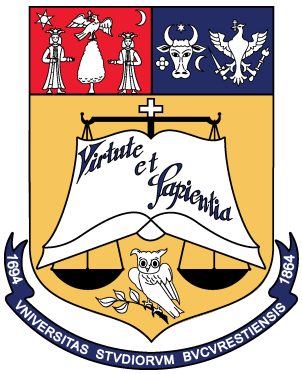


UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS
AND
COMPUTER SCIENCE

COMPUTER SCIENCE SPECIALIZATION

Bachelor thesis

SMARTFORMS API: AN AUTOMATED TOOL FOR FORM GENERATION AND PARSING

Student
Theodor Pierre Moroianu

Thesis Coordinator
Prof. dr. Alin Ştefănescu

Bucharest, June 2022

Abstract

Am dezvoltat un framework care extrage in format text continutul unor formulare completate de mână şi fotografiate. Produsul complet este API-ul utilizat de o aplicaţie web modernă (*Web 2.0*) numită "*Smart Forms*", care oferă următoarele functionalităţi: autentificarea utilizatorilor folosind *OAuth2*, generarea formularelor şi descarcarea acestora în format *PDF*, completarea formularelor online şi extragerea automată a răspunsurilor in format editabil din pozele unui formular completat de mână. Aplicaţia oferă de asemenea control deplin (citire, creare, modificare şi ştergere) asupra formularelor şi a răspunsurilor.

Împreună cu o aplicaţie client (implementată în cadrul proiectului de licenţa al unui coleg de an), produsul rezultat constituie o alternativă a serviciilor precum *Google Forms* sau *Survery Monkey*. Aplicaţia este accesibilă pe internet la adresa https://smartforms.ml/, dar poate fi şi găzduită local de utilizatori, codul fiind open-source, disponibil la adresa https://github.com/TeamUnibuc/SmartForms şi distribuit sub o licenţă *MIT* permisivă. Prin capacitatea acesteia de-a citi şi parsa formulare imprimate şi completate de mână, aplicaţia poate fi utilă pentru profesori, secretariate sau institutii care doresc să digitalizeze automat cantităţi mari de date completate de mână.

Abstract

I developped a framework able to extract in a text format the pictures of forms filled by hand. The final product is the API used by a modern web application (*Web 2.0*), called "*SmartForms*", offering the following functionalities: authenticating users using *OAuth2*, generating forms and making them downloadable in a *PDF* format, allowing users to fill forms online, and the ability to automatically parse answers written by hand on printed forms. The framework offers complet control (create, read, update and delete) over forms and answers.

Together with the client application (implemented within the thesis project of a colegue), the resulting product presents an alternative to services such as *Google Forms* or *Survey Monkey*. The application is available online at https://smartforms.ml/, but can be also hosted locally, as the code is open-source, available at https://github.com/TeamUnibuc/SmartForms, and distributed under a permissive MIT licence. Thanks to its ability to parse forms filled by hand, the application can be useful to teachers, front-desk workers, or institutions needing to automatically digitize large amound of hand-filled data.

Contents

Chapter 1

Introduction

1.1 Functionality

SmartForms, our project, is centered around creating and digitizing forms with minimal human intervention. By using our software, users can:

- Create simple forms, downloadable as a *PDF* document (`.pdf` format), allowing either multiple-choice or personalized answers.
- Upload scans of filled forms that are automatically parsed by the framework and added to the database.
- Share a link for users to fill the form online on our website, similar to modern survey websites like *Google forms*.
- Fully control (Create, View, Edit, and Delete) forms and answers.

1.2 Use Case and Target Audience

The main use case of this software is in situations where massive amounts of physical data need to be digitized and further processed, and also whenever commercial online services like *Google Forms* or *Survery Monkey* are not envisageable for legal reasons or because of connectivity issues.

Thanks to its flexibility, our framework can solve the following scenarios:

Scenario 1: Written Exams

A teacher organizes an exam for a large number of students. The exam is comprised of questions which are either multiple choice, or require answers among a well-defined set of expected answers (e.g., for a geography course, the question "What is the capital city of Uganda?" has as admissible answer "Kampala"; similarly, in a mathematics exam, the question "What are the first 5 decimals in π" requires the answer "3.1415").

The teacher doesn't want to allow the students access to the internet, in order to avoid fraud, so using an online form is not an option. On the other hand, a written exam means that the evaluator has to manually go through each exam paper.

Our software gives the teacher two options:

1. Create and print the exam using our form-generation tool, have the students fill their answers on paper copies of the form. The teacher then scans and uploads the papers in the application.

2. Self-host the 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.

Scenario 2: Routine medical questionnaires

While accessing institutions such as hospitals, for legal reasons people may be required to fill a simple form with their name, address, contact information and health status. The front office has to manually go throughout each form and enter the details into the hospital management tool.

Using our software, the hospital can instead upload scans of the filled forms and extract automatically the content with minimal human intrervention.

Scenario 3: Legal Issues concerning Data Protection

With GDPR regulations, data protection has become a top priority for many companies.

In this scenario, our software aims to replace online forms or large email mailing lists, providing a simple, self-contained and self-hosted solution for companies not wanting to use third party questionnaire solutions.

Scenario 4: Street Surveys or Petitions

A group of volunteers ask passers-by to fill in a certain questionnaire. At the end of the day, the volunteers have to manually transcribe all of the filled questionnaires to a spreadsheet document.

Using our software, the tedious task of manually entering the information can be automated, and volunteers only have to load the questionnaires to a self-feed scanner and upload the scanned documents.

1.3 Scope of the Project

Our framework consists of the following components:

- The API routes exposing its functionalities.
- The form-generation pipeline.
- The parsing pipeline, extracting answers from scanned forms.
- The database management system connector.
- The authentication mechanism, to allow for different confidentiality levels.

Note that our framework does not have a UI, as its main focus is exposing the forms-manipulating engine via the API. An example of using the framework through an UI was offered by a student colleague [23]. He showcased my framework via the following two components:

- The *SmartForms* client, a modern web application written in *React*.
- A complete server setup, exposing both the client and the API at `https://smartforms.ml`.

As mentioned above, these components can be found in [23].

1.4 Related Work

We were unable to find another software for generating forms, downloading them as *PDF* files or similar formats, and parsing manually filled documents within the same solution. *SmartForms* seems to be the first software with those functionalities freely available on the internet.

We have found many tools for conducting surveys online. The most most known are *Google Forms*, *Microsoft Forms* or *Survey Monkey*, and a notable mention is *Lime Surveys*, which is one of the only FOSS (**F**ree **O**pen **S**ource **S**oftware) survey software we were able to find [21]. These tools however lack the parsing abilities of SmartForms.

There also exist tools with parsing abilities, like for instance *Docparser* [22]. However, *Docparser* does not support generating forms or parsing handwritten text, which makes it unfit for our usecases.

In conclusion, even by combining several other existing tools one cannot achieve all of the capabilities of *SmartForms*.

A comparison of the features provided by each tool is given in the following table:

	Google Forms	*SurveyMonkey*	*Lime Surveys*	*DocParser*	*SmartForms*
Create Forms	Yes	Yes	Yes	No	Yes
Fill Online	Yes	Yes	Yes	No	Yes
View Results	Yes	Yes	Yes	No	Yes
Export as *PDF*	No	No	No	No	Yes
Parse Scans	No	No	No	Yes	Yes
Open-Source	No	No	Yes	No	Yes
Self-Hostable	No	No	Yes	No	Yes

1.5 Overview of this Thesis

This thesis is divided into several chapters:

- In chapter 1 we describe the functionalities and usecases of the SmartForms application.
- In chapter 2 we present the main technologies used by SmartForms.
- In chapter 3, chapter 4, and chapter 5 we present the internal design of the application, the list of functionalities exposed by *SmartForms API* and the database system used.
- Chapters 6, 7 and 8 are the core of the project. They contain the main inovations of our software, and describe the process of creating, parsing and digitizing *PDF* forms.
- In the chapter 10 we describe the development process of *SmartForms*.
- Chapter 11 contains our conclusions and possible areas for future development.

As stressed above, the sections containing the most inovative content are chapter 6, chapter 7, and chapter 8. In these sections is presented the *PDF* form creation and parsing pipeline, which is in our opinion the most interesting functionality of the framework. Building the pipeline posed us many difficulties, such as acquiring a relevant dataset for training our OCR neural network, testing the pipeline in real-world scenarios, or making the pipeline resistant to invalid or malicious data.

Chapter 2

Tech Stack

2.1 Programming Language

The primary programming language used in the backend of SmartForms 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 relatively low speed. Being an interpreted language, it is slower than traditional programming languages such as *C/C++* or *Java*.

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

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, encoding the exact same algorithm, 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, the *C++* code executes our (quite naive) prime counting algorithm 60 times faster than *Python*!

While this speed discrepency might seem at first problematic, in practice the difference is less noticeable, and most of our software's computing time is spent by libraries like *OpenCV* or *Numpy*, which are under the hood implemented in *C++* (and used within *Python* thanks to language bindings).

