PEX: Developing Productions with an External Language

Version 2024.1
2024-07-02

InterSystems Corporation   1 Memorial Drive   Cambridge MA 02142   www.intersystems.com

PEX: Developing Productions with an External Language
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

Introduction to the PEX Framework

The Production EXtension (PEX) framework provides you with a choice of external languages like Java and Python that you can use to develop interoperability productions. Interoperability productions enable you to integrate systems with different message formats and communication protocols. If you are not familiar with interoperability productions, see "Introduction to Productions."

PEX provides flexible connections between business services, processes, and operations that are implemented in external languages. In addition, you can use PEX-supported languages to develop inbound and outbound adapters. The PEX framework allows you to create an entire production in an external language or to create a production that has a mix of external language components and ObjectScript components. Once integrated, the production components written in an external language are called at runtime and use the PEX framework to send messages to other components in the production.

Some reasons to use the PEX framework include:

- Creating new adapters for protocols using available libraries written in an external language.
- Using available external language libraries to perform complex analysis or calculations.
- Implementing persistent messaging and long-running business processes without using ObjectScript in your application's technology stack.

You can create the following production components using a PEX-supported language:

- Inbound Adapter
- Outbound Adapter
- Business Service
- Business Process
- Business Operation

In addition to developing the code for the production components, you will typically define, configure, and monitor the production using the Management Portal. For example, you'll use the Management Portal to create the production, configure the business services, processes, and operations, start and stop the production, and trace persistent messages running through the production.

Note: PEX replaces the Java Business Hosts feature. For information on migrating existing Java Business Hosts productions to use PEX, see the community article Migrate from Java Business Host to PEX. Java Business Hosts was deprecated in release 2019.3 and is no longer part of the product.

2

Getting Started with PEX

Using the PEX framework to incorporate external language components into an interoperability production consists of the following steps:

1. In your favorite IDE, write the business host or adapter in an external language and compile the code.
2. In the Management Portal, register the new PEX component. An ObjectScript proxy class is created for the PEX component automatically.
3. If your PEX component is a business service, business operation, or business process, use the standard wizard to add a business host to your production, specifying the ObjectScript proxy class as the class of the business host. If your PEX component is an adapter, modify your business service or business operation to reference the adapter's proxy class.

2.1 PEX Libraries

Each external language has a PEX library that includes superclasses for each type of production component. The available PEX libraries are:

External Language	PEX Library
Java	`com.intersystems.enslib.pex`
.NET	`InterSystems.EnsLib.PEX`
Python	`iris.pex`

2.2 Working with PEX Components

A PEX component consists of a remote class written in an external language and an ObjectScript proxy class that the native production uses to work with the remote class. Before adding a custom PEX adapter or business host to a production, it must be registered as a PEX component. Once registered, details about the component are available to users who are building the production. To view and register PEX components, use the Management Portal to navigate to **Interoperability** > **Configure** > **Production EXtensions Components**.

2.3 Environmental Considerations

You can use InterSystems IRIS Interoperability only within an interoperability-enabled namespace that has a specific web application. When you create classes, you should avoid using reserved package names. The following subsections give the details.

2.3.1 Production-enabled Namespaces

An *interoperability-enabled namespace* is a namespace that has global mappings, routine mappings, and package mappings that make the classes, data, and menus that support productions available to it. For general information on mappings, see “Configuring Namespaces.” (You can use the information in that section to see the actual mappings in any interoperability-enabled namespace; the details may vary from release to release, but no work is necessary on your part.)

The system-provided namespaces that are created when you install InterSystems IRIS are not interoperability-enabled, except, on the community edition, the USER namespace is an interoperability-enabled namespace. Any new namespace that you create is by default interoperability-enabled. If you clear the **Enable namespace for interoperability productions** check box when creating a namespace, InterSystems IRIS creates the namespace with productions disabled.

