記事
Mihoko Iijima · 2020年10月27日 10m read

# 【はじめてのInterSystems IRIS】Interoperability（相互運用性）：コンポーネントの作成（ビジネス・サービス）

**この記事はこちらの投稿の続きの内容です。**

前回の記事
では、システム統合に必要なコンポーネントの中から、プロダクション内の処理の調整役となるビジネス・プロセスの作成について解説しました。

今回の記事では、プロダクションの情報入力窓口である、ビジネス・サービスの作成について解説します。

- プロダクション
- メッセージ
- **コンポーネント**
  - **ビジネス・サービス**
  - ビジネス・プロセス（前回の記事）
  - ビジネス・オペレーション

いよいよ「Interoperability（相互運用性）を使ってみよう！」の最後のコンポーネントです。

ビジネス・サービスは、IRIS 外部からの送信される情報の入力窓口で、外部 I/F
に対してアダプタを使用する／しないを選択できます。

サンプルでは、3 種類のビジネス・サービスを用意しています（括弧内のリンクはサンプルコードへのリンク）。

1. ファイルインバウンドアダプタを利用したファイル用ビジネス・サービス（Start.FileBS）
2. SOAP インバウンドアダプタを利用する Web サービス用ビジネス・サービス（Start.WS.WebServiceBS）
3. アダプタを使用せず、ストアドプロシージャや REST で呼び出されるビジネス・サービス（Start.NonAdapterBS）

情報入力に使用する接続方法が異なるだけ、ビジネス・サービス数が増えますが、ビジネス・サービス内で行う処理は

**外部から入力された情報を利用して、
送信する要求メッセージを作成し、
ビジネス・コンポーネント を呼び出すだけ**

とてもシンプルです。

それでは、ファイルインバウンドアダプタを使用するコンポーネントから作成方法概要を説明します。

ビジネス・サービスはスクリプトを記述するので、VSCode
かスタジオで作成します（VSCodeの使い方についてはこちらの記事もご参照ください。）。

## 1、ファイルインバウンドアダプタを利用したファイル用ビジネス・サービス（Start.FileBS）

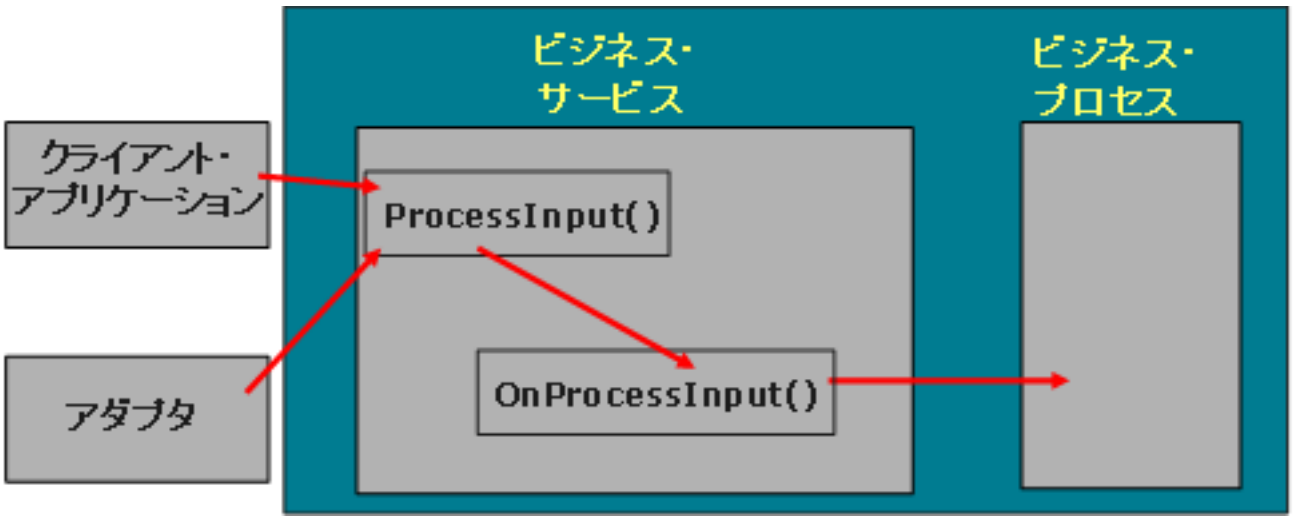
VSCode で作成する場合は、Ens.BusinessService
を継承するクラスを作成します。アダプタについては、使用する場合は、以下のように **ADAPTER**
パラメータにアダプタクラス名をしています（例はファイルインバウンドアダプタのクラスを指定しています）。
アダプタを使用しない場合は、設定は不要です。

```
Class Start.FileBS Extends Ens.BusinessService
{
Parameter ADAPTER = "EnsLib.File.InboundAdapter";
```

ファイルインバウンドアダプタでは、監視対象ディレクトリの指定をプロダクションのビジネス・サービス用設定の「ファイル・パス」で指定できます。

「ファイル・パス」に置かれたファイルが「ファイル・スペック」に指定した情報と合致する場合、ファイルをストリームオブジェクトとしてオープンし、ビジネス・サービスの **ProcessInput()**
を呼び出すときの第1引数に指定します。

