Synthèse du Projet

Nous avons un commerçant. Il gère sa boutique. Vend des produits, encaisse ses clients. Tout va pour le mieux. Mais à la fin de la journée ou de la semaine une autre réalité le rattrape : la paperasse.

Imaginez devoir entrer manuellement les informations d'une facture dans par exemple, un logiciel tierce. Vous devrez par exemple y entrez les informations de l'utilisateur. Les informations des articles, leurs noms leurs quantité, prix et autres informations et ça de manière séquentielle, en veillant à ne pas faire d' erreurs. C'est déjà très fastidieux. Mais la tâche devient en plus très lourde et chronophage si plusieurs factures doivent être traités.

Il ne s'en rend peut être pas compte mais ce temps coûte cher. De plus chaque lignes de ces factures renferme une mine d'or, des informations pourrait l'aider à mieux vendre.

Mais tant qu'il reste coincé à faire de la saisie manuelle… Toutes ces infos dorment. Et les opportunités avec.

C'est là que mon application intervient. Elle vise à transformer chaque factures image, en données exploitables, automatiquement. Puis à centraliser ces infos dans une interface claire et intelligente.

📌 1. Préparation du Projet

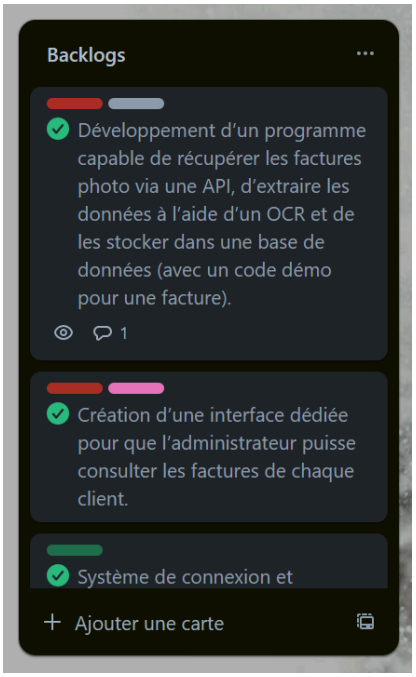
Le projet représentait un défi conséquent, avec un brief détaillé et une échéance très serrée d'environ mois. Face à ce challenge, j'ai immédiatement compris l'importance cruciale d'une organisation rigoureuse. J'ai d'abord analysé en détail les besoins du client afin de déterminer clairement les fonctionnalités attendues pour l'application.

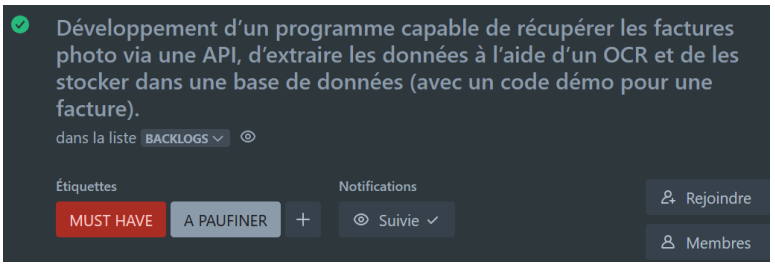
Pour faciliter la gestion du projet, j'ai structuré ces fonctionnalités en trois catégories : **Must have** (indispensables), **Nice to have** (souhaitables) et **Dream** (optionnelles).

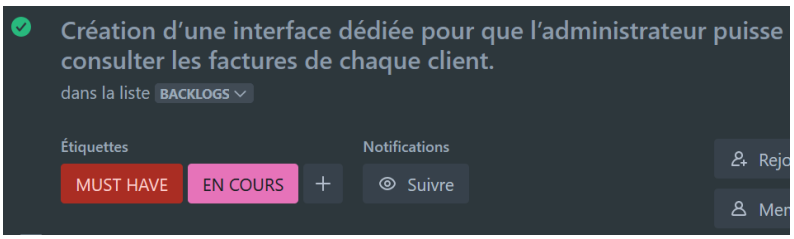
Must Have	Nice To Have	Dream
Programme capable de récupérer les factures photo via une API, d'extraire les données à l'aide d'un OCR et de les stocker dans une base de données	Système de connexion et d'inscription	interface permettant à l'utilisateur de consulter l'historique de ses factures.
interface dédiée pour que	Système de monitoring	Dashboard de monitoring

l'administrateur puisse consulter les factures de chaque client.		
		système de clustering de client
		Système de recommandation et propositions de produits

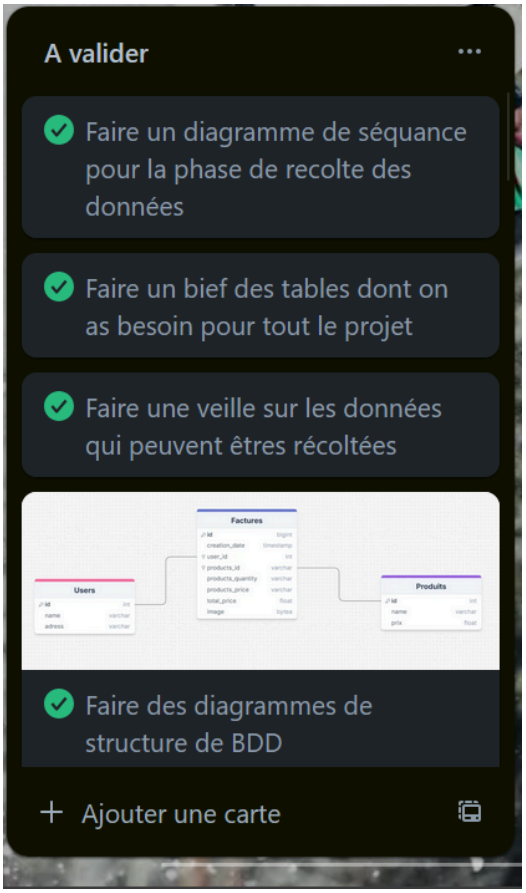
Pour piloter efficacement l'avancement, j'ai utilisé Trello afin de gérer mes tâches avec clarté. J'ai créé des backlogs avec des étiquettes indiquant les priorités, ce qui m'a permis de concentrer mes efforts sur les tâches essentielles.