Important: All system-provided namespaces are overwritten upon reinstallation or upgrade. For this reason, InterSystems recommends that customers always work in a new namespace that you create. For information on creating a new namespace, see “Configuring Data.”

2.3.2 Web Application Requirement

Also, you can use a production in a namespace only if that namespace has an associated web application that is named `/csp/`*namespace*, where namespace is the namespace name. (This is the default web application name for a namespace.) For information on defining web applications, see Applications.

2.3.3 Reserved Package Names

In any interoperability-enabled namespace, avoid using the following package names: Ens, EnsLib, EnsPortal, or CSPX. These packages are completely replaced during the upgrade process. If you define classes in these packages, you would need to export the classes before upgrading and then import them after upgrading.

Also, InterSystems recommends that you avoid using any package names that *start* with Ens (case-sensitive). There are two reasons for this recommendation:

- When you compile classes in packages with names that start with Ens, the compiler writes the generated routines into the ENSLIB system database. (The compiler does this because all routines with names that start with Ens are mapped to that database.) This means that when you upgrade the instance, thus replacing the ENSLIB database, the upgrade removes the generated routines, leaving only the class definitions. At this point, in order to use the classes, it is necessary to recompile them.

  In contrast, when you upgrade the instance, it is not necessary to recompile classes in packages with names that do not start with Ens.

- If you define classes in packages with names that start with Ens, they are available in all interoperability-enabled namespaces, which may or may not be desirable. One consequence is that it is not possible to have two classes with the same name and different contents in different interoperability-enabled namespaces, if the package name starts with Ens.

3

About Business Hosts and Adapters

This topic discusses information that applies to all business hosts and adapters written in an external language.

3.1 Creating Runtime Variables

The PEX framework allows you to use the Management Portal to specify runtime values of variables in the remote class, giving you the ability to re-use a PEX component in different productions. A variable that is declared in the remote class appears in the Management Portal as a setting of the corresponding business host. For example, if a Python class of a business service declares `Min = int(0)`, then the production's business service has a **Min** setting. At runtime, the variable is set to the value of the setting in the Management Portal.

If your remote class declares variables that you do not want to appear in the Management Portal, you can add metadata to hide the variable.

3.1.1 Variable Metadata

You can add metadata to your remote class that affects the Management Portal setting that is created for a variable. The syntax of this metadata depends on your external language: Java uses an annotation, .NET uses an attribute, and Python uses a special method named *`variableName`*`_info`. For example, the following code specifies that a setting `MyVariable` is required and adds a description that appears as a tool tip for the setting:

Java

```
@FieldMetadata(IsRequired=true,Description="Name of the company")
public String MyVariable;
```

.NET

```
[FieldMetadata(IsRequired=true,Description="Name of the company")]
public string MyVariable;
```

Python

```
MyVariable = int(0)

@classmethod
def MyVariable_info(self) -> \
 {
    'Description': "Maximum value",
    'IsRequired': True
  }:
  pass
```

Using this syntax, the following elements can be added to metadata to control the behavior of a Management Portal setting:

Metadata Element	Description
Description	A description of the setting that appears as a tool tip in the Management Portal.
Category	Groups the setting under a category heading. For example, if `Category="Basic"` is added to a Java class, then the setting is grouped under the **Basic** section of settings.
DataType	Overrides the variable data type, associating the setting with a new data type, for example, `Ens.DataType.ConfigName`.
IsRequired	If true, a value must be provided for the setting. Defaults to false.
ExcludeFromSettings	If true, it prevents a variable from appearing in the Management Portal as a setting. Defaults to false.

3.2 PEX Component Metadata

Just as you can add metadata to individual variables in a remote class, you can add metadata that applies to the entire PEX component. This metadata is used to provide information about the component in the Management Portal, where it appears on the Production EXtensions page and the Production Configuration page under **Informational Settings**.

Metadata Element	Description
Description	A description of the PEX component.
Info URL	A URL associated with the PEX component, for example, a web page providing detailed information about the component.

