Article
Guillaume Rongier · Mars 16, 2022 21m de lecture

Open Exchange

Formation Ensemble / Interoperability

Cette formation s'adresse aux débutants qui souhaitent découvrir le framework IRIS Interoperability. Nous utiliserons Docker et VSCode.

GitHub : https://github.com/grongierisc/formation-template

1. Formation Ensemble / Interoperability

Le but de cette formation est d'apprendre le cadre d'interopérabilité d'InterSystems, et en particulier l'utilisation de :
* Productions
* Messages
* Opérations commerciales
* Adaptateurs
* Processus métier
* Services commerciaux
* Services et opérations REST

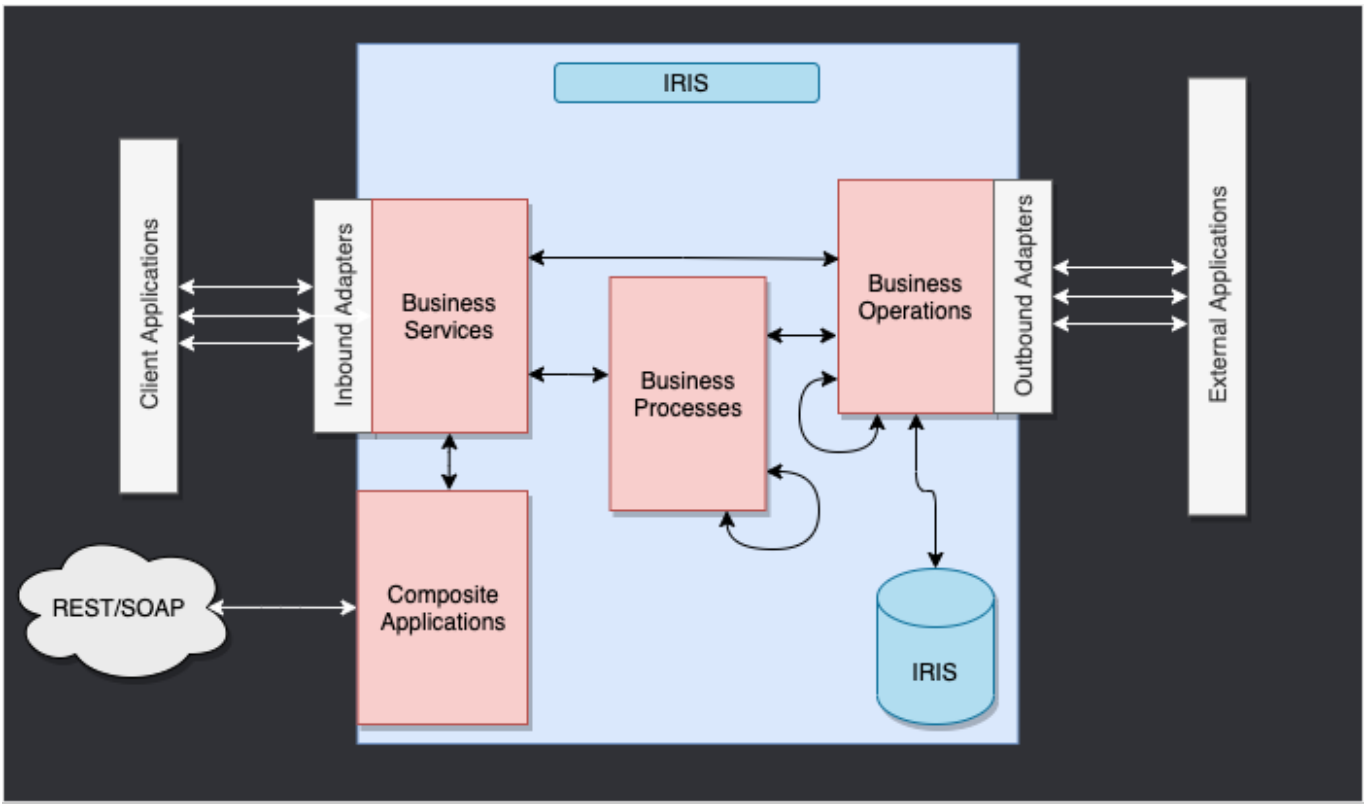
TABLE DES MATIÈRES :