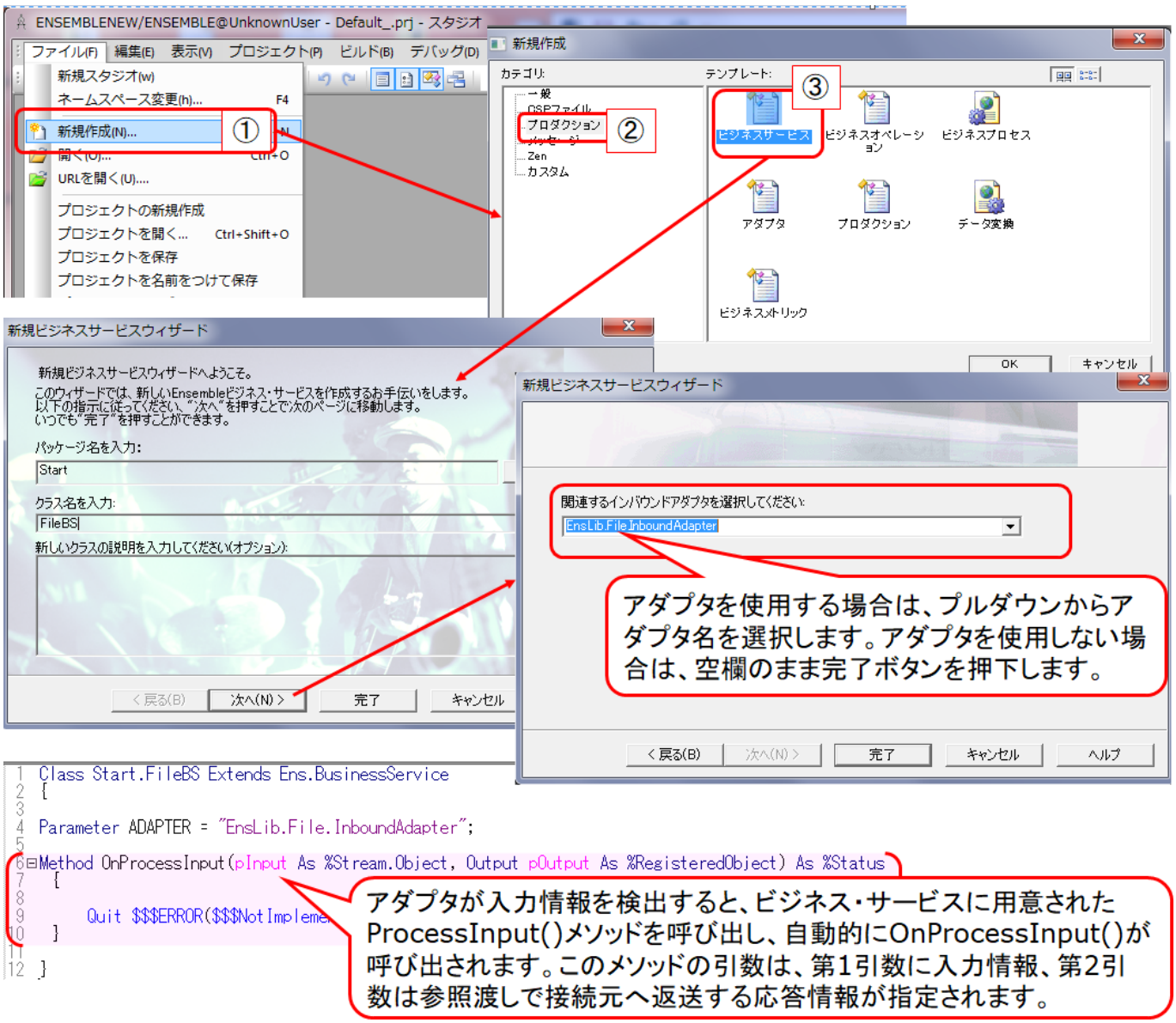
**ProcessInput()** が起動すると自動的に **OnProcessInput()** が呼び出されます。この時、**OnProcessInput()**
には、**ProcessInput()** で渡された引数がそのまま渡ります。




**OnProcessInput()** では、第1引数に渡されるファイルストリームオブジェクトから情報を取得し、次のコンポーネントへ渡すメッセージを作成し、次のコンポーネントを呼び出す処理を記述したら基本ロジックは終了です。

---

【メモ】
スタジオの場合は、新規作成メニューのビジネス・サービスウィザードを起動し、アダプタを選択して完了ボタンを押下します。




---

**OnProcessInput()** のメソッド定義は以下の通りです。

```
Method OnProcessInput(pInput As %Stream.Object, Output pOutput As %
RegisteredObject) As %Status
```

**pInput** には、テキストファイルの場合 %Steam.FileCharacter クラス、バイナリファイルの場合 %Stream.FileBinary クラスのインスタンスが渡されます。

サンプルでは、テキスト形式のファイルが入力される予定で、1依頼1行として、複数行の依頼も受け付けるように記述しています。

End Of File を検知すると1を設定する **AtEnd プロパティ**を利用してループ処理を行います。
ループ内では、ファイルの中身を1行ずつ情報を取得できる **ReadLine()**
**メソッド**を使い、行を読み取ります（ファイルアダプタ詳細は、ドキュメントをご参照ください）。

