

**BRIEF 2C      02/2023**

## **Documentation Technique**

Ugo FOK-YIN (03)  
Pascal INDICE (04)

Lien GitHub du projet: [UgoDIA/Brief-2C](#)

# Sommaire

<b>A) Technologies et outils utilisés:</b>	<b>3</b>
<b>B) Maquette et diagramme de cas d'utilisation</b>	<b>4</b>
Maquette/Graphe de dialogue: Figma	4
Diagramme cas d'utilisation:	4
<b>C)Base de données et persistance des données</b>	<b>6</b>
Arborescence dossiers avec Django:	6
<b>D)Détails des fonctionnalités</b>	<b>9</b>
1)Upload:	9
2)Résultat:	12
3)API avec Django REST Framework:	17
4)Identification:	21
5)Monitoring:	23
<b>E) Conclusion</b>	<b>25</b>
1)Ce qui a été fait:	25
2)Ce qui est à faire:	25

## A) Technologies et outils utilisés:

### **Framework Django:**

Django est un framework de développement d'application web open source qui s'inspire du principe MVT.

[Documentation de Django](#) | [Documentation de Django](#)



### **PostgreSQL:**

PostgreSQL est un SGBD orienté objet puissant et open/source qui est capable de prendre en charge en toute sécurité les charges de travail de données les plus complexes.

[Documentation PostgreSQL 15.0](#)



### **HTML/CSS et Bootstrap:**

Bootstrap.css est un framework CSS qui organise et gère la mise en page d'un site web.

[Get started with Bootstrap · Bootstrap v5.2](#)



### **Numpy:**

NumPy est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux

[NumPy documentation — NumPy v1.24 Manual](#)



### **Tensorflow/keras:**

Développé par Google, TensorFlow est une plateforme de deep learning open source conçue pour concevoir des réseaux de neurones artificiels. Quant à Keras, c'est une API de haut niveau taillée pour le prototypage rapide de réseaux de neurones artificiels pouvant recourir à TensorFlow comme socle de mise en œuvre.

[TensorFlow 2 quickstart for beginners](#) | [TensorFlow Core Getting started \(keras.io\)](#)



## **jQuery:**

jQuery est une bibliothèque JavaScript libre et multiplateforme créée pour faciliter l'écriture de scripts côté client dans le code HTML des pages web.

[jQuery API Documentation](#)



## **DataTables:**

DataTables est un plugin jQuery qui peut être utilisé pour ajouter des contrôles interactifs et avancés aux tableaux HTML de la page Web. Cela permet également de rechercher, trier et filtrer les données du tableau en fonction des besoins de l'utilisateur.

[Manual \(datatables.net\)](#)



## **Django REST framework:**

Django REST Framework est un outil puissant et flexible utilisé pour créer des API Web avec Django.

[Quickstart - Django REST framework](#)



## **B) Maquette et diagramme de cas d'utilisation**

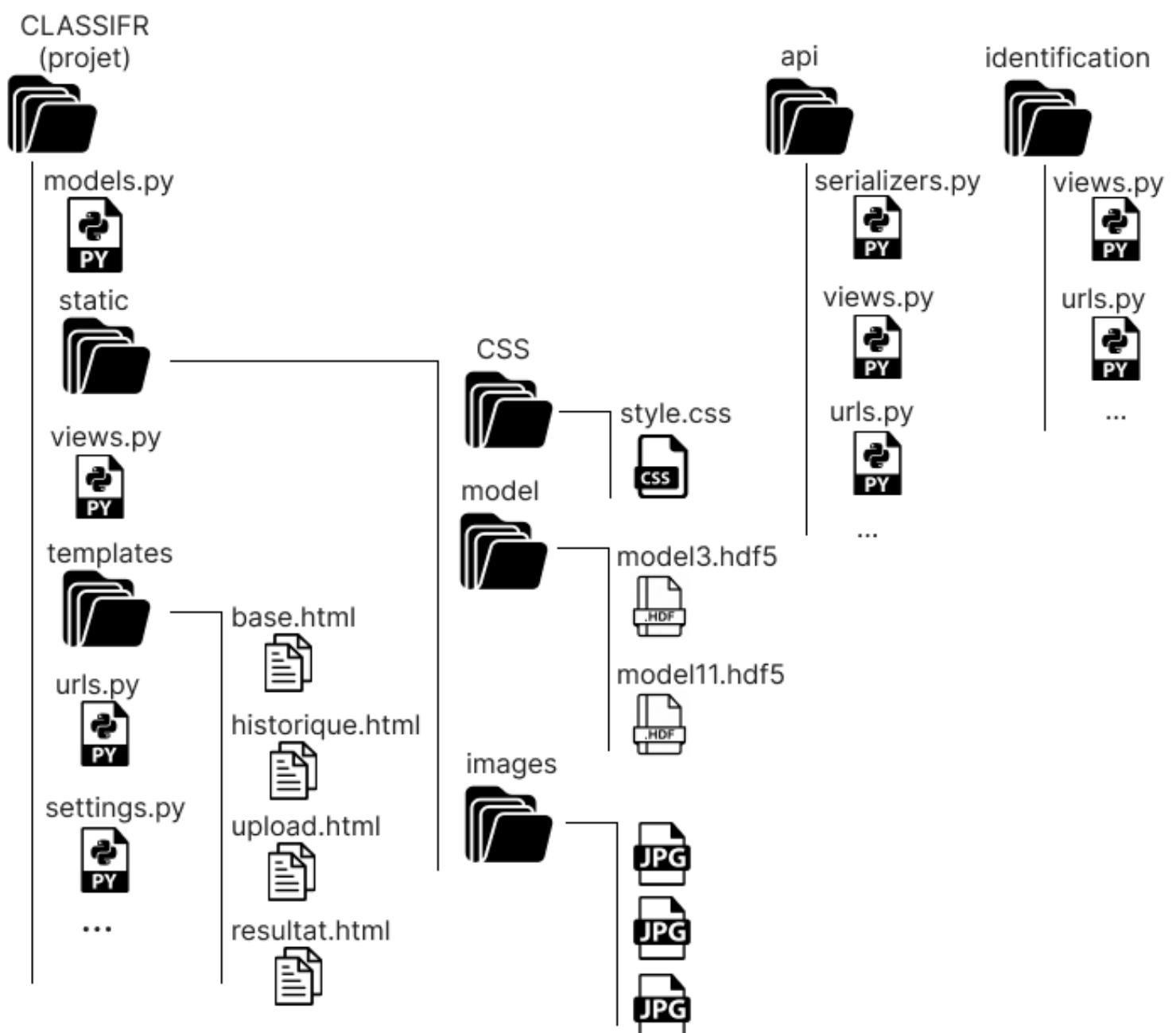
**Maquette/Graphe de dialogue:** [Figma](#)