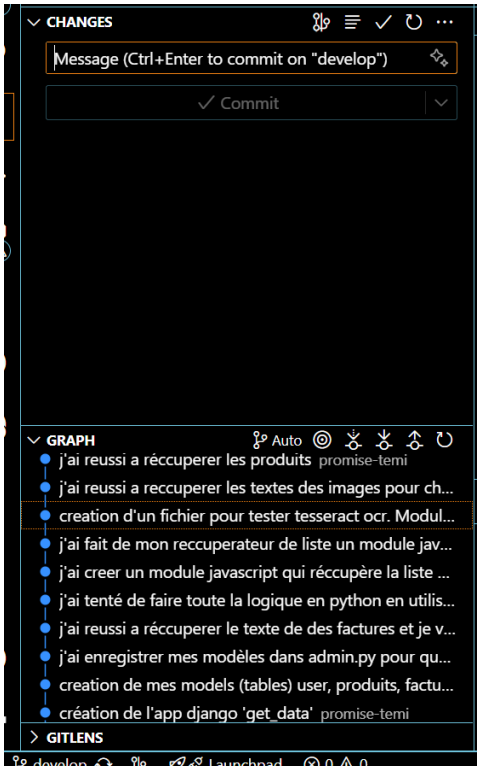


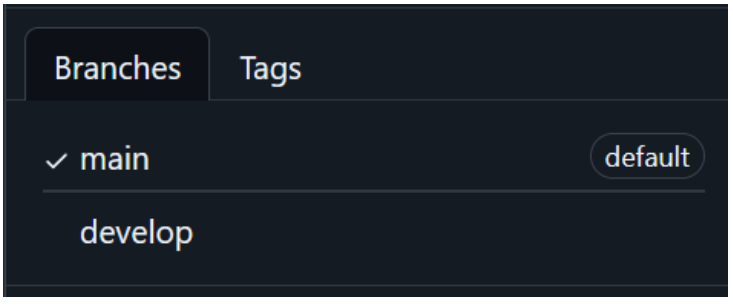


La gestion des tâches s'est faite de manière itérative et adaptative : au fur et à mesure du projet, j’ajoutais dans Trello des tâches précises accompagnées de détails sur les décisions prises, les découvertes effectuées et les ressources mobilisées.



Parallèlement, j’ai adopté une organisation Git claire avec deux branches principales : `develop` pour les développements réguliers et `main` pour les versions stables et livrables. Chaque commit sur `develop` était documenté de manière précise afin de conserver une traçabilité complète de l'évolution du projet.



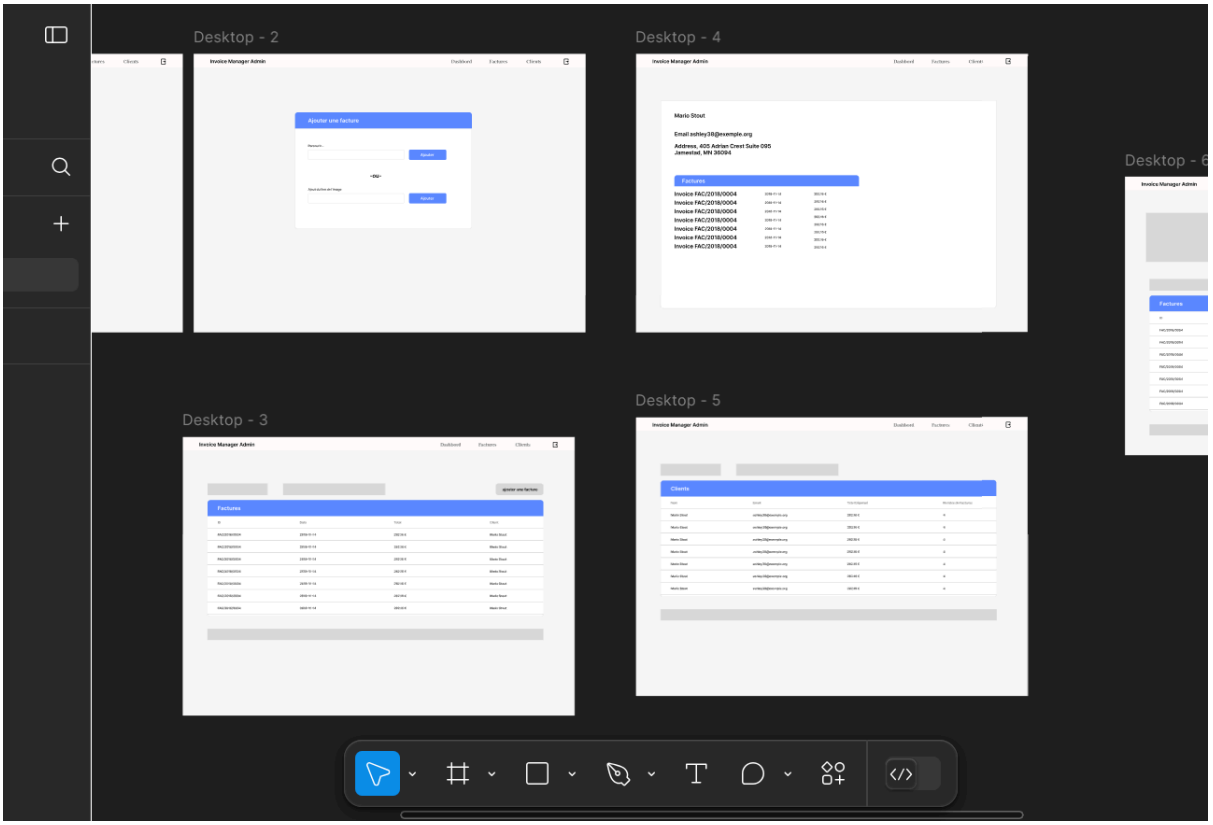


Figma & Bootstrap – Conception d’interface efficace

Pour éviter de perdre du temps pendant du développement, j’ai utilisé **Figma** pour préparer une idée claire de ce à quoi devaient ressembler les interfaces de l’application.