1行ずつ情報を取得しながらメッセージを作成し、他コンポーネントを呼び出す
**..SendRequestAsync()** メソッドを実行します。

メソッド実行時、第1引数に呼び出したいコンポーネント名を文字列で指定し、第2引数には作成した要求メッセージを指定します。
なお、**..SendRequestAsync()**は非同期呼び出し のため、応答を待ちません。

メモ：同期呼び出し用に SendRequestSync() もあります。

サンプルコードは以下の通りです。

```
13  Method OnProcessInput(pInput As %Stream.Object, Output pOutput As %RegisteredObject) As %Status
14  {
15      set st=$$$OK
16      #dim ex As %Exception.AbstractException
17      try {
18          // AtEndプロパティがに1が設定されない限りループしながら1行ずつ実行する
19          while '(pInput.AtEnd) {
20              set record=pInput.ReadLine()
21              set request=##class(Start.Request).%New()
22              set request.Product=$piece(record,",",1)
23              set request.Area=$piece(record,",",2)
24              set st=..SendRequestAsync("Start.WeatherCheckProcess",request)
25              $$$THROWONERROR(ex,st)
26          }
27      }
28      catch ex {
29          set st=ex.AsStatus()
30      }
31      Quit st
32  }
```

要求メッセージ（Start.Request）を作成します。
サンプルでは、以下の形式で指定される予定です。
**購入商品名,都市名**
カンマ区切りデータの操作に便利な**$piece()**関数を使用します。

他コンポーネントの呼び出しには、各コンポーネントに用意のある**SendRequestAsync()**メソッドを使用します。第1引数にはプロダクションに登録されているコンポーネント名を指定します。第2引数には要求メッセージを指定します。

ご参考：上記例文中の $piece()関数の使い方の解説

**$piece( “ 文字列 ” , ” 区切りマーク ” , ” ポジション番号 ” )**

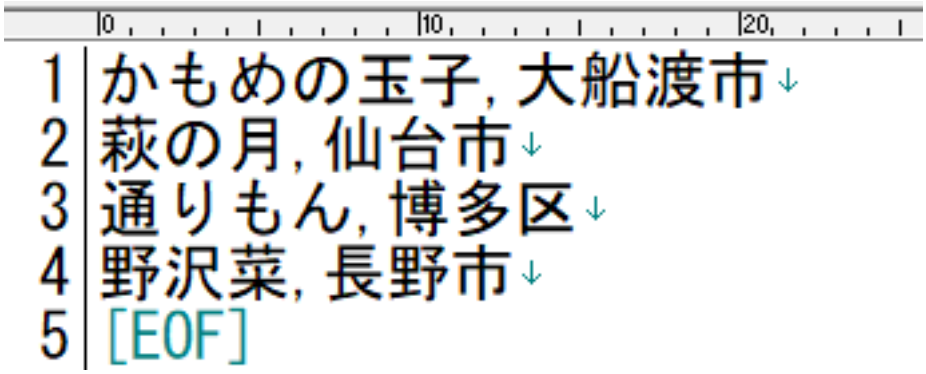
区切りマーク付き文字列の設定／取得が行える関数で、サンプルでは、カンマ区切りデータの1番目と2番目の値を取得するため、以下の構文で記述されています。