**Diagramme cas d'utilisation:**



## C) Base de données et persistance des données

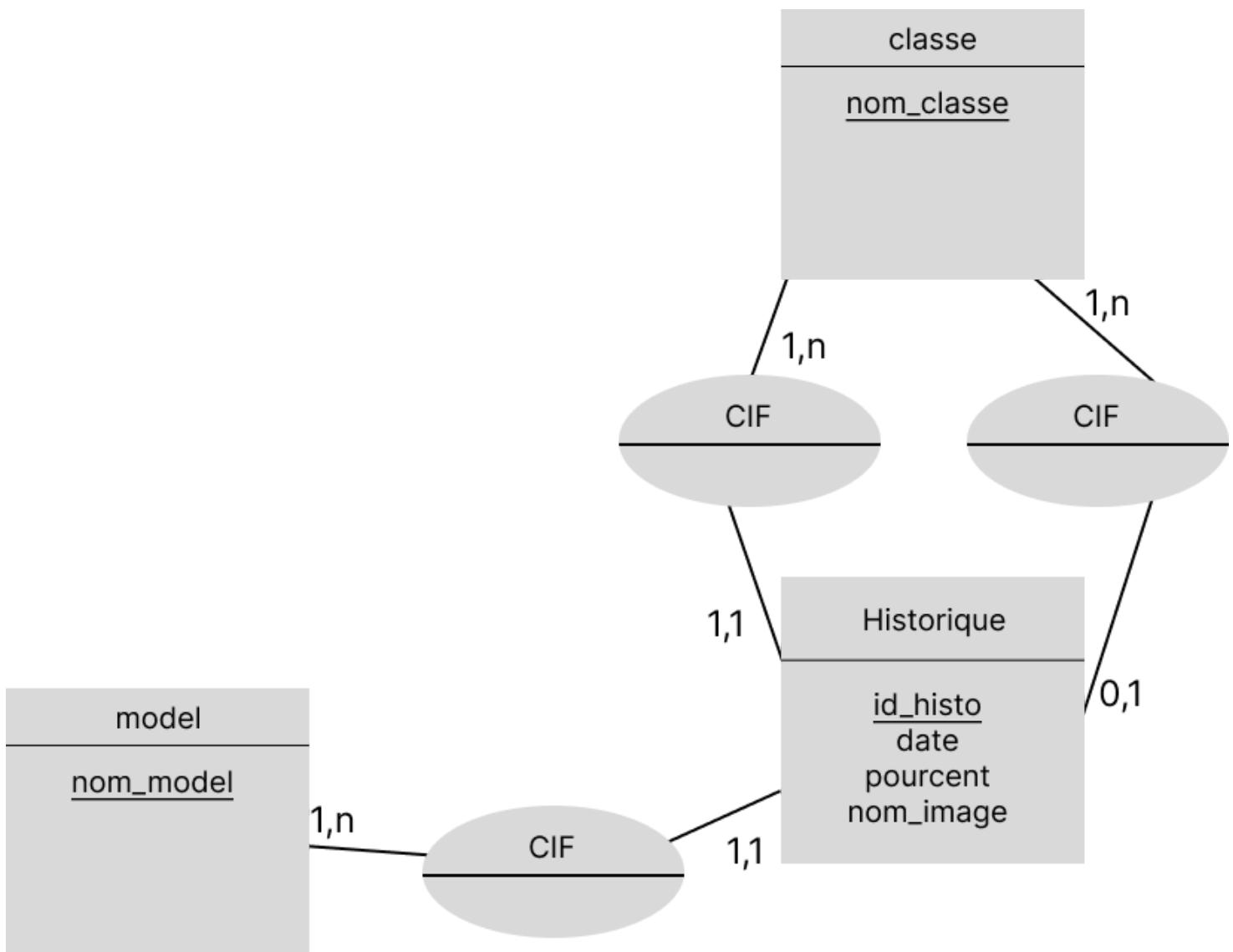
### Arborescence dossiers avec Django:

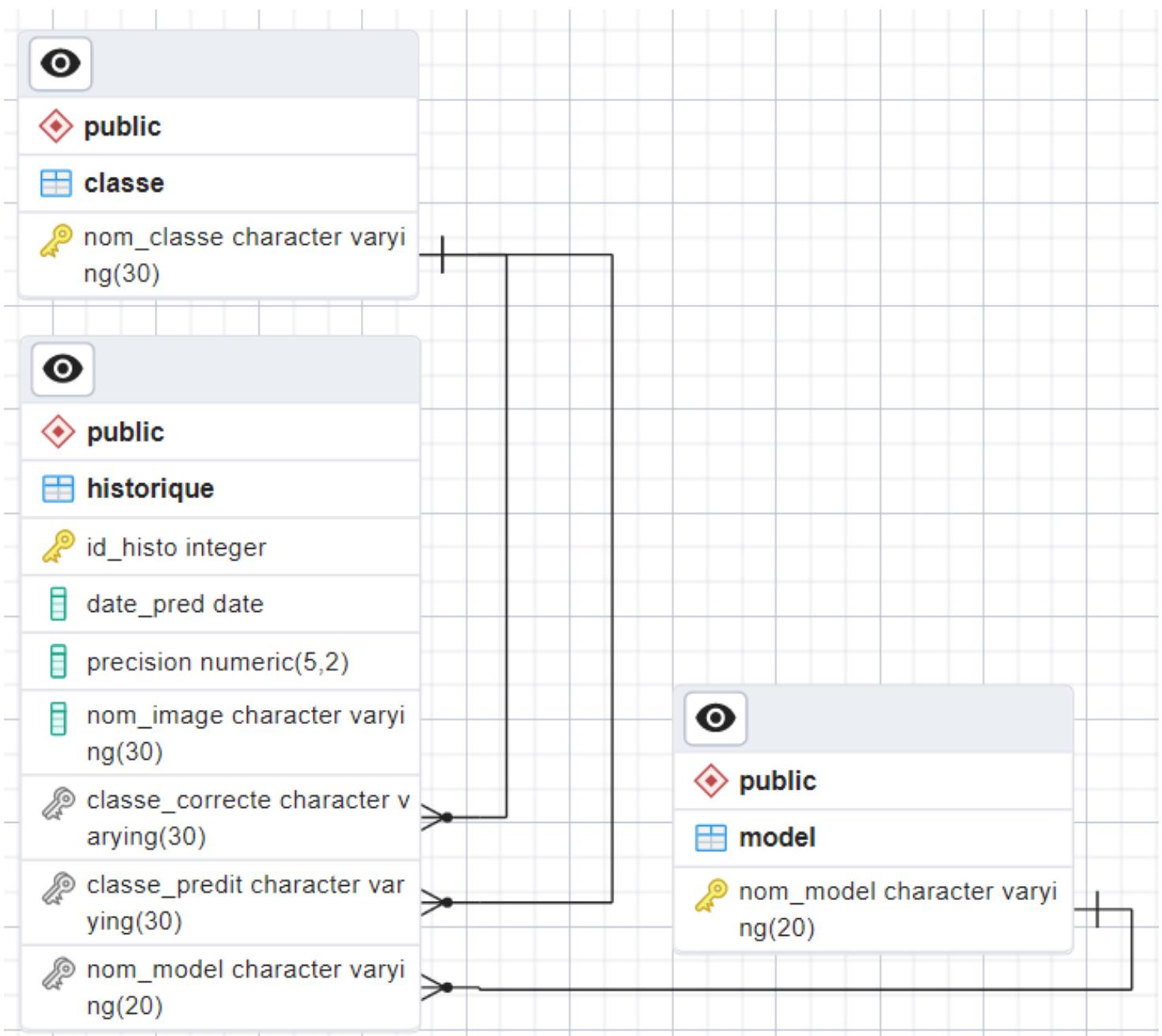


## Dictionnaire de données:

Attributs	Types de données	Valeur Null	Définition	Entité
<u>nom_classe</u>	varchar(30)		nom de classe	classe
<u>id_histo</u>	integer		numéro identifiant historique	historique
<u>nom_model</u>	varchar(20)		nom du modèle	model
date	date		date de la classification	historique
précision	decimal(5,2)		précision de la prédiction	historique
nom_image	varchar(30)		Nom de l'image enregistrée	historique

## MCD/MPD:

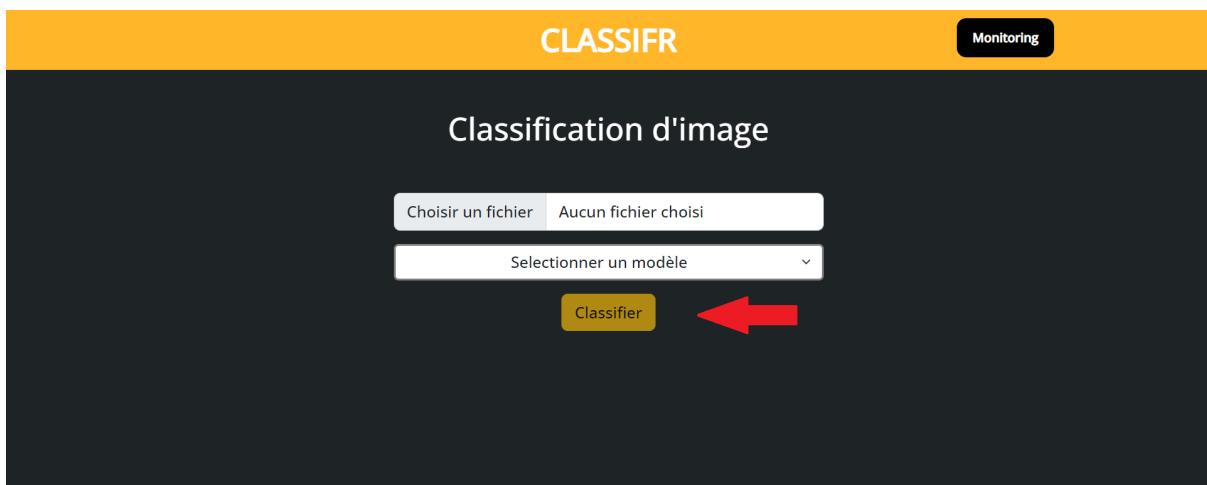




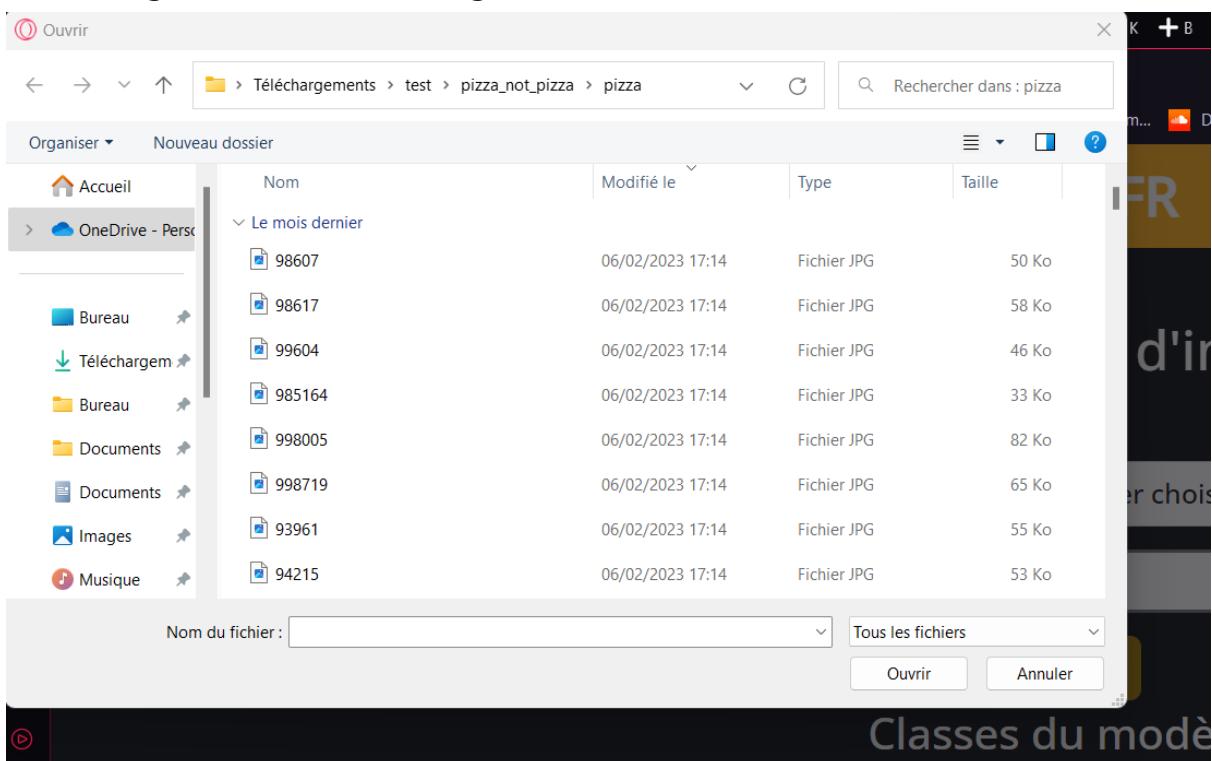
## D) Détails des fonctionnalités

### 1) Upload:

La page d'accueil renvoie directement l'utilisateur à la classification d'image. Celui-ci a accès à un sélecteur de fichier, ainsi qu'à une liste déroulante des modèles présents dans l'application. Tant qu'une image & qu'un modèle ne sont pas sélectionnés, la classification n'est pas accessible.



