accessing the internet, and having them submit their answers via our online form submission service.

2. Create and print the exam using our form generation tool, and have the students fill their answers on a printed copy of the form. Then upload a scan of their answers and let the software extract them.

Scenario 2: Covid-19 Tracking

While entering institutions such as hospitals, for legal reasons people are asked to fill a simple form with their name, address and contact information.

In case of emergency, workers have to manually go throughout each form to contact the people concerned.

Using our software, workers can simply upload a scan of the filled forms, and download the entire list of email addresses, making contacting the affected people easier and more reliable.

Scenario 3: Data Protection Legal Issues

With the recent GDPR regulations, data protection is becoming for many companies one of the top priorities.

In this scenario, our software aims to replace online forms or endless email mailing lists, providing a simple, self-contained and self-hosted solution for companies which do not want to use 3rd party questionnaire solutions.

1.3 Scope of the Project

The software is made in collaboration with another student from my cohort.

The complete software is made out of three main components:

- The API server, receiving and answering to requests.
- The Client, running on the users' browsers as a modern web application.
- The environment, connecting the two services from above and making the app visible online.

My project's scope is the API server and the services behind it. It includes:

- Implementing the API routes.
- Implementing the form generation pipeline.
- Setting up the database management system.
- Implementing the parsing pipeline, extracting answers from scanned forms.
- Implementing the authentication mechanism, to allow for different confidentiality levels.

My project does NOT include:

- Implement any user-facing elements.
- Connecting the two services or exposing the application to the internet.

1.4 Related Work

Chapter 2

Tech Stack

2.1 Programming Language

The primary programming language used in the backend is *Python 3*. *Python 3* offers a great flexibility, native support for essential datastructures used in web development such as *JSON* (JavaScript Object Notation) and easy to follow syntax.

The main reason we chose *Python* over more conventional languages for web servers is the abundance of packages available to install via *Pip*, the Python Package Manager.

The main drawback of *Python* is its speed. Being an interpreted language it is slower than traditional languages such as *C/C++* or *Java*.

To illustrate the speed difference, we ran a simple program in both *Python* and *C++*, to see the difference.

The *Python* code is:

```python
N = 300000

primes = []

for i in range(2, N):
    is_prime = True
    for d in primes:
        is_prime = is_prime and i % d != 0
    if is_prime:
        primes.append(i)

print(f"There are {len(primes)} prime numbers up to {N}")
```

The *C++* code, running the exact same instructions is:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int N = 300'000;
    vector <int> primes;
```

```
    for (int i = 2; i < N; i++) {
        bool is_prime = true;
        for (auto d : primes)
            is_prime = is_prime && (i % d);
        if (is_prime)
            primes.push_back(i);
    }

    cout << "There are " << primes.size()
        << " prime numbers up to " << N << '\n';
}
```

Running them gives us:

```
teo@fedora ~/P/T/L/tech-stack> time python ciur.py
There are 25997 prime numbers up to 300000
------------------------------------------------------
Executed in  173.56 secs    fish           external
   usr time  173.12 secs  992.00 micros  173.12 secs
   sys time    0.03 secs    0.00 micros    0.03 secs

teo@fedora ~/P/T/L/tech-stack> g++ -O2 ciur.cpp -o ciur-cpp
teo@fedora ~/P/T/L/tech-stack> time ./ciur-cpp
There are 25997 prime numbers up to 300000
------------------------------------------------------
Executed in    2.57 secs    fish           external
   usr time    2.56 secs  896.00 micros    2.56 secs
   sys time    0.00 secs    0.00 micros    0.00 secs
```

In other words, *Python* runs our (quite naive) prime counting algorithm 60 times slower than the same *C++* code!

While this might seem problematic, in practical cases the difference is less noticeable, and most of our server's computing time is spent by libraries like *OpenCV* or *Numpy*, which are implemented in *C++*, essentially offering us *C++* speeds in *Python*.

2.2 Web Server Framework

The framework we used for implementing the API server is *FastAPI*. *FastAPI* is a scalable, lightweight and efficient *Python* framwork, whose main advantages are:

- Automatic validation, serialization and deserialization of the data sent and received by the API endpoints.
- Automatic generation of the *OpenAPI* (Swagger) specifications, which can then be used as a reference point for using the API in the frontend.
- Asynchronous programming and multiple threads, making it fit for CPU-intensive tasks.

2.3 Database Management System

As our application has to store information such as existing users, created forms or answers to a persistent medium, we integrated a database management system to it.

We decided to use *MongoDB*, a document-oriented *NoSQL* system.

2.4 Image Manipulation Framework

For parsing scanned documents, correctly identifying the location of the answers on them and manipulating the images we use *OpenCV*, one of the best computer vision frameworks available together with *Numpy*, the industry standard to data manipulation in *Python*.

While *OpenCV* and *Numpy* are written in *C++* for performance reasons, *Python* bindings allow us to fully use their extensive set of functionality without leaving *Python* land.

2.5 Machine Learning Framework

The parsing part of *SmartForms* is done with the help of machine learning, achieved using a convolutional neural network.

The neural network is implemented in *Pytorch*, one of the best deep learning libraries.

Chapter 3

Design of the Application

3.1 Application Modules

To simplify the structure of the project and make the code easier to understand, the server is divided into different modules, each one having a different scope (single-responsibility principle). The modules are:

- The database module, handling the connection with the *MongoDB* database.
- The OCR module, taking care of the neural network and character prediction.
- The PDF processor module, which generates and parses the PDF forms.
- The router module, handling the API calls.
- The storage module, in which internal types are defined.
- The testing module, where unit and integration tests are defined.

Each module is designed to act as an independent unit, interacting with the others only with the help of a reduced number of function calls / *Singleton* interfaces.

Modules are written according to the standard *Python* conventions:

- The source files for each module is located in a folder with the module's name.
- At the root of the folder, a file named `__init__.py` is created. This file is then populated with all of the functionalities the module needs to export.

A sample `__init__.py` file is:

```
# backend/sources/smart_forms_types/__init__.py
"""
Types used for storing / processing documents.
"""