Ma démarche

- J’ai réalisé des maquettes **fonctionnalité par fonctionnalité**, au fur et à mesure du projet (démarche itérative)
- L’objectif n’était pas de faire du design parfait, mais de créer un **repère visuel rapide** (souvent sous forme de **zonings**)
- Ces maquettes m’ont permis de **mieux anticiper la structure HTML/CSS** à coder et de ne pas partir à l’aveugle



Pour accélérer le développement, j’ai utilisé **Bootstrap** :

- Grâce à ses composants prêts à l’emploi, j’ai pu gagner beaucoup de temps
- Le style est resté simple, mais fonctionnel et responsive

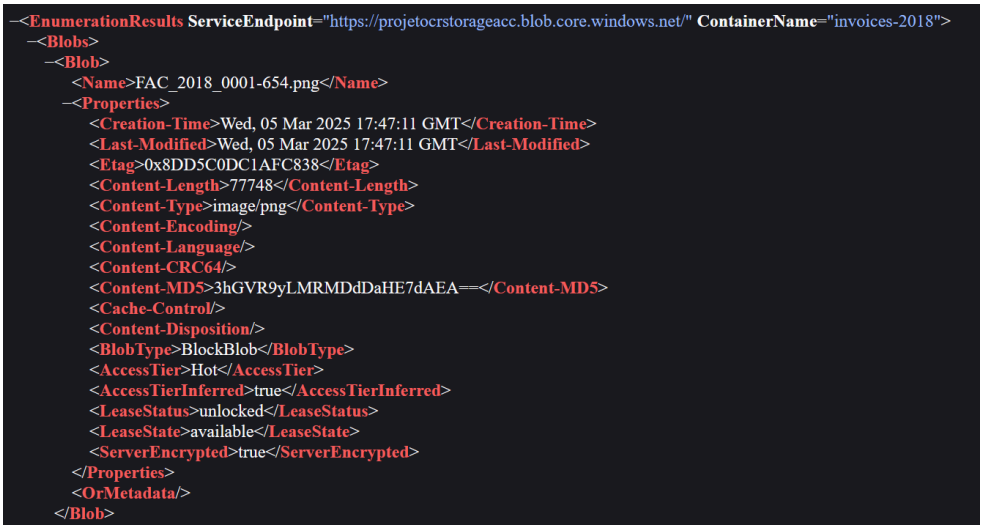
Grace à ca j’avais une interface que je pouvait implémenter suffisamment rapidement

📌 2. Fonctionnalité Principale : Extraction et Traitement des Données

Afin de développer l’application, j’avais besoin de données réelles : dans ce cas, des factures. Pour le contexte, on nous avait fourni des images de factures, stockées dans une base Azure. Chaque facture était accessible via un lien dont il suffisait de modifier l’ID. Un fichier XML regroupait tous les IDs nécessaires.

J’ai donc automatisé la récupération de ces factures grâce à un script Python, qui les téléchargeait dans un dossier local.

Capture d’un bout du xml:

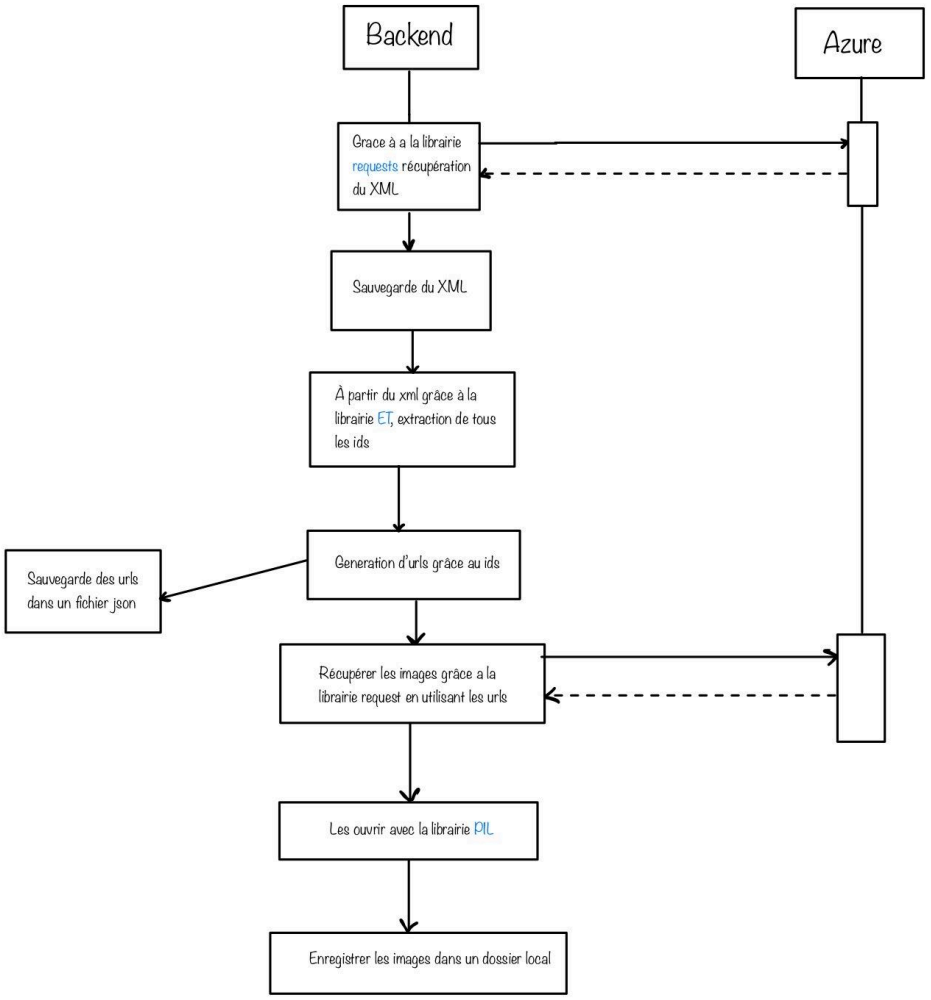


```
−<EnumerationResults ServiceEndpoint="https://projetocrstorageacc.blob.core.windows.net/" ContainerName="invoices-2018">
  −<Blobs>
    −<Blob>
       <Name>FAC_2018_0001-654.png</Name>
      −<Properties>
          <Creation-Time>Wed, 05 Mar 2025 17:47:11 GMT</Creation-Time>
          <Last-Modified>Wed, 05 Mar 2025 17:47:11 GMT</Last-Modified>
          <Etag>0x8DD5C0DC1AFC838</Etag>
          <Content-Length>77748</Content-Length>
          <Content-Type>image/png</Content-Type>
          <Content-Encoding/>
          <Content-Language/>
          <Content-CRC64/>
          <Content-MD5>3hGVR9yLMRMDdDaHE7dAEA==</Content-MD5>
          <Cache-Control/>
          <Content-Disposition/>
          <BlobType>BlockBlob</BlobType>
          <AccessTier>Hot</AccessTier>
          <AccessTierInferred>true</AccessTierInferred>
          <LeaseStatus>unlocked</LeaseStatus>
          <LeaseState>available</LeaseState>
          <ServerEncrypted>true</ServerEncrypted>
       </Properties>
       <OrMetadata/>
    </Blob>
```