Le choix d'un fichier ouvre une fenêtre contextuelle qui redirige l'utilisateur vers ses dossiers locaux afin de lui permettre le téléchargement d'une image.



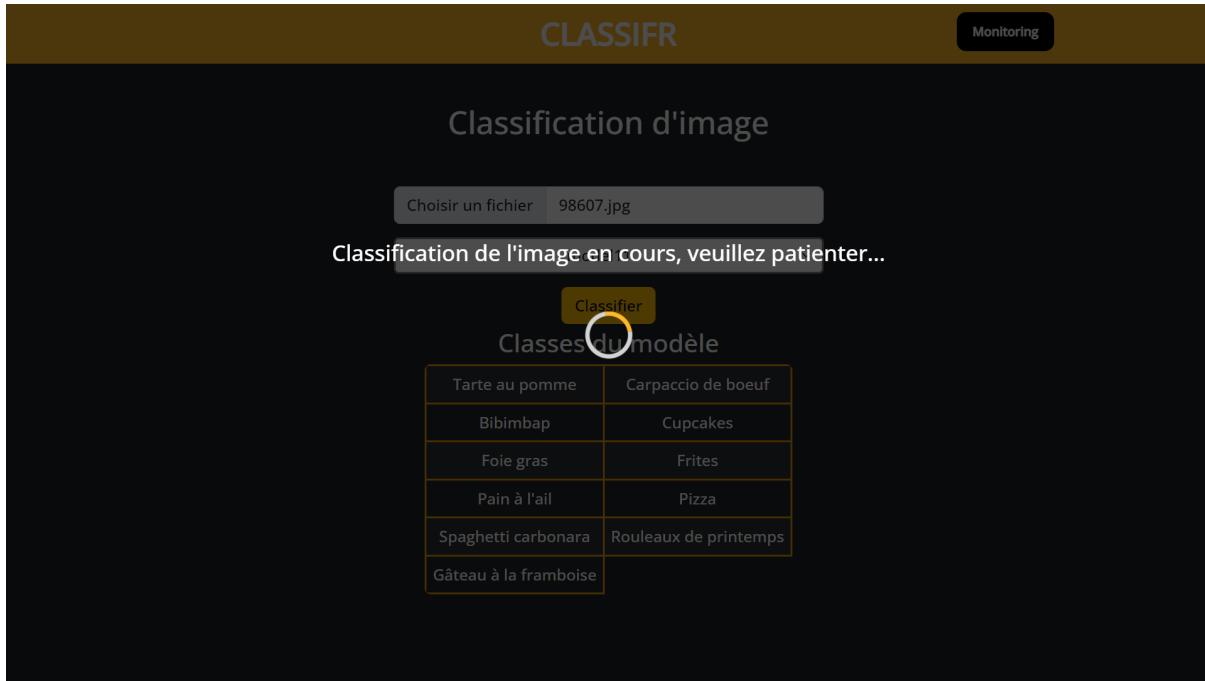
La liste déroulante ‘Sélectionner un modèle’ donne accès aux différents modèles présents dans l’application.



Une fois le modèle sélectionné, une liste des classes présentes dans le modèle apparaît alors. Lorsque l’image & le modèle sont sélectionnés, la classification de l’image est alors disponible.



Enfin, une fois la classification lancée, un écran de chargement apparaît durant la prédiction de la classe de l’image par le modèle.



## **2) Résultat:**

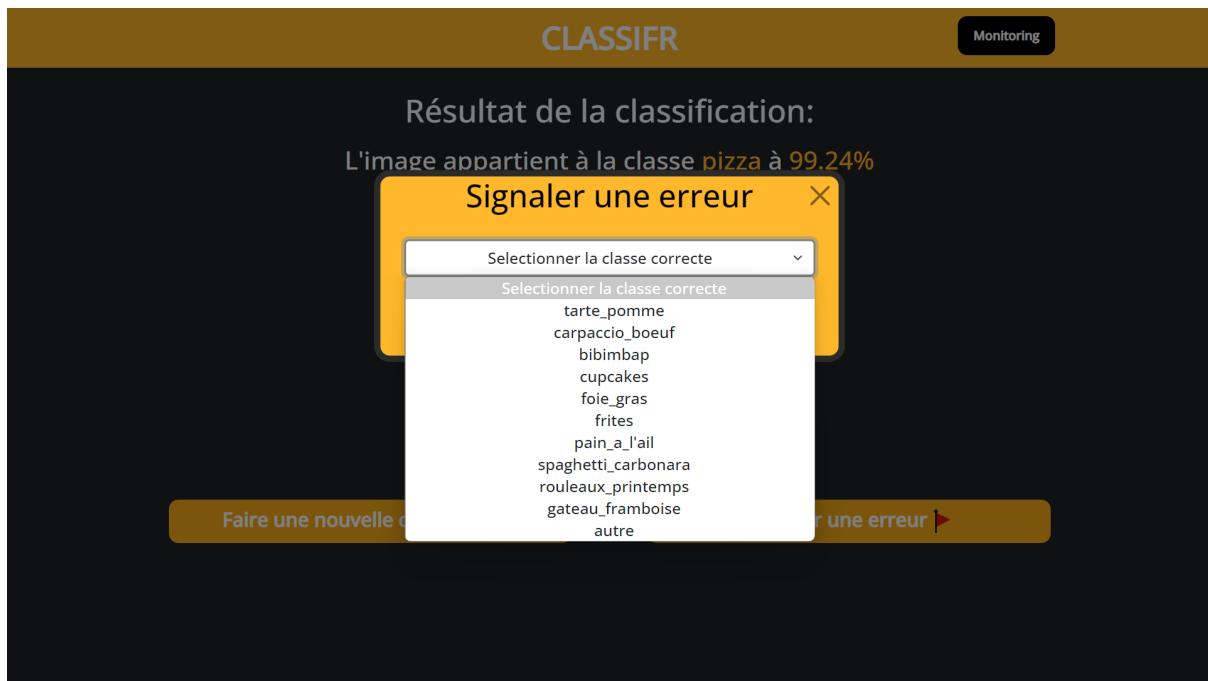
Arrivé sur la page de résultat, en plus d'un retour de l'image importée par l'utilisateur, ce dernier a accès à la classe prédite et au score de précision obtenu par le modèle. Il a également la possibilité de réaliser une nouvelle prédition, le réorientant vers la page de classification standard.