The syntax to include this class metadata depends on your external language. Java uses an annotation and .NET uses an attribute. Python uses a docstring and a special variable. The following code adds these metadata elements to the remote class of a PEX component:

Java

```
@ClassMetadata(Description="Custom Java Business Service",InfoURL="http://www.mycompany.com")
public class MyBusinessService extends com.intersystems.enslib.pex.BusinessService {
```

.NET

```
[ClassMetadata(Description ="Custom .NET Business Service", InfoURL="www.mycompany.com")]
public class MyBusinessService : InterSystems.EnsLib.PEX.BusinessService
```

Python

```
class MyBusinessService(iris.pex.BusinessService):
  """ Custom Business Service in Python """

  INFO_URL = "http://www.mycompany.com"
```

4

PEX Messaging

Within the PEX framework, most messages sent *between business hosts* are objects instantiated from one of the following subclasses:

Language	Message Class
Java	`com.intersystems.enslib.pex.Message`
.NET	`InterSystems.EnsLib.PEX.Message`
Python	`iris.pex.Message`

You simply add properties to your subclass, and then pass instantiated objects of the subclass using methods like `SendRequestAsync()` and `SendRequestSync()`. Within InterSystems IRIS, the message object written in the PEX-supported language corresponds to an object of class `EnsLib.PEX.Message`, which makes the message persistent and dynamic. By manipulating the object of type `EnsLib.PEX.Message`, you can reference any property in the external PEX object. Internally, the PEX object is represented as JSON as it passes between business hosts, so viewing the messages in the Management Portal are displayed in JSON format.

Though you can use other message objects, they must still be persistent if they are being passed between business hosts. To pass an object to a built-in ObjectScript component, you use the type IRISObject and map it to the persistent object expected by that component. For example, if you are sending a message to the EnsLib.PubSub.PubSubOperation, you would map the IRISObject to EnsLib.PubSub.Request. Trying to pass a non-persistent object as a message between business hosts results in a runtime error.

Objects sent from an inbound adapter to a business service are arbitrary and do not need to be persistent.

Note: The messages sent are limited to the InterSystems IRIS maximum string length.

5

Inbound Adapters in the PEX Framework

Business services use inbound adapters to receive specific types of input data. You can write a custom inbound adapter that is used by an ObjectScript business service, or it can be used by a business service that is also written in a PEX-supported language. For general information related to all production components written in an external language, see About Business Hosts and Adapters.

5.1 Developing a Custom Adapter

To begin the process of writing a custom inbound adapter in an external language, extend one of the following classes:

Language	Class
Java	`com.intersystems.enslib.pex.InboundAdapter`
.NET	`InterSystems.EnsLib.PEX.InboundAdapter`
Python	`iris.pex.InboundAdapter`

5.1.1 Implementing Abstract Methods

Typically, the inbound adapter's `OnTask()` method performs the main function of the adapter. At runtime, the `OnTask()` method is called at the interval specified in the settings for the business host that is using the adapter. From within `OnTask()`, call `BusinessHost.ProcessInput()` to dispatch an object to the associated business service's `ProcessInput` method. For example, a simple adapter might include:

Java

```
public void OnTask() throws Exception {
  SimpleObject request = new SimpleObject("message #"+(++runningCount));
  // send object to business service's ProcessInput() method
  String response = (String) BusinessHost.ProcessInput(request);
  return;
}
```

.NET

```
public override void OnTask()
{
  SimpleObject request = new SimpleObject("message #" + (++runningCount));
  // send object to business service's ProcessInput() method
  string response = (string)BusinessHost.ProcessInput(request);
  return;
}
```

Python

```
def OnTask(self):
  msg = "this is message # %d" %self.runningCount
  request = demo.SimpleObject(msg)
  # send object to business service's ProcessInput() method
  response = self.BusinessHost.ProcessInput(request)
  return
```

The object sent from the adapter's `BusinessHost.ProcessInput` call to the business service's `ProcessInput` method is arbitrary and does not need to be persistent within InterSystems IRIS. The same is true for the object returned by the business service's `ProcessInput` method to the adapter.