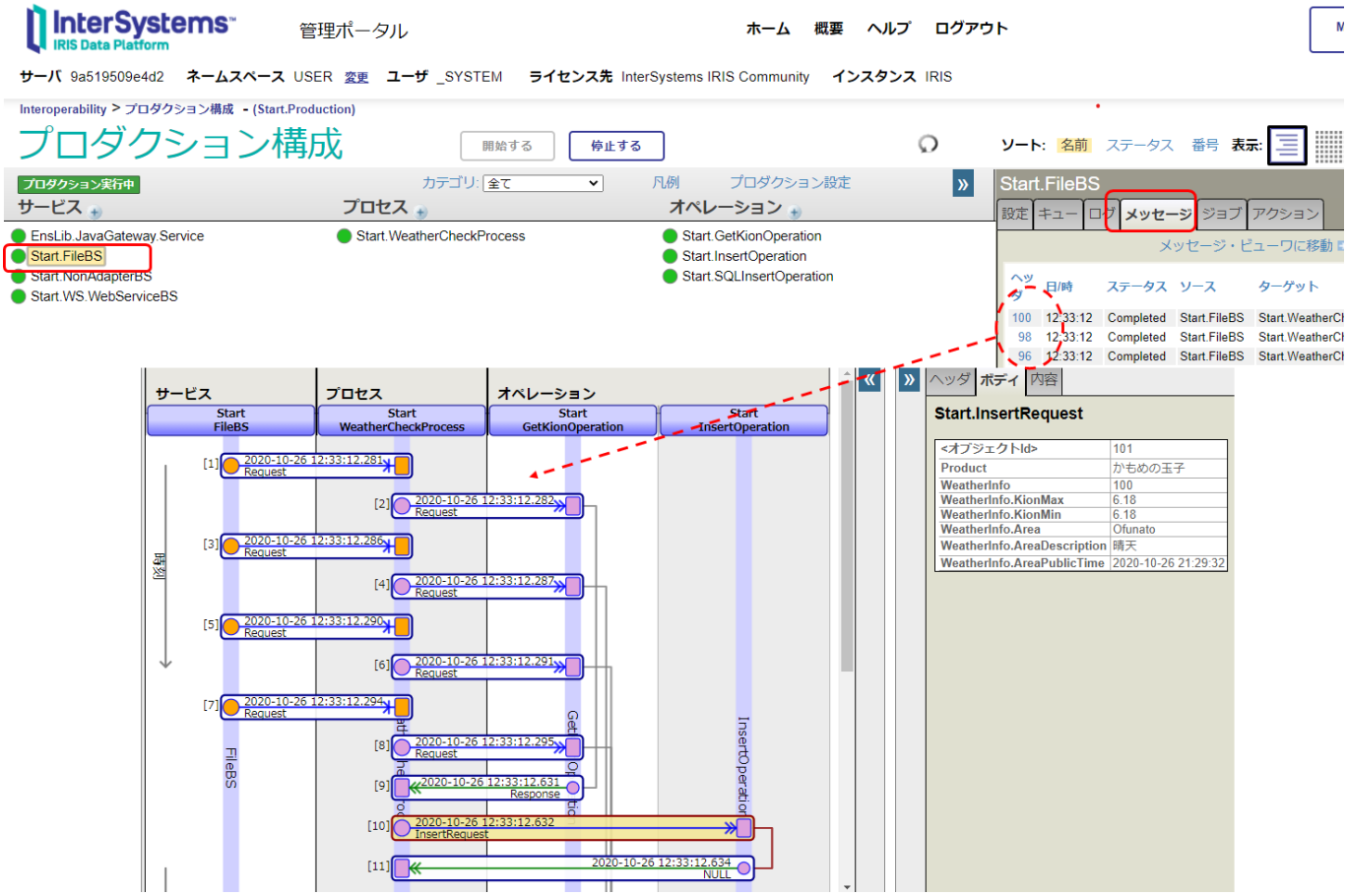
```
set request.Product=$piece(record,",",1)
set request.Area=$piece(record,",",2)
```

では、上記説明に登場した Start.FileBS の動作を確認します。

サンプルプロダクションでは「ファイル・パス」に **/irisdev/src** 「ファイル・スペック」に **check.txt** を指定しています。同じディレクトリをご用意いただくか、別ディレクトリに変更いただき、check.txtファイルに以下の形式（購入商品名,都市名）でサンプルデータを登録してください。

※ サンプルのコンテナを利用されている場合は、git clone で作成したディレクトリ以下の src ディレクトリ以下にある [Test用-check.txt] をリネームしてご利用ください。




## 2、SOAP インバウンドアダプタを利用する Web サービス用ビジネス・サービス（Start.WS.WebServiceBS）

つづいて、Web サービス用ビジネス・サービスの作成概要をご説明します。

Web サービス用ビジネス・サービスクラスは、Web サービス提供側＝ Web サービスサーバとして動作します。

サンプルでは、Web
サービスクライアントから情報を送信してもらうため、今回のサンプルプロダクションに合わせて Web
サービスメソッドに2つの引数を用意しています。Web メソッドでは、引数に入力された情報を利用してメッセージクラスを作成し、他コンポーネントを呼び出しています。

```
2  Class Start.WS.WebServiceBS Extends EnsLib.SOAP.Service
3  {
4
5  /// ウェブサービスの名前.
6  Parameter SERVICENAME = "WebServiceBS";
7
8  /// TODO: これを実際のSOAPネームスペースに変更します。
9  /// ウェブサービス用のSOAPネームスペース
10 Parameter NAMESPACE = "http://tempuri.org";
11
12 /// 参照されているクラスのネームスペースは WSDL に使用されます。
13 Parameter USECLASSNAMESPACES = 1;
14
15 /// TODO: 引数および実装を追加します。
16 /// CallEnsemble
17 ///   第1引数：購入商品名／第2引数：都市名
18 Method CallEnsemble(product As %String, Are As %String = "長野市") As %Status [ WebMethod ]
19 {
20     #dim ex As %Exception.AbstractException
21     set st=$$$OK
22     try {
23         set request=##class(Start.Request).%New()
24         set request.Area=AreCode
25         set request.Product=product
26         set st=..SendRequestAsync("Start.WeatherCheckProcess",request)
27         $$$THROWONERROR(ex,st)
28     }
29     catch ex {
30         set st=ex.AsStatus()
31         do ..ReturnMethodStatusFault(st)
32     }
33     Quit st
34 }
```

Webサービスクラスを作成するため、スーパークラスにEnsLib.SOAP.Service を指定します。

Webメソッドでは、以下の引数を用意しています。
**第1引数：購入商品名、**
**第2引数：都市名**
Webメソッド内でメッセージを作成し、**..SendRequestAsync()** を呼び出し次のコンポーネントへメッセージを送信します。
Webサービス用ビジネス・サービスでは、**OnProcessInput()を実装しなくても**、直接次のコンポーネントを ..SendRequestAsync() で呼び出すことができます。