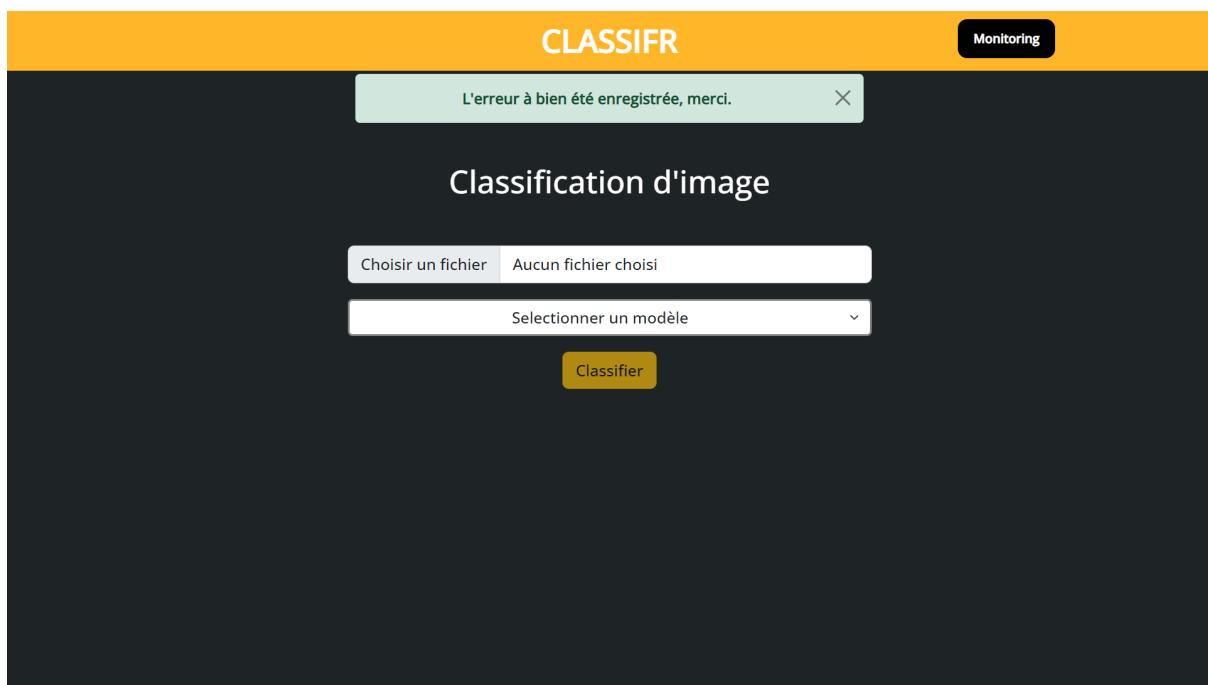
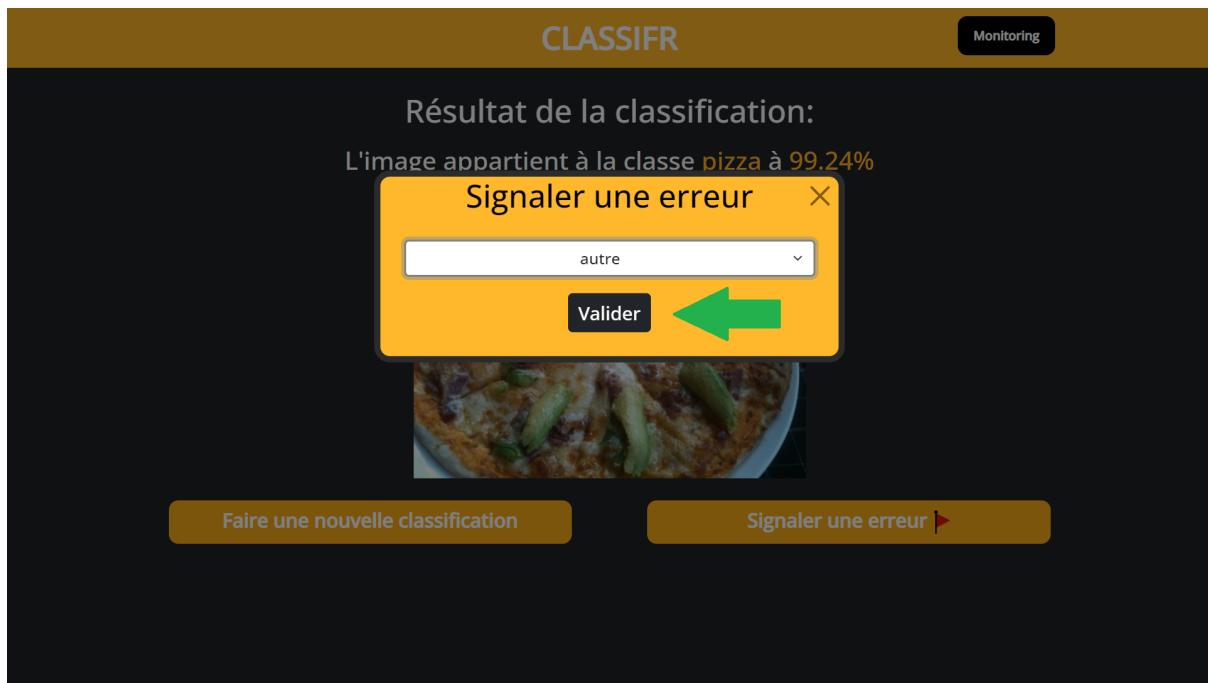
Lorsque la classe prédite par le modèle ne correspond pas à l'utilisateur, ce dernier a la possibilité d'utiliser l'option 'Signaler une erreur', qui va alors ouvrir un modal contenant une liste déroulante des autres classes présentes dans le modèle afin de proposer une correction.



Si aucun des choix des classes du modèle ne lui convient, il a alors la possibilité de sélectionner la classe 'autre'.



Une fois la bonne classe sélectionnée, il est possible de valider la correction, et celle-ci sera alors enregistrée en base de données.



### 3) API avec Django REST Framework:

Toute la partie monitoring de l'application va utiliser des endpoint API donc pour commencer on va mettre en place un serializer pour pouvoir convertir notre modèle Historique en format json.

Les serializers permettent de convertir des données complexes telles que des ensembles de requêtes et des instances de modèle en types de données Python natifs qui peuvent ensuite être facilement rendus en JSON, XML ou d'autres types de contenu.

```
from rest_framework import serializers
from classiffr.models import Historique

class HistoSerializer(serializers.ModelSerializer):
    class Meta:
        model=Historique
        fields='__all__'
```

On importe serializers depuis le module rest\_framework, puis on définit une classe que l'on appelle HistoSerializer qui va hériter de la classe ModelSerializer, on précise ensuite le nom du modèle et les champs que l'on veut sérialiser.

Dans notre views on va pouvoir importer le HistoSerializer que l'on a créé, on importe également la classe response et le décorateur api\_view.