import uuid

def generate_uuid():
    """
    generates an unique ID.
```

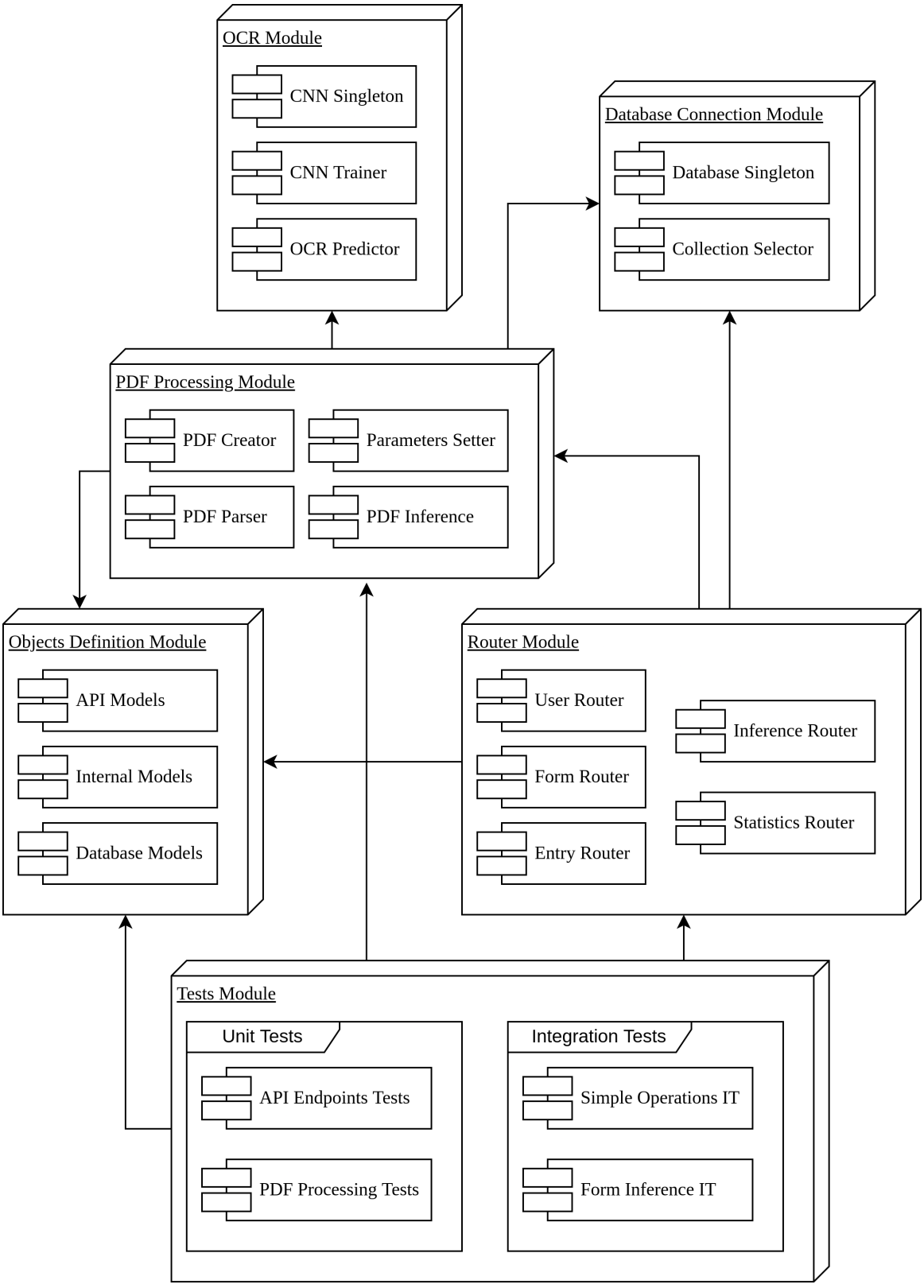


Figure 3.1: UML Diagram Of SmartForms's Backend Modules

```
    the probability of collisions between different
    uuids is beyond negligable.
    """
```

```
    return str(uuid.uuid1())

from smart_forms_types.pdf_form import *
from smart_forms_types.models import *
from smart_forms_types.user import *
from smart_forms_types.inference_dataset import *
```

Chapter 4

Detailed Overview of the API

4.1 What is an API, and why do we need one?

The *API* – or *Application Programming Interface* is an interface acting like a bridge between applications, essentially allowing for an over-the-network communication between two applications.

The most common form of *API* implementations are made with the help of the *HTTP* protocol, an application-layer protocol on top of *TCP*. For showcasing the utility of using an *API*, we can make a simple request to get the current time:

```
teom@lapttop ~> curl "http://worldtimeapi.org/api/timezone/Europe/Bucharest"\
                  | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   395  100   395    0     0   1220      0 --:--:-- --:--:-- --:--:--  1222
{
    "abbreviation": "EEST",
    "client_ip": "5.12.1.116",
    "datetime": "2022-05-20T12:17:44.160367+03:00",
    "day_of_week": 5,
    "day_of_year": 140,
    "dst": true,
    "dst_from": "2022-03-27T01:00:00+00:00",
    "dst_offset": 3600,
    "dst_until": "2022-10-30T01:00:00+00:00",
    "raw_offset": 7200,
    "timezone": "Europe/Bucharest",
    "unixtime": 1653038264,
    "utc_datetime": "2022-05-20T09:17:44.160367+00:00",
    "utc_offset": "+03:00",
    "week_number": 20
}
```

This shell script uses the *curl* program, used for querying content over the network, which:

1. Breaks down the *URL* we requested (`http://worldtimeapi.org/api/timezone/Europe/Buchare`
   into two components:

- The domain – `http://worldtimeapi.org`, which in turn is changed into an *IP* address with the help of a *DNS* server.
- The resource path (or route) – `/api/timezone/Europe/Bucharest`, used by the API server to determine what information we want to receive.

2. Sends the request to the API server.

3. Receives back the payload sent by the API server, which is piped into the *Python JSON* displaying tool to be easy to read.

In the example above, the *API* function we called doesn't have any side-effects – it doesn't change the internal state of the server. However, in most practical cases we need to be able to alter the stored information.

4.2 SmartForms *API*

The entire *SmartForms* backend software is aimed at being able to respond to API calls. In this section we will go over all of the available endpoints of the *API*, exposing all of the user-facing functionalities of the application. They are divided into categories, each one determined by its path and served by a different router.

Note that we will only give a short description of the available endpoints (the *HTTP Verb* and the path used). To see additional details such as parameters or security restrictions, please check the resources mentioned in the appendix.

4.2.1 User Router

This router exposes functionalities relating to the authentication, registration and deletion of accounts from the platform.

Its endpoints are:

- `GET /api/user` – shows a simple message telling the user if he/she is logged in, and promps it to sign in or sign out.
- `GET /api/user/login` and codeGET /api/user/auth – both endpoints help the user to sign-in / sign-up.
- `GET /api/logout` – signs out the user.
- `DELETE /api/delete-account` – deletes the user's account, his/her forms and his/her entries.

4.2.2 Form Router

The *Form Router* handles the PDF form creation / modification / deletion and retrieval.

Its endpoints are:

- `POST /api/form/preview` – receives a description of a form (questions, title, author etc) and returns a preview of the final PDF form. This is especially useful as users tend to try out multiple formats before settling for a form.

- `POST /api/form/create` – similar to the *preview* endpoint, but commits the created form to the database.
- `POST /api/form/list` – returns a list of available forms, depending on filtering criteria.
- `GET /api/description/{formId}` – returns the description of a given form.
- `GET /api/form/pdf/{formId}` – returns the *Base64* encoding of a given PDF form.
- `DELETE /api/form/delete/{formId}` – deletes the form and all of its associated answers.
- `PUT /api/form/online-access/{formId}` – updates the visibility of the form (for instance if anyone can submit an answer to it online).

Note that it is **not** possible to change the actual content of a form. This is by design, as having multiple versions of the same form would be a hassle. The *update* of a form is done on the client-side, by simply cloning the description of the initial form and creating a new one.

4.2.3 Entry Router

Similarly to the *Form Router*, the *Entry Router* contains API endpoints manipulating entries (or answers).

Its endpoints are:

- `POST /api/entry/create` – adds a new answer to a given form.
- `DELETE /api/entry/delete/{entryId}` – deletes the entry with the given Id.
- `PUT /api/entry/edit` – deletes the entry with the given Id.
- `GET /api/entry/view-entry/{entryId}` – returns the information provided in the given answer.
- `POST /api/entry/view-form-entries` – returns a list of forms respecting a set of given criteria.

4.2.4 Inference Router

The single scope of the *Inference Router* is to receive scanned PDF documents, *zip* compressed files, images etc. which are then parsed.

When the software is able to detect a form, it parses it and extracts the answer, which is then added to the database. The endpoint is `POST /api/inference/infer`.

Another functionality of the *Inference Router* is to extract character data:

1. When an answer is uploaded, the `infer` endpoint saves for each character the corresponding sub-image of the answer.
2. If the answer is later modified by a `/api/entry/edit` call, then an annotated character datapoint is generated.

3. The OCR neural network is trained on the generated dataset.

In other words, if the inference failed, a character is wrongly identified. If someone updates the answer with the correct one, then the software finds the difference and generates a new labeled image it can then use to fine-tune the OCR neural network.

4.2.5 Statistics Router

The *Statistics Router* is aimed at providing additional information about *SmartForms*.

Currently, the only available endpoint is `GET /api/statistics/global`, which returns information about the total number of forms and entries, but the router is implemented to simplify adding more advanced statistics in the future.

4.3 Usage Scenario

We will now present a simple usage scenario. We will communicate with the API using the `curl` command to better illustrate its functionality, but please note that in normal circumstances users only interact with the client, which performs the API calls in their behalf using the `fetch API`.

Let's create a new form, with the title "*Sample Form*", and two questions, one asking for the user's gender and the other for his/her name:

```
teo@fedora ~> curl -X 'POST' \
                   'http://smartforms.ml:5000/api/form/create' \
                   -H 'accept: application/json' \
                   -H 'Content-Type: application/json' \
                   -d '{
                "title": "Sample Form",
                "description": "Sample form to showcase the API.",
                "questions": [
                  {
                    "title": "Gender",
                    "description": "Boy or Girl",
                    "choices": [
                      "Boy",
                      "Girl"
                    ]
                  },
                  {
                    "title": "Name",
                    "description": "Enter your name, in uppercase.",
                    "maxAnswerLength": 36,
                    "allowedCharacters": "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
                  }
                ],
                "canBeFilledOnline": true,
                "needsToBeSignedInToSubmit": true,
```

```
                "creationDate": "2022-05-24T07:29:36.216Z"
            }'
```

We get back from the API the following *JSON* object:

```
{
    "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
    "formPdfBase64": "JVBERi0xLjMKMyAwIG9iago8PC9UeXBl...."
}
```

We have now added the form to the database, and received back its Id and a *Base64* encoding of the Pdf document. Let's decode it.

We first have to manually copy the content of the `formPdfBase64` field, and we can then run:

```
teo@fedora ~> pbpaste | base64 -d > form.pdf
teo@fedora ~> open form.pdf
```

Please note that `pbpaste` is a non-standard utility outputing the content of the clipboard. The `open` command opened a Pdf viewer, with our form:

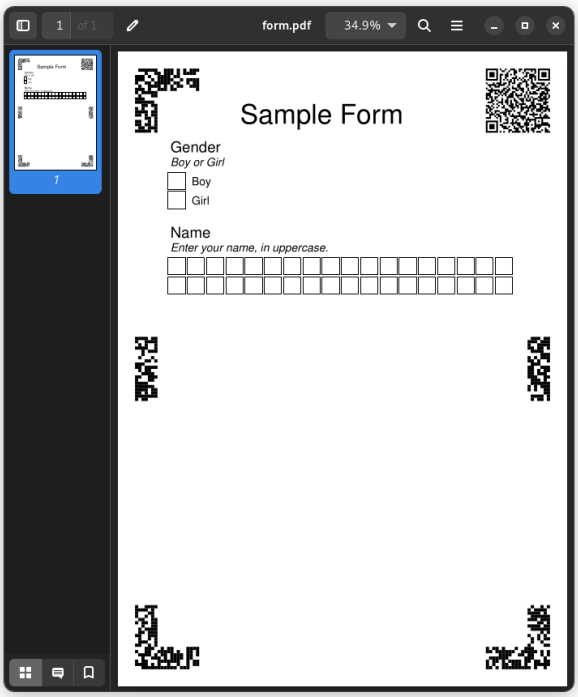


Figure 4.1: Sample form created with the SmartForms API

The features of the Pdf form are:

- The questions, where users are prompted to either write an 'X', '*' or a similar symbol for multiple choice questions, and capital letters for regular questions.
- Lateral markers, forming a binary grid of small squares generated at random, which are added to maximize the number of features we can extract to match a scan / picture of the form with the original.

- The QR code (top right corner), playing a double role:
  - Users can scan the form, which sends them to the webpage where they can fill the form online.
  - The parsing software uses the QR code to extract the ID of the form, to know what it should look for.

We can now add an answer directly from the API:

```
teo@fedora ~> curl -X 'POST' \
                    'http://smartforms.ml:5000/api/entry/create' \
                    -H 'accept: application/json' \
                    -H 'Content-Type: application/json' \
                    -d '{
                "answerId": "",
                "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
                "authorEmail": "",
                "answers": [
                  "X ",
                  "Theodor Moroianu"
                ]
              }'
# the command returns the following JSON:
{
    "entryId":"entry-04896360-db67-11ec-a6ed-dca6325bcf52"
}
```

We can also print the form, and fill it by hand. We can then pass to the inference API the following scan:

We then get back the following data:

```
{
  "entries": [
    {
      "answerId": "entry-b7ef7ca8-db69-11ec-a510-704d7ba4fc42",
      "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
      "authorEmail": "theodor.moroianu@gmail.com",
      "answers": [
        "X ",
        "THEODOR MOROIANU                    "
      ],
      "creationDate": "2022-05-24T16:59:24.561788"
    }
  ],
  "errors": []
}
```

SmartForms is able to:

1. Correctly detect, from the QR code, the form present in the scan.

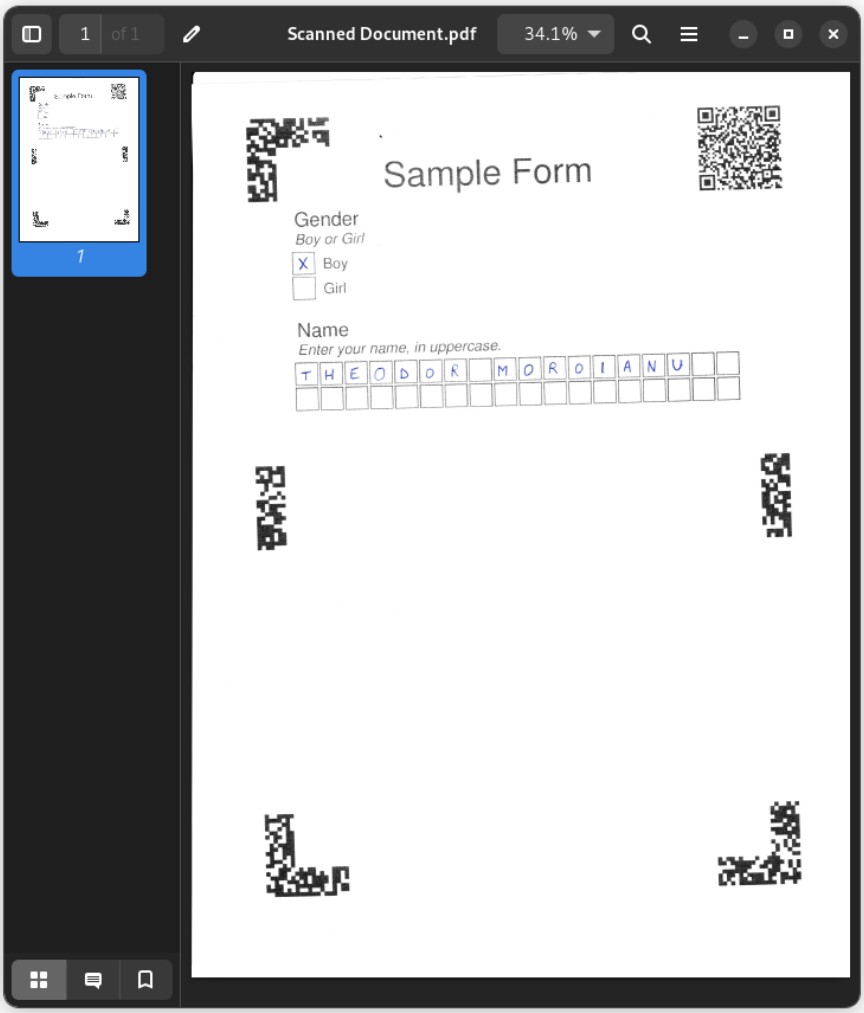


Figure 4.2: Scan of a manually filled form

2. Extract each individual answer square.

3. Run the squares through an OCR neural network, to predict the most plausible character.

4. Add the answers to the database and return them to the users.

It should be noted that the process is not flawless, and is discussed in more details in the Form Parsing section.

We can now query the API to retrieve our two entries:

```
teo@fedora ~> curl -X 'POST' \
                    'http://smartforms.ml:5000/api/entry/view-form-entries' \
                    -H 'accept: application/json' \
                    -H 'Content-Type: application/json' \
                    -d '{
                "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
```

```
                "offset": 0,
                "count": 2
              }' | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   568  100   485  100    83   1473    252 --:--:-- --:--:-- --:--:--  1726
{
    "entries": [
        {
            "answerId": "entry-04896360-db67-11ec-a6ed-dca6325bcf52",
            "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
            "authorEmail": "theodor.moroianu@gmail.com",
            "answers": [
                "X ",
                "Theodor Moroianu"
            ],
            "creationDate": "2022-05-24T13:40:04.586000"
        },
        {
            "answerId": "entry-b7ef7ca8-db69-11ec-a510-704d7ba4fc42",
            "formId": "af61ce94-db35-11ec-a6ed-dca6325bcf52",
            "authorEmail": "theodor.moroianu@gmail.com",
            "answers": [
                "X ",
                "THEDDOR MOROIANU                    "
            ],
            "creationDate": "2022-05-24T16:59:24.561000"
        }
    ],
    "totalFormsCount": 2
}
```

We can, in a similar fashion, edit / delete answers and forms.

Chapter 5

Storage System

5.1 Choosing MongoDB

We explored multiple database systems for our application. For choosing the most suited one, we made a list of requirements:

- We need an easy to set-up, efficient system we can install on relatively small devices.
- We need a system well integrated within the *Python* ecosystem.
- We need a system able to run both locally and in the cloud, to accommodate the different working scenarios of *SmartForms*.

On the other hand, we do not need:

- A system able to process massive amounts of data, as the database is mainly used for storing form descriptions and form answers.
- A system able to perform complex `joins` or similar operations typically done with the help of a *DML* (Data Manipulation Language).

The database system we decided to use within *SmartForms* is *MongoDB*, a document-based, No-SQL database storing data as *JSON* objects.

The connection to *MongoDB* is done in the `database` module. The URI, username and password are specified with the help of the `.env` secrets file, which in the current configuration connects *SmartForms* to a database stored on the cloud:

```
# Connect to Mongo Cloud
MONGO_USER='smart-forms-user'
MONGO_PASSWORD='*********'
MONGO_CLUSTER='cluster0.t96gc.mongodb.net'
MONGO_DB_NAME='SmartForms'
```

5.2 Information Stored in MongoDB

MongoDB databases, contain multiple collections storing entries, similar to SQL databases containing tables storing rows.

We set-up multiple databases:

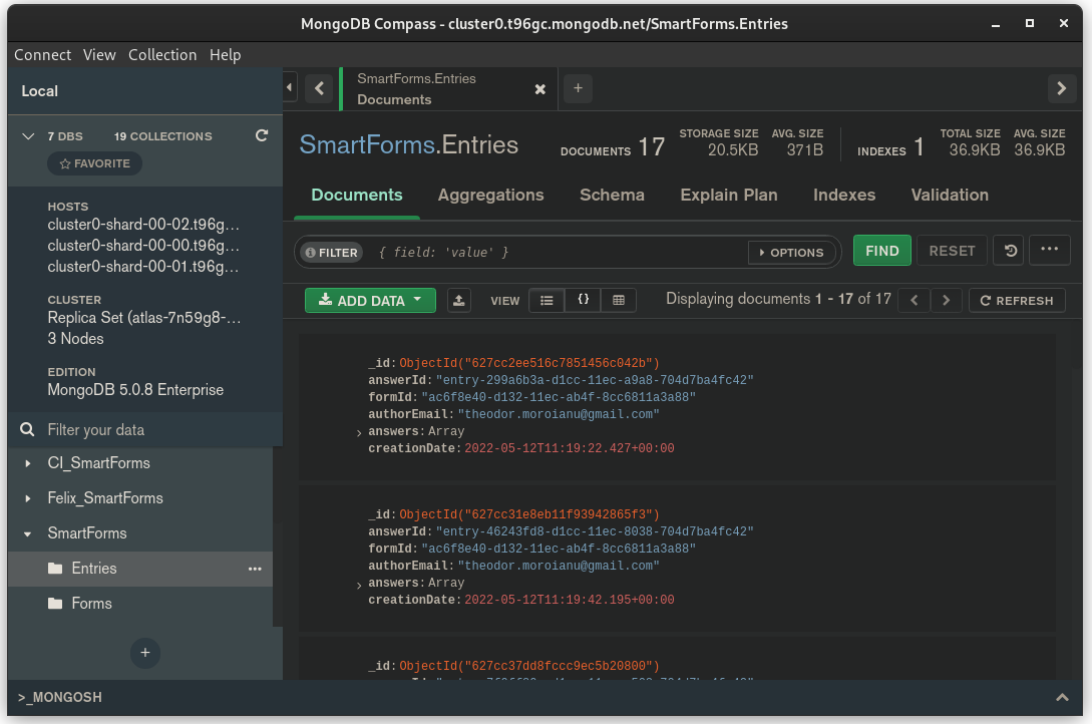


Figure 5.1: Compass – the official MongoDB viewer

- A database used for production.
- Another database used for the *CI/CD* pipelines.
- Databases used for local development and testing without disturbing the other environments.

Databases store the information used by *SmartForms*. As forms and answers requested by the users are unpredictable, and *SmartForms* is built to be able to run on relatively low-end devices, locally caching data for lower latency didn't make sense. As such, any CRUD (Create, Read, Update or Delete) operations performed on forms, answers or users are directly committed to the database.

The database stores:

- The registered users. The information saved is the name, email, date of registration and last sign-in, and an URL to a profile picture if available. To be compliant with data-protection laws like the GDPR, we only use the personal data for authentication purposes, and delete all of an user's records when he/she deletes his/her account.
- The created forms – for which we store the description, owner, creation date and the internal representation of the corresponding PDF document.
- The answers, both added directly with the API or uploaded as a scan or a picture.
- Images of the squares extracted from pictures and scans.
- A dataset of labeled characters which can be used for improving the OCR neural network.

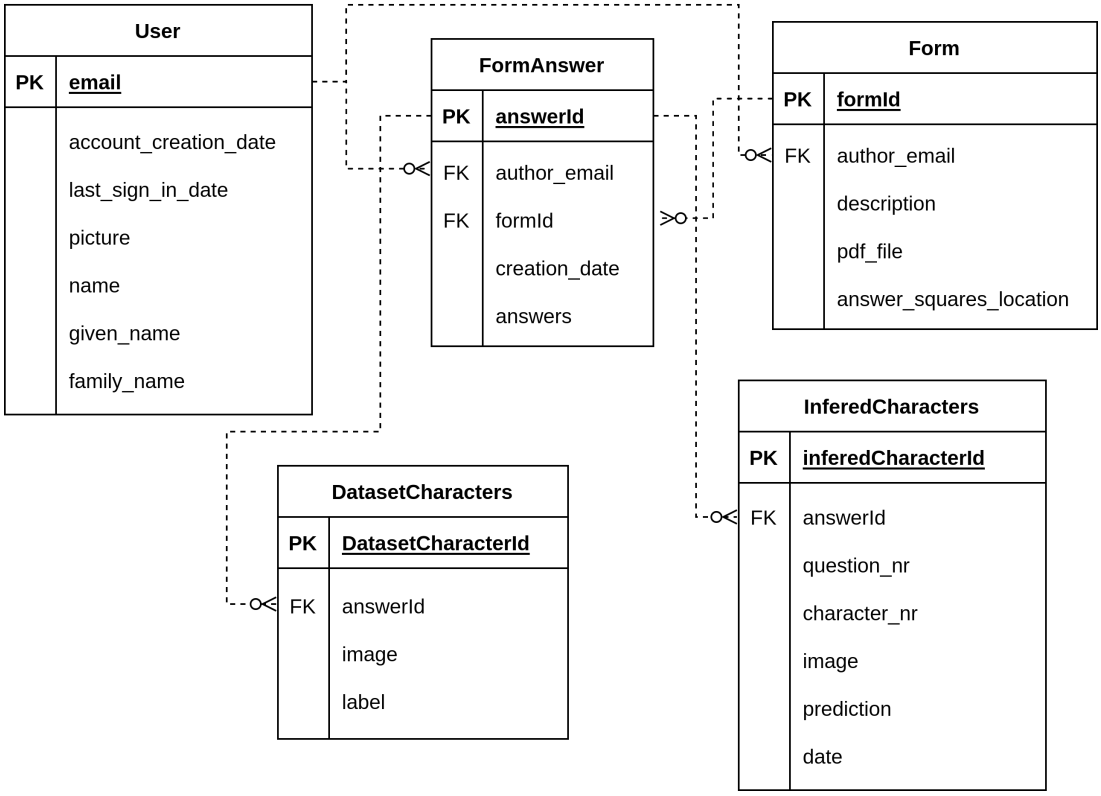


Figure 5.2: Diagram of the MongoDB Database

The last two collection – namely the `DatasetCharacters` and the `InferedCharacters` are used for automatically labeling characters, thus generating a dataset we can then use for fine-tuning the OCR neural network. A more detailed explanation of the process is done in the *Form Parsing* section.

Chapter 6

Pdf Forms Creation

6.1 Objective and Constraints

The primary objective when generating a *PDF* form is getting a concise, easy to understand and professional looking document, which can then be efficiently parsed and digitized.

The main contraints and details we have to consider are:

- While in most usecases users want to print their forms on *A4* paper, we should also make forms printable on smaller or larger paper.
- We want to allow both scans and pictures, made by a wide range of devices, in which brightness, color saturation and contrast differ.
- Modern smartphone cameras, because of the shrinkage of the size of their optical instruments, add slight distortions to the pictures (lines become curves). As each camera is different, fixing the distortions ourselves is not feasable.
- Even though most printers advertise the ability to print edge-to-edge, most modern printers aren't able to print anything too close to the edge of the paper.
- As most office printers only print in black and white for efficiency reasons, we shouldn't include color in our documents.
- People have extremely diverse styles of handwriting, which makes parsing cursive text with high accuracy difficult, even with modern technology.
- Forms can have varying lengths, from a simple question to multiple pages.

6.2 Layout of a Pdf Form

To not violate any of the constraints mentioned above, we embedd multiple components in the documents.

6.2.1 Questions

Questions are the most obvious element of the forms. They are made out of a question title – or simply speaking the question, a question subtitle (or explanation) which is optional, and input zones depending on the type of question.

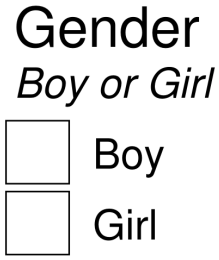


Figure 6.1: Sample multiple-choice question

If the question is a multiple choice question, where the user has to select a subset of the given possibilities (e.g. "Boy" or "Girl"), the each option is printed on a different line, with a square the user can tick to select the question.

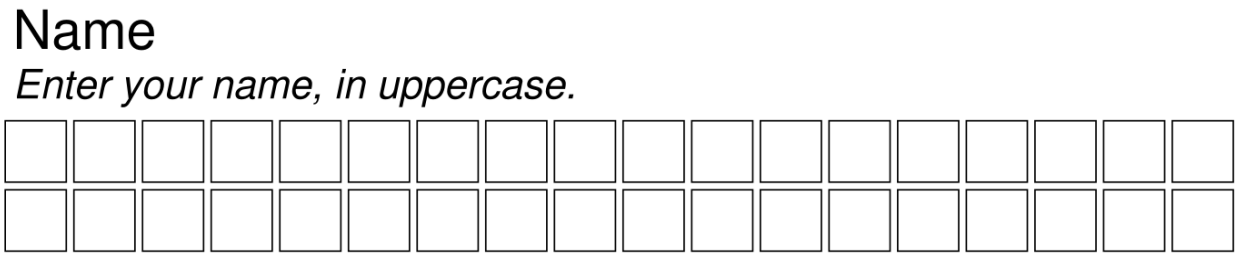


Figure 6.2: Sample text question

If the question is a text question, then a user-specified number of squares are printed. People filling the form can then write one letter per square.

The main advantages of this approach are:

- Multiple-choice and text questions can be processed similarly, as we simply want to check if boxes from the multiple choice questions contain an 'X', 'V', '*' or similar marker.