Web
サービスクラスを定義するとテスト用画面が用意されますが、デフォルトでは表示しない設定になっています。

IRIS へログインし（またはターミナルを起動し）、プロダクションがあるネームスペースに移動して以下実行してください。

ご参考：
https://docs.intersystems.com/irislatestj/csp/docbook/DocBook.UI.Page.cls?KEY=GSOAPservicecatalogpage

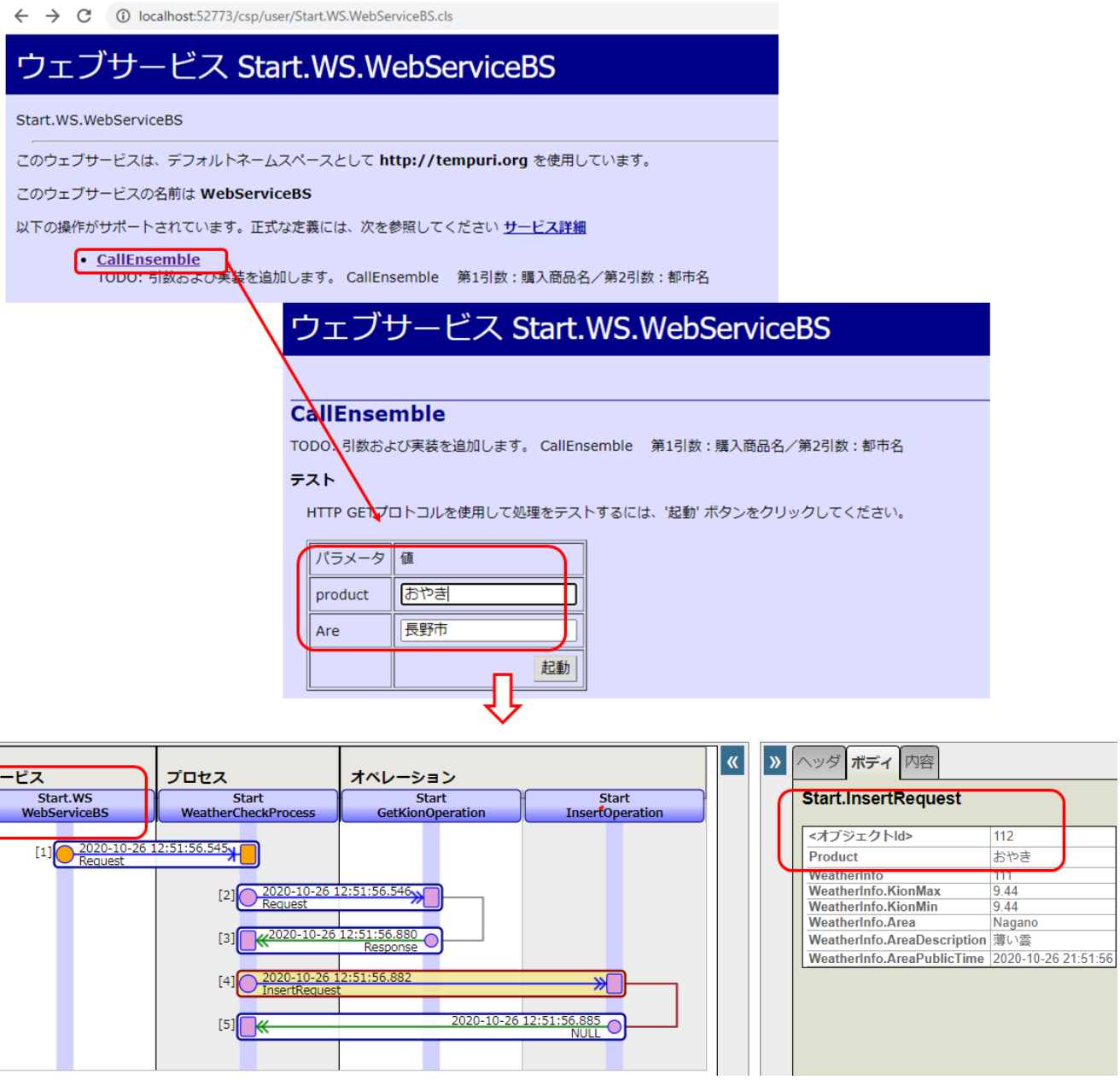
以下、サンプルコードの **docker-compose up -d** でコンテナを開始した環境での設定例です（%SYS ネームスペースで実行します）。

```
set $namespace="%SYS"
set ^SYS("Security","CSP","AllowClass","/csp/user/","%SOAP.WebServiceInfo")=1
set ^SYS("Security","CSP","AllowClass","/csp/user/","%SOAP.WebServiceInvoke")=1
```

**【注意】**
実行文には大文字小文字の区別がありますので、注意して記述してください。また、プロダクションを使用するネームスペースによって指定文字が変わります。例文はサンプルをUSERネームスペースへインポートした前提で記述しています。
もし、サンプルコードをABCネームスペースにインポートしている場合は、第4番目の添え字は "/csp/abc/" を指定します。