lien vers une image:

https://projetocrstorageacc.blob.core.windows.net/invoices-2018/FAC_2018_0001-654.png?sv=2019-12-12&ss=b&srt=sco&sp=rl&se=2026-01-01T00:00:00Z&st=2025-01-01T00:00:00Z&spr=https&sig=%2BjCi7n8g%2F3849Rprey27XzHMoZN9zdVfDw6CifS6Y1U%3D

J'ai automatisé la récupération de ces factures avec un script Python qui les téléchargeait automatiquement dans un dossier local.



📌 3. Sélection de l’OCR et Prétraitement des Images

Pour extraire automatiquement les données textuelles des factures, j'ai évalué plusieurs solutions d’OCR :

- **Tesseract.js** : Complexe à configurer et maintenir (gestion difficile de l’asynchrone JavaScript).
- **EasyOCR** : Facile d'utilisation, mais résultats imprécis.

- **Pytesseract** : Outil finalement retenu, performant même sans prétraitement, basé sur Python. installation fastidieuse mais en vaut la peine. J’ai dû l’installer dans mon

environnement virtuel (venv), configurer l'exécutable puis l'ajouter à mon PATH. J'ai découvert que j'avais accidentellement installé PyTesseract directement sur ma machine, en dehors du venv, ce qui le rendait accessible partout. C'est cette découverte qui m'a poussé à dockeriser mon application : je voulais reproduire ce fonctionnement sans imposer aux futurs utilisateurs une installation laborieuse qu'ils ne maîtriseraient pas nécessairement.

Cet article m'a été très utile pour l'installation de pytesseract :

https://www.datacamp.com/fr/tutorial/optical-character-recognition-ocr-in-python-with-pytesseract

Pour améliorer la qualité de la reconnaissance de texte (OCR), j'ai ajouté des paramètres spécifiques à Pytesseract :

- config = "--psm 6" pour que tesseract traite l'image comme un bloc de texte uniforme, aligné en lignes, sans chercher à détecter plusieurs colonnes ou structures complexes. bref il lit l'image ligne par ligne.

- lang = "eng" parce que les factures étaient en anglais et je souhaitais que Pytesseract les interprète correctement.

```
#image to text grace à pytesseract
text = pytesseract.image_to_string(image_final, lang='eng', config="--psm 6")
```

De plus, j'ai mis en place un pipeline de prétraitement des images avec OpenCV pour optimiser davantage les résultats.

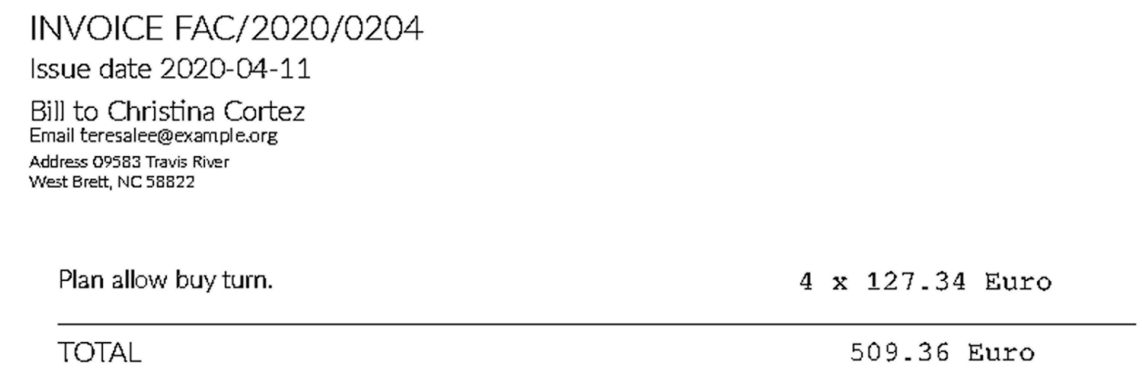
- Masquage du QR code

- Conversion en niveaux de gris

- Seuillage adaptatif

- Renforcement de la netteté

Voici l'exemple d'une image prétraité :