- This format is writing-style agnostic, as users are forced to write each letter to a specific location in a specific format.
- The forms are easy to understand and fill-in.

Note that the exact location of each square is saved in the database, to allow later pinpointing of the exact pixels of each character.

6.2.2 Markers

Lateral matching helpers, or markers, are irregular patterns we print on the corners.

Similar shapes are often printed on objects that robots or software agents need to be able to identify, due to their vast number of descriptors. *SmartForms* uses a *ORB* feature extractor[6], which relies on intensity differences between adjacent pixels.

With the help of the *ORB* feature extractor, we can then match extracted features of the blank Pdf document with a picture or scan, to get an accurate rectangular representation of the scanned image.

Figure 6.3: Top left marker

6.2.3 QR Code

QR codes (shorthand for Quick Response codes) are visual machine barcodes. QR codes are machine-readable, use the Reed–Solomon error correction algorithm[1], and can store arbitrary data, from Wi-Fi details to contacts or simple text.

Figure 6.4: QR code with content "*https://smartforms.ml/view-form/af61...*"

We use QR codes for multiple reasons:

- Due to their apparent randomness, they also help the feature extraction of the matching process.
- They offer an easy-to-follow link to the *SmartForms* website, which users can open to fill the form online.
- It allows the matching software to extract the form ID, to figure out which form is being parsed.

6.2.4 Preview Notice

Users tend to commit their work more often than is strictly required: people writing documents save them every few minutes, developers writing code compile it to check for mistakes, and, to no surprise, people generating forms tend to preview intermediate results.

As such, adding each intermediate result to the database doesn't make much sense. On the other hand, users have to easily see if a form is valid or not. This is why, on forms which are only made for preview, we print an additional "*PREVIEW*" banner.

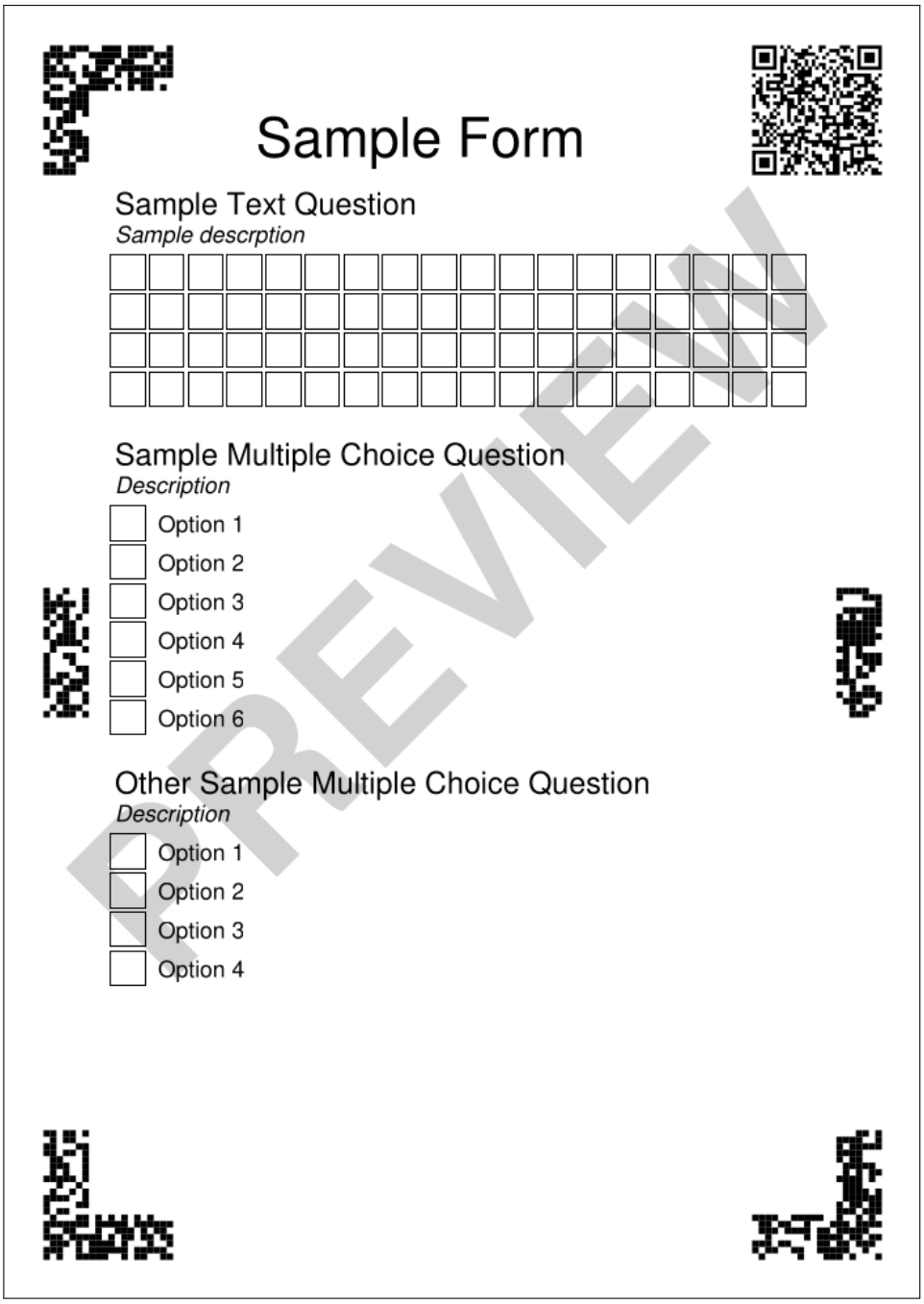


Figure 6.5: Form with "*PREVIEW*" banner.

6.2.5 Multi-page Support

If the form has many questions, then the Pdf document might spread among multiple pages. In such situations, each page gets a unique ID encoded within the QR code, to allow the parsing pipeline to order them accordingly.

The lateral markers and the optional preview notice are present on all the pages.

Chapter 7

Form Parsing

7.1 How Data is Loaded

The API relies on the `multipart/form-data`[2] format to receive reliably larger files than what is usually permitted over the *HTTP* protocol.

To facilitate the uploading of filled forms, we accept:

- A list of images, in any of the standard image formats (`.jpeg`, `.png`, `.webp` etc).
- A list of PDF documents.
- A *Zip* archive, the content of which is processed recursively.

Due to the high flexibility and permissive formats supported by *SmartForms*, virtually any structured folder of forms can be zipped, uploaded and parsed.

Once the backend receives the files, it automatically deflates all zip files, and:

1. Considers each PDF as a folder, whose content are the pages of the document (each page is considered as a single picture).
2. Splits all the images into groups, according to their containing folder.
3. Parses each group as a single form.

7.2 Finding the Template

Given a set of images (obtained directly from the user, from a zip file or extracted from a page of a PDF document), the backend has to figure out which form is being parsed.

To do this quickly and reliably, we look for the QR code inserted on each document. Finding the QR code is itself a challenge, as most libraries used for scanning QR codes (like the ones used on mobile phones) expect a close-up picture of the code. However, because we can't know the orientation, size or location of the form in the received picture, we can't reliably determine the exact location of the QR code.

For solving this issue, we use `Zbar`, an open-source bar reading library written in *C*. While the application isn't written in *Python* for compatibility and performance reasons, the `pyzbar` package provides *Python* bindings we can use.

Zbar is highly optimized for real-time barcodes and QR codes scanning, and can be also used from a terminal. By passing it the form we created above we get:

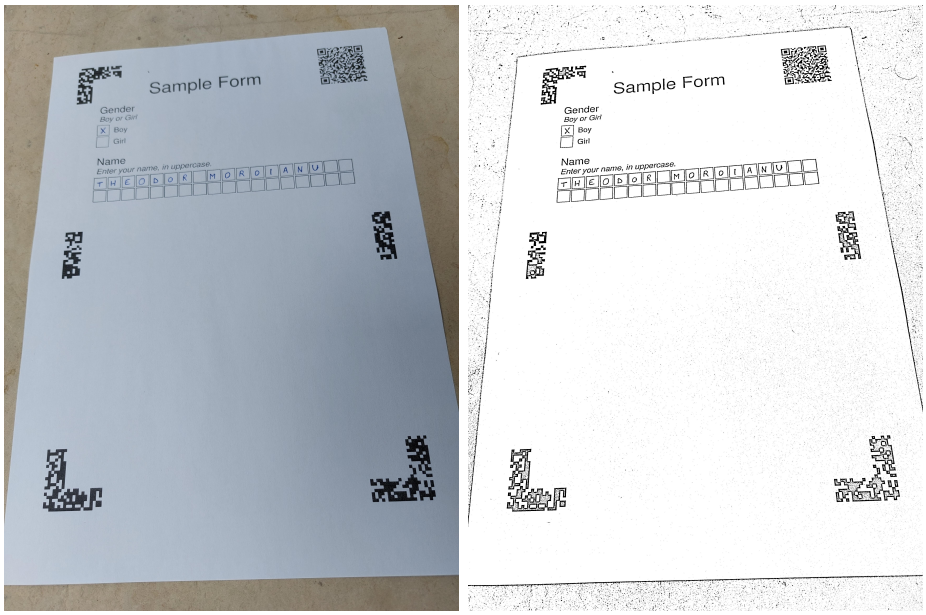
```
teo@fedora ~> zbarimg Scanned\ Document.pdf
QR-Code:
https://smartforms.ml/view-form/af61ce94-db35-11ec-a6ed-dca6325bcf52

