記事
Toshihiko Minamoto · 2022年9月10日 49m read

Python のみを使用した InterSystems のインターオペラビリティフレームワーク

この**フォーメーション**は私の GitHub にあり、30 分で csv ファイルと txt ファイルの読み取りと書き込み方法、Postgres を使った**IRIS データベース**と**リモートデータベース**の挿入とアクセス方法、FLASK API の使用方法について説明します。これらすべてに、PEP8 命名規則に従った、Python のみのインターオペラビリティフレームワークを使用します。

このフォーメーションは、ほとんどをコピー＆ペースト操作で実行でき、グローバル演習を行う前に、ステップごとの操作が説明されています。
記事のコメント欄、Teams、またはメール（lucas.enard@intersystems.com）でご質問にお答えします。

このフォーメーションに関するあらゆる点において、ご意見やご感想をお送りいただけると幸いです。

1. Ensemble / Interoperability のフォーメーション

このフォーメーションでは、Python および特に以下を使用した InterSystems のインターオペラビリティフレームワークを学習することを目標としています。
* 本番環境
* メッセージ
* ビジネスオペレーション
* アダプター
* ビジネスプロセス
* ビジネスサービス
* REST サービスと運用

目次:

2. フレームワーク

以下は、IRIS フレームワークです。

IRIS 内部のコンポーネントは、本番環境を表します。 インバウンドアダプターとアウトバウンドアダプターは、様々な種類のフォーマットをデータベースの入力と出力として使用できるようにします。
複合アプリケーションにより、REST
サービスなどの外部アプリケーションを通じて本番環境にアクセスできます。

これらのコンポーネントを繋ぐ矢印は**メッセージ**で、 リクエストかレスポンスを表します。

3. フレームワークの適用

ここでは、CSV ファイルから行を読み取り、IRIS データベースに .txt ファイルで保存します。

次に、外部データベースにオブジェクトを保存できるようにするオペレーションを db-api を使って追加します。
このデータベースは Docker コンテナに配置されており、Postgres を使用します。

最後に、複合アプリケーションを使用して、新しいオブジェクトをデータベースに挿入する方法またはこのデータベースを照会する方法を確認します（ここでは、REST サービスを使用します）。

この目的に合わせて構成されたフレームワークは、以下のようになります。

WIP

4. 前提条件

このフォーメーションでは、以下の項目が必要です。
* VSCode: https://code.visualstudio.com/
* VSCode 用の InterSystems アドオンスイート: https://intersystems-community.github.io/vscode-objectscript/installation/
* Docker: https://docs.docker.com/get-docker/
* VSCode 用の Docker アドオン
* 自動的に行われるもの: Postgres の要件
* 自動的に行われるもの: Flask の要件

5. セットアップ

5.1. Docker コンテナ

InterSystems イメージにアクセスするために、次の URL に移動してください: http://container.intersystems.com。
InterSystems の資格情報にリンクすると、レジストリに接続するためのパスワードを取得できます。 Docker の VSCode アドオンのイメージタブで、［レジストリを接続］を押し、汎用レジストリとして上記の URL（http://container.intersystems.com）を入力すると、資格情報の入力を求められます。
このログインは通常のログインですが、パスワードはウェブサイトから取得したものを使用します。

これが済むと、コンテナを構築・作成（指定された docker-compose.yml と Dockerfile を使用）できるようになります。

5.2. 管理ポータルと VSCode

このリポジトリは、VS Code 対応です。

ローカルにクローンした formation-template-python を VS Code で開きます。

指示されたら（右下に表示）、推奨される拡張機能をインストールしてください。

5.3. コンテナ内でフォルダを開く

コーディングする前に、コンテナ*内部*にアクセスしていることが**非常に重要です**。
これには、docker が VSCode を開く前に有効である必要があります。
次に VSCode 内で指示されたら（右下に表示）、コンテナ内のフォルダを開き直すと、その中にある Python コンポーネントを使用できるようになります。
これを初めて行う場合、コンテナの準備が整うまで数分かかる場合があります。

詳細情報はこちらをご覧ください。

リモートのフォルダを開くと、その中で開く VS Code とターミナルで、コンテナ内の Python コンポーネントを使用できます。 /usr/irissys/bin/irispython を使用するように構成してください。

5.4. コンポーネントの登録

本番環境に対して Python で作成しているコンポーネントを登録するには、grongier.pex.utils モジュールから registercomponent 関数を使用する必要があります。

重要: コンポーネントはすでに登録済みです（グローバル演習を除く）。
情報までに、またグローバル演習で使用できるように、以下の手順でコンポーネントを登録します。
これには、プロジェクトで作業している際に、最初に組み込み Python コンソールを使用して手動でコンポーネントを追加することをお勧めします。

これらのコマンドは、misc/register.py ファイルにあります。
このコマンドを使用するには、まずコンポーネントを作成してから、VSCode でターミナルを起動する必要があります（5.2 と 5.3 の手順を実行している場合は、自動的にコンテナ内に移動しています）。
IrisPython コンソールを起動するには、以下のように入力します。

```
/usr/irissys/bin/irispython
```

次に、以下を入力します。

```
from grongier.pex._utils import register_component
```

そして、以下のようにして、コンポーネントを登録できます。

```
register_component("bo","FileOperation","/irisdev/app/src/python/",1,"Python.FileOperation")
```

この行は、bo モジュール内にコーディングされている FileOperation クラスを登録します。このファイルは /irisdev/app/src/python/（このコースに従って作業している場合のパス）にあり、管理ポータルでは Python.FileOperation という名前を使用しています。

コンポーネントが登録済みである際にこのファイルの名前、クラスまたはパスを変更しない場合、VSCode でそれらを変更することが可能です。登録し直す必要はありません。
管理ポータルでの再起動は、必ず行ってください。

5.5. 完成ファイル

フォーメーションのある時点でやり方がわからなくなったり、さらに説明が必要となった場合は、GitHub の solution ブランチにすべての正しい内容と動作する本番環境をご覧いただけます。