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from classiffr.models import Classe,Model,Historique
from .serializers import HistoSerializer
from django.db import connection

@api_view(['GET'])
def getHisto(request):
    null=request.query_params.get('null',None)
    histo=Historique.objects.all()
    if null == "false":
        histo=Historique.objects.exclude(classe_correcte=None)
    serializer=HistoSerializer(histo, many=True)
    return Response(serializer.data)
```

### getHisto:

La première fonction view que l'on va créer permet de faire une requête afin de récupérer tout l'historique avec possibilité de filtrer à partir de l'url pour récupérer que les mauvaises classifications.

Le décorateur `api_view` permet de spécifier la/les méthode(s) autorisée(s) et dans notre cas ce sera GET.

`null` permet de récupérer la valeur du paramètre “`null`” depuis l'url de la requête et par défaut la valeur sera `None`.

On récupère ensuite toutes les instances du modèle Historique dans la variable `histo`.

Si `false` a été envoyé depuis le paramètre `null` de l'url, on change `histo` pour seulement récupérer les mauvaises classifications.

On crée ensuite une instance `serializer` avec la class `HistoSerializer` avec comme argument le queryset `histo` et `many=True` qui signifie qu'on sérialise tous les objets.

Enfin les données serialisées sont retournées en une réponse json.

Pour finir on définit l'url de l'endpoint.

```
urlpatterns = [
    path('api/histo/', views.getHisto, name="histo"),
```

### updateHisto:

Pour le deuxième endpoint, on veut pouvoir modifier une instance du modèle Historique à partir d'un ID.

```
@api_view(['POST'])
def updateHisto(request,pk):
    histo=Historique.objects.get(id_histo=pk)
    serializer=HistoSerializer(instance=histo,data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data)
```

Cette fois, vu qu'on envoie des données on va préciser que la fonction view n'accepte que les requêtes POST.

On ajoute un paramètre `pk` à la fonction qui va correspondre à la clé primaire d'un objet du modèle Historique que l'on veut mettre à jour. On retrouve l'objet avec comme clé primaire la valeur du paramètre `pk` qu'on assigne à une variable `histo`.

On crée ensuite une nouvelle instance serializer avec la classe HistoSerializer avec comme argument instance=histo et data=request.data, ce qui va permettre de déserialiser les données envoyées qui sont au format json de base vers un objet modèle qui peut être enregistré dans la base de données.

Pour finir on vérifie si les données déserialisées qui ont été reçues sont valides puis on sauvegarde l'objet et on retourne les nouvelles données dans la réponse.

Pour la définition de l'url on ajoutera donc le paramètre pk.

```
path('api/updatehisto/<str:pk>', views.updateHisto, name="histoUpdate"),
```

### **deleteHisto:**

Pour cet endpoint on veut pouvoir supprimer une instance du modèle Historique à partir d'un ID.

```
@api_view(['DELETE'])
def deleteHisto(request,pk):
    histo=Historique.objects.get(id_histo=pk)
    histo.delete()
    return Response("Ligne supprimée")
```

On précise dans le décorateur que cette fonction view n'accepte que les requêtes DELETE.

Comme pour l'update, on ajoute pk en paramètre de la fonction pour pouvoir récupérer la clé primaire de l'objet Historique que l'on veut supprimer.

On récupère l'objet Historique qui correspond à la valeur de pk que l'on l'assigne à la variable histo puis on supprime l'instance.

Enfin on retourne une response avec un message de confirmation.

Pour l'url, on ajoute aussi le paramètre pk.

```
path('api/deletehisto/<str:pk>', views.deleteHisto, name="histoDelete"),
```

## createHisto et getHistoDet:

```
@api_view(['GET'])
def getHistoDet(request,pk):
    histo=Historique.objects.get(id_histo=pk)
    serializer=HistoSerializer(histo, many=False)
    return Response(serializer.data)

@api_view(['POST'])
def createHisto(request):
    serializer=HistoSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data)
```

createHisto permet de créer des nouvelles instances dans le modèle Historique et getHistoDet permet de récupérer qu'un objet du modèle en fonction du paramètre pk donné.

On ne va se servir des ces endpoints pour l'instant mais voici leur urls.

```
path('api/histo/<str:pk>', views.getHistoDet, name="histoDet"),
path('api/createhisto/', views.createHisto, name="histoCreate"),
```

## stats:

Cet endpoint nous sera utile pour remplir un tableau dans l'onglet stats de la page de monitoring.

On ne va pas passer par l'ORM de django mais plutôt utiliser connection.cursor() on a donc pas besoin de serializer, on va juste envoyer un dictionnaire comme response.

```
from django.db import connection
```

```
@api_view(['GET'])
def getstats(request):
    result = [getModelStats('model3'),getModelStats('model11')]
    return Response(result)
```

#### **4)Identification:**

Lorsqu'un admin va vouloir observer les performances des modèles, il va pouvoir cliquer sur un bouton Monitoring en haut à droite de la page et sera redirigé vers une page de login pour s'authentifier afin d'accéder à la page de monitoring.



Il s'agit d'un simple formulaire qui va être envoyé vers le back end afin de vérifier si l'identifiant et le mot de passe correspondent à ceux enregistrés dans la base de données dans la table auth\_user générée par Django.

```
def loginq(request):
    if request.method == "POST":
        username = request.POST['identifiant']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('historique')
        else:
            messages.error(request, ("Mauvais identifiant ou mot de passe, veuillez ressayer."))
            return redirect('login')
    else:
        return render(request, 'login.html')

