**Пофайловое описание структуры проекта.**

## ****main.py****

**Главный файл проекта.**  
Это центральная точка входа в систему, которая объединяет в себе все ключевые компоненты:

* взаимодействие с RabbitMQ (очередями),
* вызовы OCR пайплайна (распознавание паспортов),
* выдачу результатов в ответ,
* возможность работы **в двух режимах**:
  1. **API сервис (FastAPI)** — принимает HTTP-запросы с изображением паспорта (в base64), кладёт задачу в очередь RabbitMQ и ждёт ответа от воркера.
  2. **Очередь-воркер** — умеет работать как сервис обработки, т.е. сам подписывается на очередь задач, достаёт изображение и обрабатывает его, а затем кладёт результат в ответную очередь.

### Основные задачи:

1. **Приём запросов** (API):
   * Клиент (например, Postman) отправляет JSON:

{"image\_b64": "<base64\_данные>", "meta": {"client":"demo"}}

main.py кладёт задачу в очередь passport.tasks.

Ждёт ответа максимум 30 секунд.

Если ответ пришёл → возвращает JSON с результатами OCR.

Если нет → возвращает status=timeout.

**Работа в режиме воркера:**

* Подписывается на очередь задач (passport.tasks).
* Получает картинку.
* Запускает пайплайн распознавания: детекция паспорта, определение поворота, OCR полей, верификация.
* Формирует структурированный JSON с результатами, например:

{

"fields": {

"surname": {"value": "ИВАНОВ", "confidence": 0.99},

"name": {"value": "ИВАН", "confidence": 0.99},

"series": {"value": "4510", "confidence": 0.99},

"number": {"value": "123456", "confidence": 0.99}

}

}

Отправляет результат обратно в reply.<correlation\_id>.

1. **Сохранение результатов в отладочных целях**  
   В директории results/ сохраняет промежуточные и итоговые изображения:
   * повернутые,
   * с нарисованными bbox,
   * с подписями полей.

## ****api\_server.py****

Файл отвечает **чисто за HTTP API** (FastAPI).  
Фактически, часть его логики встроена уже в main.py, но этот модуль можно выделять отдельно, чтобы держать **веб-сервер** отдельно от логики обработки.

### Основные функции:

* Принимает POST-запросы (/v1/process) с base64 изображением.
* Валидирует вход через Pydantic.
* Проверяет API-ключ (X-API-Key).
* Кладёт задачу в RabbitMQ и ждёт ответа.
* Отвечает JSON-структурой с результатом.

Используется, когда хочется запускать **только API без воркера**.

## ****worker\_stub.py****

Тестовый (заглушечный) воркер.  
Нужен для отладки RabbitMQ взаимодействия без запуска полноценного OCR пайплайна.

### Что делает:

* Подписывается на очередь задач (passport.tasks).
* При получении сообщения просто отвечает фиксированным JSON-заглушкой:

{

"image": "passport\_123.jpg",

"fields": {"name": {"text": "АЛЕКСЕЙ", "conf\_ocr": 97.2}}

}

* Поддерживает correlation\_id и reply\_to.

Этот модуль позволил быстро убедиться, что API ↔ RabbitMQ ↔ Worker связка работает, ещё до интеграции OCR.

## ****utils.py****

Вспомогательный модуль с набором универсальных функций.  
Работает с изображениями (OpenCV).

### Основные функции:

* **rotate\_keep\_canvas** — поворот изображения на угол без обрезки.
* **best\_by** — выбор детекции по имени класса с максимальной уверенностью.
* **minrect\_from\_poly\_or\_bbox** — вычисление минимального прямоугольника вокруг объекта.
* **angle\_from\_vertical** — оценка угла наклона текста относительно вертикали.
* **crop\_from\_xyxy** — вырезание ROI из bbox с небольшим расширением.

Используется практически во всех стадиях пайплайна.

## ****detection.py****

Модуль для работы с YOLO-моделью.

### Что делает:

* Загружает модель (YOLOv8 / Ultralytics).
* Запускает model.predict на изображении.
* Возвращает список детекций (bounding boxes + confidence).