2.2 Web Server Framework

The framework we used for implementing the API server is *FastAPI* [15]. *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* [16] (Swagger) specifications, which can then be used as a reference point for using the API in the frontend.
- Asynchronous programming and multithreading, making it fit for CPU-intensive tasks.

2.3 Database Management System

As our application has to store in a persistent medium information such as existing users, already created forms and answers, we had to integrate in it a database management system.

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 chose *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 functionalities without leaving *Python* land.

2.5 Machine Learning Framework

The character recognition functionality of *SmartForms* is achieved by means of machine learning techniques, using a convolutional neural network. This neural network is implemented and trained in *PyTorch*, one of the best deep learning libraries currently available [9].

The necessity and usage of each of the mentioned tools and frameworks is further discussed in the following sections.

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, ilustrated in Figure 3.1, 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 data 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 or Singleton interfaces.

The 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.
    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 *
```

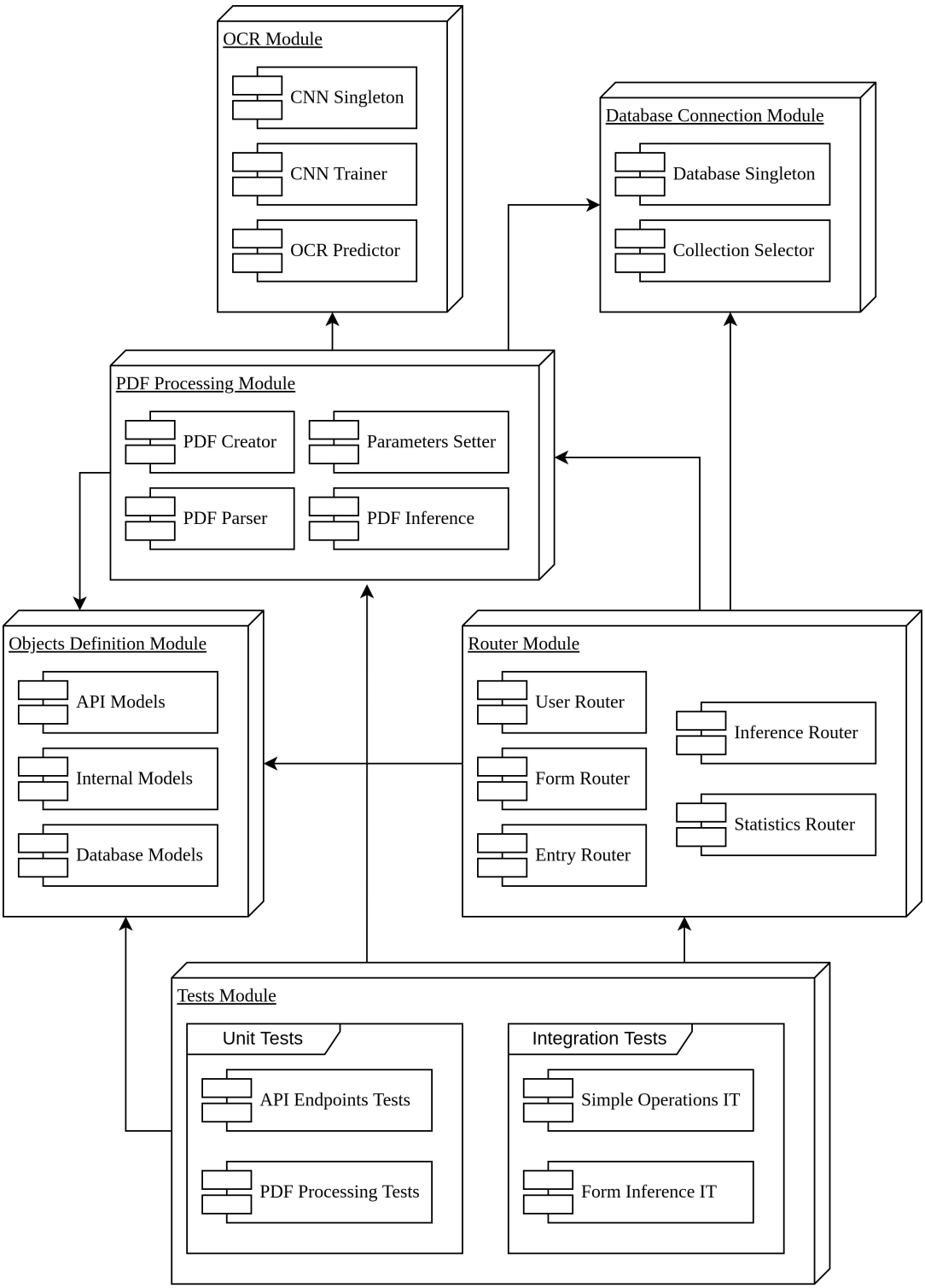


Figure 3.1: UML Diagram Of SmartForms's Backend Modules

Chapter 4

Detailed Overview of the API

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

The API – or **A**pplication **P**rogramming **I**nterface – is an interface acting like a bridge between applications, essentially allowing for an over-the-network communication between them.

The most common API implementations are made with the help of the *HTTP* protocol, an application-layer protocol on top of the *TCP/IP* stack. 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/Bucharest) 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 a payload, which is piped into the *Python JSON* displaying tool to be easier to read.

In the example above, the API function we called does not have any side-effects – it doesn't change the internal state of the server. However, in most practical cases, the API exposes functions mutating 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*, presenting 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 informing users whether they are logged in or not, and prompts them to sign in/out.
- `GET /api/user/login` and `GET /api/user/auth` – both endpoints help the user to sign-in and sign-up.
- `GET /api/logout` – signs out the user.
- `DELETE /api/delete-account` – deletes users' accounts, forms and answers.

4.2.2 Form Router

This router handles the creation, modification, deletion, and retrieval of forms. 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 particularly handy since users tend to try out multiple formats before settling for a final 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 the *PDF* representation of a given 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 whether one can submit an answer to it or not).

Note that once a form is created, it is no longer possible to change its content. 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 purpose of the *Inference Router* is to receive pictures of forms filled by hand, which are then parsed. When the software detects 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, in order to generate a character dataset:

1. When an answer is uploaded, the `infer` endpoint saves for each character the corresponding sub-image containing its pixels.
2. If the answer is later modified, then an annotated character datapoint is generated.
3. The OCR neural network is trained on the generated dataset.

In other words, the inference may fail, leading to a character being wrongly identified. In such cases, if the user updates the answer field by providing a corrected answer, then the software spots the difference and generates a new labeled character image which will subsequently be used 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 designed to make possible the addition of more advanced statistics in future versions of the application.

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, however note that in normal circumstances users only interact with the client, which performs the API calls on their behalf using the `fetch API`.

Let us create a new form with the title "*Sample Form*" consisting of two questions, one asking for the user's gender and the second prompting 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 its *PDF* representation. To 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
```

Here `pbpaste` is a non-standard utility outputing the content of the clipboard. The `open` command opened a *PDF* viewer with our form, as illustrated in Figure 4.1.

The features of this *PDF* document are:

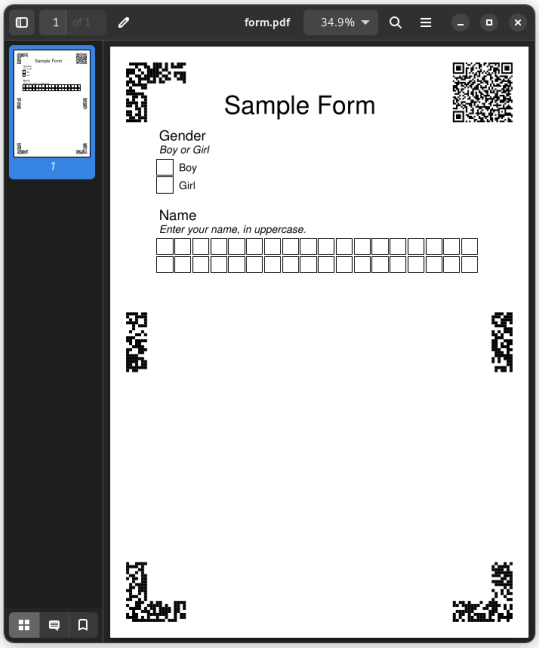


Figure 4.1: Sample form created with the SmartForms API

- The questions, where users are prompted to either write an 'X', '*' or a similar symbol for multiple choice questions, and words for regular questions, one character per box.
- Lateral markers, forming a binary grid of randomly generated small squares, which are added to maximize the number of features we can rely on when matching a scan or a picture of the form with the original document.
- The QR code (top right corner), playing a double role:
  - Users can scan the form, being thus redirected to a webpage where they can fill the form online.
  - The parsing software uses the QR code to extract the ID of the form, allowing it to retrieve the original form from the database.