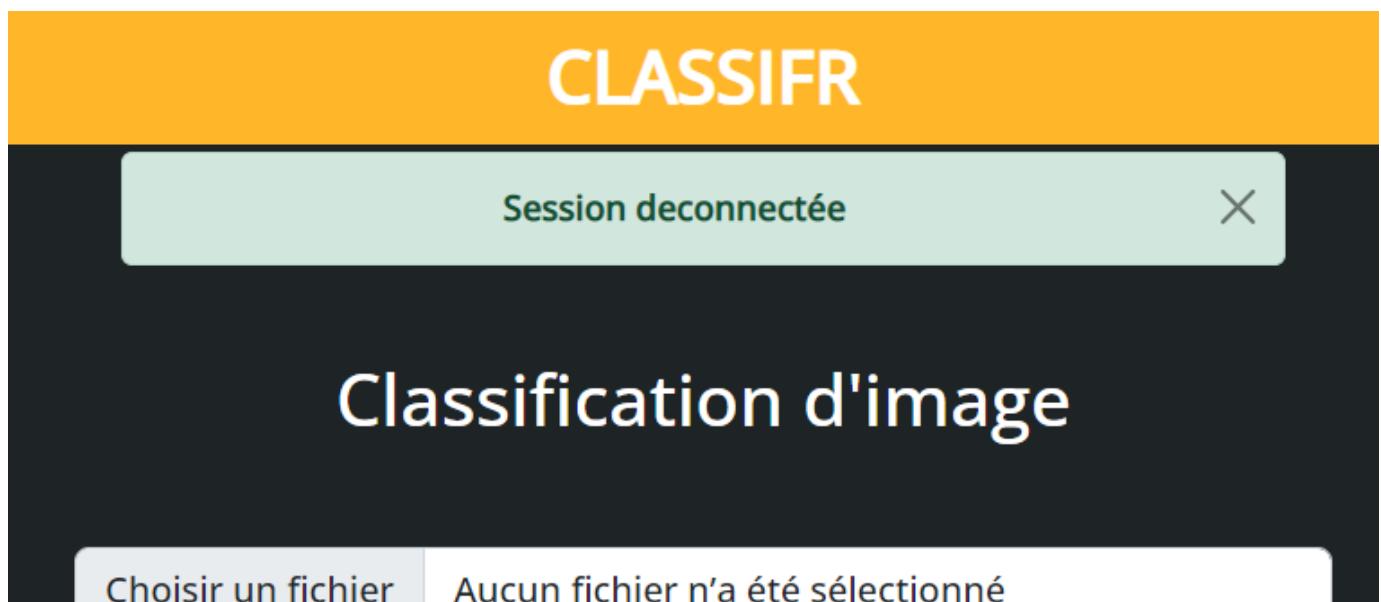
def logoutq(request):
    logout(request)
    messages.success(request, ("Session déconnectée"))
    return redirect('upload')
```

Dans le cas où il y a une erreur, un message apparaîtra en haut de la page.



Et si tout est correct, l'admin sera redirigé vers la page de monitoring.

A tout moment l'admin peut se déconnecter en cliquant sur le bouton déconnexion et sera redirigé vers la page d'accueil publique avec un message de confirmation.



## 5)Monitoring:

Sur la page de monitoring, il y a 3 différents onglets qui affichent différentes pages.

**CLASSIFR**

Deconnexion

Historique ⏲ Erreurs 🚨 Stats 💡

Afficher 10 lignes Filtrer:

ID	Date	Image	Résultat	Précision	Classe correcte	Modèle utilisé	Actions
131	2023-03-08		bibimbap	99.84%	✓	model11	<button>Editer</button> <button>X</button>
130	2023-03-08		gateau_framboise	81.86%	✓	model11	<button>Editer</button> <button>X</button>
129	2023-03-08		gateau_framboise	99.80%	✓	model11	<button>Editer</button> <button>X</button>
128	2023-03-08		carpaccio_boeuf	83.67%	✓	model11	<button>Editer</button> <button>X</button>

Affichage de 10 lignes sur 119 Précédent 1 2 3 4 5 ... 12 Suivant

## Historique:

Pour commencer, l'onglet par défaut est l'onglet “Historique” qui affiche du plus récent au plus ancien toutes les classification effectuées dans un tableau construit avec l'aide du plugin DataTables.

Sur chaque ligne du tableau il y a 2 boutons, le bouton “Editer” permet d'afficher un modal avec un formulaire qui permet de modifier la colonne “Classe correcte” qui signifie qu'une classification est bonne ou mauvaise avec comme information la classe correcte.

**CLASSIFR**

Historique ⏲ Erreurs 🚨 Stats 💡

Afficher 10 lignes Filtrer:

ID	Date	Image	Résultat	Précision	Classe correcte	Modèle utilisé	Actions
131	2023-03-08		Modification	Cupcakes	Valider	model11	<button>Editer</button> <button>X</button>
130	2023-03-08		gateau_framboise	99.80%	✓	model11	<button>Editer</button> <button>X</button>
129	2023-03-08		gateau_framboise	99.80%	✓	model11	<button>Editer</button> <button>X</button>
128	2023-03-08		carpaccio_boeuf	83.67%	✓	model11	<button>Editer</button> <button>X</button>

Affichage de 10 lignes sur 119 Précédent 1 2 3 4 5 ... 12 Suivant

En fonction du modèle utilisé, les options du formulaire s'adaptent et une fois le formulaire validé l'historique est mis à jour.  
Le deuxième bouton permet tout simplement de supprimer toute une ligne de l'historique après avoir cliqué sur un bouton de confirmation sur un modal.

# CLASSIFR

Deconnexion

**Historique** **Erreurs** **Stats**

Afficher 10 lignes Filtrer:

ID	Date	Image	Résultat	Précision	Classe correcte	Modèle utilisé	Actions
131	2023-03-08		Voulez-vous vraiment supprimer X cette ligne ?				<button>Editer</button> <button>X</button>
130	2023-03-08						<button>Editer</button> <button>X</button>
129	2023-03-08		gateau_framboise	99.80%	✓	model11	<button>Editer</button> <button>X</button>
128	2023-03-08		carpaccio_boeuf	83.67%	✓	model11	<button>Editer</button> <button>X</button>

Affichage de 10 lignes sur 111 Précédent 1 2 3 4 5 ... 12 Suivant

## Erreurs:

# CLASSIFR

Deconnexion

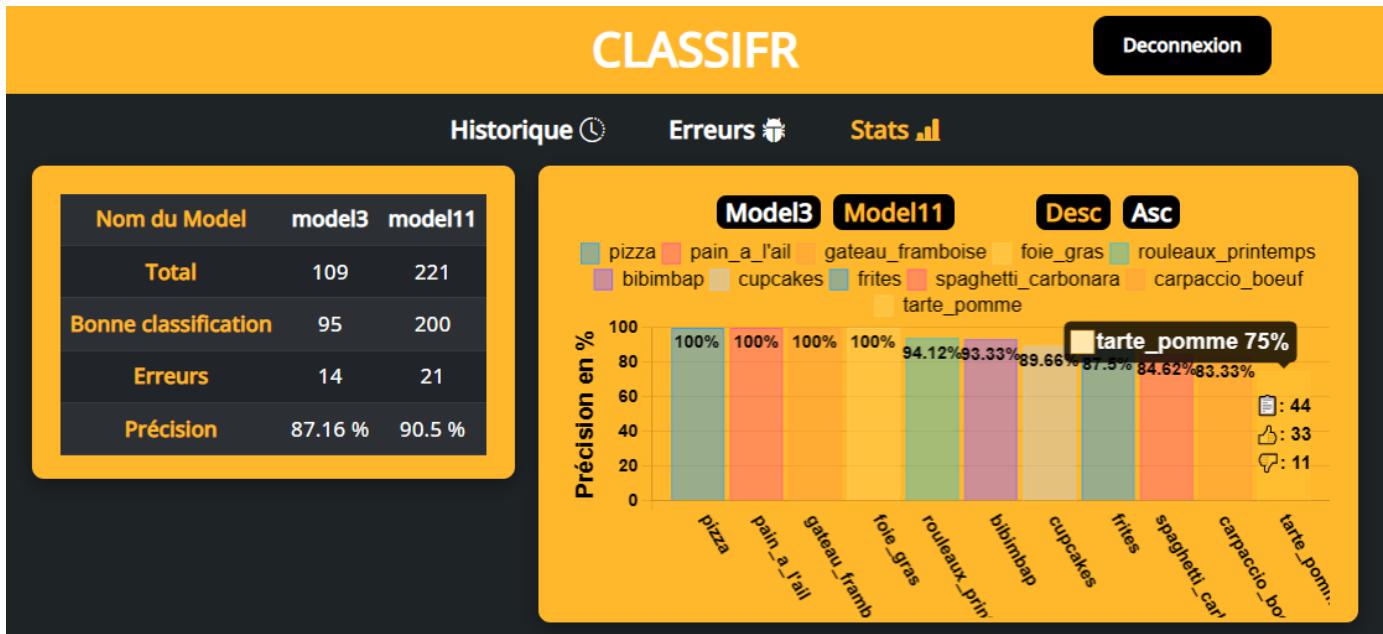