Каждая детекция оформлена как словарь:

{

"name": "photo",

"conf": 0.92,

"xyxy": [x1, y1, x2, y2],

"poly": None

}

Это основной источник информации для локализации полей паспорта.

## ****orientation.py****

Модуль, который отвечает за **определение ориентации паспорта** (правильный угол).

### Основные задачи:

1. **Stage 1 (Sweep ±30°):**
   * Пробует разные углы (±1…30°).
   * Считает score сцены по ключевым объектам (passport, mrz, photo).
   * Выбирает лучший угол.
2. **Stage 2 (K90 Layout):**
   * Проверяет 0°, 90°, 180°, 270°.
   * Считает метрики (расположение MRZ внизу, фото слева и т.д.).
   * Выбирает лучший.
3. **Stage 3 (Photo Mode):**
   * Специальная логика, если попалась «фото-страница».
4. **draw\_oriented** — визуализация: рисует прямоугольники и подписи на картинке.

Без этого модуля OCR бы часто работал на перевёрнутых или боковых фото.

## ****ocr.py****

Модуль, отвечающий за **OCR (распознавание текста)**.  
Работает через **EasyOCR**, но с кастомной обученной моделью для паспортов.

### Основные функции:

* **preprocess\_for\_ocr** — готовит несколько вариантов изображения (базовый, инверсия, бинаризация).
* **easyocr\_lang\_and\_params** — подбирает allowlist и параметры OCR под разные поля.
* **ocr\_text\_and\_conf** — прогоняет картинку через OCR на всех вариантах предобработки, выбирает лучший результат.
* **process\_field\_with\_retry** — делает несколько попыток, чтобы повысить уверенность.
* **postprocess\_field** — минимальная очистка и валидация текста.

Это ключевой модуль, который превращает вырезанный ROI в реальный текст.

## ****verifier.py****

Модуль постобработки и коррекции распознанного текста.  
Здесь впервые подключается **LLM (T5 русская модель)** для исправления ошибок OCR.

### Основные задачи:

1. Формирует **промпт** из полей паспорта.
2. Отправляет его в LLM, которая должна выдать JSON с исправленными полями.
3. Чистит ответ модели и парсит JSON.
4. Делает **дополнительную валидацию**:
   * Формат серии и номера.
   * Код подразделения (XXX-XXX).
   * Даты (ДД.ММ.ГГГГ).
   * Пол («МУЖ.» / «ЖЕН.»).
   * MRZ (44 символа).
   * Капитализация ФИО.

Таким образом, verifier превращает «сырые» результаты OCR в аккуратный, стандартизированный JSON.

# Функциональное описание проекта (как что собственно работает)

## 1) Способы запуска

* **CLI-режим (по умолчанию)**: python main.py path/to/passport.jpg
  + Обрабатывает один файл и пишет результат в path/to/passport.ocr.json, плюс отладочные картинки в results/.
* **Сервис-режим** (воркер RabbitMQ): python main.py --service
  + Слушает очередь passport.tasks, принимает задания и отвечает в reply.<correlation\_id> (или в reply\_to, если оно задано).

## 2) Клиент → HTTP (если используем API)

1. Клиент отправляет POST на /v1/process с телом:

{ "image\_b64": "<BASE64>", "meta": { "client": "demo" } }

1. **api\_server.py** валидирует base64, формирует задачу и кладёт её в очередь passport.tasks:
   * Добавляет correlation\_id (UUID) и создаёт эксклюзивную ответную очередь reply.<correlation\_id>.
   * Публикует сообщение с reply\_to=reply.<correlation\_id> и этим же correlation\_id.
2. API ждёт до RPC\_TIMEOUT\_SEC (по умолчанию 30 сек) первый ответ с совпадающим correlation\_id.
   * Пришёл ответ → отдаёт клиенту JSON.
   * Не пришёл → возвращает {status:"timeout"}.

Если API не используем, а тестируем локально — сразу идём к шагу 4 (CLI).