6. 本番環境

本番環境は、Iris
上のすべての作業の土台であり、**サービス**、**プロセス**、および**オペレーション**をまとめるフレームワーク
の外殻として考える必要があります。
本番環境内のすべては関数を継承します。この関数は、このクラスのインスタンスの作成時に解決する oninit 関数と、インスタンスがキルされたときに解決する onteardown 関数です。 これは、書き込みを行うときに変数を設定する際、または使用された開いているファイルをクローンする際に役立ちます。

ほぼすべてのサービス、プロセス、およびオペレ
ーションを持つ**本番環境**は、**すでに作成済み**であることに注意してください。
指示されたら、ユーザー名の SuperUser とパスワードの SYS を使用して接続してください。

本番環境が開いていない場合は、[Interoperability] > [Configure] メニューに移動して、[Production] をクリックし、 次に [Open] をクリックして iris / Production を選択します。

これを行ったら、ビジネスオペレーションに直接進むことができます。

ただし、本番環境の作成方法に興味がある場合は、以下のようにして作成することができます。
管理ポータルに移動し、ユーザー名: SuperUser、パスワード: SYS を使用して接続します。
次に、[Interoperability] と [Configure] メニューに進みます。

次に、[New] を押して [Formation] パッケージを選択し、本番環境の名前を設定します。

本番環境を作成した直後、[Operations] セクションの真上にある [Production Settings（本番環境の設定）] をクリックする必要があります。 右のサイドバーメニューで、[Settings] タブの [Development and Debugging（開発とデバッグ）] で [Testing Enabled（テストを有効）] をアクティブにします（忘れずに [Apply] を押してください）。

この最初の本番環境で、ビジネスオペレーションを追加していきます。

7. ビジネスオペレーション

ビジネスオペレーション（BO）は、IRIS
から外部アプリケーション/システムにリクエストを送信できるようにする特定のオペレーションです。
必要なものを IRIS に直接保存するためにも使用できます。
BO には、このインスタンスがソースからメッセージを受信するたびに呼び出される onmessage
関数もあるため、フレームワークで確認できるように、情報を外部クライアントと送受信することが可能です。

これらのオペレーションは、VSCode でローカルに作成します。つまり、src/python/bo.py ファイルです。
このファイルを保存すると、IRIS でコンパイルされます。

ここでの最初のオペレーションでは、メッセージのコンテンツをローカルデータベースに保存し、同じ情報をローカルの .txt ファイルに書き込みます。

まずは、このメッセージを保存する方法を作成する必要があります。

7.1. オブジェクトクラスの作成

dataclass を使用して、メッセージの情報を格納することにします。

すでに存在する src/python/obj.py ファイルを以下のようにします。
インポート:

```
from dataclasses import dataclass
```

コード:

```
@dataclass
class Formation:
    id_formation:int = None
    nom:str = None
    salle:str = None

@dataclass
class Training:
    name:str = None
    room:str = None
```

Formation クラスは、csv の情報を格納して「8. ビジネスプロセス」に送信する Python
オブジェクトとして使用し、Training クラスは、「8. ビジネスプロセス
」から複数のオペレーションに情報を送信し、Iris データベースに保存するか .txt
ファイルに書き込むために使用されます。

7.2. メッセージクラスの作成

これらのメッセージには、7.1 で作成された obj.py ファイルにある Formation オブジェクトまたは Training
オブジェクトが含まれます。

メッセージ、リクエスト、およびレスポンスはすべて grongier.pex.Message
クラスの継承であることに注意してください。

すでに存在する src/python/msg.py ファイルを以下のようにします。
インポート:

```
from dataclasses import dataclass
from grongier.pex import Message

from obj import Formation,Training
```

コード:

```
@dataclass
class FormationRequest(Message):
    formation:Formation = None

@dataclass
class TrainingRequest(Message):
    training:Training = None
```

繰り返しますが、FormationRequest クラスは、csv の情報を格納して「8. ビジネスプロセス」に送信するメッセージとして使用し、TrainingRequest クラスは、「8. ビジネスプロセス」から複数のオペレーションに情報を送信し、Iris データベースに保存するか .txt ファイルに書き込むために使用されます。

7.3. オペレーションの作成

必要な要素がすべて揃ったので、オペレーションを作成できるようになりました。
すべてのビジネスオペレーションは、grongier.pex.BusinessOperation クラスを継承していることに注意してください。
すべてのオペレーションは、src/python/bo.py に保存されます。これらを区別できるよう、ファイルに現時点で確認できるように複数のクラスを作成する必要があります。オペレーションのすべてのクラスはすでに存在していますが、現時点ではほぼ空の状態です。

オペレーションがメッセージ/リクエストを受信すると、各関数のシグネチャに指定されたメッセージ/リクエストの種類に応じて自動的に正しい関数にメッセージ/リクエストを送信します。
メッセージ/リクエストの種類が処理されない場合は、onmessage 関数に転送されます。

すでに存在する src/python/bo.py ファイルを以下のようにします。
インポート:

```
from grongier.pex import BusinessOperation
import os
import iris

from msg import TrainingRequest,FormationRequest
```

FileOperation クラスのコード:

```
class FileOperation(BusinessOperation):
    """
    ????????????????????????????????
    """
```

```
    def on_init(self):
        """
        ?????????????????? path ??????????????path ????????????? /tmp ??????????
???
        ???filename ??????????????toto
.csv ???????
        :return: None
        """
        if hasattr(self,'path'):
            os.chdir(self.path)
        else:
            os.chdir("/tmp")
        return None

    def write_training(self, request:TrainingRequest):
        """
        ??????????????????

        :param request: The request message
        :type request: TrainingRequest
        :return: None
        """
        romm = name = ""
        if request.training is not None:
            room = request.training.room
            name = request.training.name
        line = room+" : "+name+"\n"
        filename = 'toto.csv'
        self.put_line(filename, line)
        return None

    def on_message(self, request):
        return None


    def put_line(self,filename,string):
        """
        ????????????????????????

        :param filename: The name of the file to write to
        :param string: The string to be written to the file
        """
        try:
            with open(filename, "a",encoding="utf-8",newline="") as outfile:
                outfile.write(string)
        except Exception as error:
            raise error
```