設定が完了したら、以下の URL にアクセスします。

http://localhost:52773/csp/user/Start.WS.WebServiceBS.cls




WSDL を Web サービスクライアントへ提示する場合は、以下のURLの末尾に WSDL=1 を指定してください。

http://localhost:52773/csp/user/Start.WS.WebServiceBS.cls?WSDL

## 3,アダプタを使用せず、ストアドプロシージャや REST で呼び出されるビジネス・サービス（Start.NonAdapterBS）

次は、アダプタを使用しないビジネス・サービス（Start.NonAdapterBS）をご紹介します。

```
Class Start.NonAdapterBS Extends Ens.BusinessService
{

Method OnProcessInput(request As Start.Request, Output pOutput As %RegisteredObject) As %Status
{
    set st=$$$OK
    #dim ex As %Exception.AbstractException
    try {
        set st=..SendRequestAsync("Start.WeatherCheckProcess",request)
        $$$THROWONERROR(ex,st)
    }
    catch ex {
        set st=ex.AsStatus()
    }
    Quit st
}

}
```

第1引数にメッセージ用オブジェクトが渡るように定義しています。

アダプタを使用するビジネス・サービスでは、アダプタが情報を検知すると、ビジネス・サービスの ProcessInput() メソッドを呼び出します。

アダプタを使用しない場合も、ProcessInput() メソッドを呼び出せばいいのですが、このメソッドは外部公開していないため、アダプタを使用しないビジネス・サービスを実装する場合、ProcessInput()を実行するための方法を検討する必要があります。

サンプルでは、以下2種類の方法を利用しています。

- ストアドプロシージャ（Start.Utilsクラス）
- REST 用ディスパッチクラス（Start.REST）→ こちらの記事で実行したサービスです。

では、ストアドプロシージャの例をご紹介します。

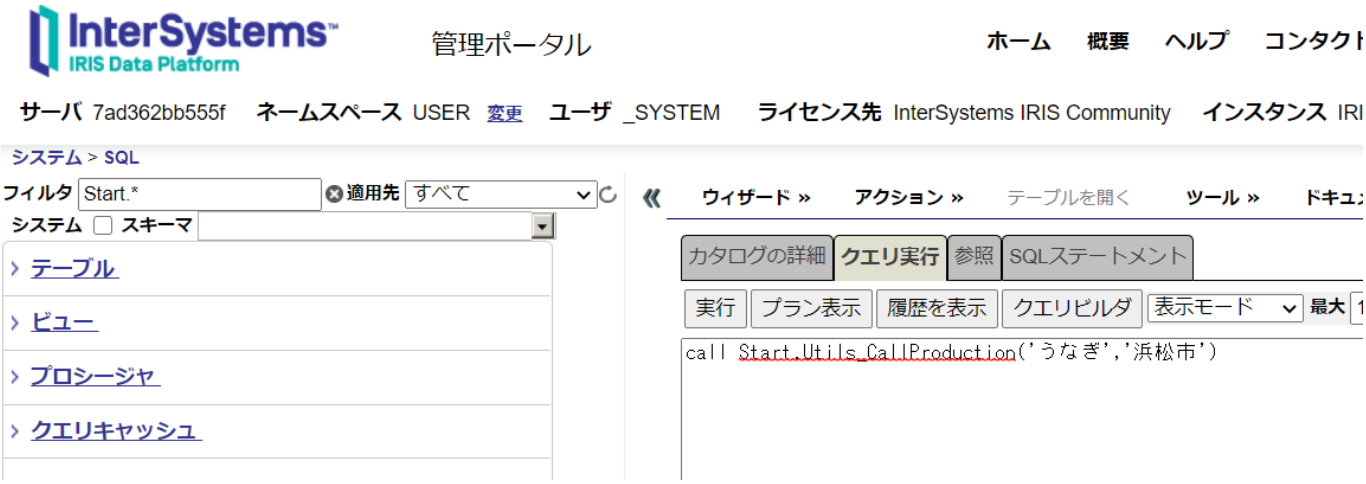
```
ClassMethod CallProduction(product As %String, areacode As %String) As %Status [ SqlProc ]
{
    #dim %sqlcontext As %Library.ProcedureContext
    #dim ex As %Exception.AbstractException
    set st=$$$OK
    try {
        // ビジネスサービスのオブジェクト生成 ：第1引数はビジネスサービスのプロダクション登録名
        set st=##class(Ens.Director).CreateBusinessService("Start.NonAdapterBS",.BS)
        // 戻り値がエラーの場合はcatchへ移動
        $$$THROWONERROR(ex,st)

        set request=##class(Start.Request).%New()
        set request.Product=product
        set request.Area=areacode

        // ビジネスサービスに情報配信
        set st=BS.ProcessInput(request)
        // 戻り値がエラーの場合はcatchへ移動
        $$$THROWONERROR(ex,st)
    }
    catch ex {
        // ストアドプロシージャ呼び出し元へ返す情報の設定
        set %sqlcontext.%SQLCODE=ex.AsSQLCODE()
        set %sqlcontext.%Message=ex.AsSQLMessage()
        set st=ex.AsStatus()
    }
    quit st
}
```