We can now add an answer directly using the API (the `/api/entry/create` route):

```
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 alternately print the form, fill it by hand and scan it. We can then pass to the inference API the scan showed in Figure 4.2.

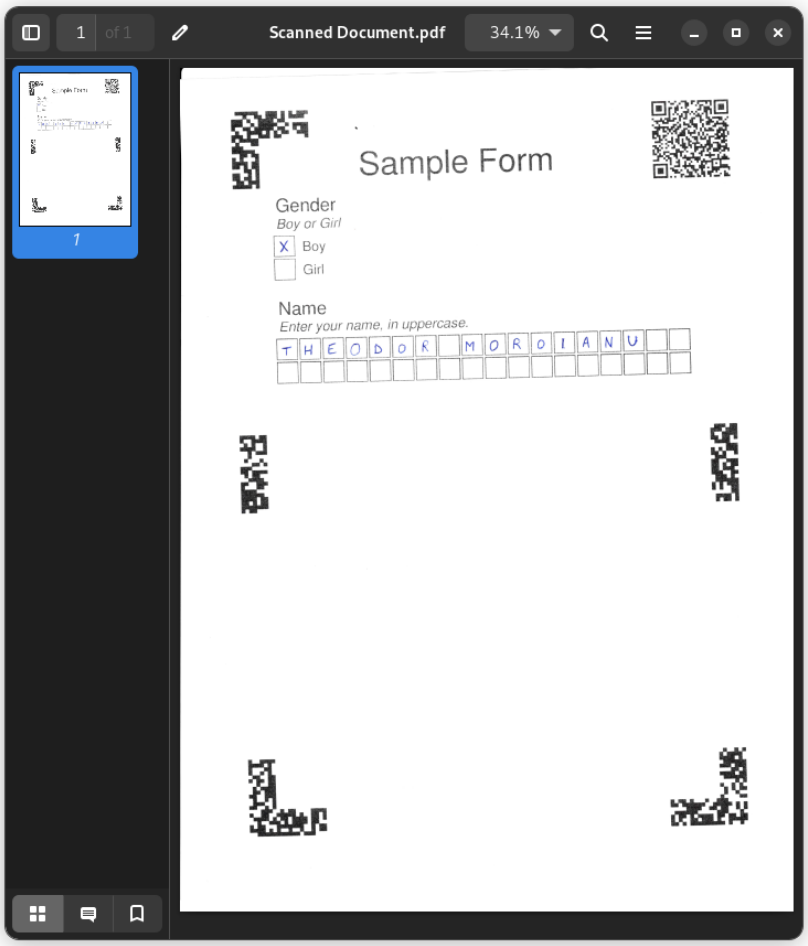


Figure 4.2: Scan of a manually filled form

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 ID of the form present in the scan.
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 user.

It should be noted that the process is not entirely flawless. This aspect is discussed in detail in Chapter 7 dealing with Form Parsing.

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
}
```

In a similar fashion, the API allows us to edit or 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 established 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 in 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 created multiple databases:

- A database used for production.
- Another one used for the *CI/CD* pipelines.

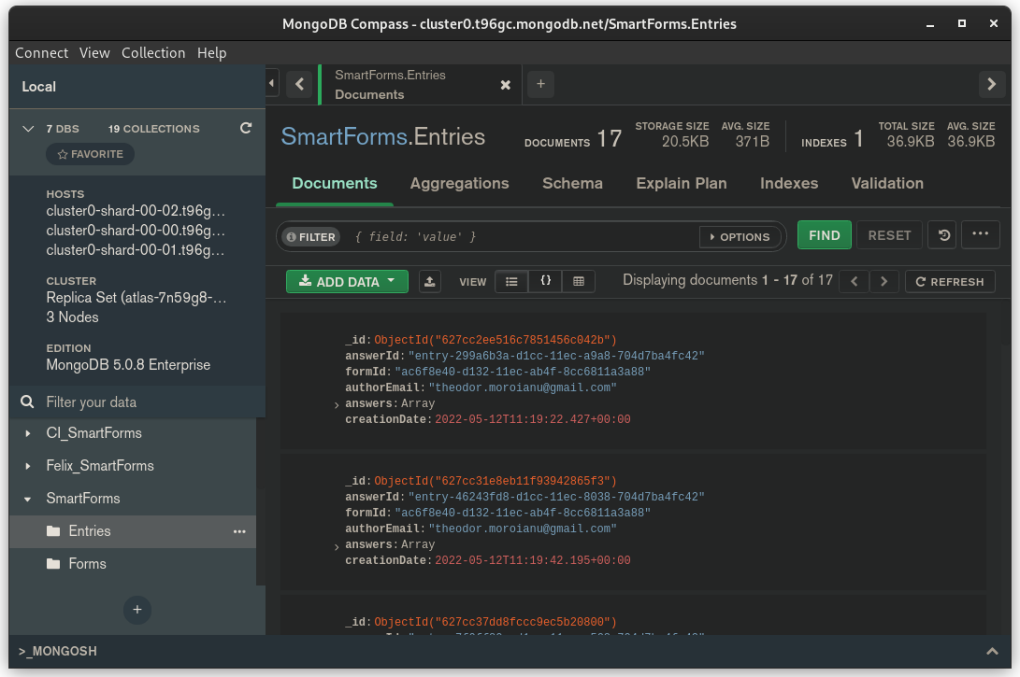


Figure 5.1: Compass – the official MongoDB viewer

- 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 records of deleted accounts.
- 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 individual characters extracted from pictures and scans.
- A dataset of labeled characters which can be used for improving the OCR neural network.

The last two collections – 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.

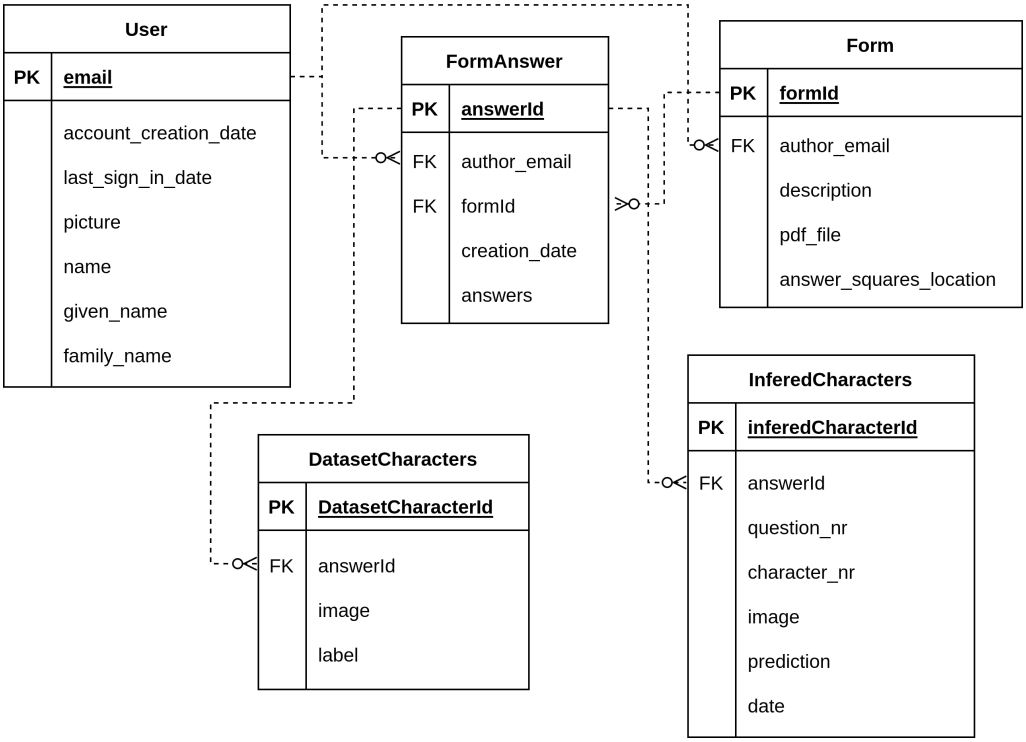


Figure 5.2: Diagram of the MongoDB Database

Chapter 6

PDF Forms Creation

6.1 Objective and Constraints

Forms are not internally stored as *PDF* documents. Users do not actually need to use *PDF* documents at all, as one can create, share, and fill forms from the application itself. However, being able to have a printed copy of the form can be useful.

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 constraints 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 internally is not feasible.
- Even though most printers advertise the ability to print edge-to-edge, most modern printers are not able to print anything too close to the edges of the paper.
- As most office printers only print in black and white for efficiency reasons, we should not 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 vary in length from a simple question to multiple pages.

6.2 Layout of a PDF Form

To conform to the constraints mentioned above, we embed multiple components in the documents.

6.2.1 Questions

Questions are the most obvious element of the forms. They consist of a question title (or the question *per se*), an optional question subtitle (or explanation), and input zones depending on the type of question.

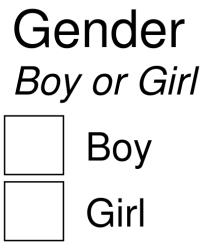


Figure 6.1: Sample multiple-choice question

If the question is multiple choice as in Figure 6.1, where the user has to select a subset of the given possibilities (e.g. "Boy" or "Girl"), each option is printed on a different line, with a square the user can tick to select the answer.