ご覧のとおり、FileOperation が msg.TrainingRequest タイプのメッセージを受信すると、request のシグネチャが TrainingRequest であるため、writetraining 関数に送信されます。
この関数では、メッセージが格納する情報が toto.csv ファイルに書き込まれます。

path はオペレーションのパラメーターであるため、filename
を管理ポータルで直接変更できる変数にし、基本値を toto.csv とすることができます。
このようにするには、oninit 関数を以下のように編集する必要があります。

```
    def on_init(self):
```

```
        if hasattr(self,'path'):
            os.chdir(self.path)
        else:
            os.chdir("/tmp")
        if not hasattr(self,'filename'):
            self.filename = 'toto.csv'
        return None
```

次に、オペレーションに直接コーディングして filename = 'toto.csv' を使用する代わりに、self.filename を呼び出します。
すると、writetraining 関数は、以下のようになります。

```
    def write_training(self, request:TrainingRequest):
        romm = name = ""
        if request.training is not None:
            room = request.training.room
            name = request.training.name
        line = room+" : "+name+"\n"
        self.put_line(self.filename, line)
        return None
```

独自の filename の選択方法にいては、「7.5 テスト」をご覧ください。

情報を .txt ファイルに書き込めるようになりましたが、iris データベースはどうなっているでしょうか？
src/python/bo.py ファイルでは、IrisOperation クラスのコードは以下のようになっています。

```
class IrisOperation(BusinessOperation):
    """
    iris ??????????????????????????
    """

    def insert_training(self, request:TrainingRequest):
        """
        `TrainingRequest` ?????????????? `iris.training` ?????????
        `TrainingResponse` ??????????

        :param request: ?????????????????
        :type request: TrainingRequest
        :return: TrainingResponse ?????
        """
        sql = """
        INSERT INTO iris.training
        ( name, room )
        VALUES( ?, ? )
        """
        iris.sql.exec(sql,request.training.name,request.training.room)
        return None

    def on_message(self, request):
        return None
```

ご覧のとおり、IrisOperation が msg.TrainingRequest
タイプのメッセージを受信すると、このメッセージが保有する情報は、iris.sql.exec IrisPython 関数によって SQL クエリに変換されて実行されます。 このメソッドによって、メッセージは IRIS
ローカルデータベースに保存されます。

これらのコンポーネントは、事前に本番環境に**登録済み**です。

情報までに、コンポーネントを登録する手順は、5.4. に従い、以下を使用します。

```
register_component("bo","FileOperation","/irisdev/app/src/python/",1,"Python.FileOper
ation")
```

および、以下を使用します。

```
register_component("bo","IrisOperation","/irisdev/app/src/python/",1,"Python.IrisOper
ation")
```

7.4. 本番環境へのオペレーションの追加

オペレーションは、こちらで事前に登録済みです。
ただし、オペレーションを新規作成する場合は、手動で追加する必要があります。

今後の参考までに、オペレーションの登録手順を説明します。
登録には、管理ポータルを使用します。 [Operations] の横にある [+] 記号を押すと、[Business Operation Wizard（ビジネスオペレーションウィザード）] が開きます。
そのウィザードで、スクロールメニューから作成したばかりのオペレーションクラスを選択します。

新しく作成したすべてのオペレーションに対して必ず実行してください！

7.5. テスト

オペレーションをダブルクリックすると、オペレーションが有効化されるか、再起動して変更内容が保存されます。
重要: この無効化して有効化し直す手順は、変更内容を保存する上で非常に重要な手順です。
重要: その後で、Python.IrisOperation **オペレーション**を選択し、右のサイドバーメニューの [Actions] タブに移動すると、**オペレーション**を**テスト**することができます。
（うまくいかない場合は、テストを有効化
し、本番環境が開始していることを確認してから、本番環境をダブルクリックして再起動をクリックし、オペレーションをリロードしてください）。

IrisOperation については、テーブルは自動的に作成済みです。
情報までに、これを作成するには、管理ポータルを使用します。[System Explorer（システムエクスプローラー）] > [SQL] > [Go] に移動して、Iris データベースにアクセスします。 次に、[Execute Query（クエリを実行）] を開始します。

```
CREATE TABLE iris.training (
    name varchar(50) NULL,
    room varchar(50) NULL
)
```

管理ポータルのテスト機能を使用して、前に宣言したタイプのメッセージをオペレーションに送信します。 すべてがうまくいけば、仮想トレースを表示すると、プロセスとサービスとオペレーション間で何が起きたかを確認できるようになります。
Request Type として使用した場合:

```
Grongier.PEX.Message
```

%classname として使用した場合:

```
msg.TrainingRequest
```

%json として使用した場合:

```
{
    "training":{
        "name": "name1",
        "room": "room1"
    }
}
```

ここでは、メッセージがプロセスによってオペレーションに送信され、オペレーションがレスポンス（空の文字列）を送り返すのを確認できます。
以下のような結果が得られます。

FileOperation については、以下のように管理ポータルの %settings に path を入力できます（7.3. の filename に関するメモに従った場合は、設定に filename を追加できます）。

```
path=/tmp/
filename=tata.csv
```

結果は以下のようになります。

繰り返しますが、Python.FileOperation **オペレーション**を選択して、右サイドバーメニューの [Actions] タブに移動すると、**オペレーション**を**テスト**できます。
（うまくいかない場合は、テストを有効化し、本番環境が起動していることを確認してください）。
Request Type として使用した場合:

```
Grongier.PEX.Message
```

%classname として使用した場合:

```
msg.TrainingRequest
```

%json として使用した場合:

```
{
    "training":{
        "name": "name1",
        "room": "room1"
    }
}
```

結果は以下のようになります。

オペレーションが動作したかどうかを確認するには、toto.csv（7.3. の filename に関するメモに従った場合は tata.csv）ファイルと Iris データベースにアクセスして変更内容を確認する必要があります。
次の手順を実行するには、コンテナの中に移動する必要がありますが、5.2. と 5.3 を実行した場合には、省略できます。
toto.csv にアクセスするには、ターミナルを開いて、以下を入力する必要があります。

```
bash
```

```
cd /tmp
```

```
cat toto.csv
```

または、必要であれば "cat tata.csv" を使用してください。
重要: ファイルが存在しない場合、管理ポータルでオペレーションを再起動していない可能性があります。
再起動するには、オペレーションをクリックして再起動を選択してください（または、無効化してからもう一度ダブルクリックして有効化してください）。
もう一度テストする必要があります。