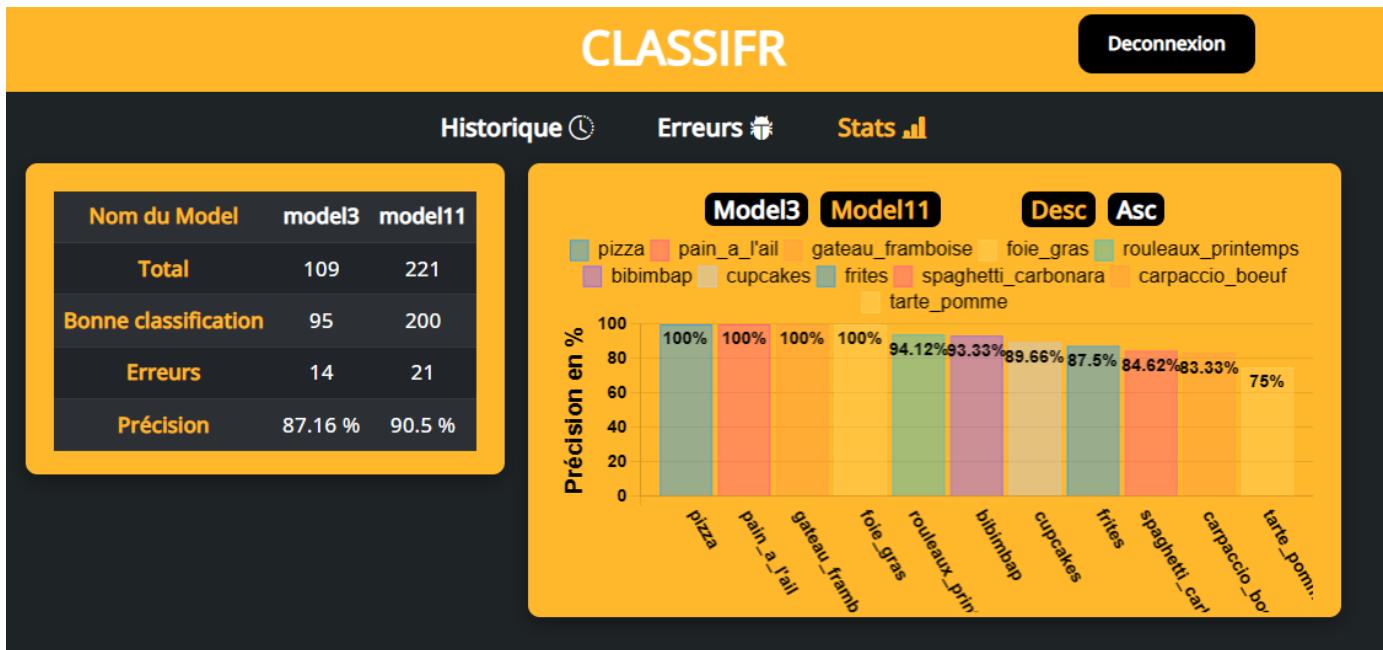
**Historique** **Erreurs** **Stats**

Afficher 10 lignes Filtrer:

ID	Date	Image	Résultat	Précision	Classe correcte	Modèle utilisé	Actions
131	2023-03-08		bibimbap	99.84%	cupcakes	model11	<button>Editer</button> <button>X</button>
120	2023-03-08		pizza	99.90%	cupcakes	model11	<button>Editer</button> <button>X</button>
53	2023-03-07		omelette	94.91%	pizza	model3	<button>Editer</button> <button>X</button>
45	2023-03-07		rouleaux_printemps	97.26%	tarte_pomme	model11	<button>Editer</button> <button>X</button>

Affichage de 9 lignes sur 9 Précédent 1 Suivant

## Stats:



## E) Conclusion

### 1) Ce qui a été fait:

- Mise en place de la base de données de PostgreSQL
- Authentification & contrôle d'accès
- Upload simple avec sélection de modèle & prédiction
- Stockage des images post-upload
- Historique des uploads
- Monitoring & suivi des statistiques de prédictions des modèles

### 2) Ce qui est à faire:

#### Classification avec plusieurs images:

L'import et la prédiction de plusieurs images peut être réalisé, cependant le signalement de l'erreur ne fonctionne que pour la dernière image.



Une solution alternative actuelle pour signalement d'erreur consiste à modifier les classes directement depuis l'historique des uploads.

The screenshot shows a web-based application titled "CLASSIFR". At the top right is a "Deconnexion" button. Below the title are three navigation links: "Historique" (with a clock icon), "Erreurs" (with a red exclamation mark icon), and "Stats" (with a bar chart icon). A search bar labeled "Filtrer:" is positioned at the top right of the main content area.

The main content is a table with the following columns: ID, Date, Image, Résultat, Précision, Classe correcte, Modèle utilisé, and Actions. The table contains four rows of data:

ID	Date	Image	Résultat	Précision	Classe correcte	Modèle utilisé	Actions
54	2023-03-10		tarte_pomme	97.32%	✓	model11	<button>Editer</button> <button>X</button>
53	2023-03-10		bibimbap	97.93%	✓	model11	<button>Editer</button> <button>X</button>
52	2023-03-10		pizza	99.61%	✓	model11	<button>Editer</button> <button>X</button>
51	2023-03-10		pain_a_l'ail	56.46%	✓	model11	<button>Editer</button> <button>X</button>

Below the table, a message says "Affichage de 10 lignes sur 54". At the bottom right are page navigation buttons: "Précédent", a yellow "1", and numbers 2, 3, 4, 5, 6, followed by "Suivant".

### **Développement de la partie admin:**

Cela relèverait de l'intégration de nouveaux modèles au sein de l'application, ainsi que l'update des modèles présents avec possibilités de réentrainement à partir des images récoltés durant les uploads des utilisateurs, ou encore l'ajout de nouvelles classes au sein de ses modèles.

### **Développement de la partie admin:**

Cela relèverait de l'intégration de nouveaux modèles au sein de l'application, ainsi que l'update des modèles présents avec possibilités de réentrainement à partir des images récoltés durant les uploads des utilisateurs, ou encore l'ajout de nouvelles classes au sein de ses modèles.

### **Réorientation architecture RESTful:**

Retravailler l'application dans le but de lui donner une architecture RESTful.