- 1. **Ensemble / Formation Interoperability **
- 2. Framework
- 3. Adapter le framework
- 4. Prérequis
- 5. Mise en place
    - 5.1. Conteneurs Docker
    - 5.2. Portail de gestion
    - 5.3. Sauvegarde de la progression
- 6. Productions
- 7. Opérations
    - 7.1. Création de notre classe de stockage
    - 7.2. Création de notre classe de messages
    - 7.3. Création de notre opération
    - 7.4. Ajout de l'opération à la production
    - 7.5. Test
- 8. Processus métier
    - 8.1. BP simple
    - 8.1.1. Création du processus
    - 8.1.2. Modifier le contexte d'un BP
    - 8.2. BP lisant les lignes CSV
    - 8.2.1. Création d'une carte d'enregistrement
    - 8.2.2. Création d'une transformation de données
    - 8.2.3. Ajout de la transformation des données au processus métier
    - 8.2.4. Configuration de la production
    - 8.2.5. Test de
- 9. Accéder à une base de données externe à l'aide de JDBC
    - 9.1. Création de notre nouvelle opération
    - 9.2. Configurer la production

- 9.3. Test
- 9.4. Exercice
- 9.5. Solution
- 10. Service REST
  - 10.1. Créer le service
  - 10.2. Ajouter notre BS
  - 10.3. Tester
- Conclusion

2. Cadre de travail

Voici le Framework IRIS.



Les composants à l'intérieur d'IRIS représentent une production. Les adaptateurs entrants et les adaptateurs sortants nous permettent d'utiliser différents types de formats comme entrée et sortie pour notre base de données. Les applications composites nous donneront accès à la production via des applications externes comme les services REST.

Les flèches entre tous ces composants sont des **messages**. Ils peuvent être des demandes ou des réponses.

3. Adapter le framework

Dans notre cas, nous allons lire des lignes dans un fichier csv et les enregistrer dans la base de données IRIS.

Nous ajouterons ensuite une opération qui nous permettra de sauvegarder également des objets dans une base de données externe, en utilisant JDBC. Cette base de données sera située dans un conteneur docker, en utilisant postgre.

Enfin, nous verrons comment utiliser des applications composites pour insérer de nouveaux objets dans notre base de données ou pour consulter cette base (dans notre cas, via un service REST).

Le framework adapté à notre objectif nous donne :

4. Prérequis

Pour cette formation, vous aurez besoin de :
* VSCode : https://code.visualstudio.com/
* La suite d'addons InterSystems pour vscode : https://intersystems-community.github.io/vscode-objectscript/installation/
* Docker : https://docs.docker.com/get-docker/
* L'addon docker pour VSCode.

5. Mise en place

5.1. Conteneurs Docker

Afin d'avoir accès aux images InterSystems, nous devons nous rendre à l'url suivante : http://container.intersystems.com. Après nous être connectés avec nos identifiants InterSystems, nous obtiendrons notre mot de passe pour nous connecter au registre. Dans l'addon docker VScode, dans l'onglet image, en appuyant sur connect registry et en entrant la même url que précédemment (http://container.intersystems.com) comme registre générique, il nous sera demandé de donner nos identifiants. L'identifiant est l'identifiant habituel mais le mot de passe est celui que nous avons obtenu sur le site Web.

De là, nous devrions pouvoir construire et composer nos conteneurs (avec les fichiers docker-compose.yml et Dockerfile donnés).