Iris データベースにアクセスするには、管理ポータルにアクセスし、[System Explorer（システムエクスプローラー）] > [SQL] > [Go] を選択する必要があります。 次に、[Execute Query（クエリを実行）] を開始します。

```
SELECT * FROM iris.training
```

8. ビジネスプロセス

ビジネスプロセス（BP）は、本番環境のビジネスロジックです。 リクエストをプロセスしたり、本番環境の他のコンポーネントにそのリクエストをリレーしたリするために使用されます。
BP には、インスタンスがソースからリクエストを受信するたびに呼び出される onrequest

関数もあるため、情報を受信して処理し、正しい BO に送信することができます。

これらのプロセスは、VSCode でローカルに作成します。つまり、src/python/bp.py ファイルです。
このファイルを保存すると、IRIS でコンパイルされます。

8.1. 単純な BP

サービスから受信する情報を処理し、適切に配信する**ビジネスプロセス**を作成しましょう。
オペレーションを呼び出す単純な BP を作成することにします。

この BP は情報をリダイレクトするだけであるため、これを Router と呼び、src/python/bp.py に作成します。
インポート:

```
from grongier.pex import BusinessProcess

from msg import FormationRequest, TrainingRequest
from obj import Training
```

コード:

```
class Router(BusinessProcess):

    def on_request(self, request):
        """
        ??????????????????????????????????????
        TrainingRequest ?????? FileOperation ? IrisOperation ???????IrisOperation ? 1
 ?????PostgresOperation ???????

        :param request: ??????????????
        :return: None
        """
        if isinstance(request,FormationRequest):

            msg = TrainingRequest()
            msg.training = Training()
            msg.training.name = request.formation.nom
            msg.training.room = request.formation.salle

            self.send_request_sync('Python.FileOperation',msg)
            self.send_request_sync('Python.IrisOperation',msg)
        return None
```

Router は FormationRequest タイプのリクエストを受信し、TrainingRequest タイプのメッセージを作成して IrisOperation と FileOperation オペレーションに送信します。
メッセージ/リクエストが求めているタイプのインスタンスでない場合、何も行わず、配信しません。

これらのコンポーネントは、事前に本番環境に**登録済み**です。

情報までに、コンポーネントを登録する手順は、5.4. に従い、以下を使用します。

```
register_component("bp","Router","/irisdev/app/src/python/",1,"Python.Router")
```

8.2. 本番環境へのプロセスの追加

プロセスは、こちらで事前に登録済みです。
ただし、プロセスを新規作成する場合は、手動で追加する必要があります。

今後の参考までに、プロセスの登録手順を説明します。
登録には、管理ポータルを使用します。 [Process] の横にある [+] 記号を押すと、[Business Process Wizard（ビジネスプロセスウィザード）] が開きます。
そのウィザードで、スクロールメニューから作成したばかりのプロセスクラスを選択します。

8.3. テスト

プロセスをダブルクリックすると、プロセスが有効化されるか、再起動して変更内容が保存されます。
重要: この無効化して有効化し直す手順は、変更内容を保存する上で非常に重要な手順です。
重要: その後で、**プロセス**を選択し、右のサイドバーメニューの [Actions] タブに移動すると、**プロセス**を**テスト**することができます。
（うまくいかない場合は、テストを有効化し、本番環境が開始していることを確認してから、本番環境をダブルクリックして再起動をクリックし、プロセスをリロードしてください）。

こうすることで、プロセスに msg.FormationRequest タイプのメッセージを送信します。 Request Type として使用した場合:

```
Grongier.PEX.Message
```

%classname として使用した場合:

```
msg.FormationRequest
```

%json として使用した場合:

```
{
    "formation":{
        "id_formation": 1,
        "nom": "nom1",
        "salle": "salle1"
    }
}
```

すべてがうまくいけば、仮想トレースを表示すると、プロセスとサービスとプロセス間で何が起きたかを確認できるようになります。
ここでは、メッセージがプロセスによってオペレーションに送信され、オペレーションがレスポンスを送り返すのを確認できます。

9. ビジネスサービス

ビジネスサービス（BS）は、本番環境の中核です。 情報を収集し、ルーターに送信するために使用されます。
BS には、フレームワークの情報を頻繁に収集する onprocessinput 関数もあるため、REST API やその他のサービス、またはサービス自体などの複数の方法で呼び出してサービスのコードをもう一度実行することが可能です。
BS には、クラスにアダプターを割り当てられる getadaptertype
関数もあります。たとえば、Ens.InboundAdapter は、サービスがその onprocessinput を 5
秒おきに呼び出すようにすることができます。

これらのサービスは、VSCode でローカルに作成します。つまり、python/bs.py ファイルです。
このファイルを保存すると、IRIS でコンパイルされます。

9.1. 単純な BS

CSV を読み取って、msg.FormationRequest
として各行をルーターに送信するビジネスサービスを作成しましょう。

この BS は csv を読み取るため、これを ServiceCSV と呼び、src/python/bs.py に作成します。
インポート:

```
from grongier.pex import BusinessService

from dataclass_csv import DataclassReader

from obj import Formation
from msg import FormationRequest
```

コード:

```
class ServiceCSV(BusinessService):
    """
    csv ????? 5 ?????????????????? Python Router ??????????
    """

    def get_adapter_type():
        """
        ?????????
        """
        return "Ens.InboundAdapter"

    def on_init(self):
        """
        ???????????????? path ????????????????
        path ????????????? '/irisdev/app/misc/' ???????
        :return: None
        """
        if not hasattr(self,'path'):
            self.path = '/irisdev/app/misc/'
        return None

    def on_process_input(self,request):
        """
        formation.csv ?????????????FormationRequest ??????????
        Python.Router ??????????

        :param request: ??????????
        :return: None
```

```
        """
        filename='formation.csv'
        with open(self.path+filename,encoding="utf-8") as formation_csv:
            reader = DataclassReader(formation_csv, Formation,delimiter=";")
            for row in reader:
                msg = FormationRequest()
                msg.formation = row
                self.send_request_sync('Python.Router',msg)
        return None
```