By default, the object sent from the adapter to the business service is serialized into JSON and received by the service as an IRISObject type. However, if the adapter and business service share a connection, the business service can receive and return the same object type, which has advantages. For details, see Sharing a Connection.

At a minimum, your adapter must implement the superclass' `OnInit`, `OnTearDown`, and `OnTask` methods. For details on these and other methods of an inbound adapter, see the PEX API Reference.

5.2 Registering the Adapter

Once you are done writing the code for the PEX adapter, you are ready to register it. Registering the adapter generates an ObjectScript proxy class that a business service can use to identify the adapter and defines the external language server that the production uses to connect to the adapter. For details on registering the adapter, see Registering a PEX Component.

5.3 Adding the Adapter to a Business Service

A PEX adapter can be used by a PEX business service or by a native ObjectScript business service. The process of configuring a business service so it uses the PEX adapter varies depending on the type of business service. Both scenarios require the adapter to be registered as a PEX component.

When a PEX business service is using the PEX adapter, the remote class of the business service uses a method to identify the adapter. For details, see Using an Inbound Adapter.

Like all native ObjectScript business services, a native business service using a PEX adapter identifies the adapter using an `ADAPTER` parameter. In this case, the `ADAPTER` parameter is set to the name of the PEX adapter's ObjectScript proxy class. By default, this proxy class shares the name of the adapter's remote class, but a custom proxy name might have been defined when the adapter was registered.

6

PEX Outbound Adapters

Business operations use outbound adapters to send out specific types of data from the production. You can write a custom outbound adapter that is used by an ObjectScript business operation, or it can be used by a business operation that is also written in a PEX-supported language. For general information related to all production components written in an external language, see About Business Hosts and Adapters.

6.1 Developing a Custom Adapter

To write an outbound adapter in an external language, extend one of the following classes:

Language	Class
Java	`com.intersystems.enslib.pex.OutboundAdapter`
.NET	`InterSystems.EnsLib.PEX.OutboundAdapter`
Python	`iris.pex.OutboundAdapter`

Within the outbound adapter, you can create all the methods you need to successfully send out data from the production. Each of these methods can be called from the business operation associated with the adapter. The business operation can call a method with arguments of arbitrary objects and literals. For details on how the business operation calls the adapter methods, see Invoking Adapter Methods.

6.1.1 Implementing Abstract Methods

In addition to creating methods that send out data from the production, your remote outbound adapter class must implement a few abstract methods. For details about these methods, see PEX API Reference.

6.2 Registering the Adapter

Once you are done writing the code for the PEX adapter, you are ready to register it. Registering the adapter generates an ObjectScript proxy class that a business operation can use to identify the adapter and defines the external language server that the production uses to connect to the adapter. For details on registering the adapter, see Registering a PEX Component.

6.3 Adding the Adapter to a Business Operation

A PEX adapter can be used by a PEX business operation or by a native ObjectScript business operation. The process of configuring a business operation so it uses the PEX adapter varies depending on the type of business operation. Both scenarios require the adapter to be registered as a PEX component.

When a PEX business operation is using the PEX adapter, the remote class of the business operation uses a method to identify the adapter. For details, see Using an Outbound Adapter.

Like all native ObjectScript business operations, a native business operation using a PEX adapter identifies the adapter using an `ADAPTER` parameter. In this case, the `ADAPTER` parameter is set to the name of the PEX adapter's ObjectScript proxy class. By default, this proxy class shares the name of the adapter's remote class, but a custom proxy name might have been defined when the adapter was registered.

7

Business Services in the PEX Framework

Business services connect with external systems and receive messages from them through an inbound adapter. For general information related to all production components written in Java, see About Business Hosts and Adapters.