Ens.Directorクラスを使用してビジネス・サービスのインスタンスを生成します。

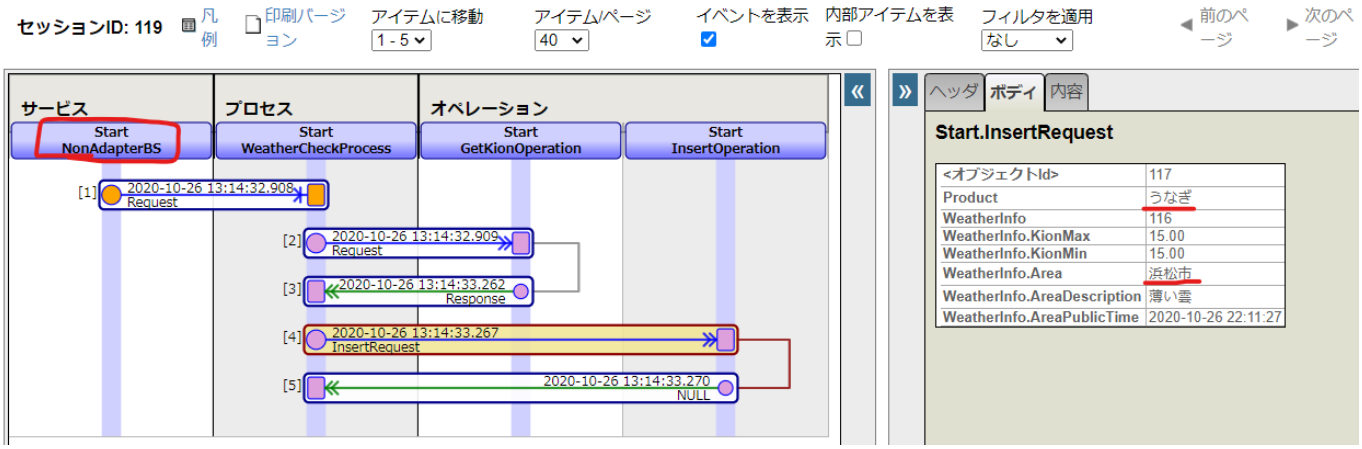
ビジネス・サービスのProcessInput()を呼び出します。

プロダクションにアダプタを使用しないビジネス・サービス（Start.NonAdapterBS）を追加した後で（サンプルでは追加済に状態）以下ストアドプロシージャを実行します。

**call Start.UtilsCallProduction('うなぎ','浜松市')**




実行結果のトレースは以下の通りです。




続いて、REST 用ディスパッチクラス作成例をご紹介します。

```
1  /// RESTを使ってビジネスサービスを呼び出す例
2  Class Start.REST Extends %CSP.REST
3  {
4  XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
5  {
6  <Routes>
7  <Route Url="/weather/:product/:arecode" Method="GET" Call="WeatherCheck"/>
8  </Routes>
9  }
   Debug this method
10 ClassMethod WeatherCheck(product As %String, areacode As %String) As %Status
11 {
12     #dim ex As %Exception.AbstractException
13     #dim %response As %CSP.Response
14     set st=$$$OK
15     try {
16         set %response.ContentType="application/json"
17         set %response.CharSet="utf-8"
18
19         // ビジネスサービスのオブジェクト生成 ：第1引数はビジネスサービスのプロダクション登録名
20         set st=##class(Ens.Director).CreateBusinessService("Start.NonAdapterBS",.BS)
21         // 戻り値がエラーの場合はcatchへ移動
22         $$$THROWONERROR(ex,st)
23         set request=##class(Start.Request).%New()
24         set request.Product=product
25         set request.Area=areacode
26
27         // ビジネスサービスに情報配信
28         set st=BS.ProcessInput(request)
29         // 戻り値がエラーの場合はcatchへ移動
30         $$$THROWONERROR(ex,st)
31         // 以下クライアントへ戻す情報
32         set json={}
33         set json.Message="天気情報確認：依頼しました"
34         set json.Status=1
35         write json.%ToJSON()
36     }
37     catch ex {
38         set st=ex.AsStatus()
39     }
40     quit $$$OK
41 }
```

渡されるURLからどのメソッドを呼び出すか定義します。

Ens.Directorクラスを使用してビジネス・サービスのインスタンスを生成します。

ビジネス・サービスのProcessInput()を呼び出します。

XData Url Map に定義されている XML は、REST 呼び出しのときの URL に対して、どのメソッドが呼び出されるかを定義しています。
サンプルは、**GET 要求で /weather/第1引数（購入商品名）/第2引数（都市名）** の URL が渡されると、**WeatherCheck()** メソッドを呼び出す定義を記述しています。

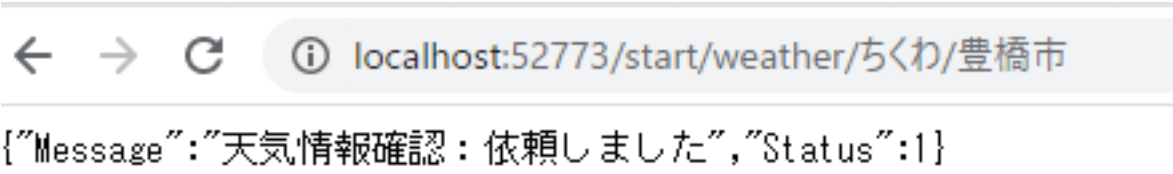
```
<Route Url="/weather/:product/:arecode" Method="GET" Call="WeatherCheck"/>
```