INVOICE FAC/2020/0204

Issue date 2020-04-11

Bill to Christina Cortez

Email teresalee@example.org

Address 09583 Travis River

West Brett, NC 58822

Plan allow buy turn.	4 x 127.34 Euro
TOTAL	509.36 Euro

Ces deux articles m'ont été très utiles :

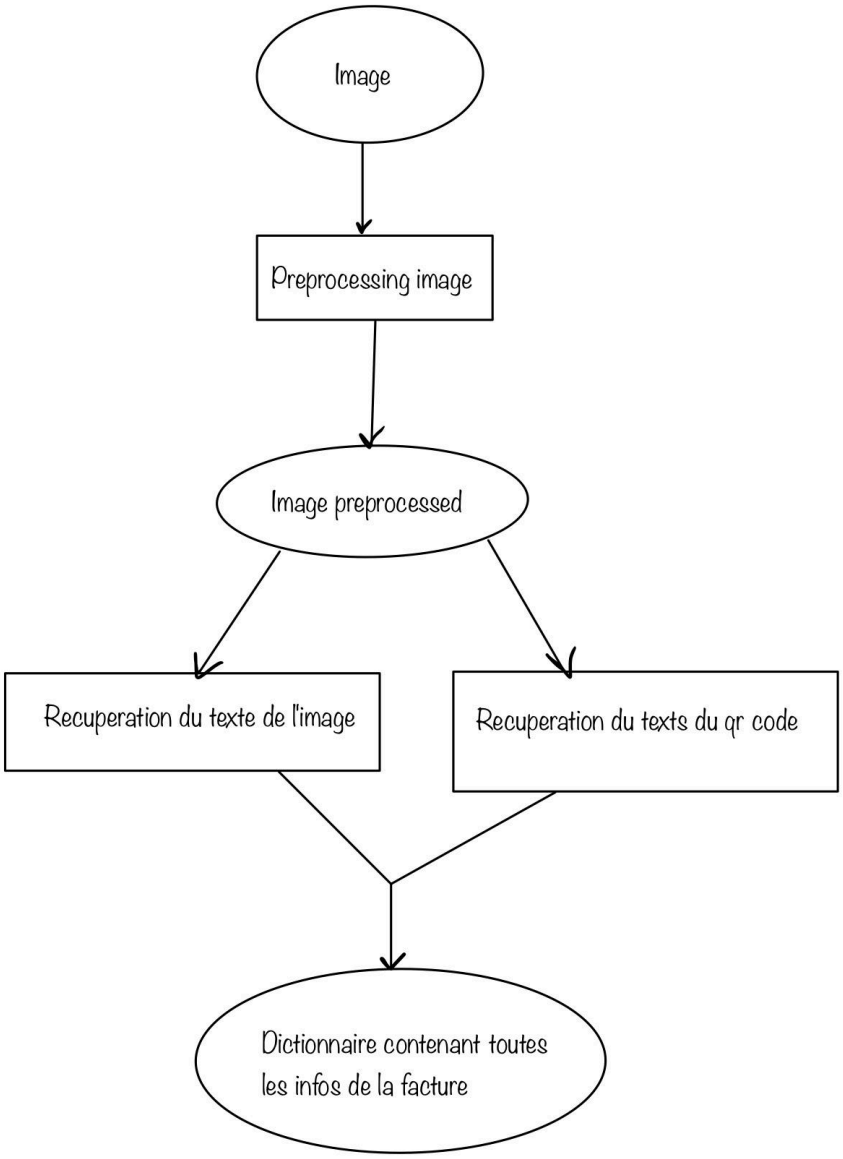
https://www.researchgate.net/publication/363769945_Improve_OCR_Accuracy_with_Advanced_Image_Preprocessing_using_Machine_Learning_with_Python

https://medium.com/@Hiadore/preprocessing-images-for-ocr-a-step-by-step-guide-to-quality-recovery-923b6b8f926b

J'ai aussi identifié des erreurs récurrentes mineures (comme la confusion entre les crochets "[" et la lettre "l") et les ai corrigées avec un simple `replace()`.