7.1 Developing a Business Service

To write a business service in an external language, extend one of the following classes:

Language	**Class**
Java	`com.intersystems.enslib.pex.BusinessService`
.NET	`InterSystems.EnsLib.PEX.BusinessService`
Python	`iris.pex.BusinessService`

There are three ways of implementing a business service:

1. Polling business service with an adapter — The production framework at regular intervals calls the adapter's OnTask() method, which sends the incoming data to the ProcessInput() method of the business service, which, in turn calls the OnProcessInput method with your business service code. Your custom code must implement the OnProcessInput method to handle data from the adapter, not ProcessInput.
2. Polling business service that uses the default adapter — In this case, the framework calls the default adapter's OnTask method with no data. The OnProcessInput() method then performs the role of the adapter and is responsible for accessing the external system and receiving the data.
3. Nonpolling business service — The production framework does not initiate the business service. Instead custom code in either a long-running process or one that is started at regular intervals initiates the business service by calling the Director.CreateBusinessService() method. For more details, see Director.

7.1.1 Implementing Abstract Methods

After extending the PEX class, you need to implement abstract methods for the business service.

When developing a polling business service with an adapter, the `OnProcessInput()` takes an arbitrary object from the adapter, and returns an arbitrary object. These arbitrary objects do not need to be persistent.

For more details about the abstract methods that need to be implemented, see PEX API Reference.

7.2 Using an Inbound Adapter

Within a production, a business service uses an inbound adapter to communicate with systems outside the production. When developing a PEX business service, you can include a special method in the remote class to define which inbound adapter the business service uses. This inbound adapter can be a PEX adapter or a native ObjectScript adapter.

The method used to specify the inbound adapter for the PEX business service is `getAdapterType()`. For example, if the PEX business service uses a custom PEX inbound adapter, your remote class might include:

Java

```
public String getAdapterType() {
  return "com.demo.pex.MyInboundAdapter";
}
```

.NET

```
public override string getAdapterType() {
  return "Demo.PEX.MyInboundAdapter";
}
```

Python

```
def getAdapterType():
  return "demo.PEX.MyInboundAdapter"
```

When using a PEX adapter, the `getAdapterType` method should return the name of the ObjectScript proxy class that was specified when the adapter was registered. By default, this proxy name is the same as the remote class, but a custom name might have been defined.

If you do not include `getAdapterType` in the remote class, the business service uses the standard `Ens.InboundAdapter` adapter. If your business service does not use an adapter, return an empty string.

7.3 Using the Business Service

Once you have finished developing the remote class of the PEX business service, you can complete the following steps to integrate the business service into an interoperability production:

1. Register the PEX business service by navigating to **Interoperability** > **Configure** > **Production EXtensions Components**. For details, see Registering a PEX Component.
2. Open the production and use the standard wizard to add a business service. In the **Service Class** field, select the ObjectScript proxy class of the PEX component. By default, the name of this proxy class matches the remote class, but a custom name might have been defined when the component was registered.

8

PEX Business Processes

Business processes allow you to define business logic, including routing and message transformation. Business processes receive messages from other business hosts in the production for processing.

For general information related to all production components written in an external language, see About Business Hosts and Adapters.

8.1 Developing a Business Process

To write business process in an external language, extend one of the following classes:

Language	Class
Java	`com.intersystems.enslib.pex.BusinessProcess`
.NET	`InterSystems.EnsLib.PEX.BusinessProcess`
Python	`iris.pex.BusinessProcess`

8.1.1 Implementing Abstract Methods

After extending the business process class, you need to implement some abstract methods. For details on these methods, see PEX API Reference.