FlaskService はそのままにし、ServiceCSV のみを入力することをお勧めします。

ご覧のとおり、ServiceCSV は、独立して機能し、5
秒おき（管理ポータルのサービスの設定にある基本設定で変更できるパラメーター）に onprocessinput
を呼び出せるようにする InboundAdapter を取得します。

サービスは、5 秒おきに formation.csv を開き、各行を読み取って、Python.Router に送信される
msg.FormationRequest を作成します。

これらのコンポーネントは、事前に本番環境に**登録済み**です。

情報までに、コンポーネントを登録する手順は、5.4. に従い、以下を使用します。

```
register_component("bs","ServiceCSV","/irisdev/app/src/python/",1,"Python.ServiceCSV"
)
```

9.2. 本番環境へのサービスの追加

サービスは、こちらで事前に登録済みです。
ただし、サービスを新規作成する場合は、手動で追加する必要があります。

今後の参考までに、サービスの登録手順を説明します。
登録には、管理ポータルを使用します。 [service] の横にある [+] 記号を押すと、[Business Service
Wizard（ビジネスサービスウィザード）] が開きます。
そのウィザードで、スクロールメニューから作成したばかりのサービスクラスを選択します。

9.3. テスト

サービスをダブルクリックすると、サービスが有効化されるか、再起動して変更内容が保存されます。
重要: この無効化して有効化し直す手順は、変更内容を保存する上で非常に重要な手順です。
前に説明したように、サービスは 5 秒おきに自動的に開始するため、ここでは他に行うことはありません。
すべてがうまくいけば、視覚的トレースを表示すると、プロセスとサービスとプロセス間で何が起きたかを確認することができます。
ここでは、メッセージがサービスによってプロセスに送信され、プロセスによってオペレーションに送信され、オペレーションがレスポンスを送り返すのを確認できます。

10. db-api による外部データベースへのアクセス

このセクションでは、外部データベースにオブジェクトを保存するオペレーションを作成します。 db-api
を使用し、その他の Docker コンテナをセットアップして Postgres を設定します。

10.1. 前提条件

Postgres を使用するには psycopg2 が必要です。これは、単純なコマンドで Postgres
データベースに接続できるようにする Python モジュールです。
これは自動的に完了済みですが、情報までに説明すると、Docker コンテナにアクセスし、pip3 を使って
psycopg2 をインストールします。
ターミナルを開始したら、以下を入力します。

```
pip3 install psycopg2-binary
```

または、requirements.txt にモジュールを追加して、コンテナを再構築できます。

10.2. 新しいオペレーションの作成

新しいオペレーションは、src/python/bo.py ファイルの他の 2 つのオペレーションの後に追加してください。
以下は、新しいオペレーションとインポートです。
インポート:

```
import psycopg2
```

コード:

```
class PostgresOperation(BusinessOperation):
    """
    ??????? Postgres ??????????????????????
    """

    def on_init(self):
        """
        Postgres ????????????????????????????????
        :return: None
        """
        self.conn = psycopg2.connect(
        host="db",
        database="DemoData",
        user="DemoData",
        password="DemoData",
        port="5432")
        self.conn.autocommit = True

        return None

    def on_tear_down(self):
        """
        ????????????????
        :return: None
        """
        self.conn.close()
        return None

    def insert_training(self,request:TrainingRequest):
```

```
        """
        ??????? Postgre ?????????????

        :param request: ?????????????????
        :type request: TrainingRequest
        :return: None
        """
        cursor = self.conn.cursor()
        sql = "INSERT INTO public.formation ( name,room ) VALUES ( %s , %s )"
        cursor.execute(sql,(request.training.name,request.training.room))
        return None

    def on_message(self,request):
        return None
```

このオペレーションは、最初に作成したオペレーションに似ています。 msg.TrainingRequest
タイプのメッセージを受信すると、psycopg モジュールを使用して、SQL リクエストを実行します。
これらのリクエストは、Postgres データベースに送信されます。

ご覧のとおり、接続はコードに直接書き込まれています。コードを改善するために、他のオペレーションで前に行ったようにし、host、database、およびその他の接続情報を変数にすることができます。基本値を db や DemoData にし、管理ポータルで直接変更できるようにします。
これを行うには、以下のようにして oninit を変更します。

```
    def on_init(self):
        if not hasattr(self,'host'):
          self.host = 'db'
        if not hasattr(self,'database'):
          self.database = 'DemoData'
        if not hasattr(self,'user'):
          self.user = 'DemoData'
        if not hasattr(self,'password'):
          self.password = 'DemoData'
        if not hasattr(self,'port'):
          self.port = '5432'

        self.conn = psycopg2.connect(
        host=self.host,
        database=self.database,
        user=self.user,
        password=self.password,
        port=self.port)

        self.conn.autocommit = True

        return None
```

これらのコンポーネントは、事前に本番環境に**登録済み**です。

情報までに、コンポーネントを登録する手順は、5.4. に従い、以下を使用します。

```
register_component("bo","PostgresOperation","/irisdev/app/src/python/",1,"Python.Post
gresOperation")
```

10.3. 本番環境の構成

オペレーションは、こちらで事前に登録済みです。
ただし、オペレーションを新規作成する場合は、手動で追加する必要があります。

今後の参考までに、オペレーションの登録手順を説明します。
登録には、管理ポータルを使用します。 [Operations] の横にある [+] 記号を押すと、[Business Operation Wizard（ビジネスオペレーションウィザード）] が開きます。
そのウィザードで、スクロールメニューから作成したばかりのオペレーションクラスを選択します。

その後、接続を変更する場合は、オペレーションの [parameter] ウィンドウで、[Python] の %settings に変更するパラメーターを追加すれば完了です。 詳細については、「7.5 テスト」の 2 つ目の画像をご覧ください。

10.4. テスト

オペレーションをダブルクリックすると、オペレーションが有効化されるか、再起動して変更内容が保存されます。
重要: この無効化して有効化し直す手順は、変更内容を保存する上で非常に重要な手順です。
重要: その後で、**オペレーション**を選択し、右のサイドバーメニューの [Actions] タブに移動すると、**オペレーション**を**テスト**することができます。
（うまくいかない場合は、テストを有効化し、本番環境が開始していることを確認してから、本番環境をダブルクリックして再起動をクリックし、オペレーションをリロードしてください）。