5.2. Portail de gestion

Nous allons ouvrir un portail de gestion. Il nous donnera accès à une page web où nous pourrons créer notre production. Le portail devrait être situé à l'url : http://localhost:52775/csp/sys/UtilHome.csp ?$NAMESPACE=IRISAPP. Vous aurez besoin des informations d'identification suivantes :

IDENTIFIANT : SuperUser

MOT DE PASSE : SYS

5.3. Sauvegarde de la progression

Une partie des choses que nous allons faire sera sauvegardée localement, mais tous les processus et productions sont sauvegardés dans le conteneur docker. Afin de persister tout notre progrès, nous devons exporter chaque classe qui est créée par le portail de gestion avec l'addon InterSystems ObjectScript :

Nous devrons sauvegarder notre production, le plan d'enregistrement, les processus métier et le transfert de données de cette manière. Après cela, lorsque nous fermerons notre conteneur docker et le recomposerons, nous aurons toujours toute notre progression sauvegardée localement (c'est, bien sûr, à faire après chaque changement via le portail). Pour le rendre à nouveau accessible à IRIS, nous devons compiler les fichiers exportés (en les sauvegardant, les addons InterSystems s'occupent du reste).

6. Productions

Nous pouvons maintenant créer notre première production. Pour cela, nous allons passer par les menus [Interoperability] et [Configurer] :

Nous devons ensuite appuyer sur [Nouveau], sélectionner le paquet [Formation] et choisir un nom pour notre production :

Immédiatement après avoir créé notre production, nous devons cliquer sur [Paramètres de production] juste au-dessus de la section [Opérations]. Dans le menu latéral de droite, nous devrons activer [Test activé] dans la partie [Développement et débogage] de l'onglet [Paramètres] (n'oubliez pas de cliquer sur [Appliquer]).

Dans cette première production, nous allons maintenant ajouter des opérations commerciales.

7. Opérations

Une Business Operation (BO) est une opération spécifique qui nous permettra d'envoyer des requêtes depuis IRIS vers une application / un système externe. Elle peut également être utilisée pour enregistrer directement dans IRIS ce que nous voulons.

Nous allons créer ces opérations en local, c'est-à-dire dans le fichier Formation/BO/. En sauvegardant les fichiers, nous les compilerons dans IRIS.

Pour notre première opération, nous allons sauvegarder le contenu d'un message dans la base de données locale.

Nous devons d'abord avoir un moyen de stocker ce message.

7.1. Création de notre classe de stockage

Les classes de stockage dans IRIS étendent le type %Persistent. Elles seront enregistrées dans la base de données interne.

Dans notre fichier Formation/Table/Formation.cls nous avons :

```
Class Formation.Table.Formation Extends %Persistent
{

Property Name As %String;

Property Salle As %String;

}
```

Notez que lors de l'enregistrement, des lignes supplémentaires sont automatiquement ajoutées au fichier. Elles sont obligatoires et sont ajoutées par les addons InterSystems.

7.2. Création de notre classe de messages

Ce message contient un objet Formation, situé dans le fichier Formation/Obj/Formation.cls :

```
Class Formation.Obj.Formation Extends (%SerialObject, %XML.Adaptor)
{

Property Nom As %String;

Property Salle As %String;
```

```
}
```

La classe Message utilisera cet objet Formation, src/Formation/Msg/FormationInsertRequest.cls :

```
Class Formation.Msg.FormationInsertRequest Extends Ens.Request
{

Property Formation As Formation.Obj.Formation;

}
```

7.3. Création de notre opération

Maintenant que nous avons tous les éléments dont nous avons besoin, nous pouvons créer notre opération, dans le fichier Formation/BO/LocalBDD.cls :

```
Class Formation.BO.LocalBDD Extends Ens.BusinessOperation
{

Parameter INVOCATION = "Queue";

Method InsertLocalBDD(pRequest As Formation.Msg.FormationInsertRequest, Output pRespo
nse As Ens.StringResponse) As %Status
{
    set tStatus = $$$OK

    try{
        set pResponse = ##class(Ens.Response).%New()
        set tFormation = ##class(Formation.Table.Formation).%New()
        set tFormation.Name = pRequest.Formation.Nom
        set tFormation.Salle = pRequest.Formation.Salle
        $$$ThrowOnError(tFormation.%Save())
    }
    catch exp
    {
        Set tStatus = exp.AsStatus()
    }

    Quit tStatus
}

XData MessageMap
{
<MapItems>
    <MapItem MessageType="Formation.Msg.FormationInsertRequest">
        <Method>InsertLocalBDD</Method>
    </MapItem>
</MapItems>
}

}
```

La MessageMap nous donne la méthode à lancer en fonction du type de demande (le message envoyé à

l'opération).

