

# TMA POM

## Dossier d'Architecture Technique

***Réf. POM3-CS-2025-DAT-POM-020***

	Nom	Société	Fonction	Date	Visa
Rédigé par :	C. Mertz	CS Group	Chef de projet	06/03/2025	
Validé par :	I. Besson	CS Group	Skill Manager	18/03/2025	
Pour application :	E. Le Pape	Service Central Vigicrues	Chef de projet	20/05/2025	

**CS GROUP**  
**6 rue Brindejonc des Moulinais**  
**Parc de la Grande Plaine**  
**BP 15872**  
**31506 Toulouse Cedex 5**

ED.	RÉV.	DATE	MOTIF
01	00	06/03/25	Création du document

# Sommaire

## Table des matières

- 1. Architecture générale.....5
  - 1.1 Vue d'ensemble de l'architecture.....5
  - 1.2 Interaction entre composants.....5
  - 1.3 Documents applicables.....5
  - 1.4 Glossaire.....6
- 2. Architecture applicative.....7
  - 2.1 Backend Symfony.....7
    - 2.1.1 Interactions entre le backend et les IHMs.....7
    - 2.1.2 Composants techniques.....8
    - 2.1.3 Modularité.....8
    - 2.1.4 Gestion des rôles et sécurité.....9
    - 2.1.5 Structure du projet.....10
    - 2.1.6 Mécanisme de logs.....12
  - 2.2 IHM POM complète.....12
    - 2.2.1 Composants techniques.....12
  - 2.3 IHM de pilotage des modèles.....13
    - 2.3.1 Composants techniques.....13
    - 2.3.2 Structure du projet.....14
  - 2.4 Base de données PostgreSQL.....14
    - 2.4.1 Interactions entre le backend Symfony et la base de données.....14
    - 2.4.2 Composants techniques.....15
  - 2.5 Programmes d'interface.....15
    - 2.5.1 Interaction entre la POM et les serveurs de calcul.....15
    - 2.5.2 Composants techniques.....16
    - 2.5.3 Structure du projet.....16
    - 2.5.4 Structure applicative de la librairie PomInterface.....17
    - 2.5.5 Structure applicative des modèles analyse/prévision.....18
    - 2.5.6 Mécanisme de logs.....18
  - 2.6 Intégration avec d'autres systèmes.....19

## Liste des tableaux

- Tableau 1 : Documents applicables.....6
- Tableau 2 : Glossaire.....6

## Liste des figures

- Figure 1: Architecture applicative : affichages des IHMs.....7
- Figure 2: Architecture applicative : échanges avec la base de données.....15

Figure 3: Architecture applicative : interaction entre la POM et les PIx.....16

Figure 4: Librairie PomInterface : processors génériques.....18

Figure 5: Librairie PomInterface : Model analyse / prévision.....18

Figure 6: Architecture applicative : échanges avec la PHYC et la BDImage.....20

# 1. Architecture générale

## 1.1 Vue d'ensemble de l'architecture

L'architecture de l'application repose sur un backend Symfony servant deux interfaces distinctes. Le backend, développé avec Symfony, agit comme une couche centrale qui gère la logique métier, expose des API REST, et rend des templates Twig.

L'architecture est modulaire et suit une approche en couches :

- ✓ **Backend :**
  - ↳ Symfony pour la logique métier, les API et la gestion des données
  - ↳ Sert deux types de frontend :
    - **Frontend IHM POM classique** : rendu côté serveur avec des templates Twig
    - **Frontend IHM de pilotage des modèles** : rendu indirectement via un template Twig qui charge le fichier **main.js** généré par React (application SPA, Single Page Application)
- ✓ **Base de données** : PostgreSQL pour le stockage relationnel des données
- ✓ **Programmes d'interface** : librairie Python permettant de lancer les modèles hydrologiques utilisés (GRP, Mascaret/Telemac, Plathynes), voir des modèles plus spécifiques développés par certains SPC.

## 1.2 Interaction entre composants

La nature des interactions entre les composants sont :

- ✓ **Symfony <==> Twig** : rendu via TwigBundle.
- ✓ **Symfony <==> React** : API REST
- ✓ **Symfony <==> PostgreSQL** : Doctrine ORM
- ✓ **POM <==> Serveurs de calcul** : SSH pour échanges de fichiers et lancement de calculs

La POM interagit avec trois composants externes :

- ✓ **POM <==> PHYC** : web-services SOAP et API REST
- ✓ **POM <==> BDImage** : web-services
- ✓ **POM <==> Superviseur National** : HTTP

Ces interactions sont décrites plus en détails dans la suite du document.

## 1.3 Documents applicables

Document	Date	Version
[SPEC-POM] Spécifications Fonctionnelles Détaillées ref ; POM3-CS-2021-DS-POM-009	23/09/2024	3.4-00

[MIEX-POM] Manuel d'installation et d'exploitation POM-MIEX ref : POM3-CS-2021-MIEX-POM-010	28/11/2023	3.0-05

**Tableau 1 : Documents applicables**

## 1.4 Glossaire

Acronyme	Signification
MVC	Modèle-Vue-Contrôleur
ORM	Object-relational Mapping
PDO	PHP Data Object
SPA	Simple page Application
SQL	Structured Query Language
SSO	Single Sign-On

**Tableau 2 : Glossaire**

## 2. Architecture applicative

### 2.1 Backend Symfony

Il s'agit du composant principal de la POM.