8.1.2 Persistent Properties

Within InterSystems IRIS, native business processes are persistent objects. For the lifespan of these business processes, properties are stored and are accessible during each callback. By default, a PEX business process lacks this characteristic; each method is called in a separate instance and the values of variables are not preserved. However, you can mimic the persistence of a native business process by making the properties of the PEX business process persistent. Use the following syntax to persist a property of a PEX business process:

Java

```
// Use annotation to create a persistent property
// Strings, primitive types, and their boxed types can be persisted

@Persistent
public integer runningTotal = 0;
```

.NET

```
// Use attribute to create a persistent property
// Strings, primitive types, and their boxed types can be persisted

[Persistent]
public int runningTotal = 0;
```

Python

```
#Set class variable to create persistent property
#Only variables of types str, int, float, bool and bytes can be persisted

PERSISTENT_PROPERTY_LIST=["myVariable1","myVariable2"]
```

Within InterSystems IRIS, persistent properties are saved in the corresponding instance of the business process. Persistent properties are restored before each callback and are saved after each callback.

8.2 Using the Business Process

Once you have finished developing the remote class of the PEX business process, you can complete the following steps to integrate the business process into an interoperability production:

1. Register the PEX business process by navigating to **Interoperability** > **Configure** > **Production EXtensions Components**. For details, see Registering a PEX Component.
2. Open the production and use the standard wizard to add a business process. In the **Business Process Class** field, select the ObjectScript proxy class of the PEX component. By default, the name of this proxy class matches the remote class, but a custom name might have been defined when the component was registered.

9

PEX Business Operations

Business operations connect with external systems and send messages to them via an outbound adapter.

For general information related to all production components written in a PEX-supported language, see About Business Hosts and Adapters.

9.1 Developing a Business Operation

To write a business operation in an external language, extend one of the following classes:

Language	Class
Java	`com.intersystems.enslib.pex.BusinessOperation`
.NET	`InterSystems.EnsLib.PEX.BusinessOperation`
Python	`iris.pex.BusinessOperation`

9.1.1 Implementing Abstract Methods

At runtime, the `OnMessage()` method is called when the business operation receives a message from another business host. From within this method, the business operation can call any of the methods defined in the outbound adapter associated with the business operation. Parameters for calls from a business operation to an outbound adapter do not need to be persistent.

For details on other abstract methods that need to be implemented, see PEX API Reference.

9.2 Using an Outbound Adapter

Within a production, a business operation uses an outbound adapter to communicate with systems outside the production. When developing a PEX business operation, you can include a special method in the remote class to define which outbound adapter the business operation uses. This outbound adapter can be a PEX adapter or a native ObjectScript adapter.

The method used to specify the outbound adapter for the PEX business operation is `getAdapterType()`. For example, if the PEX business operation uses a custom PEX outbound adapter, your remote class might include:

Java

```
public String getAdapterType() {
  return "com.demo.pex.MyOutboundAdapter";
}
```

.NET

```
public override string getAdapterType() {
  return "Demo.PEX.MyOutboundAdapter";
}
```

Python

```
def getAdapterType():
  return "demo.PEX.MyOutboundAdapter"
```

When using a PEX adapter, the `getAdapterType` method should return the name of the ObjectScript proxy class that was specified when the adapter was registered. By default, this proxy name is the same as the remote class, but a custom name might have been defined.

9.2.1 Invoking Adapter Methods

A business operation uses its outbound adapter by invoking methods defined in the adapter's code. The syntax for invoking these methods varies depending on whether the business operation is a PEX component or a native ObjectScript class. For details on invoking the PEX adapter methods from a native business operation, see Accessing Properties and Methods from a Business Host.

If your business operation is a PEX component, use `Adapter.invoke()` to call the adapter's method. Its signature is:

`Adapter.invoke("`*`methodName`*`", `*`arguments`*`)`

Where:

- *methodName* specifies the name of the method in the outbound adapter to be executed.
- *arguments* contains the arguments of the specified method.