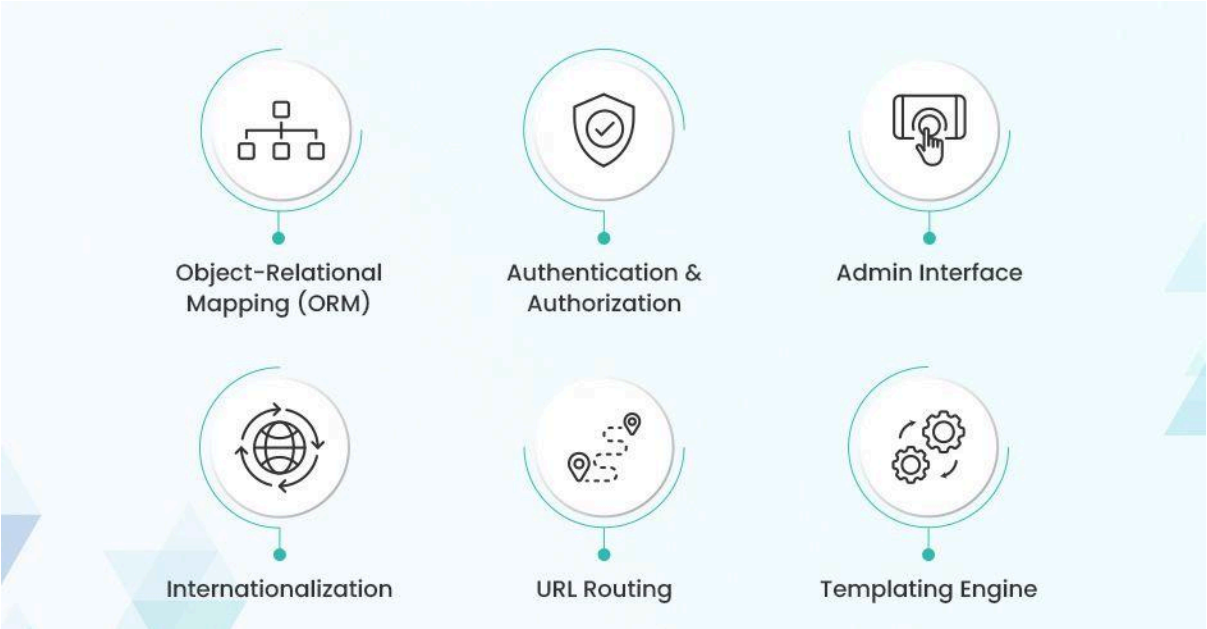
Pour récupérer les données d'un code QR, j'ai fait appel à la bibliothèque OpenCV, qui permet non seulement de détecter les codes QR, mais aussi d'en extraire les informations qu'ils contiennent.

```python
# Initialiser le détecteur de QR codes
detector = cv2.QRCodeDetector()        You, 3 weeks

# Détecter et décoder le QR code
data, bbox, _ = detector.detectAndDecode(image)
if data:
    print("Données du QR code :", data)
    datas = data.lower().split('\n')
    arranged_datas = {
        'invoice_time': get_time(datas),
        'genre' : get_genre(datas),
        'birth' : get_birth(datas)
```



📌 4. Intégration avec Django

Pour l'application finale, j'ai choisi Django pour sa robustesse et ses fonctionnalités natives : gestion des URLs, ORM, templates HTML dynamiques, authentification intégrée et une interface admin prête à l’emploi.



Quand on crée un projet Django, plusieurs fichiers et dossiers sont générés automatiquement.

1. Le dossier du projet Django (souvent nommé comme le projet lui-même)

Ce dossier contient la **configuration principale** du projet. À l'intérieur, on retrouve notamment :

- **`settings.py`** : c'est le fichier de configuration du projet. On y définit les paramètres généraux comme la base de données utilisée, les applications installées, les chemins des fichiers statiques, la langue, le fuseau horaire, etc.

- **`urls.py`** : ce fichier contient les **routes principales** du projet. C'est ici qu'on relie des URLs à des vues. On peut aussi inclure les fichiers `urls.py` des différentes applications ici.

- **`wsgi.py`** et **`asgi.py`** : ces fichiers sont utilisés pour déployer le projet. Ils permettent à Django de communiquer avec un serveur web. En développement, tu n'as pas souvent besoin d'y toucher.

2. Le fichier `manage.py`

Il se trouve à la racine du projet. C'est un outil en ligne de commande pour interagir avec Django. Par exemple, tu peux l'utiliser pour lancer le serveur (`python manage.py runserver`), créer des applications, exécuter des migrations, créer un superuser, etc.

3. Les applications Django (apps)

Dans Django, un projet est généralement composé de plusieurs **apps**. Chaque app représente une **fonctionnalité ou une partie logique** du projet (par exemple : blog, utilisateurs, panier, etc.).

Quand on crée une app avec `python manage.py startapp nom_de_l_app`, Django génère une structure de base :

- **`models.py`** : on définit ici les **modèles** de données (les tables de la base de données).
- **`views.py`** : on y écrit les **vues**, c'est-à-dire la logique qui va répondre à une requête et retourner une réponse (HTML, JSON, etc.).
- **`urls.py`** *(à créer si nécessaire)* : permet de définir les **routes propres à l'app**, qu'on peut ensuite inclure dans le fichier `urls.py` principal du projet.
- **`admin.py`** : permet de personnaliser l'interface d'administration Django pour les modèles définis.
- **`apps.py`** : configuration de base de l'app, rarement modifiée manuellement.
- **`migrations/`** : dossier où Django stocke les fichiers de migration pour les modèles (utile pour créer ou modifier la base de données).

4. Template, Static, Forms (souvent ajoutés)

Dans chaque app ou dans un dossier global, on peut aussi ajouter :

- **`templates/`** : dossiers contenant les fichiers HTML utilisés par les vues.
- **`static/`** : pour les fichiers statiques (CSS, JavaScript, images…).
- **`forms.py`** *(à créer)* : pour définir des formulaires personnalisés (liés ou non à des modèles).

📌 Création de ma première app

J'ai créé le projet Django et l'app principale `get_data` pour gérer l'extraction et le traitement des factures. Dans cette app, dans le fichier models.py, j'ai implémenté les modèles principaux (`Produit`, `User`, `Facture`, `FactureProduct`) et utilisé les commandes Django classiques (`makemigrations`, `migrate`) pour gérer les migrations.

Exemple de modèle:

```python
class User(models.Model):
    name = models.CharField(max_length=250)
    email = models.CharField(max_length=300, null=True, blank=False, unique=True)
    adress = models.CharField(max_length=500)
    city = models.CharField(max_length=100, null=True)
    state = models.CharField(max_length=100, null=True)
    postal_code = models.CharField(max_length=10, null=True)
    type = models.CharField(max_length=10, default="user")
    genre = models.CharField(max_length=10, null=True)
    birth_day = models.DateField(null=True)
```