Il intègre la logique métier.

Il gère l'authentification des utilisateurs, les interfaces avec les composants externes, la gestion des routes et le rendu de l'IHM POM complète, le lancement des calculs de l'ensemble des modèles, en mode temps-réel et temps différé et il expose un ensemble d'API REST pour l'IHM de pilotage des modèles.

Deux frontends coexistent :

- ✓ **IHM POM complète** : il s'agit d'une interface traditionnelle **rendue côté serveur** par Symfony à l'aide de templates Twig. Les pages sont générées dynamiquement et servies directement aux utilisateurs via des requêtes HTTP classiques.
- ✓ **IHM de pilotage des modèles, « prévi » ou « prévisionniste »** : il s'agit d'une interface moderne de type SPA construite avec **React**. Ce frontend est intégré au backend Symfony via un template Twig spécifique qui charge le fichier **main.js** généré par la compilation de React. Une fois chargé dans le navigateur, React prend le contrôle du **rendu côté client** et interagit avec le backend via des appels API REST pour les données dynamiques.

#### 2.1.1 Interactions entre le backend et les IHMs

Le backend Symfony joue un rôle dual : il rend directement l'IHM (Twig) et initialise l'IHM React via Twig, tout en fournissant les endpoints API nécessaires au fonctionnement de React. Les utilisateurs accèdent aux deux IHMs via un navigateur web, chaque IHM offrant une expérience distincte mais intégrée au même système.

Le schéma suivant illustre la prise en charge des deux IHMs par le backend Symfony.

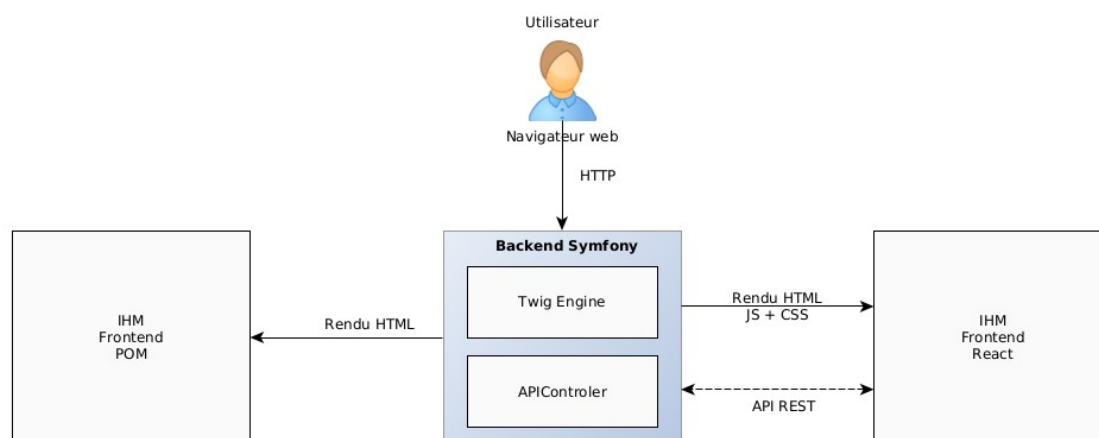


Figure 1: Architecture applicative : affichages des IHMs

Les requêtes HTTP des utilisateurs sont prises en charge par le backend Symfony, qui agit comme point d'entrée unique. Lorsqu'un utilisateur accède à l'application via un navigateur, une requête HTTP est envoyée à Symfony. Pour l'IHM POM complète, le moteur Twig génère un rendu HTML côté serveur, directement renvoyé au navigateur. Pour l'IHM React, Symfony utilise un template Twig spécifique pour livrer une page HTML incluant le fichier **main.js**, généré

par la compilation de React. Une fois chargé dans le navigateur, React prend le contrôle et effectue des appels API REST vers Symfony pour les données dynamiques.

Ce schéma d'architecture est progressivement complété dans la suite du document lors de la description des composants de l'application.

## 2.1.2 Composants techniques

- ✓ **Framework : Symfony 5.4.0 (évolution de Symfony 2.0)**
- ✓ **PHP : 7.4 (évolution de 5.3.3)**
- ✓ **Dépendances**
  - ↳ **Doctrine ORM** : doctrine/orm:2.13.3
  - ↳ **Monolog** : monolog/monolog:2.8.0

La liste des dépendances sont définies dans le fichier **composer.json**

## 2.1.3 Modularité

Le système repose sur le framework Symfony qui autorise un développement par modules « bundles ». Ces modules peuvent être cloisonnés ou cohabiter en interaction faible. Dans le cas présent, l'architecture repose sur la mise en place des « bundles » suivants :