## 3) Воркер (main.py --service) получает задачу из RabbitMQ

1. В service\_run() устанавливается соединение с RabbitMQ:
   * объявляет входную очередь passport.tasks,
   * подготавливает потребителя (consume).
2. На каждое сообщение вызывается handle\_message():
   * извлекает correlation\_id и reply\_to,
   * читает payload (JSON),
   * достаёт изображение:
     + либо из image\_b64 (декод + cv2.imdecode),
     + либо из image\_path (читает с диска).

Если изображение не удалось прочитать → формирует ответ {status:"error", error:"bad\_or\_missing\_image"} и публикует его обратно (см. шаг 9).

## 4) Параметры пайплайна

Перед запуском распознавания применяются параметры из payload["params"] (если есть) или дефолты:

* model — путь к YOLO-модели (по умолчанию runs/segment/train5/weights/best.pt);
* imgsz, conf, iou — параметры предсказания YOLO;
* device — "0" (GPU) или "cpu";
* dataset\_root — папка-накопитель для кропов EasyOCR (dataset\_for\_easyocr);
* fms\_csv — путь к справочнику подразделений (helper\_files/fms\_unit.csv).

Разово при старте сервиса инициализируется словарь подразделений:

from fms\_units import init\_fms\_units

init\_fms\_units(args.fms\_csv)

## 5) Запуск пайплайна: run\_pipeline\_on\_bgr(...)

На вход — исходное изображение **BGR** (OpenCV). На выход — структурированный словарь результатов.

### 5.1. Инициализация моделей

* Загружается YOLO (детекция полей) — YOLO(model\_path).
* Инициализируется EasyOCR Reader для ['ru','en'] (и отдельная конфигурация для MRZ), GPU/CPU — по параметру device.

### 5.2. Stage 1 — Автоповорот «тонкой» подстройкой

Функции из **stages.py** и **detection.py**:

* choose\_best\_angle\_by\_sweep(model, bgr0, imgsz, conf, iou, device)
  + Пробует набор маленьких углов (±1..±30°), на каждом запускает YOLO и считает «оценку сцены» (scene\_score\_basic).
  + Возвращает лучший угол и соответствующие детекции.
* Применяет поворот к исходной картинке (если надо).

Зачем: поймать мелкий поворот, чтобы дальше аккуратнее выставить поля и увеличить шанс хорошего OCR.

### 5.3. Stage 1.5 — Нормализация без зеркал

* normalize\_without\_mirror(bgr1, dets\_best)
  + Если MRZ оказался «вверху», разворачивает картинку на 180° (паспорт должен быть «как читаемая книжка»).

### 5.4. Stage 2 — Выбор лучшего k\*90° (0/90/180/270)

* choose\_best\_k90\_layout(bgr2, dets2):
  + Проверяет 4 поворота.
  + Для каждого поворота пересчитывает детекции (координаты).
  + Оценивает композицию (scene\_score\_passport\_layout): MRZ снизу и длинный по горизонтали, фото слева-снизу, серия вертикальная справа и т.д.
  + Забирает тот, где всё «правдоподобнее».

### 5.5. Stage 3 — Спец-режим «photo mode»

* Если **нет** page1, но **есть** photo, включаем choose\_best\_k90\_photo\_mode(...):
  + Другие правила компоновки (фото портрет, MRZ ниже фото и т.д.).
  + Этот шаг нужен для «половинчатых» разворотов, где попала, например, только страница с фото.

### 5.6. Итог после переориентации

* Получаем «финальную сцену» (final\_img, final\_dets) и помечаем флаг used\_stage = "stage2" | "stage3".

## 6) Распознавание полей (OCR)

Проходит по всем найденным полям (кроме «служебных» — passport/page1/page2/photo). На каждый бокс:

### 6.1. Вырезание ROI

* crop\_from\_xyxy(final\_img, d["xyxy"], expand\_frac=...) (из **utils.py**)
  + Для «длинных» текстов (issued\_by, birth\_place) захватывает больше рамку, чтобы не отрезать буквы.