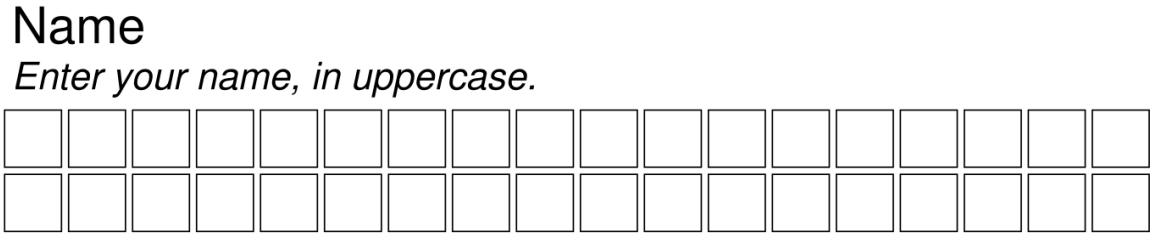


Figure 6.2: Sample text question

If the question is a text question as in Figure 6.2, 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 a 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 intuitive and easy to fill.

Note that the exact location of each square is saved in the database, to allow for 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.

Figure 6.3: Top left marker

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.

6.2.3 QR Code

QR codes (shorthand for Quick Response codes) are visual machine barcodes. QR codes are machine-readable, use a powerful correction algorithm [1] which makes them readable even when damaged, and can store arbitrary content, from Wi-Fi details to binary data to simple text.

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

We use QR codes for several reasons:

- Due to their apparent randomness, they help the matching process similarly to lateral markers.
- 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. Similarly, 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*" watermark.

6.2.5 Multi-page Support

If the form contains 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.

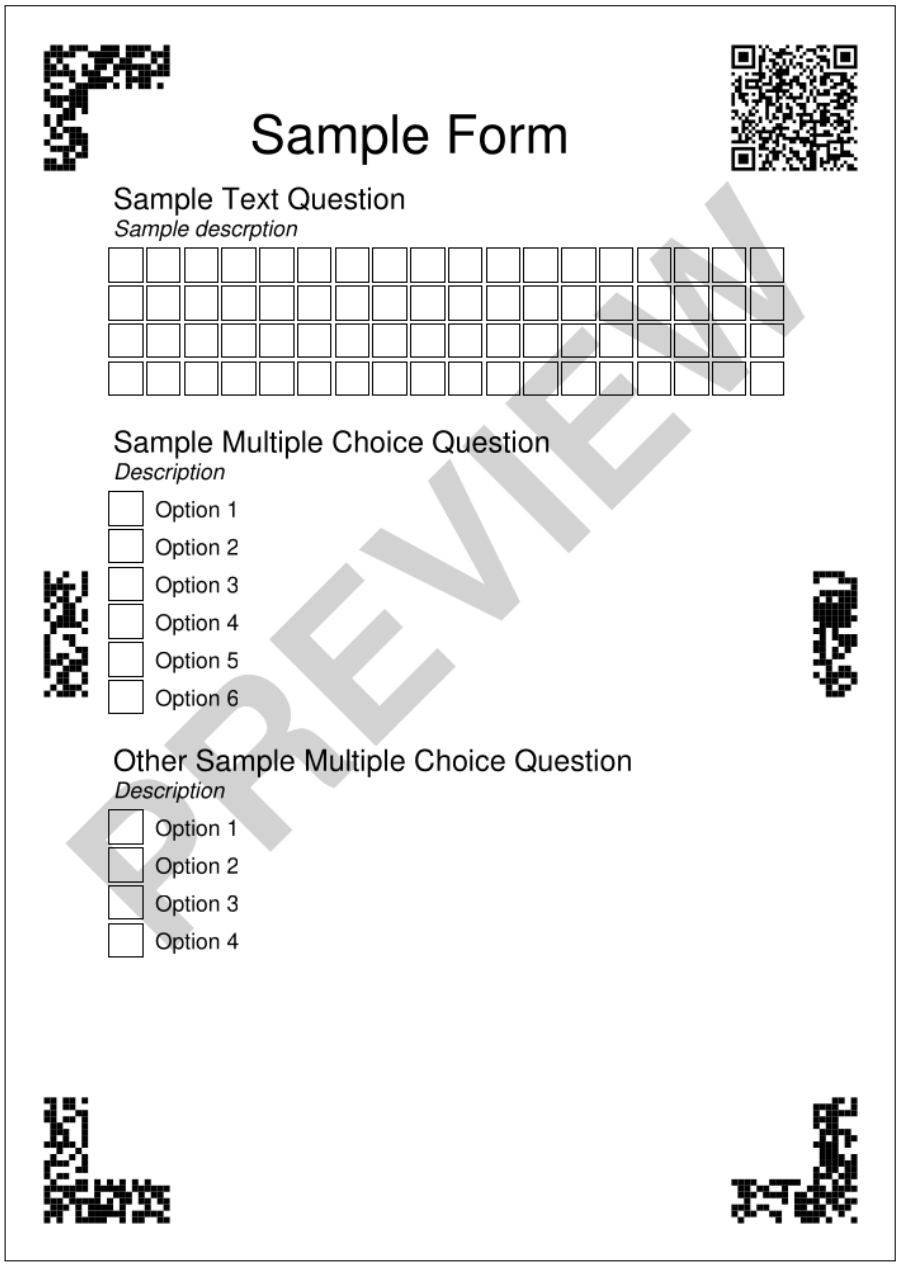


Figure 6.5: Form with "*PREVIEW*" watermark

Chapter 7

Form Parsing

When users have a batch of handfilled documents to digitize, they just have to upload the files to *SmartForms API* and let it parse the documents automatically.

7.1 How Data is Loaded

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

To facilitate the uploading of filled forms, we accept:

- Images in any standard image format (`.jpeg`, `.png`, `.webp` etc).
- *PDF* documents.
- *ZIP* archives, 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. We illustrate below the action of *Zbar* on a scanned form:

```
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
```

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 actual ID, we simply have to remove the "*https://smartforms.ml/view-form/*" prefix. *Pyzbar* operates in a similar way as its command-line counterpart, 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

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]:

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

However, even for grayscale images we cannot simply apply a standard binary threshold, due to possible differences in brightness. Figure 7.1 is an optical ilusion, where squares marked with **A** and **B** seem different (one is black and the other white), but their pixels have the same color, which makes it impossible to separate them using a binary threshold. Generally, separating dark and light regions is not a trivial task, if we want to take into account lighting and brightness. Similarly, applying a binary threshold considering pixel intensities larger than a fixed K black and intensities smaller than K white in our scanned forms will lead to misclassified pixels, where darker empty regions of the form might be classified as black, and lighter text or graphics of the form might get classified as white.

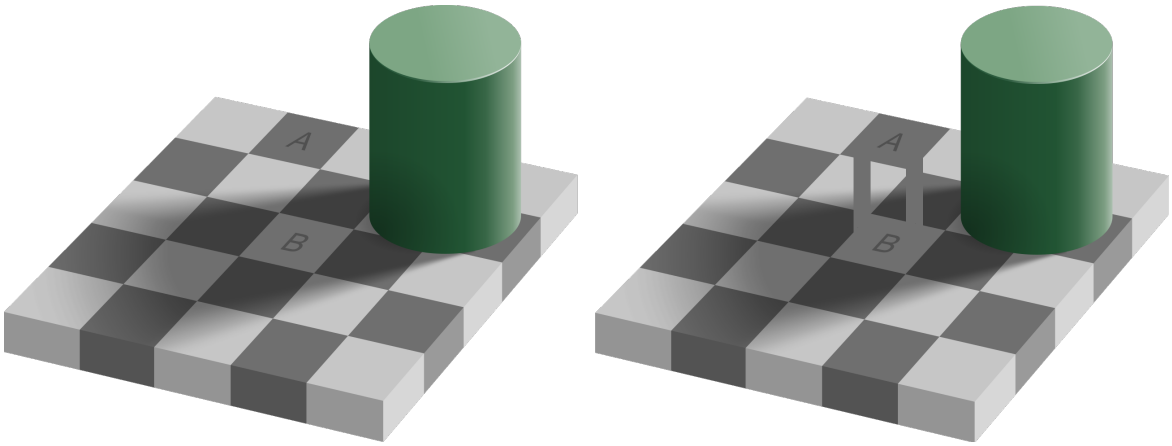
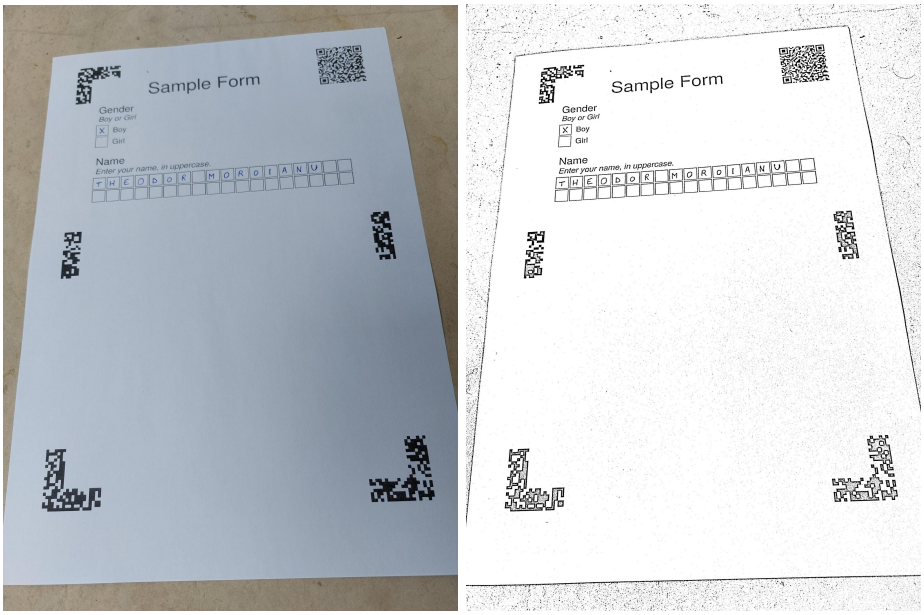


Figure 7.1: Checker Shadow Illusion [5]

Figure 7.2 (a) shows what an unprocessed picture or scan of our form might look like. Our first step is converting it into grayscale. To further convert the resulting grayscale image to binary, we use an adaptive thresholding technique [4], which for each pixel 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 7.2 (b).