Comme nous pouvons le voir, si l'opération a reçu un message du type Formation.Msg.FormationInsertRequest, la méthode InsertLocalBDD sera appelée. Cette méthode enregistrera le message dans la base de données locale d'IRIS.

7.4. Ajout de l'opération à la production

Nous devons maintenant ajouter cette opération à la production. Pour cela, nous utilisons le portail de gestion. En appuyant sur le signe [+] à côté de [Opérations], nous avons accès à l'[Assistant d'opérations commerciales]. Là, nous choisissons la classe d'opération que nous venons de créer dans le menu déroulant.

9.3. Test

Un double clic sur l'opération nous permettra de l'activer. Après cela, en sélectionnant l'opération et en allant dans les onglets [Actions] dans le menu latéral droit, nous devrions être en mesure de tester l'opération (si ce n'est pas le cas, consultez la partie création de la production pour activer les tests / vous devrez peut-être démarrer la production si elle est arrêtée).

En faisant cela, nous allons envoyer à l'opération un message du type que nous avons déclaré plus tôt. Si tout se passe bien, les résultats devraient être comme indiqué ci-dessous :

L'affichage de la trace visuelle nous permettra de voir ce qui s'est passé entre les processus, les services et les opérations. Ici, nous pouvons voir le message envoyé à l'opération par le processus, et l'opération renvoyant une réponse (qui est juste une chaîne vide).

8. Processus métier

Les processus métier (BP) sont la logique métier de notre production. Ils sont utilisés pour traiter les demandes ou relayer ces demandes à d'autres composants de la production.

Les processus métier sont créés dans le portail de gestion :

8.1. BP simple

8.1.1. Création du processus

Nous sommes maintenant dans le concepteur de processus métier. Nous allons créer un BP simple qui appellera notre opération :

8.1.2. Modifier le contexte d'un BP

Un BP possède un **Contexte**. Il est composé d'une classe de requête, la classe de l'entrée, et d'une classe de réponse, la classe de la sortie. **Les processus métier n'ont qu'une entrée et une sortie**. Il est également possible d'ajouter des propriétés.

Puisque notre BP ne sera utilisé que pour appeler notre BO, nous pouvons mettre comme classe de requête la classe de message que nous avons créée (nous n'avons pas besoin d'une sortie puisque nous voulons juste insérer dans la base de données).

Nous avons ensuite choisi la cible de la fonction d'appel : notre BO. Cette opération, étant **appelée** a une propriété **callrequest**. Nous devons lier cette callrequest à la requête de la BP (elles sont toutes deux de la classe Formation.Msg.FormationInsertRequest), nous faisons cela en cliquant sur la fonction d'appel et en utilisant le constructeur de requête :

Nous pouvons maintenant sauvegarder cette BP (dans le package « Formation.BP » et sous le nom « InsertLocalBDD » ou « Main », par exemple). Tout comme les opérations, les processus peuvent être instanciés et testés via la configuration de production, pour cela ils doivent être compilés au préalable (sur l'écran Concepteur de processus métier).

Pour l'instant, notre processus ne fait que transmettre le message à notre opération. Nous allons le complexifier pour que le BP prenne en entrée une ligne d'un fichier CSV.

8.2. Lecture de lignes CSV par un BP

8.2.1. Création d'une carte d'enregistrement

Afin de lire un fichier et de mettre son contenu dans un fichier, nous avons besoin d'une Carte d'enregistrement (RM). Il existe un Record Mapper spécialisé pour les fichiers CSV dans le menu [Interoperability > Build] du portail de gestion :

Nous créerons le mapper comme suit :

Nous allons créer le mappeur comme ceci :

Maintenant que la carte est créée, nous devons la générer (avec le bouton Générer). Nous devons maintenant avoir une Transformation de données à partir du format de la carte des enregistrements et un message d'insertion.