- ✓ **Bundles**
  - ↳ **Symfony** : FrameworkBundle, SecurityBundle, TwigBundle, MonologBundle, DoctrineBundle.
  - ↳ **Personnalisés** :
    - **CommonBundle** : prend en charge certaines fonctions communes aux autres « bundles », avec notamment la sécurité (**PomAuthenticator**), les rôles, les droits, le journal des événements, les favoris, ...
    - **ModelBundle** : prend en charge tous les aspects de la modélisation
    - **PrevisionBundle** : prend en charge le calcul des prévisions
    - **AdminBundle** : prend en charge toutes les fonctions d'administration
    - **WSBundle** : gère les accès aux web-services utilisés par la POM (PHYC, BDImage)
    - **MockBundle** : permet de gérer des modèles bouchon ou naïf
    - **DoctrineBundle** : gère l'extension de Doctrine<sup>1</sup>
- ✓ **Configuration** :
  - ↳ **parameters.ini** : paramètres dynamiques
- ✓ **Structure générale** :
  - ↳ Un ensemble de contrôleurs pour les routes Twig et les endpoints API
  - ↳ Des entités mappées avec Doctrine pour la base de données
  - ↳ Un ensemble de services pour encapsuler la logique métier

La POM exploite des composants réalisés pour des problématiques spécifiques :

- ✓ **PomData** : composant de manipulation de données
- ✓ **PomTools** : ensemble de classes utilitaires de manipulation d'objets php (dates, string, commandes, XML, ...)

<sup>1</sup> C'est un patch de Doctrine. Dans l'idéal, il serait préférable de s'en passer pour éviter d'avoir à le mettre à jour lors des montées de version, mais jusqu'à présent nous avons fait le choix de le conserver.

- ✓ **PomXML** : composants de gestion des fichiers XML, organisé par format :
  - ↳ **XMLBdImage** : composant de lecture / écriture de fichiers au format XML BD Image
  - ↳ **XMLExchange** : composant de lecture / écriture de fichiers au format XML d'échange POM
  - ↳ **XMLParameters** : composant de lecture / écriture de fichiers au format XML POM de paramétrage du modèle
  - ↳ **XMLProgression** : composant de lecture / écriture de fichiers au format XML POM de progression du modèle
  - ↳ **XMLSandre** : composant de lecture / écriture de fichiers au format XML Sandre.

### 2.1.4 Gestion des rôles et sécurité

Deux modes d'authentification sont disponibles :

- ✓ **Cerbere** : SSO externe avec ticket via l'API PHYC
- ✓ **PHYC** : authentification via web-service PHYC (WsBdhService)
- ✓ Le mode **PHYC** est à utiliser uniquement en cas de secours si le mode **Cerbere** n'est pas disponible. C'est à l'utilisateur de basculer d'un mode à l'autre. La bascule d'un mode à l'autre ne vaut pour cet utilisateur et depuis son navigateur (pas d'impact de son choix sur les autres utilisateurs).

La sécurisation est configurée via le fichier **app/config/security.yml** de Symfony et réalisée via la classe **PomAuthenticator** :

- ✓ **Dépendances** :
  - ↳ **EntityManager** : gestion des entités Userpom
  - ↳ **WsBdhService** : appels aux services web pour PHYC
  - ↳ **UserProviderInterface** : interface Symfony qui fournit la gestion des utilisateurs pour l'authentification
  - ↳ **TokenStorageInterface** : interface Symfony qui permet de stocker et gérer le jeton d'authentification
  - ↳ **RouterInterface** : interface Symfony qui permet de gérer les routages de l'application
- ✓ **Modes d'authentification**
  - ↳ **Cerbere (SSO)**
    - Utilise un ticket reçu via « ?ticket=. »
    - Vérifie le ticket via une API externe « **cerbere\_checkSession** » /api/validerticket?ticket=[ticket])
    - Récupère **idsession** et **cdcontact** (JSON)
    - Redirige vers Cerbere « **cerbere\_newSession** » /api/cerbere?service=[url] si pas de ticket
    - Cache en session avec expiration « **cerbere\_sessionCache** »
  - ↳ **PHYC (Login/Mot de passe)**
    - La page d'authentification est gérée via la route **/auth/login**
    - Reçoit les paramètres **auth\_login** et **auth\_password** via **/auth/login**
    - Appelle les web-services PHYC **Authentifier** ou **AuthentifierAlias** pour valider l'utilisateur
    - Stocke **idsession**, **cdcontact**, et **mot de passe** en session
    - Redirige vers **/auth/login** si le cache est expiré
  - ↳ **Logique commune**

- Charge l'utilisateur via **cdcontact** dans **Userpom**
- Définit un profil par défaut (PREV, MOD, RMOD, ADM) si absent
- Met à jour **lastconnexion**, **bdhsessionid**, et **pwd**

#### ↳ Succès d'authentification

- Appelle web-service PHYC **publierContactListe** pour récupérer **cdintervenant**.
- Met à jour l'entité **Userpom** et persiste via **EntityManager**

Des rôles sont utilisés par le système de sécurité pour gérer les autorisations d'accès des utilisateurs. La granularité des rôles permet un contrôle d'accès à des sections spécifiques et permet une flexibilité puisque cela peut se faire sans changer le code applicatif.

Les rôles standards Symfony sont utilisés :

- ✓ **ROLE\_USER** : utilisateur authentifié de base
- ✓ **ROLE\_ADMIN** : administrateur avec privilèges élevés

Des rôles spécifiques à la POM sont définis :

- ✓ **ROLE\_VIEW** : consultation des modèles
- ✓ **ROLE\_EDIT** : modification des modèles
- ✓ **ROLE\_VIEWPROG** : consultation des programmations
- ✓ **ROLE\_EDITCHAIN** : gestion des enchaînements de calcul
- ✓ **ROLE\_SERV** : administrations des serveurs/plateformes
- ✓ ...

cf [SPEC-POM] pour l'ensemble des rôles et la répartition par profil.