(a) Original (b) Processed

Figure 7.2: Picture converted to binary

7.4 Changing Image Perspective

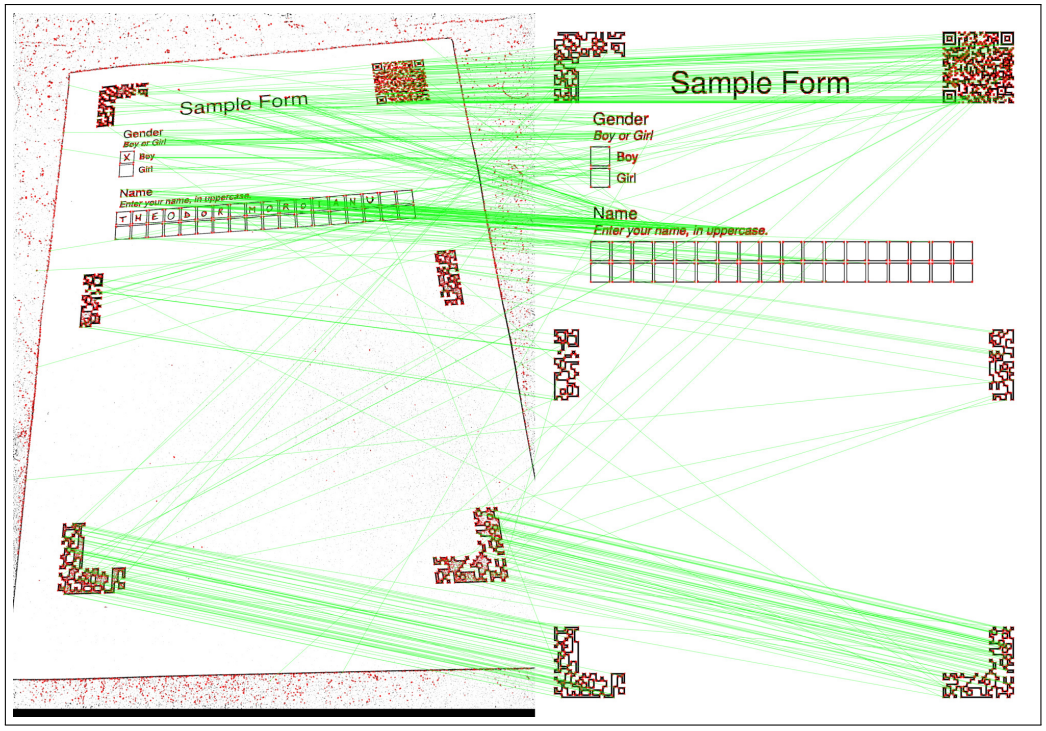


Figure 7.3: Matches found with ORB

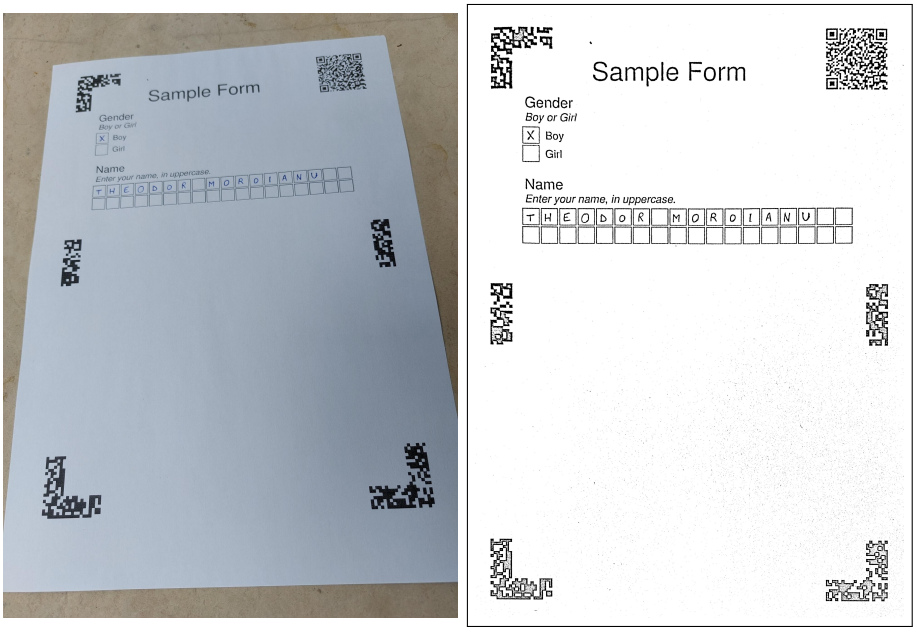
For parsing the form, we have to be able to match specific pixels of the picture / scan of the

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 several issues arise:

- 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 $1:1$, then a scaling factor is needed in order to be able to match the picture and the template.
- If the picture only preserves proportions then, in addition to the scaling factor, rotation and translation transformations (Euclidean isometries) are also required.
- If the picture does not keep the same proportions, then we have to find a homography between the two.

As finding homographies between two images is a common task in Computer Vision, *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.



(a) Original (b) Corrected

Figure 7.4: Original and 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 phenomenon does not appear for scanned documents.

7.5 Extracting Answer Squares

Extracting squares containing an answer from a form is straightforward once we have a corrected image. As we save the position of squares where users write 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
            - page: 0
        ...
```

Figure 7.5: Marked Answer Squares

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 red in Figure 7.5 the extracted answer squares.

Figure 7.6 is a flowchart of the entire parsing pipeline. As one can see, each step is independent and self-contained, which adds stability to the software and makes the source code easier to follow.

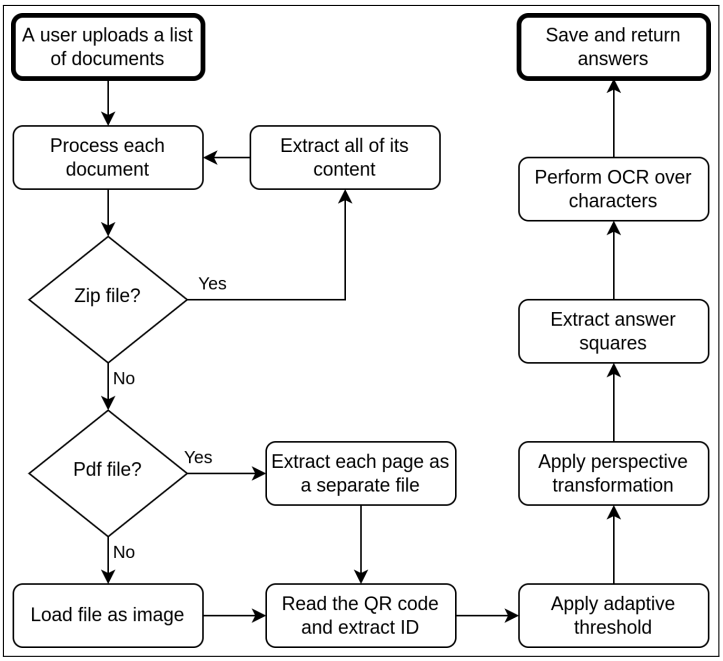


Figure 7.6: Parsing flowchart

Chapter 8

OCR Neural Network Design and Training

8.1 Why Use a CNN

We use a neural network in order to be able to extract from a small square, as the ones marked in Figure 7.5, the character written inside.

While traditional methods (such as the HOG [7] approach) perform reasonably well, we decided to design, train and use a convolutional neural network (CNN) [8], as it often outperforms classical 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 identify characters.

8.2 CNN Architecture

We played with various common architectures, both pretrained and untrained, such as the *VGG* [11] or the *Resnet* [10] architectures. However, such architectures are way to complex for our relativly simple task of classifying around 100 characters. We therefore decided to build and train our own convolutional netural network, using the *Pytorch* framework.

The layers definition of our network is:

```
# backend/sources/ocr/network.py
model = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=(5, 5), padding=(2, 2)),
    nn.ReLU(),

    nn.Conv2d(16, 32, kernel_size=(3, 3), padding=(1, 1)),
    nn.ReLU(),

    nn.BatchNorm2d(32),
    nn.MaxPool2d((2, 2)),

    nn.Conv2d(32, 64, kernel_size=(3, 3), padding=(1, 1)),
    nn.ReLU(),

    nn.Conv2d(64, 64, kernel_size=(3, 3), padding=(1, 1)),
    nn.ReLU(),
```

```
    nn.Conv2d(64, 64, kernel_size=(3, 3), padding=(1, 1)),
    nn.ReLU(),

    nn.BatchNorm2d(64),
    nn.MaxPool2d((2, 2)),

    nn.Flatten(),
    nn.Dropout(0.2),

    nn.Linear((IMAGE_SIZE // 4)**2 * 64, 1024),
    nn.ReLU(),

    nn.Linear(1024, len(CHARACTERS))
)
```

Our network consists of:

- Six layers of two-dimensional convolutions, with square kernels of sizes 3 and 5, over 16, 32 and 64 channels.
- A *ReLU* [12] activation layer after each convolution.
- Batch normalization and dimensionality reduction (max pools) layers.
- Two linear layers, acting as the classificator part of the model.

As we will show in the *Training* section, this architecture is more than enough to accuratly perform our classification task. Its small size also allows it to load and run fast. The total size of the model is under 14Mb, which, by today's standards, is really light. This allows our model to run smoothly on the *CPU* as well as the *GPU*, which in turn allows *SmartForms* to run on low-end devices and reduces the power usage.

8.3 Dataset Used

An easy solution to get a dataset would have been to simply create one. This has however major drawbacks:

- We have to create over 62 classes ($26 * 2$ lowercase and uppercase letters, and 10 digits, not counting special characters such as '`@`', '`!`', '`+`', spaces etc.). Making from scratch a dataset large enough to correctly train a network would be extremly time-consumming.
- The network is meant to recognize any handwritting. Training it on a dataset made by only a handful of people means that characters won't have a large enough diversity to correctly generalize (different people write characters such as '`1`', '`7`', '`a`', '`e`' etc. differently).
- Similarly, the dataset would be dependent on the type of pen and ink used (which can have different thikness and darkness), the scanning device / phone camera model (due to post-processing all modern capturing devices do), etc. Making a robust dataset would mean spending a lot of time, effort and resources to vary as many different parameters as possible.

To fix this issue, the obvious solution is to use an already made dataset. We chose the *EMNIST* [13] dataset, which contains over 600.000 handwritten digits, lowercase and uppercase letters. Sadly, we were unable to find a dataset containing non-alphanumeric characters.

To account for whitespaces, we enhanced the EMNIST dataset with 10.000 peppered (with random black noise) blank images. This is necessary as the network needs to be able to detect if a certain square contains or not a character.

For optimization reasons, we pre-processed the entire dataset, resizing images to the size our network expects as input, and adding the whitespaces. A small snippet of code showing an overview of the dataset pre-processing is the following:

```
# backend/sources/ocr/generate_dataset.py

images: np.ndarray, labels: np.ndarray = [], []

# importing and loading the entire dataset
# we do NOT use tensorflow, tensorflow_datasets
# is a separate package for managing datasets
import tensorflow_datasets as t_d
ds = t_d.load('emnist', shuffle_files=True)

# resizing images from the dataset and adding them
# to the `images` and `labels` lists
# [...]

# generating a list of 10000 blank images
# and adding them to the `images` and `labels` list
whitespaces = []
for i in range(10000):
    whitespaces.append((np.zeros((28, 28), dtype=np.uint8), 62))
# [...]

# saving the entire dataset as a numpy binary file
# at the location `DATASET_PATH`
os.makedirs(DATASET_PATH)

imgs = np.stack(imgs)
labels = np.stack(labels)

np.save(IMAGES_PATH, imgs)
np.save(LABELS_PATH, labels)
```

This preprocessing helps with the loading time by greatly reducing the size of the dataset. While adding the whitespace images might seem redundant at this stage, it greatly simplifies the future processing of the dataset, and barely affects its size, increasing it by a mere $\sim 1.6\%$.

8.4 Data Processing and Augmentation

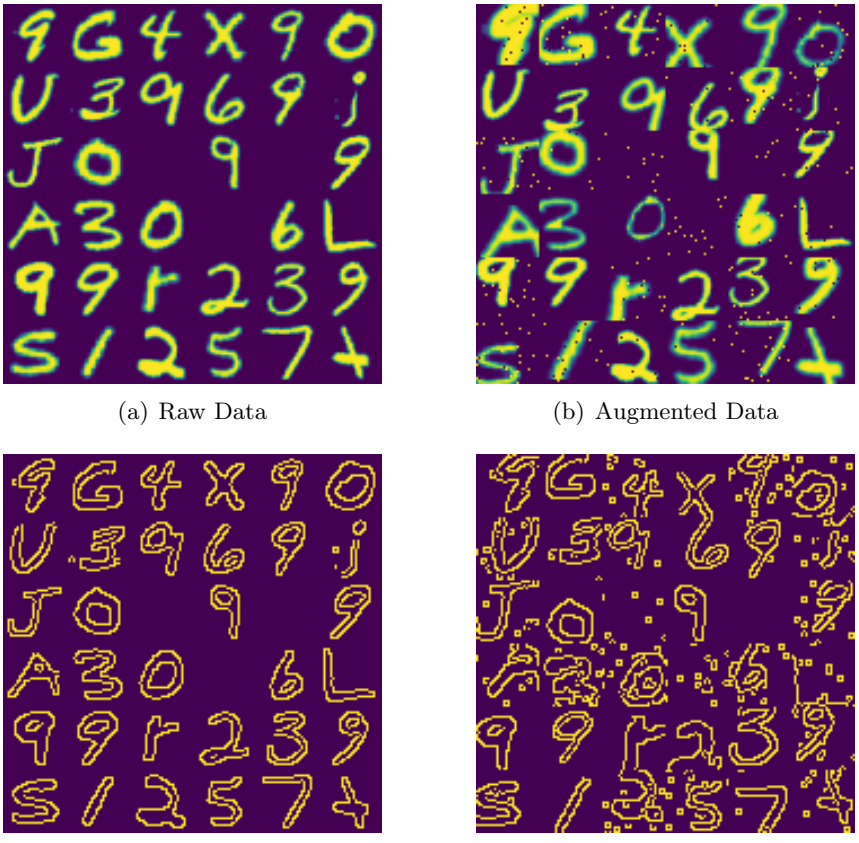
The main issue we had to fight against is that images from the EMNIST dataset do not look completly similar to the ones we need to process. This means the data we train on is not the same as the data we want our model to predict.

To fix this issue, we heavily rely on data augmentation and post-processing. The augmentation techniques we use are:

- Image erosion and dilation (increasing / reducing the white zone).

- Image resizing – randomly cropping borders of the images or padding it, essentially zooming into / out of the image.
- Random translations. This is especially usefull as most EMNIST data is perfectly centered in the image, but ours is not.
- Random rotations by a relatively small angle.
- Salt and pepper random noise.

For post-processing, we apply the Canny edge detector, which returns a smoothed contour of the image. After applying data augmentation on the EMNIST dataset and post-processing on both EMNIST and our images we found no visual difference between the two. We built a small handmade dataset, and confirmed that the accuracy of our model on this dataset is similar to the EMNIST data.



(a) Raw Data

(b) Augmented Data

(c) Processed Raw Data

(d) Processed Augmented Data

Figure 8.1: Data Augmentation of the EMNIST dataset

In Figure 8.1 we can see the different stages of pre-processing of our dataset. We train the network on the processed augmented data (d), but validate against and expect real world data to be similar to (c). Blank characters represent whitespaces (' ').

8.5 Training

We trained the network using the standard *Pytorch* method (defining a loss function, an optimizer, and looping throughout mini-batches to train the network). A notable observation is

that we had to tweak our loss function (i.e., the cross-entropy loss function) to take into account the weight of each class, as our data is not balanced.

The classification task we want our model to perform is ambiguous (for e.g., 'o', 'O' and '0' are written similarly). To counter this issue, we define a *per-class accuracy*, where for a given character we only consider characters in the same class (digits, lowercase or uppercase) – note that whitespaces are considered to belong to all 3 classes. In other words, a given character is corectly determined in our *per-class accuracy* metric iff the corect label has the highest score among all characters of the same class.

This metric is the one most relevant to the problem at hand, as questions we have to parse contain a list of allowed characters set by users (for instance, a phone number field will only allow digits). The network obtains an error rate of less than 1.5%, with which we are confortable, as the misclassified characters are for the most part poorly written characters. Figure 8.2 shows the evolution of loss and accuracy over the course of training.

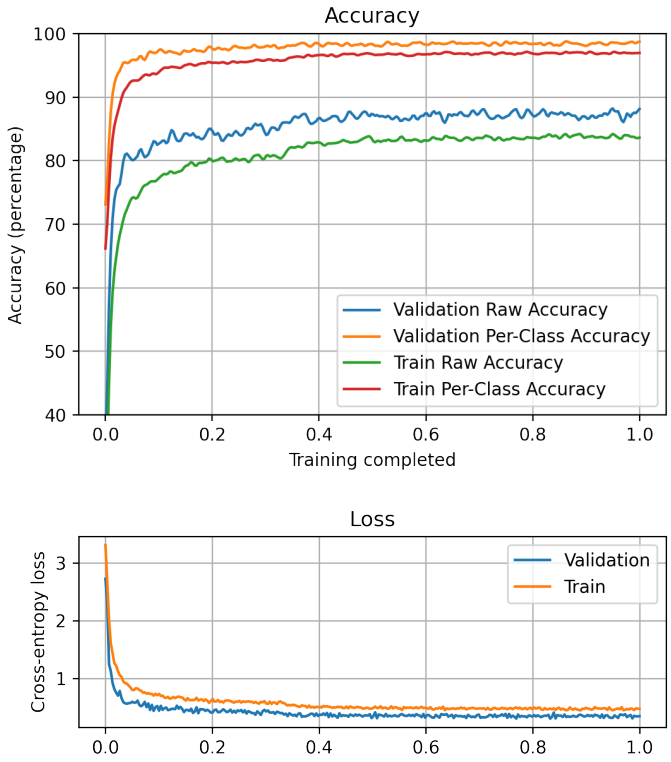


Figure 8.2: Training results of the CNN network

8.6 Self-Training

As mentioned above, the *EMNIST* dataset we are using only contains alphanumeric characters. We manually added whitespaces, but special characters such as '!', '?', '+', '-', '@' etc. are still not present in the dataset.
To mitigate this issue, we have the following approach:

1. When a form is uploaded in order to be parsed, we save an image of the characters we extracted alongside the answer.

2. If a user edits a parsed form, we assume the user corected a mistake made by the OCR network, and, for each modified character, save it in a separate database as a labled image.

An example would be the following situation:

(a) A user sends to *SmartForms API* a form to parse.

(b) The OCR network detects a character as an '`O`'.

(c) The user later updates the answer, changing, among others, the '`O`' into a '`Q`'.

(d) *SmartForms API* then knows that the image it originally labeled as a '`O`' is in fact a '`Q`', and adds the labeled image to the database.

3. Maintainers of the application can train the network again, taking into account the newly created dataset.

This approach is ideal, as SmartForms generates a dataset from the same universe as the characters it has to parse, and over time less and less mistakes are made due to the larger and larger dataset.

Chapter 9

HTTP/S Server and Secure Authentication

9.1 FastAPI Session Middleware

Our entire API backend service is written using FastAPI, *a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints* [15]. FastAPI helps us to validate data received, and automatically generates the OpenAPI [16] specification of our service.

In order to authenticate users we use a session middleware. This allows us to use a stateless authentication mechanism, where all the authentication data is stored within an encrypted cookie on the user's browser.

An API endpoint looks similar to this one, which shows a short message to the users and allows them to login and logout of their account:

```
@app.get('/')
async def home(request: Request):
    """
    Shows the user its authentication status, and allows him to
    check the documentation, login or logout.
    """
    # retrieve the user's details from the session
    # if the user is not logged in, then the `user` object is None
    user = request.session.get('user')
    email = user['email'] if user is not None else 'Not signed in'

    html = (
        f"<pre>Email: {email}</pre><br>" +
        "<a href='/api/docs'>documentation</a><br>" +
        ("<a href='/api/user/logout'>logout</a>"
            if user is not None else
            "<a href='/api/user/login/'>login</a>")
    )
    return HTMLResponse(html)
```

By default, *FastAPI* only handles requests over *HTTP*, but one can use a web-server such as *Nginx* [19] to act as a HTTP endpoint and proxy for our service.

The setup currently running on the https://smartforms.ml website, made as part of a different project [23], is the following:

- *SmartForms API* is running on `0.0.0.0:5000`.
- *Nginx* is binded to `0.0.0.0:443` and `0.0.0.0:80`, and proxies any requests made to the `/api/` route to `localhost:5000`, which is the API server.

Please note that both services are binded to the meta-address `0.0.0.0`, which makes them accessible over all networking interfaces. This means in particular that the API server is exposed to the interned by accessing the `http://smartforms.ml:5000` URL (please note the connection is `HTTP` and not `HTTPS`). The unsecure connection might seem problematic, but, as it can only be used by users forging requests, this is not a security issue, and can be stopped by binding the API service to `localhost` instead. Figure 9.1 shows a diagram of the entire service.

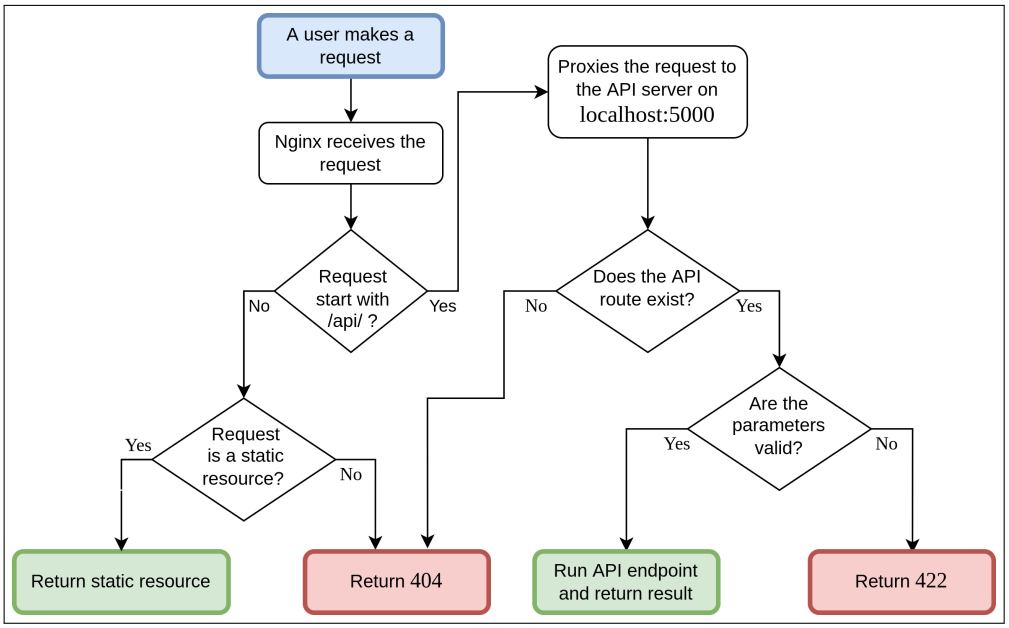


Figure 9.1: `https://smartforms.ml/` service diagram

9.2 Oauth2 Authentication Framework

As SmartForms is a small application, requiring users to create an account with a username and password is tidious. We decided to use the *OAuth v2.0* [18] (**O**pen **Auth**entication) protocol, which allows users to login using a third party account, such as a Facebook, Google or Microsoft account.

Due to the difficulty to register SmartForms as an OAuth service on most platforms (we need to show a proof we own a domain, ask for human verification, add a credit card etc) we have only registered our application on Google Cloud.

This means that in its current state SmartForms only allows authentication using a Google account, but allowing additional authentication services only requires a few lines of code besides registering SmartForms on said services.

Chapter 10

Tests, Source Control and CI/CD

10.1 Testing

To ensure the stability of the application and speed up the development process we decided to use the *TDD* (**T**est **D**riven **D**evelopment) approach: each API endpoint and all features of SmartForms are thoroughly tested by unit tests.

We also have integration tests, running the application end-to-end (generating forms, creating, modifying and uploading answers).

The only components of the application which are not tested by unit tests are:

- The CNN training module, which is technically part of the software, but is only runned manually or if the network model is not found, which only happens due to human intervention.
- The `user` router, which can hardly be tested as it performs requests to Google Cloud. Testing the endpoints would require creating a test account on Google, extracting its authentication cookies, and manually forging a request to authenticate it on SmartForms, which is almost impossible and strongly forbidden by the Google terms & conditions.

The total coverage of our unit tests is of 85%, which might seem low, but most of the untested lines of code live either in the two modules mentioned above, or hard to reach sections which were manually tested during development.

10.2 Git

We are using Git[14] as a source control software, and use the Github platform as a remote. The source code can be viewed online at `https://github.com/TeamUnibuc/SmartForms/tree/main/backend`, where the complete history of the project (commits, pull requests and issues) is public. The repository contains *SmartForms API* (our project), as well as the *SmartForms* client [23], due to their tight connection.

At the time of writting, the repository (backend and frontend combined) has over 300 PRs and commits on the main branch, 35 active branches and a total of 36.000 added and 14.000 deleted lines of code thoughout its history.

10.3 CI/CD

The *SmartForms* repository is set-up using **C**ontiguous **I**ntegration and **C**ontiguous **D**eployment pipelines. They are used in the following scenario:

1. A developer wants to add a new feature to the project. He branches off from the master branch and implements the feature on the newly created branch.

2. Once the feature is finished, the developer makes a *pull request* – also called *merge request*, asking for his/her squashed commits to be applied to the master branch (instead of applying the squashed commits, a rebase or a simple merge are also possible, but less common).

3. The **CI** pipeline kicks in, automatically running a series of checks and tests.

4. If the tests succeed, and the branch is merged, the **CD** pipeline starts, pushing the entire code from the master branch to production.

The contiguous deployment pipeline is written and mentained by ????[23], but the integration one is written by us:

```yaml
# .github/workflows/backend.yml

name: Run Backend Unittests

on:
    push:
        branches: [ main ]
    pull_request:
        branches: [ main ]
        types: [ opened, synchronize ]
jobs:
    build:
        # avoids running AFTER a merged PR
        if: "!contains(github.event.head_commit.message,
                                        'Merge pull request #')"
        runs-on: ubuntu-latest

        steps:
            - uses: actions/checkout@v2
            - name: Set up Python 3.10
              uses: actions/setup-python@v2
              with:
                  python-version: "3.10"
            - name: Install dependencies
              run: |
                sudo apt-get install -y libzbar0 poppler-utils
                python -m pip install --upgrade pip
                pip install flake8 pytest
                pip install -r backend/requirements.txt
            - name: Lint with flake8
                ...
            - name: Test with unittest
              env:
                MONGO_USER: ${{ secrets.MONGO_USER }}
                MONGO_PASSWORD: ${{ secrets.MONGO_PASSWORD }}
                ...
              run: |
                (cd backend/sources/tests/ && python -m unittest)
```

The code shown above does the following:

1. Checks if the code being tested is part of a PR, and stops if not.
2. Installs required packages on the machine – both *Debian* packages with `apt`, and *Python* packages with `pip`.
3. Syntactically checks the code with `flake8`, a *Python* linter.
4. Runs the backend unit and integration tests.

Chapter 11

Conclusion and Future Work

11.1 Possible Enhancements

The framework presented in this project is a MVP of a modern online forms and surveys generation and parsing tool. There are many directions in which the project can be extended.

11.1.1 User Roles

The *SmartForms* API does not currently have any support for user roles, such as moderators or administrators. This is purely by design, as we did not consider it essential for our product: as we use MongoDB, any modifications to users, forms or answers can easily be made directly from the database.

Adding user roles would however make the application more attractive to large corporations, where each employee has a specific set of attributions and authorizations.

11.1.2 Stateless PDF Forms

An important improvement we could have made, but decided against due to the centralized design of our application, is to save any form-related information directly on QR codes.

Currently, QR codes store the ID of the form they are printed onto, which is then used to retrieve from the database details about the form, in order to parse it. We can shortcut and completly eliminate the need for a database (in the matching process) by storing on the QR codes all the details of the form.

While the amount of information to store might seem excessive, people already managed to fit an entire GUI game on single QR code[20], which offers up to a few kilobytes of storage, depending on the error-correction level.

11.1.3 Multithreading

An important improvement speed-wise would be introducing multi-threading to our parsing pipeline. While the OCR model is hard to be run in parallel as it might be running on the GPU, preprocessing such as QR code extraction, adaptive thresholding, feature extraction & matching and even dataloaders for the CNN could be ran in parallel, to fully utilize more than one CPU core.

11.1.4 Offline Login System

SmartForms relies on online third party authentication. While authentication checks can be disabled altogether by editing the '`.env`' file, being able to manage the access to forms and answers could be useful, even in offline scenarios.

11.1.5 Additional Input Formats

Another addition to *SmartForms* could be the introduction of additional input formats, such as a large square for a signature, or a more advanced question format, mimicking questionaires of the form "*When I go to school I have to walk ..... km and it takes ..... minutes*", in which text and answer boxes are mixed.

11.2 Conclusions

SmartForms was an idea that I came up with while helping a front desk worker transcribe a form I had just filled into his computer, one of the situations *SmartFroms* aims to solve. The API, form generation, and form parsing modules went through multiple iterations of approaches and technologies used before ending up in the current state, which we hope is the perfect balance of simplicity and usability, as *SmartForms API* aims to offer a simple yet effective interface for managing questionaires and surverys.

The hardest part of the project has been by far the form parsing pipeline, due to many reasons, such as:

- As data is comming from users, we cannot assume that the file format is recognized, files are not corrupted, images contain a valid form, the form exists etc.
- The process uses real-world images that cannot be accuratly created digitally, which translated into a lot of printed forms, scans with various settings, pictures under different lighting conditions and angles and so forth.
- We had to make the existing dataset used for training the OCR CNN mimic our data. To check the real accuracy of the network, we had to also create a small verification dataset from scratch.

We have not talked too much about the client of *SmartForms*, as it is not part of this project, but the two components (*SmartForms API* and the *SmartForms* client) work perfectly together, making the inner workings of the API completly transparent to the user.

While the framework can still be improved, we believe it is already fit for being used by people or institutions for which *SmartForms* presents attractive features.

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*, n.d., 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 preprint arXiv:1511.08458, 2015.

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

[10] K. He, X. Zhang, S. Ren, J. Sun, 2016, *Deep residual learning for image recognition*, 2016 IEEE conference on computer vision and pattern recognition, IEEE (2016), 770–778.

[11] K. Simonyan, A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, preprint arXiv:1409.1556, 2014.

[12] A.F. Agarap, *Deep learning using rectified linear units (relu)*, preprint arXiv:1803.08375, 2018.

[13] G. Cohen, S. Afshar, J. Tapson, A. Schaik, *EMNIST: an extension of MNIST to handwritten letters*, preprint arXiv:1702.05373, 2017.

[14] *Git*, n.d., accessed 30 May 2022, https://git-scm.com/.

[15] *FastAPI*, n.d., accessed 31 May 2022, https://fastapi.tiangolo.com/.

[16] *OpenAPI Specification*, 20 February 2020, accessed 31 May 2022, https://swagger.io/specification/.

[17] J.S. Murdoch, *Hardened stateless session cookies*, International Workshop on Security Protocols, Springer (2008), 93–101.

[18] D. Hardt, *RFC 6749: The OAuth 2.0 authorization framework*, RFC Editor, 2012.

[19] *Nginx*, n.d., accessed 1 June 2022, https://nginx.org/en/.

[20] MattKC, *Can you fit a whole game into a QR code?*, accessed 2 June 2022, https://www.youtube.com/watch?v=ExwqNreocpg, 2020.

[21] *Lime Survey*, n.d., accessed 5 June 2022, https://github.com/LimeSurvey/LimeSurvey.

[22] *Convert PDF Forms to Excel, CSV or Webhooks*, n.d., accessed 5 June 2022, https://docparser.com/solutions/form-pdf-to-text/.

[23] F. Puscasu, *Design and implementation of an application for generating and parsing forms with modern methodologies of software development*, University of Bucharest, 2022.

Appendix A

Manual Deployment of SmartForms

SmartForms API is currently deployed at http://smartforms.ml:5000, and the companion *SmartForms* client is deployed at https://smartforms.ml:443 – or https://smartforms.ml.

It is however possible to host both the client and the API server locally. The steps to do so are:

1. Download the *SmartForms* repository.
2. Install dependencies.
3. Set-up the environmnent file.
4. Start the services.

Step 1: Download the repository

There are two ways to download the repository:

- Clone the repository with *Git*: `git clone https://github.com/TeamUnibuc/SmartForms`.
- Manually download and deflate a *ZIP* archive of the repository by visiting https://github.com/TeamUnibuc/SmartForms/archive/refs/heads/main.zip.

Step 2: Install dependencies

The dependencies unusally not available out-of-the-box on most systems of *SmartForms* are:

- A *Conda* distribution, such as *Anaconda* or *Miniconda*.
- The *Zbar* library, downloadable from http://zbar.sourceforge.net/, and installable as `libzbar0` on most UNIX systems.

Note that you may have additional unresolved dependencies, which will need to be manually installed.

Step 3: Set-up the environment

To set-up the *SmartForms API* environment, you need to create the file `backend/.env`, which can be done by renaming the `backend/.env.sample` file.

The required fields of the `.env` file are:

- `MONGO_USER`, `MONGO_PASSWORD`, `MONGO_CLUSTER`, and `MONGO_DB_NAME` which are the username and password of the *MongoDB* user, URI of the database and the database name. The database itself can be either installed locally by following the steps described at https://www.mongodb.com/docs/manual/installation/ or on the cloud (https://www.mongodb.com/cloud).

- `GOOGLE_CLIENT_ID` and `GOOGLE_CLIENT_SECRET`, which are provided by Google when registering an *OAuth* application (https://console.cloud.google.com/apis/credentials/).
- `COOKIES_SECRET`, a secret passphrase to encrypt and decrypt session cookies (which are stored on the client-side due to our stateless design), allowing among others to not sign out users when restarting the server.
- `FRONTEND_URL`, the URL the client is available at (for e.g. https://smartforms.ml). If you are not planning to use a client, please use the address of the backend instead. This variable is used to redirect the user though login.
- `FORM_ID_PREFIX`, the prefix added to form IDs when generating QR codes, which can be usefull when users scan the QR code.

Step 4: Start the services

To build and start *SmartForms API*, you have to:

1. Install the *SmartForms* conda environment.
2. Enable the environment.
3. Build and start the API.

The steps can be done on a UNIX system with the following commands:

```
$ conda env create -f conda_environment.yaml
$ conda activate SmartForms
$ ./deploy_script.sh
```

Note that only the API server is started. The client service is built into static files, which need to be served with a server such as *Nginx*.