### 6.2. Спец-случай для series\_number

* Серия/номер обычно вертикальны — поворачиваем кроп на -90°, чтобы OCR читал по строке.

### 6.3. Предобработка + OCR

* preprocess\_for\_ocr(crop, field) — несколько вариантов (контраст, инверсия+адаптивный порог, Оцу).
* ocr\_text\_and\_conf(variants, field, reader) — гоняем EasyOCR на каждом варианте, собираем среднюю уверенность и выбираем лучший текст.
* postprocess\_field(field, text) — лёгкая чистка (лишние пробелы и т.п.).
* Результат (текст, уверенность, bbox) складывается в ocr\_results[field].

### 6.4. Подсказки по ФИО без автозамены

* Для fam, name, sername вызывается correct\_field(..., aggressive=True) (из **post\_ocr\_corrector.py**):
  + Возвращает **кандидатов** из словарей (rapidfuzz/levenshtein), но **не подменяет** распознанное.
  + Кандидаты пишутся в ocr\_results[field]["candidates"] — пригодятся для UI/ручного подтверждения.

### 6.5. Сохранение кропов в датасет

* write\_crop\_to\_dataset(dataset\_root, field, crop, text) — сохраняет png в dataset\_for\_easyocr/<поле>/... + дописывает gt.txt.  
  Это пополняет тренировочную выборку для последующих дообучений.

## 7) Словарь подразделений (issued\_code ↔ issued\_by)

* Собираем распознанные issued\_code и issued\_by.
* resolve\_issued\_by(code\_text, issued\_by\_text) (из **fms\_units.py**):
  + Нормализует код (поддерживает 610-041, 610 041, 610041 → 610-041).
  + Ищет этот код в справочнике (helper\_files/fms\_unit.csv).
  + Если коду соответствует **несколько** названий — сравнивает их с распознанным issued\_by\_text по похожести и выбирает лучший.
  + В ocr\_results["issued\_code"]["text"] кладёт нормализованный код, а в ocr\_results["issued\_by"]["text"] — самое «правдоподобное» название из справочника (плюс метаинформацию dict\_meta: откуда взялось, какие кандидаты сравнивались и т.д.).

8) Сбор итогов: «сырой» результат

{

"used\_stage": "stage2" | "stage3",

"fields\_raw": {

"<field>": {

"text": "...",

"conf\_ocr": ...,

"conf\_det": ...,

"bbox\_xyxy": [...],

"crop\_path": "...",

"candidates": [...], # если ФИО

"dict\_meta": {...}, # если issued\_by подтянули из словаря

},

...

}

}

9) format\_fields\_for\_api(fields\_raw) трансформирует структуру к виду:

{

"<field>": {

"value": "<текст>",

"confidence": 0.93,

"candidates": [...], // если есть

"dict\_meta": {...} // если есть

}

}

## 10) Ответ клиенту (и отладочные файлы)

### В CLI-режиме

* Пишется path/to/passport.ocr.json.
* В results/ сохраняются:
  + финальное изображение с нарисованными «цветными» коробками и подписями (stage2 / stage3),
  + промежуточные (по желанию: .seg\_oriented.jpg, .final.jpg и т.д.).

### В сервис-режиме

* Формируется ответ вида:

{

"status": "ok",

"result": {

"image": "image\_b64|path",

"used\_stage": "stage3",

"fields": { ... } // формат из п.8

}

}

Публикуется в reply\_to (если был задан) либо в reply.<correlation\_id>.

* Если в пайплайне случилась ошибка — отправляется {status:"error", error:"pipeline\_failed", detail:"..."}.