Les rôles sont répartis dans les profils suivants :

- ✓ **Prévisionnistes** (PREV)
- ✓ **Modélisateurs** (MOD)
- ✓ **Responsable de la modélisation** (RMOD)
- ✓ **Administrateurs** (ADM)

## 2.1.5 Structure du projet

Afin d'assurer une répartition claire des responsabilités de chaque « bundles », la structure suivante est utilisée :

- ✓ **Command (src/CS/PomXXXXXXBundle/Command/)** :
  - ↳ Gèrent les commandes console Symfony (ex : pom:lock, pom:unlock, ....)
  - ↳ Appellent les services appropriés
- ✓ **Controllers (src/CS/PomXXXXXXBundle/Controller/)** :
  - ↳ Gèrent les requêtes HTTP
  - ↳ Appellent les services appropriés
  - ↳ Retourne les réponses (vues Twig ou JSON pour les API)
- ✓ **Repositories (src/CS/PomXXXXXXBundle/DAO/)** :

- ↳ Gèrent les interactions avec la base de données via Doctrine
- ↳ Définissent des méthodes de récupération spécifiques (requêtes optimisées)
- ✓ **Entités (src/CS/PomXXXXXXBundle/Entity/) :**
  - ↳ Représentent les modèles de données et leurs relations
  - ↳ Utilisent les annotations Doctrine pour la gestion des colonnes et des contraintes
- ✓ **Formulaires (src/CS/PomXXXXXXBundle/Form/) :**
  - ↳ La définition des formulaires de l'IHM
- ✓ **Services (src/CS/PomXXXXXXBundle/Services/) :**
  - ↳ L'ensemble des services contiennent la logique métier
  - ↳ Encapsulent les traitements complexes
  - ↳ Injectés dans les contrôleurs via l'auto-wiring
- ✓ **Templates (src/CS/PomXXXXXXBundle/ressources/views) :**
  - ↳ Fichiers Twig utilisés pour le rendu côté serveur
  - ↳ Structurés par sections pour faciliter la réutilisation

```
pom-symfony-project/
|
├─ app/
|   ├── AppKernel.php
|   ├── bootstrap.php
|   └─ config/
|       ├── config.yml
|       ├── parameters.ini
|       ├── routing.yml
|       ├── security.yml
|       └─ vesions.ini
|   └─ Resources
|       └─ Migrations
├─ src/
|   └─ CS/
|       ├── PomXXXXXXBundle/
|       │   ├── Command/
|       │   ├── Contrroller/
|       │   ├── DAO/
|       │   ├── Entity/
|       │   ├── Form/
|       │   └─ Resources/
```

```

| | | | └─ config
| | | | └─ views
| | | └─ Services/
| composer.json

```

### 2.1.6 Mécanisme de logs

Un système de trace permet de suivre l'exécution à des fins d'analyse. Les logs produits sont structurés afin de diagnostiquer les erreurs. Les logs sont spécialisées par type d'opérations.

- ✓ **Trace** : utilitaire de trace en fichier à des fins d'analyse.
  - ↳ Avec des classes dérivées **TraceFtp**, **TraceSsh** et **TraceWs**
  - ↳ Séparation des traces par type d'opération dans des fichiers de logs spécifiques : **pom\_trace.log**, **pom\_trace\_ftp.log**, **pom\_trace\_ssh.log** et **pom\_trace\_ws.log**
  - ↳ Seules les traces FTP, SSH et WS sont activées par défaut.
  - ↳ L'activation des traces générales doit se faire manuellement, par exemple :

```
$trace = new Trace("Début runExecution");
```

## 2.2 IHM POM complète

Le frontend Twig définit un ensemble de composants d'interface réutilisables afin d'assurer l'homogénéité de l'application en utilisant l'héritage Twig pour centraliser la structure des pages et garantir une cohérence visuelle.

Il s'agit des composants suivants :

- ✓ Boutons
- ✓ Formulaires
- ✓ Types de champs standardisés
- ✓ Champs de saisie avec auto-complétion
- ✓ Messages d'alerte et notifications
- ✓ Tableaux et affichages structurés

L'organisation des fichiers dans chaque bundle est la suivante :

- **Default** : pages principales (index, préférences, licences)
- **Form** : gestion des formulaires et champs communs
- **Help** : aide en ligne et documentation utilisateur
- **Layout** : structure générale et messages d'information (erreurs, alertes, navigation)
- **Report** : rapports et affichage des derniers éléments
- **Security** : pages d'authentification et gestion des accès

## 2.2.1 Composants techniques

- ✓ **Technologie** : Twig 2.15.3.
- ✓ **Configuration** : Templates dans `src/CS/*/Resources/views/*`

## 2.3 IHM de pilotage des modèles

### 2.3.1 Composants techniques

- ✓ **Technologie**
  - **Framework:** React 18.2.0
  - **Langage:** TypeScript 5.2.2
  - **Build tool:** Vite 5.1.4
  - **Package manager:** pnpm 9.15.2
- ✓ **Pile logicielle**
  - ↳ **Bundler** : Vite 5.1.4, port 8080.
  - ↳ **Styling** : Tailwind CSS 3.4.1.
  - ↳ **Dépendances** : Redux, React Router, Radix UI, React Flow.
- ✓ **Sécurité** : Accès API via NelmioCorsBundle, authentification Cerbere/Phyc.