8.2.2. Création d'une transformation de données

Nous trouverons le créateur de transformations de données (DT) dans le menu [Interoperability > Builder]. Nous allons créer notre DT comme ceci (si vous ne trouvez pas Formation.RM.Csv.Record, peut-être que vous n'avez pas généré la carte d'enregistrement) :

Nous pouvons pouvons maintenant mapper les différents champs ensemble :

8.2.3. Ajout de la transformation des données au processus métier

La première chose que nous devons changer est la classe de requête de la BP, puisque nous devons avoir en entrée la carte d'enregistrement que nous avons créée.

Nous pouvons alors ajouter notre transformation (le nom du processus ne change rien, d'ici nous avons choisi de le nommer Main) :

L'activité de transformation va prendre la requête de la BP (un enregistrement du fichier CSV, grâce à notre Record Mapper), et la transformer en un message FormationInsertRequest. Afin de stocker ce message pour l'envoyer à la BO, nous devons ajouter une propriété au contexte de la BP.

Nous pouvons maintenant configurer notre fonction de transformation pour qu'elle prenne l'entrée du BP et enregistre sa sortie dans la propriété nouvellement créée. La source et la cible de la transformation RmToMsg sont respectivement request et context.Msg :

Nous devons faire de même pour Call BO. Son entrée, ou callrequest, est la valeur stockée dans context.msg :

Au final, le flux dans la BP peut être représenté comme ceci :

8.2.4. Configuration de la production

Avec le signe [+], nous pouvons ajouter notre nouveau processus à la production (si ce n'est pas déjà fait). Nous avons également besoin d'un service générique pour utiliser le record map, nous utilisons

EnsLib.RecordMap.Service.FileService (nous l'ajoutons avec le bouton [+] à côté des services). Nous paramétrons ensuite ce service :

Nous devrions maintenant être en mesure de tester notre BP.

8.2.5. Test

Nous testons l'ensemble de la production comme suit :

Dans le menu Explorateur système > SQL, vous pouvez exécuter la commande

```
SELECT
ID, Name, Salle
FROM Formation_Table.Formation
```

pour voir les objets que nous venons d'enregistrer.

9. Accéder à une base de données externe à l'aide de JDBC

Dans cette section, nous allons créer une opération pour sauvegarder nos objets dans une base de données externe. Nous utiliserons l'API JDBC, ainsi que l'autre conteneur docker que nous avons mis en place, avec postgre dessus.

9.1. Création de notre nouvelle opération

Notre nouvelle opération, dans le fichier Formation/BO/RemoteBDD.cls est la suivante :

```
Include EnsSQLTypes

Class Formation.BO.RemoteBDD Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.SQL.OutboundAdapter";

Property Adapter As EnsLib.SQL.OutboundAdapter;

Parameter INVOCATION = "Queue";

Method InsertRemoteBDD(pRequest As Formation.Msg.FormationInsertRequest, Output pResp
onse As Ens.StringResponse) As %Status
{
    set tStatus = $$$OK

    try{
        set pResponse = ##class(Ens.Response).%New()
        set ^inc = $I(^inc)
        set tInsertSql = "INSERT INTO public.formation (id, nom, salle) VALUES(?, ?,
?)"
        $$$ThrowOnError(..Adapter.ExecuteUpdate(.nrows,tInsertSql,^inc,pRequest.Forma
tion.Nom, pRequest.Formation.Salle ))
    }
    catch exp
    {
```

```
        Set tStatus = exp.AsStatus()
    }

    Quit tStatus
}

XData MessageMap
{
<MapItems>
    <MapItem MessageType="Formation.Msg.FormationInsertRequest">
        <Method>InsertRemoteBDD</Method>
    </MapItem>
</MapItems>
}

}
```

Cette opération est similaire à la première que nous avons créée. Lorsqu'elle recevra un message du type Formation.Msg.FormationInsertRequest, elle utilisera un adaptateur pour exécuter des requêtes SQL. Ces requêtes seront envoyées à notre base de données postgre.

9.2. Configurer la production

Maintenant, via le portail de gestion, nous allons instancier cette opération (en l'ajoutant avec le signe [+] dans la production).