## 11) Что где реализовано (быстрая карта модулей)

* **main.py** — «дирижёр»: CLI и сервис, RabbitMQ, запуск всего пайплайна, сборка ответа, сохранение результатов.
* **detection.py** — вызов YOLO, формирование списка детекций.
* **utils.py** — базовые операции с изображениями (повороты, кропы, minAreaRect, углы, выбор лучшей детекции).
* **stages.py** — логика выбора ориентации/компоновки: sweep-углы, k\*90, photo-mode, отрисовка разметки.
* **ocr.py** — предобработка, вызовы EasyOCR, многовариантность, сбор уверенности, повторные попытки, постобработка.
* **post\_ocr\_corrector.py** — подсказки для ФИО (словарная коррекция без автозамены).
* **fms\_units.py** — нормализация кода подразделения и сопоставление «код → выдавший орган» (с учётом нескольких кандидатов).
* **verifier.py** (опционально) — LLM для «умной» коррекции, если подключаем.

## 12) Таймауты и отказоустойчивость

* API ждёт ответ до RPC\_TIMEOUT\_SEC (30 сек по умолчанию). Если не дождался — возвращает {status:"timeout"}.
* Воркер аккуратно логирует исключения и отправляет {status:"error"} вместо того, чтобы «падать».
* Критичные словари (FMS) инициализируются один раз, при ошибках — просто отключается мэппинг (не ломает пайплайн).

## 13) Что можно подкрутить (пара «ручек»)

* imgsz/conf/iou — чувствительность детекции.
* expand\_frac в crop\_from\_xyxy — ширина рамки вокруг текстовых полей.
* Порог «доверия» для повторных попыток в process\_field\_with\_retry.
* Агрессивность подсказок по ФИО (в посткорректоре).
* Приоритеты и «баллы» в scene\_score\_\* (для тонкой настройки ориентации/компоновки).
* Пути сохранения отладочных картинок (results/) и накопительных датасетов (dataset\_for\_easyocr/).

**Как это все запускать (без контейнера)**

1) убеждаемся, что у нас Ubuntu 24, все драйвера и CUDA 12.\* установлены.

2) сначала ставим зависимости, pip3 install -r requirements.txt

3) запускаем rabbitMQ, в чистом виде (включен guest аккаунт с паролем guest)

4) запускаем python3 api\_server.py

5) запускаем python3 main.py --service --imgsz 1536 --conf 0.25 --iou 0.6 --device 0

6) шлем POST на http://localhost:8085/v1/process запросы, с телом

{"image\_b64":"<base64 файла для распознавания>","meta":{"client":"demo"}}

Ответ получаем такой,

{

"request\_id": "f44a2b3d-68e3-4cbb-b071-ebeb6a909af9",

"status": "ok",

"result": {

"status": "ok",

"result": {

"image": "image\_b64",

"used\_stage": "stage3",

"fields": {

"mrz": { значение mrz

"value": "'3\" {5 5 #",

"confidence": 0.418

},

"series\_number": { серийный номер

"value": "03 15 810755",

"confidence": 1.0 уверенность

},

"birth\_date": { дата рождения

"value": "13.04.2001",

"confidence": 0.983

},

"birth\_place": { место рождения

"value": "ГОРСОЧИ КРАСНОДАРСКОГОКРАЯ",

"confidence": 1.0

},

"fam": { фамилия

"value": "БОРЕВ",

"confidence": 1.0,

"candidates": [ близкие варианты по словарям

[

"бореев",

98

],

[

"борнев",

94

],

[

"борцев",

94

],

[

"борчев",

94

],

[

"борщев",

94

]

]

},

"name": { имя

"value": "ТИМОФЕЙ",

"confidence": 0.999,

"candidates": [

[

"тимофей",

100

],

[

"тимафей",

92

],

[

"тим",

90

],

[

"мо",

88

],

[

"ти",

88

]

]

},

"sername": { отчетство

"value": "ДАНИИЛОВИЧ",

"confidence": 1.0,

"candidates": [

[

"даниилович",

100

],

[

"данилович",

100

],

[

"даниелович",

100

],

[

"даниэлович",

100

],

[

"даниялович",

100

]

]

},

"sex": { пол

"value": "МУЖ",

"confidence": 1.0

},

"issued\_by": {

"value": "",

"confidence": null,

"dict\_meta": {

"source": "ocr",

"note": "cannot normalize issued\_code",

"normalized\_code": null,

"candidates": [],

"picked": ""

}

}

}

}

},

"detail": null

}