scanned 1 barcode symbols from 1 images in 0.02 seconds
```

Please note that the text extracted by *Zbar* is not the actual ID of the form, but rather an URL to the *SmartForms* website, where, depending on permissions, users may fill the form online. To extract the real ID, we simply have to remove the "*https://smartforms.ml/view-form/*" prefix. *Pyzbar* operates in a similar way, but can be called from within *Python* code.

With the form ID, we can query the *MongoDB* database to extract all the required details about the form.

7.3 Applying Grayscale and Binary Threshold



(a) Original (b) Processed

Figure 7.1: Picture converted to binary

Most image processing algorithms rely on grayscale or binary images. As such, we first have to convert our colored image to grayscale, and then apply a binary threshold.

To convert an image to grayscale, we can independently transform each pixel from a `RGB` value to a single channel. *OpenCV*, the machine vision framework we are using, converts an image to grayscale using the following formula[3]:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

Our first step is converting a picture or scan of our form into grayscale. Figure 7.1 (a) shows what such a picture might look like. However, even with the image in grayscale we cannot simply apply a standard binary threshold, due to possible differences in brightness.

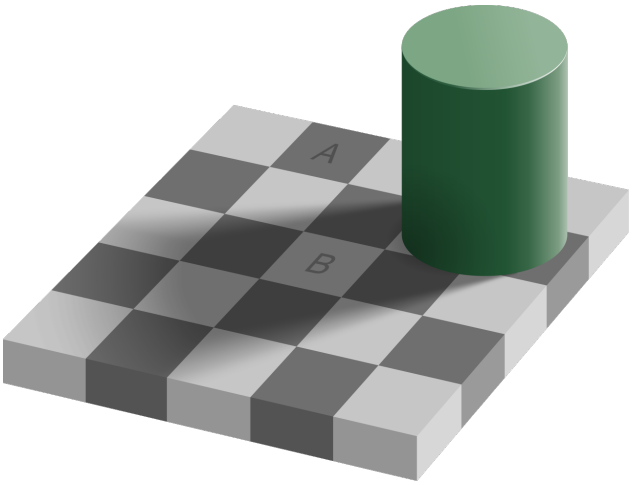


Figure 7.2: Checker Shadow Illusion[5]

As ilustrated in Figure 7.2, where squares marked with **A** and **B** seem different even though the color is the same, separating dark and light regions is not a trivial task, if we want to take into account lighing and brightness. Similarly, applying a binary threshold considering pixel intensities larger than a fixed K black and intensities smaller than K white will lead to misclassified pixels, where darker empty regions of the form might be classified as black, and similarly lighter text or graphics of the form might get classified as white.

To convert our grayscale image to binary, we use an adaptive thresholding technique[4], which for each pixels determines, based on a small region around it, if it should be black or white. Applying such an algorithm over our grayscale image gives us an image similar to Figure 7.1 (b).

7.4 Changing Image Perspective

For parsing the form, we have to be able to match specific pixels of the picture / scan of the form with the template, to figure out where locations we are interested in are. Further in this section, we will call the picture / scan of the form the *picture*, and the original form, retrieved from the database, the *template*.

Matching the picture with the template is not trivial, as things become quite messy:

- If the picture keeps the same proportions, orientation and scaling as the template, then we have a perfect 1-1 match between the two.
- If the picture keeps the same proportions and orientation, but the scaling is not 1, then a scaling factor is needed in order to be able to match the picture and the template.
- If the picture only keeps the same proportions, then in addition to the scaling factor a rotating factor is also required.
- If the picture does not keep the same proportions, then we have to find a homography between the two.

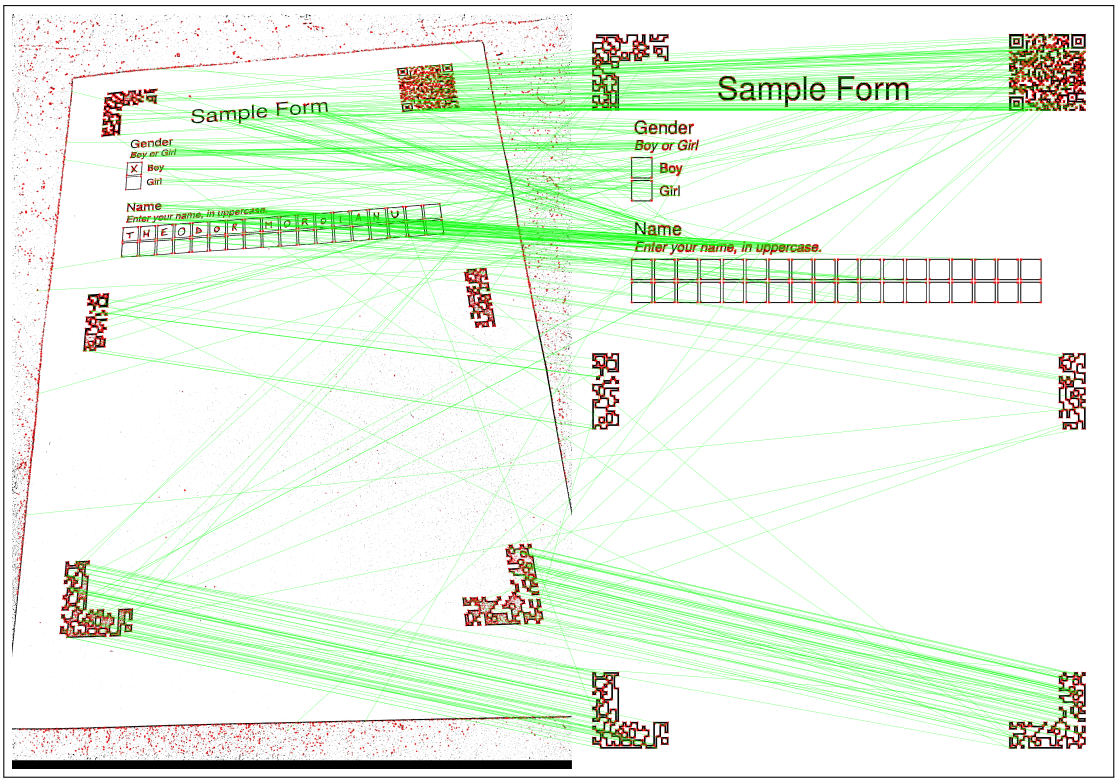


Figure 7.3: Matches found with ORB

As finding homographies between two images is a quite common task, *OpenCV* offers the ORB feature extractor[6], which is able to extract and match features of the two images, as seen in Figure 7.3. If enough matches are found, we can change the image's perspective, to get a frontal view of the form.

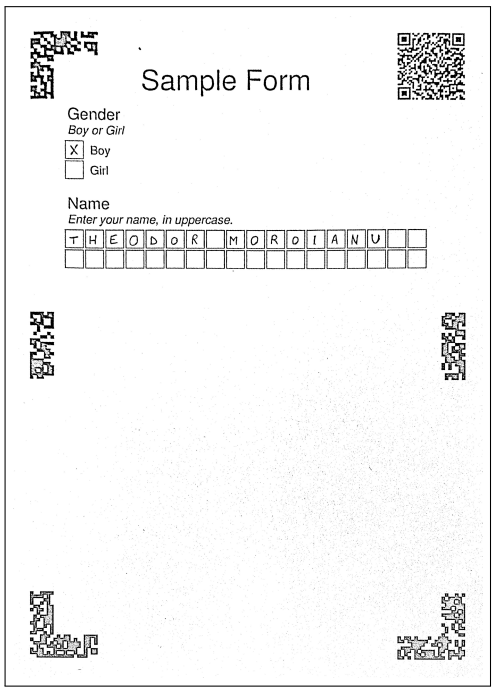


Figure 7.4: Corrected Picture

In Figure 7.4 we can see the result of our perspective transformation over the binary picture. We can observe that:

- The thresholding is not perfect, as some white areas have a few black pixels, and black text has a few white spots, which means that our OCR network has to be resistant to small noise.
- Straight lines (more specifically the boundaries of the text boxes) seem slightly curved. This is actually the case, as phone cameras add small distorsions to the images. This fenomenon does not appear on scans though.

7.5 Extracting Answer Squares

Extracting squares containing an answer from a form is straightforward, given a corrected image. As we save the position of squares where users are writing their answers when we generate the form, we simply have to extract said square.

For each form we store in the database a list with the absolute position of each answer square. An sample of the stored information is:

```
answer_squares_location:
    - 0:
        - 0:
            - width: 8
            - x: 23.5
            - y: 49.5
            - page: 0
        - 1:
            - width: 8
            - x: 32.5
            - y: 49.5
            - page: 0
        ...
    - 1:
        - 0:
            - width: 8
            - x: 23.5
            - y: 89.5
        ...
```

With the help of the information mentioned above, and of a corrected image similar to Figure 7.4, we can mark and extract each answer square of the form. We marked in Figure 7.5 the answer squares we extract in red.

Figure 7.5: Answer Squares Marked

Chapter 8

Neural Network Design and Training

8.1 Why Use a CNN

Using traditional CV approaches we have been able to isolate small images of the characters we want to extract. While traditional methods (such as HOG[7] approaches) perform reasonably well, we decided to design, train and use a convolutional neural network (CNN)[8], which often outperforms classic classificators.

We use *Pytorch*, "*an optimized tensor library for deep learning using GPUs and CPU*"[9]. *Pytorch* allows us to design, train, save and evaluate a convolutional neural network we then use to predict characters.

8.2 CNN Architecture

8.3 Dataset Used

8.4 Data Processing and Augmentation

8.5 Training

8.6 Contiguous Training

Chapter 9

HTTP/S Server, Secure Authentication

9.1 FastAPI

9.2 Session Middleware

9.3 Oauth2

Chapter 10

Tests, Source Control and CI/CD

10.1 Git

10.2 Smartforms Backend / Frontend

10.3 Testing

10.4 CI/CD

Chapter 11

Conclusion and Future Work

Stuff that can be added to enhance the user experience:
Merge with conclusion

- Add in the QR code info about the form, to not require a database.
- Add multi-threading while reading forms, to speed up the process. 23

Bibliography

[1] S. B. Wicker, V. K. Bhargava, *Reed-Solomon codes and their applications*, John Wiley & Sons, 1999.

[2] L. Masinter, *RFC2388: Returning Values from Forms: multipart/form-data*, RFC Editor, 1998.

[3] *OpenCV: Color Conversions*, n.d., accessed 29 May 2022, https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html.

[4] *OpenCV: Image Thresholding*, n.d., accessed 29 May 2022, https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

[5] E. H. Adelson, Pbroks13, *checker shadow illusion*, accessed 29 May 2022, https://commons.wikimedia.org/w/index.php?curid=75000950.

[6] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, *ORB: An efficient alternative to SIFT or SURF*, 2011 International conference on computer vision, IEEE (2011), 2564–2571.

[7] N. Dalal, B. Triggs, *Histograms of oriented gradients for human detection*, 2005 IEEE computer society conference on computer vision and pattern recognition, IEEE (2005), Vol. 1, 886–893.

[8] K. O'Shea, R. Nash, *An introduction to convolutional neural networks*, arXiv:1511.08458, 2015.

[9] *Pytorch documentation*, n.d., accessed 30 May 2022, https://pytorch.org/docs/stable/index.html.