La liste des dépendances sont définies dans le fichier **package.json**

Dans l'objectif de simplifier les déploiements, l'IHM Frontend React de pilotage des modèles est déployée sur le même serveur Apache que le backend et l'IHM Frontend POM complète.

Pour cela :

- ✓ une route **api\_pilotage** spécifique est définie au niveau de **ApiController**
- ✓ un Twig `src/CS/PomPrevisionBundle/Resources/views/Prevision/frontend.html.twig` est définie pour appeler les 3 fichiers nécessaires à l'IHM
  - ↳ **main.js**
  - ↳ **main.css**
  - ↳ **react.js**
- ✓ 3 variables d'environnement sont transmises au front React
  - ↳ **user** : une structure JSON contenant des informations sur l'utilisateur connecté :

```
• {
  "id": 18,
  "cdcontact": "1545",
  "name": "John Do",
  "urllogout": "/app.php/auth/logout",
  "role": {
    "id": 1,
    "code": "ADM",
```

```
"name": "Administrateur"
}
}
```

🔗 **BASENAME** : URL de l'IHM de pilotage : **/app.php/pilotage**

🔗 **API\_URL** : URL de base de l'API dans le backend **/app\_dev.php/prevision/api/v1/**

### 2.3.2 Structure du projet

pom-symfony-project/

```
├─ frontend/
|   ├─ src/
|   |   ├─ api/
|   |   ├─ assets/
|   |   ├─ common/
|   |   ├─ config/
|   |   ├─ domain/
|   |   ├─ features/
|   |   ├─ helpers/
|   |   ├─ store/
|   |   ├─ utils/
|   |   └─ views/
|   ├─ node_modules/
|   └─ public/
|   components.json
|   package.json
|   tailwind.config.ts
|   tsconfig.json
|   vite.config.ts
```

## 2.4 Base de données PostgreSQL

### 2.4.1 Interactions entre le backend Symfony et la base de données

Le schéma suivant illustre les échanges entre le backend Symfony et la base PostgreSQL.

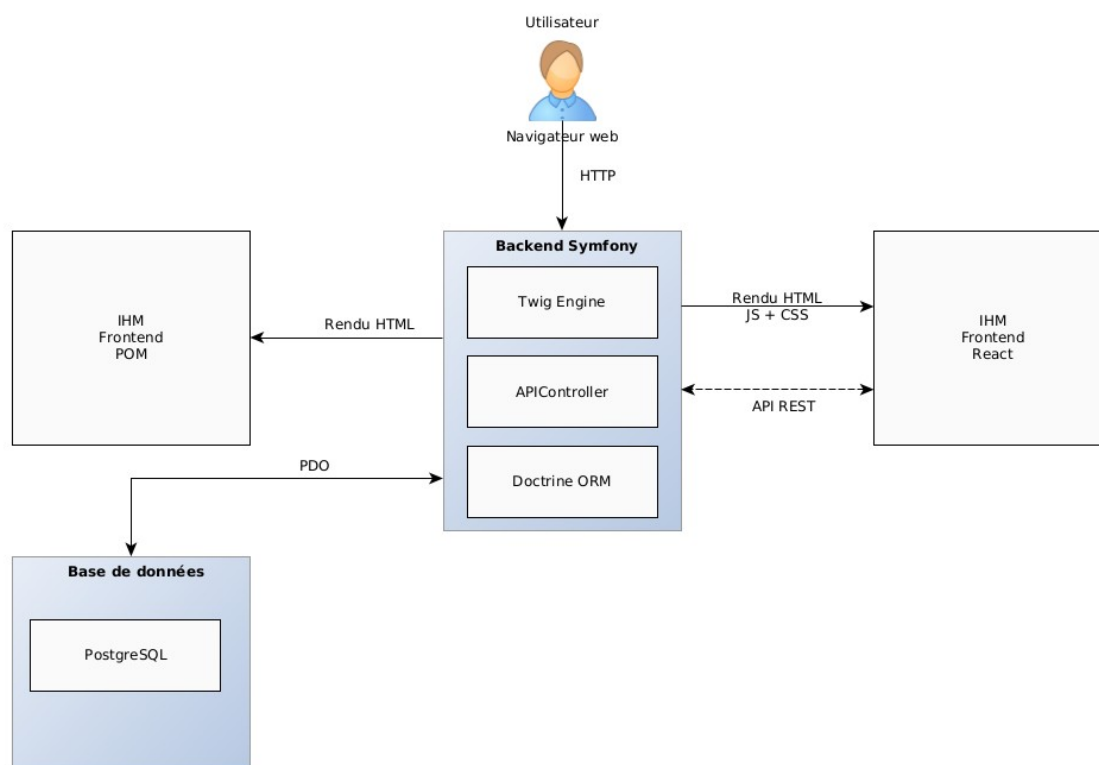


Figure 2: Architecture applicative : échanges avec la base de données

Le backend Symfony interagit avec une base de données PostgreSQL via Doctrine, configuré comme ORM principal. Lorsqu'une requête HTTP arrive (ex. depuis une IHM ou une API REST), Symfony la traite dans un contrôleur. Si des données sont nécessaires, le contrôleur appelle une entité ou un service qui utilise le EntityManager de Doctrine. Celui-ci traduit les opérations sur les entités en requêtes SQL adaptées à PostgreSQL, exécutées via PDO. Les résultats sont renvoyés sous forme d'objets entités à Symfony pour générer une réponse (Twig ou JSON).