For example, to invoke the adapter's `printString` method, add the following code to your business operation:

Java

```
public Object OnMessage(Object request) throws Exception {
  MyRequest myReq = (MyRequest)request;
  Adapter.invoke("printString", myReq.requestString);
}
```

.NET

```
public override object OnMessage(object request)
{
  MyRequest myReq = (MyRequest)request;
  Adapter.invoke("printString", myReq.RequestString);
}
```

Python

```
def OnMessage(self, messageInput):
  self.Adapter.invoke("printString", messageInput.requestString)
  return
```

When the business operation passes a primitive to the adapter, the same primitive is received by the adapter. However, by default, when the business operation passes an object to the adapter, the object is serialized into JSON and received by the adapter as an IRISObject type. If you want to change this behavior so the adapter receives and returns the same object type, see Sharing a Connection.

9.3 Using the Business Operation

Once you have finished developing the remote class of the PEX business operation, you can complete the following steps to integrate the business operation into an interoperability production:

1. Register the PEX business operation by navigating to **Interoperability** > **Configure** > **Production EXtensions Components**. For details, see Registering a PEX Component.
2. Open the production and use the standard wizard to add a business operation. In the **Operation Class** field, select the ObjectScript proxy class of the PEX component. By default, the name of this proxy class matches the remote class, but a custom name might have been defined when the component was registered.

10

Registering a PEX Component

Once you have used an external language to create the remote class of your PEX component, you need to register the class using the Management Portal. This registration defines critical information about the PEX component like the external language server that connects the remote class and the name of the ObjectScript proxy class that is created for the component. The registration process is the same regardless of whether the PEX component is an adapter or business host.

Tip: The external language server that connects your remote class must be fully operational *before* registering the PEX component. If you are using Java, you need to add a utility jar file to the server's classpath. For details about this jar file, see Connecting with External Language Servers.

To register a PEX component:

1. Open the Management Portal and navigate to **Interoperability** > **Configure** > **Production EXtensions Components**.
2. Select **Register New Component**.
3. Use the **Remote Classname** field to specify the class written in the external language, including its package.
4. If desired, use the **Proxy Name** field to specify a custom name for the ObjectScript proxy class that is created for the PEX component. The default is the name of the remote class.
5. Use the **External Language Server** field to select the external language server that the production will use to connect to the remote class. The external language server has to be fully operational, even if it is not currently running.
6. Use the **Gateway Extra Classpaths** field to specify the executable that contains the remote class. You can also add other binaries, files, and directories needed by the remote class.

Note: After selecting **Register**, the new PEX component does not appear in the list until you have selected the Refresh icon.

11

Connecting with External Language Servers

When you register the PEX component written in an external language, you specify an external language server that the production uses to communicate with the remote class. This external language server is commonly referred to as a *gateway*.

In most languages, the built-in external language server for your PEX language contains all you need to run a PEX production; you do not need to create a custom server. However, if you are using Java, the external language server must include a special utility jar file on its classpath. So your default or custom Java server must include `install-dir/dev/java/lib/1.8/intersystems-utils-version.jar` on its classpath, where *install-dir* is the directory where your InterSystems product is installed. If you are manually entering the location of this file, you can use the keyword `$$IRISHOME` to identify the installation directory.

11.1 Sharing a Connection

In some cases, you might want a PEX adapter to use the connection of its associated PEX business service or business operation rather than using a separate external language server connection. When each PEX component uses a separate connection, objects passed between the components must be serialized into JSON strings, preventing you from directly passing an object between the components. By sharing a connection, your adapter and business host can pass an object directly and get the same object back. To share a connection, the PEX adapter and its associated PEX business host must be written in the same external language.

To indicate that the PEX adapter should share a connection with its business service or business operation, use the Management Portal to select the business host's **Alternative Adapter Connection** > **Use Host Connection** setting. Once this setting has been enabled, the PEX framework ignores the external language server setting that was specified when the PEX adapter was registered.