PostGresOperation については、テーブルは自動的に作成済みです。

こうすることで、オペレーションに msg.TrainingRequest タイプのメッセージを送信します。 Request Type として使用した場合:

```
Grongier.PEX.Message
```

%classname として使用した場合:

```
msg.TrainingRequest
```

%json として使用した場合:

```
{
    "training":{
        "name": "nom1",
        "room": "salle1"
    }
}
```

以下のとおりです。

仮想トレースをテストすると、成功が表示されます。

これで、外部データベースに接続することができました。

ここまで、ここに記載の情報に従ってきた場合、プロセスやサービスは新しい PostgresOperation を呼び出さない、つまり、管理ポータルのテスト機能を使用しなければ呼び出されないことを理解していることでしょう。

10.5. 演習

演習として、bo.IrisOperation がブール値を返すように変更し、そのブール値に応じて bp.Router に bo.PostgresOperation を呼び出すようにしてみましょう。
そうすることで、ここで新しく作成したオペレーションが呼び出されるようになります。

ヒント: これは、bo.IrisOperation レスポンスの戻り値のタイプを変更し、新しいメッセージ/レスポンスタイプに新しいブール値プロパティを追加して、bp.Router に if 操作を使用すると実現できます。

10.6. ソリューション

まず、bo.IrisOperation からのレスポンスが必要です。 src/python/msg.py の他の 2 つのメッセージの後に、以下のようにして新しいメッセージを作成します。
コード:

```
@dataclass
class TrainingResponse(Message):
    decision:int = None
```

次に、そのレスポンスによって、bo.IrisOperation のレスポンスを変更し、decision の値を 1 または 0 にランダムに設定します。
src/python/bo.py で 2 つのインポートを追加し、IrisOperation クラスを変更する必要があります。
インポート:

```
import random
from msg import TrainingResponse
```

コード:

```
class IrisOperation(BusinessOperation):
    """
    ??????? iris ??????????????????????
    """

    def insert_training(self, request:TrainingRequest):
        """
        `TrainingRequest` ??????????????`iris.training` ????????
        `TrainingResponse` ???????????

        :param request: ??????????????????
        :type request: TrainingRequest
        :return: A TrainingResponse message
        """
        resp = TrainingResponse()
        resp.decision = round(random.random())
```

```
        sql = """
        INSERT INTO iris.training
        ( name, room )
        VALUES( ?, ? )
        """
        iris.sql.exec(sql,request.training.name,request.training.room)
        return resp

    def on_message(self, request):
        return None
```

次に、src/python/bp.py で bp.Router プロセスを変更します。ここでは、IrisOperation からのレスポンスが 1 の場合に PostgresOperation を呼び出すようにします。 新しいコードは以下のようになります。

```
class Router(BusinessProcess):

    def on_request(self, request):
        """
        ??????????????????????????????????????????
        ??????TrainingRequest ?????? FileOperation ? IrisOperation ?????IrisOperation
 ? 1 ???? PostgresOperation ???????

        :param request: ??????????????
        :return: None
        """
        if isinstance(request,FormationRequest):

            msg = TrainingRequest()
            msg.training = Training()
            msg.training.name = request.formation.nom
            msg.training.room = request.formation.salle

            self.send_request_sync('Python.FileOperation',msg)

            form_iris_resp = self.send_request_sync('Python.IrisOperation',msg)
            if form_iris_resp.decision == 1:
                self.send_request_sync('Python.PostgresOperation',msg)
        return None
```

非常に重要: オペレーションの呼び出しには、**send<u>r</u>equest<u>a</u>sync**　ではなく、必ず **send<u>r</u>equest<u>s</u>ync** を使用する必要があります。そうでない場合、この操作はブール値のレスポンスを受け取る前に実行されます。

テストする前に、必ず変更したすべてのサービス、プロセス、およびオペレーションをダブルクリックして再起動してください。これを行わない場合、変更内容は適用されません。

テスト後、視覚的トレースで、csv で読み取られたおよそ半数のオブジェクトがリモートデータベースにも保存されていることがわかります。
bs.ServiceCSV を開始するだけでテストできることに注意してください。リクエストは自動的にルーターに送信

され、適切に配信されます。
また、サービス、オペレーション、またはプロセスをダブルクリックしてリロードを押すか、VSCode で保存した変更を適用するには再起動を**押す必要がある**ことにも注意してください。

11. REST サービス

ここでは、REST サービスを作成して使用します。

11.1. 前提条件

Flask を使用するには、flask のインストールが必要です。これは、REST サービスを簡単に作成できるようにする Python モジュールです。 **これは、自動的に実行済みです**が、今後の情報までに説明すると、Docker コンテナ内にアクセスして iris python に flask をインストールします。
ターミナルを開始したら、以下を入力します。

```
pip3 install flask
```

または、requirements.txt にモジュールを追加して、コンテナを再構築できます。

11.2. サービスの作成

REST サービスを作成するには、API を本番環境にリンクするサービスが必要です。このために、src/python/bs.py の ServiceCSV クラスの直後に新しい単純なサービスを作成します。

```
class FlaskService(BusinessService):

    def on_init(self):
        """
        API ???????????????? target ??????????????????
        ????target ????????????? 'Python.Router' ???????
        :return: None
        """
        if not hasattr(self,'target'):
            self.target = "Python.Router"
        return None

    def on_process_input(self,request):
        """
        API ???? Python.Router ??????????????????????
        :return: None
        """
        return self.send_request_sync(self.target,request)
```

このサービスに onprocessinput を行うと、リクエストが Router に転送されます。

これらのコンポーネントは、事前に本番環境に**登録済み**です。

情報までに、コンポーネントを登録する手順は、5.4. に従い、以下を使用します。

```
register_component("bs","FlaskService","/irisdev/app/src/python/",1,"Python.FlaskServ
ice")
```