La configuration de Doctrine est définie dans **config.yml**.

### 2.4.2 Composants techniques

- ✓ **Version : PostgreSQL 13 (évolution de 9.1)**
- ✓ **Configuration**
  - ↳ Port : 5433 (non standard, configurable dans **postgresql.conf**)
  - ↳ Sécurité : **pg\_hba.conf** pour restreindre les IPs autorisées à se connecter

## 2.5 Programmes d'interface

Les programmes d'interface utilisent une librairie Python **PomInterface PI** qui définit un cadre générique permettant à la POM d'interagir avec des modèles hydrologiques.

### 2.5.1 Interaction entre la POM et les serveurs de calcul

Ce schéma illustre la mise en œuvre des échanges entre la POM et les programmes d'interface Plx, via la librairie Python PomInterface.

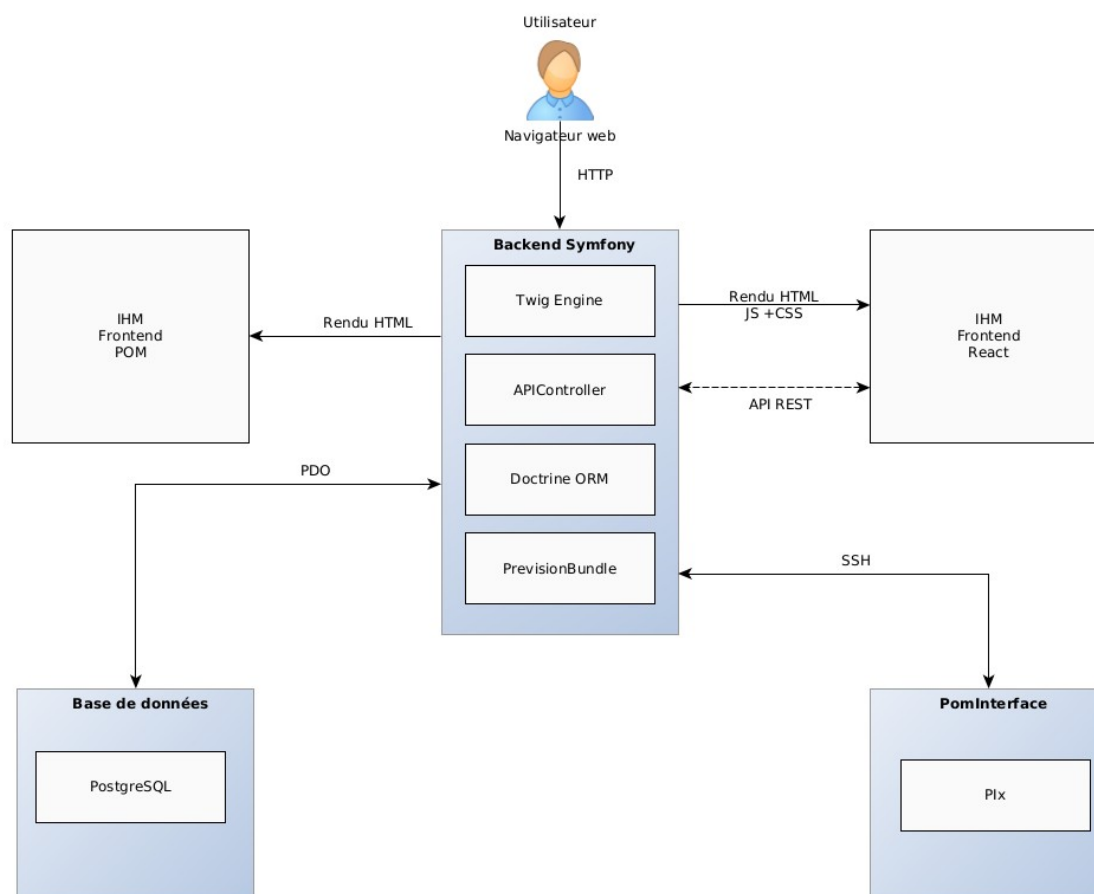


Figure 3: Architecture applicative : interaction entre la POM et les Plx

Les interactions entre la POM et les programmes d'interface se font via SSH, il s'agit :

- ✓ de la copie des fichiers nécessaires au programme d'interface
- ✓ du lancement de la commande du programme d'interface
- ✓ du contrôle de l'état de la commande lancée
- ✓ de la récupération des fichiers produits lorsque la commande est terminée

### 2.5.2 Composants techniques

- ✓ **Technologie :**

- **Langage**: Python 3.7
- **Dépendances explicites** : numpy, pandas, lxml, tomli
- **Dépendances implicites** : codecs, os, re, subprocess
- **Autres dépendances** : libhydro, libdbimage