Nous devrons également ajouter le JavaGateway pour le pilote JDBC dans les services. Le nom complet de ce service est EnsLib.JavaGateway.Service.

Nous devons maintenant configurer notre opération. Puisque nous avons mis en place un conteneur postgre, et connecté son port 5432, les valeurs dont nous avons besoin dans les paramètres suivants sont :

DSN : jdbc:postgresql://db:5432/DemoData

Pilote JDBC : org.postgresql.Driver

JDBC Classpath : /tmp/iris/postgresql-42.2.14.jar

Enfin, nous devons configurer les informations d'identification pour avoir accès à la base de données distante. Pour cela, nous devons ouvrir la visionneuse d'identifiants :

L'identifiant et le mot de passe sont tous deux DemoData, comme nous l'avons configuré dans le fichier docker-compose.yml.

De retour à la production, nous pouvons ajouter « Postgre » dans le champ [Identifiant] dans les paramètres de notre opération (il devrait être dans le menu déroulant). Avant de pouvoir le tester, nous devons ajouter le JGService à l'opération. Dans l'onglet [Paramètres], dans les [Paramètres supplémentaires] :

7.5. Test

Lors du test, la trace visuelle doit indiquer un succès :

Lors du test, la trace visuelle doit montrer un succès :

9.4. Exercice

À titre d'exercice, il pourrait être intéressant de modifier BO.LocalBDD afin qu'il renvoie un booléen qui indiquera au BP d'appeler BO.RemoteBDD en fonction de la valeur de ce booléen.

Indice : cela peut être fait en changeant le type de réponse que LocalBDD renvoie et en ajoutant une nouvelle propriété au contexte et en utilisant l'activité if dans notre BP.

9.5. Solution

Tout d'abord, nous devons avoir une réponse de notre opération LocalBDD. Nous allons créer un nouveau message, dans le fichier Formation/Msg/FormationInsertResponse.cls :

```
Class Formation.Msg.FormationInsertResponse Extends Ens.Response
{

Property Double As %Boolean;

}
```

Ensuite, nous modifions la réponse de LocalBDD par cette réponse, et définissons la valeur de son booléen de manière aléatoire (ou non) :

```
Method InsertLocalBDD(pRequest As Formation.Msg.FormationInsertRequest, Output pRespo
nse As Formation.Msg.FormationInsertResponse) As %Status
{
    set tStatus = $$$OK

    try{
        set pResponse = ##class(Formation.Msg.FormationInsertResponse).%New()
        if $RANDOM(10) < 5 {
            set pResponse.Double = 1
        }
        else {
            set pResponse.Double = 0
        }
...
```

Nous allons maintenant créer un nouveau processus (copié sur celui que nous avons fait), où nous ajouterons une nouvelle propriété de contexte, de type %Boolean :

Cette propriété sera remplie avec la valeur du callresponse.Double de notre appel d'opération (nous devons définir la [Classe de message de réponse] à notre nouvelle classe de message) :

Nous ajoutons ensuite une activité if, avec la propriété context.Double comme condition :

TRÈS IMPORTANT : nous devons décocher **Asynchrone** dans les paramètres de notre appel LocallBDD, ou l'activité if se déclenchera avant de recevoir la réponse booléenne.

Enfin, nous configurons notre activité d'appel avec comme cible le BO RemoteBDD :

Pour compléter l'activité if, nous devons faire glisser un autre connecteur de la sortie du if vers le triangle join ci-

dessous. Comme nous ne ferons rien si le booléen est faux, nous laisserons ce connecteur vide. Après avoir compilé et instancié, nous devrions être en mesure de tester notre nouveau processus. Pour cela, nous devons changer le Target Config Name de notre File Service.

Dans la trace, nous devrions avoir environ la moitié des objets lus dans le csv enregistrés également dans la base de données distante.

10. Service REST

Dans cette partie, nous allons créer et utiliser un service REST.