J'ai du ajouter mes models à admin.py pour y avoir accès dans l'interface admin

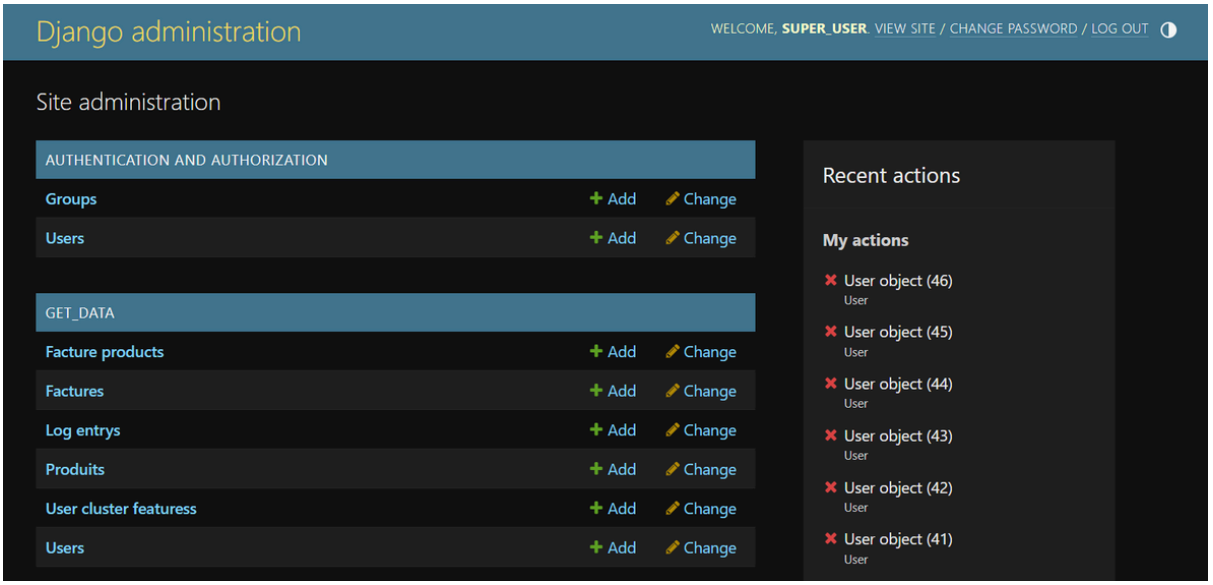
```python
from django.contrib import admin
from .models import Produit, Facture, User, FactureProduct,

# Register your models here.
admin.site.register(Produit)
admin.site.register(User)       You, 4 weeks ago • j'ai enr
admin.site.register(Facture)
admin.site.register(FactureProduct)
admin.site.register(UserClusterFeatures)
```

J'ai dailleur créé un super-admin (via la commande createsuperuser) pour accéder à l'interface Django Admin, disponible à l'URL habituelle de l'admin.

l'interface est accessible via l'url : http://127.0.0.1:8000/admin/login/

l'identifiant est : super_user

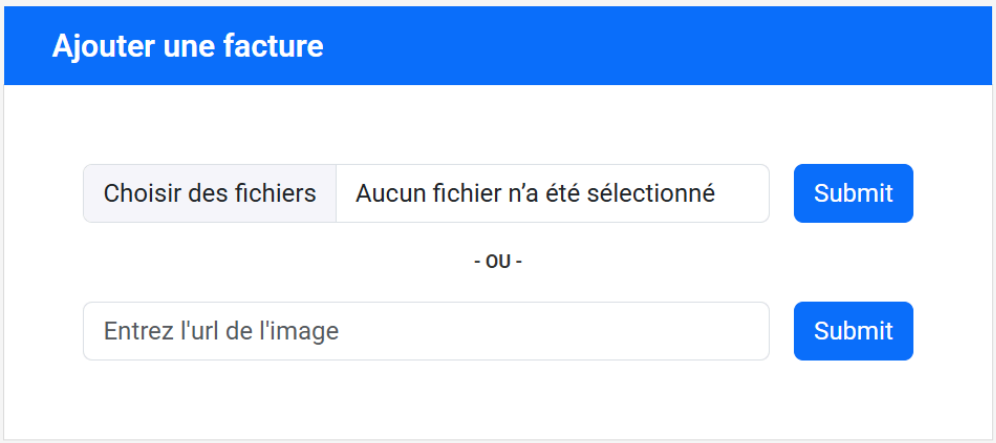
et le mot de passe est : password



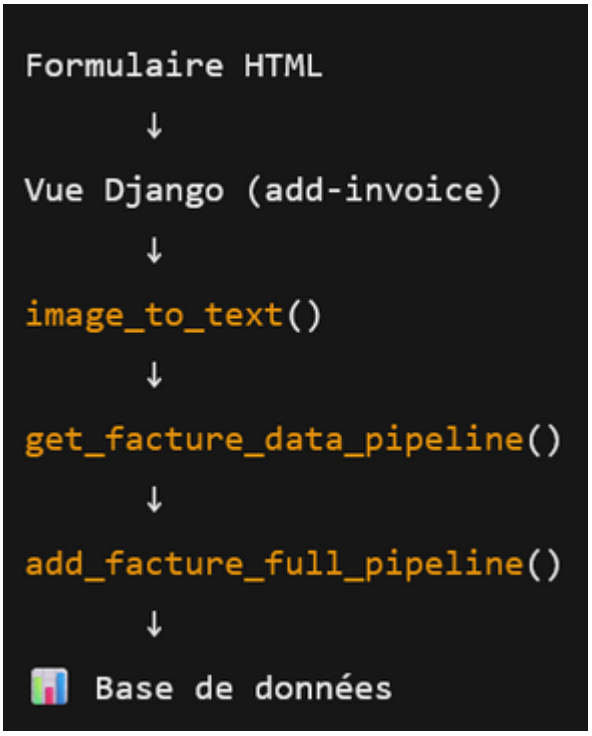
📌 5. Automatisation du Flux de Facturation

J'ai conçu une page HTML (`create-invoice.html`) proposant deux formulaires :

- Un pour téléverser les images localement (`add-invoice2`)
- Un autre pour renseigner les URLs (`add-invoice`)



Ces vues Django appellent ensuite la fonction centrale `image_to_text()`, qui traite l'image, extrait le texte via OCR (`pytesseract`), et les données du QR code avec OpenCV. Les données extraites sont structurées par la fonction `get_facture_data_pipeline()` dans un dictionnaire clair, puis insérées automatiquement dans la base avec `add_facture_full_pipeline()`. Ce pipeline vérifie également la présence d'une facture pour éviter les doublons.