REST サービスを作成するには、Flask を使って get と post 関数を管理する API を作成する必要があります。
新しいファイルを python/app.py として作成してください。

```
from flask import Flask, jsonify, request, make_response
from grongier.pex import Director
import iris

from obj import Formation
from msg import FormationRequest


app = Flask(__name__)

# GET Infos
@app.route("/", methods=["GET"])
def get_info():
    info = {'version':'1.0.6'}
    return jsonify(info)

# GET all the formations
@app.route("/training/", methods=["GET"])
def get_all_training():
    payload = {}
    return jsonify(payload)

# POST a formation
@app.route("/training/", methods=["POST"])
def post_formation():
    payload = {}

    formation = Formation()
    formation.nom = request.get_json()['nom']
    formation.salle = request.get_json()['salle']

    msg = FormationRequest(formation=formation)

    service = Director.CreateBusinessService("Python.FlaskService")
    response = service.dispatchProcessInput(msg)

    return jsonify(payload)

# GET formation with id
@app.route("/training/<int:id>", methods=["GET"])
def get_formation(id):
    payload = {}
    return jsonify(payload)

# PUT to update formation with id
@app.route("/training/<int:id>", methods=["PUT"])
```

```
def update_person(id):
    payload = {}
    return jsonify(payload)

# DELETE formation with id
@app.route("/training/<int:id>", methods=["DELETE"])
def delete_person(id):
    payload = {}
    return jsonify(payload)

if __name__ == '__main__':
    app.run('0.0.0.0', port = "8081")
```

Flask API は Director を使用して、前述のものから FlaskService
のインスタンスを作成してから適切なリクエストを送信することに注意してください。

上記の子k一度では、POST
フォーメンション関数を作成しました。希望するなら、これまでに学習したものすべてを使用して、適切な情報を get/post
するように、他の関数を作成することが可能です。ただし、これに関するソリューションは提供されていません。

11.3. テスト

Python Flask を使用して flask アプリを開始しましょう。

最後に、Router サービスをリロードしてから、任意の REST
クライアントを使用してサービスをテストできます。

（Mozilla の RESTer として）REST
サービスを使用するには、以下のようにヘッダーを入力する必要があります。

```
Content-Type : application/json
```

ボディは以下のようになります。

```
{
    "nom":"testN",
    "salle":"testS"
}
```

認証は以下のとおりです。
ユーザー名:

```
SuperUser
```

パスワード:

```
SYS
```

最後に、結果は以下のようになります。

12. グローバル演習

Iris DataPlatform
とそのフレームワーク
のすべての重要な概念について学習したので、グローバル演習で腕試しをしましょう。この演習では、新しい BS と BP を作成し、BO を大きく変更して、新しい概念を Python で探ります。

12.1. 指示

こちらの**エンドポイント** https://lucasenard.github.io/Data/patients.json
を使用して、患者と歩数に関する情報を自動的に**取得**するようにします。 次に、ローカルの csv ファイルに書き込む前に、各患者の平均歩数を計算します。

必要であれば、フォーメーション全体ま
たは必要な箇所を読むか、以下のヒントを使って、ガイダンスを得ることをお勧めします。

管理ポータルでコンポーネントにアクセスできるように、コンポーネントの登録を忘れずに行いましょう。

すべてを完了し、テストしたら、または演習を完了するのに十分なヒントを得られなかった場合は、全過程をステップごとに説明したソリューションをご覧ください。

12.2. ヒント

ここでは、演習を行うためのヒントを紹介します。
読んだ分だけヒントが得られてしまうため、必要な箇所のみを読み、毎回すべてを読まないようにすることをお勧めします。

たとえば、bs
の「情報の取得」と「リクエストによる情報の取得」のみを読み、他は読まないようにしましょう。

12.2.1. bs

12.2.1.1. 情報の取得

エンドポイントから情報を取得するには、Python の requests モジュールを検索し、json と json.dumps を使用して文字列に変換してから bp に送信します。

12.2.1.2. リクエストによる情報の取得

オンライン Python Web サイトまたはローカルの Python ファイルを使用してリクエストを使用し、取得した内容をさらに深く理解するために、出力とそのタイプを出力します。

12.2.1.3. リクエストによる情報の取得とその使用

新しいメッセージタイプと情報を保持するオブジェクトタイプを作成し、プロセスに送信して平均を計算します。

12.2.1.4. 情報の取得のソリューション

リクエストを使用してデータを取得する方法と、この場合に部分的に、それをどう処理するかに関するソリューションです。

```
r = requests.get(https://lucasenard.github.io/Data/patients.json)
data = r.json()
for key,val in data.items():
    ...
```

繰り返しますが、オンライン Python Web サイトまたはローカルの Python
ファイルでは、key、val、およびタイプを出力し、それらを使用して何をできるかを理解することが可能です。
json.dumps(val) を使用して val を格納してから、SendRequest
の後にプロセスにいるときに、json.loads(request.patient.infos) を使用してその val を取得します（val の情報を patient.infos に格納した場合）。

12.2.2. bp

12.2.2.1. 平均歩数と dict

statistics は、算術演算を行うために使用できるネイティブライブラリです。

12.2.2.2. 平均歩数と dict: ヒント

Python のネイティブ map 関数では、たとえばリスト内または辞書内の情報を分離することができます。

list ネイティブ関数を使用して、map の結果をリストに変換し直すことを忘れないようにしましょう。

12.2.2.3. 平均歩数と dict: map を使用する

オンライン Python Web サイトまたはローカルの Python
ファイルを使用して、以下のように、リストのリストまたは辞書のリストの平均を計算することができます。

```
l1 = [[0,5],[8,9],[5,10],[3,25]]
l2 = [["info",12],["bidule",9],[3,3],["patient1",90]]
l3 = [{"info1":"7","info2":0},{"info1":"15","info2":0},{"info1":"27","info2":0},{"inf
o1":"7","info2":0}]

#???????????????0/8/5/3?
avg_l1_0 = statistics.mean(list(map(lambda x: x[0]),l1))

#??????? 2 ???????5/9/10/25?
avg_l1_1 = statistics.mean(list(map(lambda x: x[1]),l1))

#12/9/3/90 ???
avg_l2_1 = statistics.mean(list(map(lambda x: x[1]),l2))

#7/15/27/7 ???
avg_l3_info1 = statistics.mean(list(map(lambda x: int(x["info1"])),l3))

print(avg_l1_0)
print(avg_l1_1)
print(avg_l2_1)
print(avg_l3_info1)
```