10.1. Créer le service

Pour créer un service REST, nous avons besoin d'une classe qui étend %CSP.REST, dans Formation/REST/Dispatch.cls nous avons :

```
Class Formation.REST.Dispatch Extends %CSP.REST
{

/// Ignore any writes done directly by the REST method.
Parameter IgnoreWrites = 0;

/// Convertir par défaut le flux d'entrée en Unicode
Parameter CONVERTINPUTSTREAM = 1;

/// Le jeu de caractères par défaut est utf-8
Parameter CHARSET = "utf-8";

Parameter HandleCorsRequest = 1;

XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <!-- Get this spec -->
  <Route Url="/import" Method="post" Call="import" />
</Routes>
}

/// Obtenir cette spécification
ClassMethod import() As %Status
{
  set tSc = $$$OK

  Try {

      set tBsName = "Formation.BS.RestInput"
      set tMsg = ##class(Formation.Msg.FormationInsertRequest).%New()

      set body = $zcvt(%request.Content.Read(),"I","UTF8")
      set dyna = {}.%FromJSON(body)

      set tFormation = ##class(Formation.Obj.Formation).%New()
      set tFormation.Nom = dyna.nom
      set tFormation.Salle = dyna.salle
```

```
      set tMsg.Formation = tFormation

      $$$ThrowOnError(##class(Ens.Director).CreateBusinessService(tBsName,.tService))

      $$$ThrowOnError(tService.ProcessInput(tMsg,.output))

  } Catch ex {
      set tSc = ex.AsStatus()
  }

  Quit tSc
}

}
```

Cette classe contient une route pour importer un objet, liée au verbe POST :

```
<Routes>
  <!-- Obtenir cette spécification -->
  <Route Url="/import" Method="post" Call="import" />
</Routes>
```

La méthode d'importation va créer un message qui sera envoyé à un service métier.

10.2. Ajouter notre BS

Nous allons créer une classe générique qui va router toutes ses sollicitations vers TargetConfigNames. Cette cible sera configurée lorsque nous instancierons ce service. Dans le fichier Formation/BS/RestInput.cls nous avons :

```
Class Formation.BS.RestInput Extends Ens.BusinessService
{

Property TargetConfigNames As %String(MAXLEN = 1000) [ InitialExpression = "BuisnessP
rocess" ];

Parameter SETTINGS = "TargetConfigNames:Basic:selector?multiSelect=1&context={Ens.Con
textSearch/ProductionItems?targets=1&productionName=@productionId}";

Method OnProcessInput(pDocIn As %RegisteredObject, Output pDocOut As %RegisteredObjec
t) As %Status
{
    set status = $$$OK

    try {

        for iTarget=1:1:$L(..TargetConfigNames, ",") {
            set tOneTarget=$ZStrip($P(..TargetConfigNames,",",iTarget),"<>W")  Contin
ue:""=tOneTarget
            $$$ThrowOnError(..SendRequestSync(tOneTarget,pDocIn,.pDocOut))
        }
    } catch ex {
        set status = ex.AsStatus()
    }
```

```
    Quit status
}

}
```

De retour à la configuration de production, nous ajoutons le service de la manière habituelle. Dans le champ [Target Config Names], nous mettons notre BO LocalBDD :

Pour utiliser ce service, nous devons le publier. Pour cela, nous utilisons le menu [Modifier l'application Web] :

10.3. Test

Enfin, nous pouvons tester notre service avec n'importe quel type de client REST :

Conclusion

Grâce à cette formation, nous avons créé une production capable de lire les lignes d'un fichier csv et de sauvegarder les données lues à la fois dans la base de données IRIS et dans une base de données externe en utilisant JDBC. Nous avons également ajouté un service REST afin d'utiliser le verbe POST pour sauvegarder de nouveaux objets.

Nous avons découvert les principaux éléments du cadre d'interopérabilité d'InterSystems.

Nous l'avons fait en utilisant docker, vscode et le portail de gestion IRIS d'InterSystems.

#Docker #Débutant #Framework #Interopérabilité #Caché #Ensemble #InterSystems IRIS
Voir l'application sur InterSystems Open Exchange

URL de la source:https://fr.community.intersystems.com/post/formation-ensemble-interoperability