📌 6. Tests et Validation

Pour valider cette fonctionnalité centrale, j'ai écrit des tests rigoureux et un peu rigides avec Django (`django.test.TestCase`) :

- `test_image_to_text_file()` pour tester les images locales
- `test_image_to_text_link()` pour les images hébergées sur Azure

Ces tests contrôlent la présence des données critiques extraites via OCR et QR code.

Pour information django utilise la syntaxe de la librairie unitest pour les tests.

📌 7. Interface d'Administration Personnalisée

J'ai créé une app distincte appelée `admin_interface` avec des vues pour afficher les factures (`factures.html`, via `getfacture()`), les détails d'une facture (`info_facture.html`, via `get_single_facture()`), les clients (`clients.html`, via `get_client()`) et les détails d'un client (`info_client.html`, via `get_single_client()`).

Toutes securisé par un système de connection. Bien qu'il n'y ait pas de système d'inscription à proprement parler, j'ai, au tout début du projet, créé un super utilisateur pour accéder à l'interface d'administration de Django.

Pour indiquer à Django que je souhaite que certaines vues soient protégées et que l'utilisateur doive absolument être connecté, j'ai simplement ajouté le décorateur `@login_required` au-dessus des vues concernées.

```python
from django.contrib.auth.decorators import login_required
```

```python
@login_required
def factures(request):
    data = getfacture()
    template = loader.get_template('factures.html')
    context = {
        'mes_factures': data,      You, last week • j'airive
    }
    return HttpResponse(template.render(context, request))
```

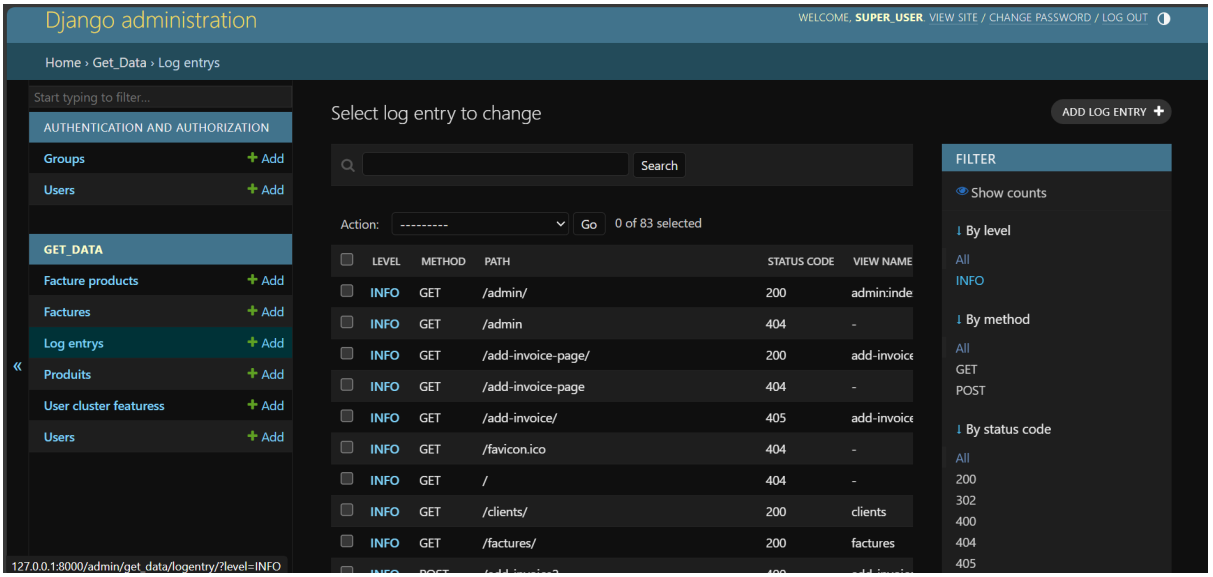
Il faut donc se connecter avec les identifiants du super utilisateur. Cependant, ce super utilisateur peut créer de nouveaux utilisateurs depuis l'interface d'administration et leur attribuer des droits spécifiques.

📌 9. Monitoring Applicatif

Pour garantir un suivi post-déploiement, j'ai implémenté un modèle `LogEntry` pour enregistrer toutes les requêtes HTTP importantes.

```python
class LogEntry(models.Model):
    level = models.CharField(max_length=20)
    message = models.TextField()
    path = models.CharField(max_length=255)
    method = models.CharField(max_length=10)
    status_code = models.PositiveSmallIntegerField()
    view_name = models.CharField(max_length=255, null=True, blank=True)
    ip_address = models.GenericIPAddressField(null=True, blank=True)
    duration = models.FloatField(null=True, blank=True)
    extra_data = models.JSONField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
```

Un middleware personnalisé (`RequestLoggingMiddleware`) capture automatiquement les détails des requêtes (méthode, URL, statut, durée) dans la base de données. Ces logs sont ensuite consultables via l'interface d'administration de Django grâce à `LogEntryAdmin`.



Pour activer ce middleware, j’ai ajouté sa référence dans le fichier `settings.py`.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    # MES MIDDLEWARES
    'get_data.middleware.RequestLoggingMiddleware',
]
```

📌 Conclusion

Cette approche rigoureuse, organisée et détaillée m'a permis de livrer une solution complète, robuste et évolutive, répondant précisément aux besoins initiaux tout en me permettant d'acquérir de solides compétences techniques et méthodologiques.