### 2.5.3 Structure du projet

```
pom-interface/
├── libpatches/
│   ├── libdbimage/
│   └── libhydro/
├── pominterface/
│   ├── common/
│   ├── logs/
│   ├── models/
│   ├── parallel/
│   ├── processors/
│   │   ├── baseprocessor.py
│   │   ├── cleanerprocessor.py
│   │   ├── initializeprocessor.py
│   │   ├── inputsprocessor.py
│   │   ├── launchprocessor.py
│   │   ├── outputsprocessor.py
│   │   ├── runconfigprocessor.py
│   │   ├── runprocessor.py
│   │   ├── scenarioprocessor.py
│   │   └── startprocessor.py
│   ├── tests/
│   ├── tools/
│   └── xmlpom/
└── setup.py
```

Le plus important dans cette structure du projet ce sont les **processors**. Grâce aux **processors**, la librairie PomInterface définit un fonctionnement générique pour les programmes d'interfaces Plx. A la charge de chaque Plx d'implémenter les **processors** en fonction des spécificités du Plx

## 2.5.4 Structure applicative de la librairie PomInterface

Le schéma suivant permet de visualiser le rôle des processors qui ont un rôle générique dans la librairie PomInterface, à savoir : **StartProc**, **InitProc**, **CleanProc**.

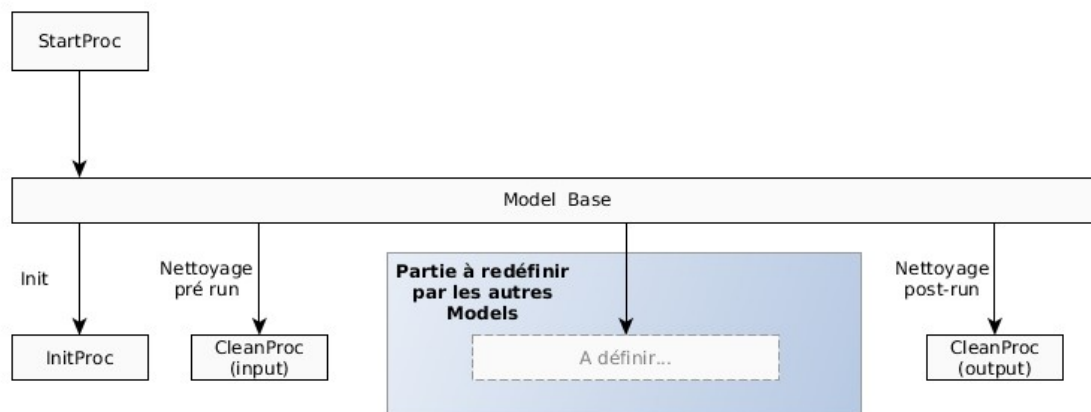


Figure 4: Librairie PomInterface : processors génériques

## 2.5.5 Structure applicative des modèles analyse/prévision

C'est le modèle le plus courant, le Plx va d'abord effectuer des calculs d'analyse séquentiel, puis des calculs de prévision avec un mécanisme de parallélisme.

Le schéma suivant illustre la structure de ce modèle et l'usage des processors : **ScenarioProc**, **InputProc**, **LaunchProc**, **RunConfigProc**, **RunProc**.

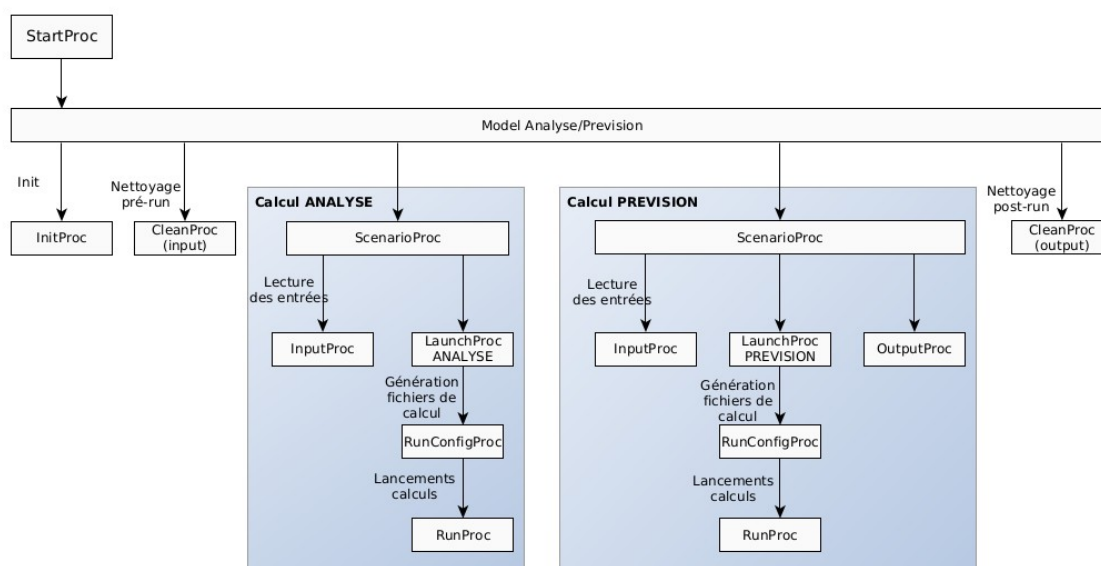


Figure 5: Librairie PomInterface : Model analyse / prévision

## 2.5.6 Mécanisme de logs

La librairie **PomInterface** définit un mécanisme de logs. Par défaut, les logs ne sont pas activés.