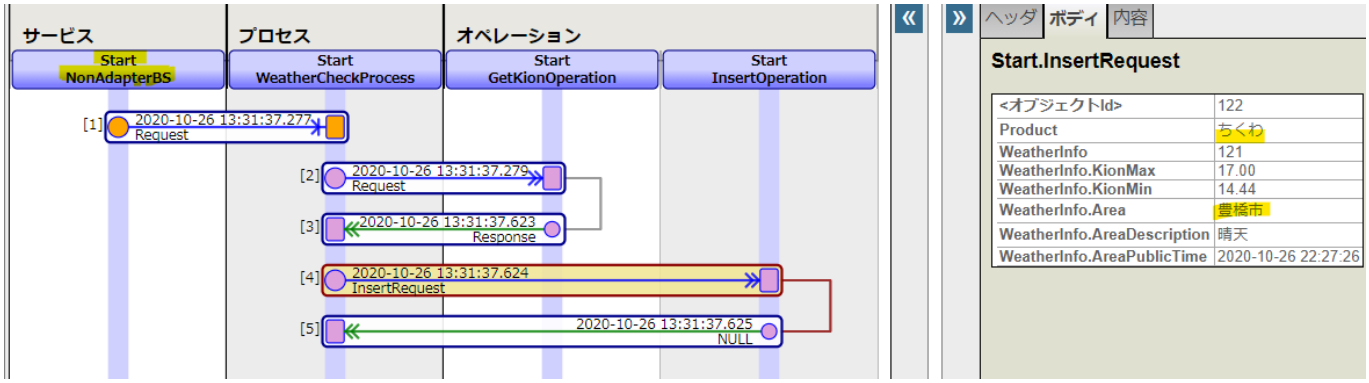
後は、 上記 URL を指定するためのベース URL
を管理ポータルのウェブアプリケーションパス設定画面で定義したら完成です。

設定詳細については、こちらの記事をご参照ください。

準備ができたら、REST で情報を送信できるビジネスサービスを利用して、情報を流してみます。

例）http://localhost:52773/start/weather/ちくわ/豊橋市




アダプタを使用しない場合、直接外部から ProcessInput() が呼び出せないため、REST やストアドプロシージャを通して実行されるロジックの中でビジネス・サービス用オブジェクトを生成し（Ens.Director クラスの CreateBusinessService() メソッドを使用） ProcessInput() を呼び出しました。

アダプタを使用する場合は、アダプタが入力を検知し専用オブジェクトに情報を格納しビジネス・サービスに渡してくれますが、アダプタを使わない場合、その部分だけが異なるだけで後の処理はほとんど一緒です。

ビジネス・サービスは、IRIS 外部から入力された情報を利用して要求メッセージを作成し、ビジネス・コンポーネントを呼び出すだけのシンプル設計です。

サンプルプロダクションを通して以下の内容が確認できました。

**プロダクションを動作させるために役割が異なるコンポーネントがある（ビジネス・サービス、ビジネス・プロセス、ビジネス・オペレーション）**

**コンポーネント間で情報を伝達するためには、メッセージを使用する**

**メッセージは削除しない限りデータベースに保存されているので、いつでもトレースできる**

**接続周りの処理を簡単にしてくれるアダプタがある**

IRIS の Interoperability（相互運用性）の使い方についての基本操作は以上の通りです。

この他、CSV
ファイルなどの形式の決まっ
たファイルの入出力に便利なレコードマップ（ご参考：FAQトピック）や、データ変換ツールなどもあります。
また、このシリーズと同様にInteroperability を利用して InterSystems
IRISでシンプルに開発するIoTアプリケーションの記事もあります。ぜひご参照ください。

この他、IRIS for Health では FHIR や HL7（SS-MIX2も含む）の送受信についてもサポートしています。

また別の記事でご説明できたらと思います。ご興味ある内容ありましたら、ぜひコメント欄にご記入ください！

最後に、Interoperability（相互運用性）の使い方についてはトレーニングコースもご用意しています。
講師と一緒にじっくり試してみたい方は、ぜひトレーニングコースへのご参加もご検討ください！

#REST API #ビジネスサービス #初心者 #相互運用性 #InterSystems IRIS #InterSystems IRIS for Health

---

**ソースURL:**
https://jp.community.intersystems.com/post/%E3%80%90%E3%81%AF%E3%81%98%E3%82%81%E3%81%A6%E3%81%AEintersystems-iris%E3%80%91interoperability%EF%BC%88%E7%9B%B8%E4%BA%92%E9%81%8B%E7%94%A8%E6%80%A7%EF%BC%89%EF%BC%9A%E3%82%B3%E3%83%B3%E3%83%9D%E3%83%BC%E3%83%8D%E3%83%B3%E3%83%88%E3%81%AE%E4%BD%9C%E6%88%90%EF%BC%88%E3%83%93%E3%82%B8%E3%83%8D%E3%82%B9%E3%83%BB%E3%82%B5%E3%83%BC%E3%83%93%E3%82%B9%EF%BC%89