12.2.2.4. 平均歩数と dict: 解答

リクエストに、日付と歩数の dict の json.dumps である infos
属性を持つ患者が保持されている場合、以下のようにして平均歩数を計算できます。

```
statistics.mean(list(map(lambda x: int(x['steps']),json.loads(request.patient.infos))
))
```

12.2.3. bo

bo.FileOperation.WriteFormation に非常に似たものを使用できます。

bo.FileOperation.WritePatient のようなものです。

12.3. ソリューション

12.3.1. obj と msg

obj.py に以下を追加できます。

```
@dataclass
class Patient:
    name:str = None
    avg:int = None
    infos:str = None
```

msg.py に以下を追加できます。
インポート:

```
from obj import Formation,Training,Patient
```

コード:

```
@dataclass
class PatientRequest(Message):
    patient:Patient = None
```

この情報を単一の obj に保持し、get リクエストから得る dict の str を直接 infos 属性に入れます。
平均は、プロセスで計算されます。

12.3.2. bs

bs.py に以下を追加できます。 インポート:

```
import requests
```

コード:

```
class PatientService(BusinessService):

    def get_adapter_type():
        """
```

```
        ???????????
        """
        return "Ens.InboundAdapter"

    def on_init(self):
        """
        API ???????????????? target ??????????????????
        target ??????????????'Python.PatientProcess' ???????
        API ???? api_url ???????? taget ???????? api_url ???????
        api_url ??????????????'https://lucasenard.github.io/Data/patients.json' ????
???
        :return: None
        """
        if not hasattr(self,'target'):
            self.target = 'Python.PatientProcess'
        if not hasattr(self,'api_url'):
            self.api_url = "https://lucasenard.github.io/Data/patients.json"
        return None

    def on_process_input(self,request):
        """
        API ????????????????????? Patient ??????????
        ???????????

        :param request: ??????????????
        :return: None
        """
        req = requests.get(self.api_url)
        if req.status_code == 200:
            dat = req.json()
            for key,val in dat.items():
                patient = Patient()
                patient.name = key
                patient.infos = json.dumps(val)
                msg = PatientRequest()
                msg.patient = patient
                self.send_request_sync(self.target,msg)
        return None
```

ターゲットと api url 変数を作成します（oninit を参照）。
requests.get が req 変数に情報を入れた後、json で情報を抽出する必要があります。これにより dat が dict になります。
dat.items を使用して、患者とその情報を直接イテレートできます。
次に、patient オブジェクトを作成し、json データを文字列に変換する json.dumps を使用して val を文字列に変換し、patient.infos 変数に入れます。
次に、プロセスを呼び出す msg.PatientRequest の msg リクエストを作成します。

コンポーネントの登録を忘れずに行いましょう。 5.4. に従い、以下を使用します。

```
register_component("bs","PatientService","/irisdev/app/src/python/",1,"Python.Patient
Service")
```

12.3.3. bp

bp.py に以下を追加できます。 インポート:

```
import statistic
```

コード:

```
class PatientProcess(BusinessProcess):

    def on_request(self, request):
        """
        ?????????PatientRequest ????????????????????????????????
        ?????? Python.FileOperation ??????????

        :param request: ??????????????????????
        :return: None
        """
        if isinstance(request,PatientRequest):
            request.patient.avg = statistics.mean(list(map(lambda x: int(x['steps']),
json.loads(request.patient.infos))))
            self.send_request_sync('Python.FileOperation',request)

        return None
```

取得したばかりのリクエストを取り、PatientRequest である場合は、歩数の平均を計算して、FileOperation に送信します。 これは患者の avg 変数に正しい情報を挿入します（詳細は、bp のヒントを参照してください）。

コンポーネントの登録を忘れずに行いましょう。 5.4. に従い、以下を使用します。

```
register_component("bp","PatientProcess","/irisdev/app/src/python/",1,"Python.Patient
Process")
```

12.3.4. bo

bo.py に以下を追加できます。FileOperation クラス内:

```
    def write_patient(self, request:PatientRequest):
        """
        ??????????????????????

        :param request: ??????????
        :type request: PatientRequest
        :return: None
        """
        name = ""
        avg = 0
        if request.patient is not None:
            name = request.patient.name
            avg = request.patient.avg
        line = name + " avg nb steps : " + str(avg) +"\n"
        filename = 'Patients.csv'
        self.put_line(filename, line)
        return None
```

前に説明したとおり、FileOperation は以前に登録済みであるため、もう一度登録する必要はありません。

12.4. テスト

7.4. を参照して、オペレーションを追加しましょう。

9.2. を参照して、サービスを追加しましょう。

次に、管理ポータルに移動し、前と同じように行います。 新しいサービスは、InboundAdapter を追加したため、自動的に実行されます。

toto.csv と同様にして、Patients.csv を確認してください。

12.5. グローバル演習のまとめ

この演習を通じて、メッセージ、サービス、プロセス、およびオペレーションの作成を学習し、理解することができました。
Python で情報をフェッチする方法とデータに対して単純なタスクを実行する方法を発見しました。

すべての完成ファイルは、GitHub の solution ブランチにあります。

13. まとめ

このフォーメーションを通じ、csv ファイルから行を読み取り、読み取ったデータを db-api を使ってローカル txt、IRIS データベース、および外部データベースに保存できる IrisPython のみを使用して、フル機能の本番環境を作成しました。
また、POST 動詞を使用して新しいオブジェクトを保存する REST サービスも追加しました。

InterSystems のインターオペラビリティフレームワークの主要要素を発見しました。

これには、Docker、VSCode、および InterSystems の IRIS 管理ポータルを使用して実行しました。

#API #Embedded Python #Python #データベース #フレームワーク #ベストプラクティス #Ensemble #InterSystems IRIS #VSCode #学習ポータル

ソースURL:
https://jp.community.intersystems.com/post/python-%E3%81%AE%E3%81%BF%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9F-intersystems-%E3%81%AE%E3%82%A4%E3%83%B3%E3%82%BF%E3%83%BC%E3%82%AA%E3%83%9A%E3%83%A9%E3%83%93%E3%83%AA%E3%83%86%E3%82%A3%E3%83%95%E3%83%AC%E3%83%BC%E3%83%A0%E3%83%AF%E3%83%BC%E3%82%AF