L'activation s'effectue en ajoutant l'option « --logs » » dans la ligne de commande de lancement du Plx.

Le fichier généré est nommé « pix\_horodate.log ».

Exemples de codes pour tracer en mode debug. Il faut utiliser la classe **LL**.

```
_ = LL.enter(_log, "RunConfigPitProcessor._check_simultaneous_launch()")
```

```
LL.debug(_log, u"Lancement calcul...")
```

## 2.6 Intégration avec d'autres systèmes

La POM interagit :

- ✓ Avec la PHYC pour l'authentification via Cerbere de l'utilisateur, pour l'extraction des données, pour l'insertion des prévisions.
- ✓ Avec la BDImage pour l'extraction des produits images d'observations et de prévisions.

Le schéma suivant illustre ces flux de données :

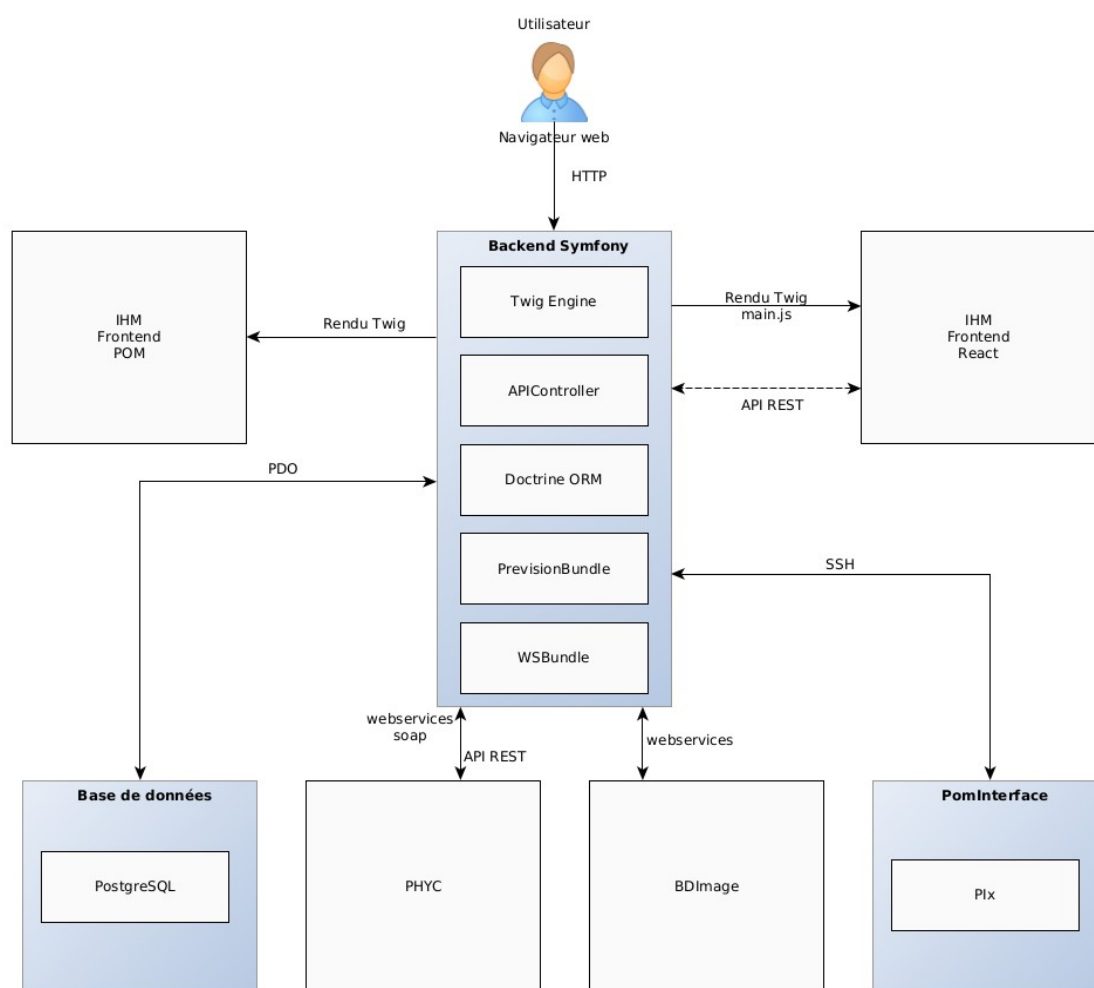


Figure 6: Architecture applicative : échanges avec la PHYC et la BDImage

L'ensemble des appels aux web-services PHYC et BDImage sont mis en œuvre dans le module « WSBundle ».

Il existe également des liens directs via HTTP entre certains formulaires de la POM de l'IHM POM complète et de l'IHM de pilotage des modèles, soit vers la page d'accueil du Superviseur National, soit vers des graphes de sites.

Il y a également des liens vers hydroportail depuis les entités et les utilisateurs.

Ces liens ne sont pas représentés sur les schémas d'architecture applicative, car ce sont des accès qui permettent de quitter la POM vers le Superviseur National.