The following examples demonstrate the advantages of using a shared connection between an adapter and its associated business host.

Business Service and Inbound Adapter

When an inbound adapter shares a connection with the business service, the business service receives the same object that was sent by the inbound adapter. For example:

Java

```
// Code from the inbound adapter:
public void OnTask() throws Exception {
  SimpleObject request = new SimpleObject("message #1");
  String response = (String) BusinessHost.ProcessInput(request);
}

// Code from the business service:
public Object OnProcessInput(Object messageInput) throws Exception {
  SimpleObject obj = (SimpleObject)messageInput;
  System.out.print("\r\n[Java] Object received: " + obj.value);
  return "...Service received " + obj.value;
}
```

In contrast, the following code shows how you would have to handle an object sent to the business service from the inbound adapter if they do not share a connection.

Java

```
// Code from the inbound adapter:
public void OnTask() throws Exception {
  SimpleObject request = new SimpleObject("message #1");
  String response = (String) BusinessHost.ProcessInput(request);
}

// Code from the business service:
public Object OnProcessInput(Object messageInput) throws Exception {
  com.intersystems.jdbc.IRISObject obj = (com.intersystems.jdbc.IRISObject)messageInput;
  System.out.print("\r\n[Java] Object received: " + obj.get("value"));
  return "...Service received" + obj.get("value");
}
```

Business Operation and Outbound Adapter

When an outbound adapter shares a connection with the business operation, the adapter receives the same object that was sent by the business operation. For example:

Java

```
// Code from the business operation:
public Object OnMessage(Object request) throws Exception {
  SimpleObject myObj = new SimpleObject("my string");
  Adapter.invoke("passObj", myObj);
  return null;
}

// Code from the associated adapter:
public Object passObj(SimpleObject obj) {
  System.out.print("\r\n[Java]Object received: " + obj.value);
  return null;
}
```

In contrast, the following code shows how you would have to handle an object sent to the adapter from the business operation if they do not share a connection.

Java

```
// Code from the business operation:
public Object OnMessage(Object request) throws Exception {
  SimpleObject myObj = new SimpleObject("my string");
  Adapter.invoke("passObj", myObj);
  return null;
}

// Code from the associated adapter:
public Object passObj(com.intersystems.jdbc.IRISObject obj) {
  System.out.print("\r\n[Java]Object received: " + obj.get(“value”));
  return null;
}
```

A

PEX API Reference

This reference lists the methods for PEX components written in any of the external languages that are supported by the PEX framework. The ObjectScript proxy classes generated for remote classes inherit from ObjectScript classes in the EnsLib.PEX package.

A.1 General Notes about Methods

As you override methods to implement a production component, keep the following in mind:

- Each production component must override all abstract methods.
- While native interoperability methods use one input argument and one output argument and return a status, the corresponding PEX methods take one input argument and return the output argument as a return value.
- All error handling for PEX methods are done with exceptions.
- For native interoperability methods that don't require persistent objects as input and output arguments, the corresponding PEX methods can also use arbitrary objects as arguments and return values. PEX utilizes forward proxy and reverse proxy of the external language server to map the arbitrary object appropriately.
- For native interoperability methods that require persistent objects as arguments, such as the methods that send messages to other processes, the corresponding PEX methods can use PEX Messages as arguments and return values. Examples of such methods are SendRequestSync, SendRequestAsync, OnRequest, OnResponse and OnMessage.
- When overriding callback methods, you should not change the formal spec of the methods even if you have customized the message class. The argument types should remain as objects.

A.2 Business Operations

The business operation can optionally use an adapter to handle the outgoing message. If the business operation has an adapter, it uses the adapter to send the message to the external system. The adapter can either be a PEX adapter or an ObjectScript adapter.

A.2.1 OnMessage() Method

The **OnMessage()** message is called when the business operation receives a message from another production component. Typically, the operation will either send the message to the external system or forwards it to a business process or another business operation. If the operation has an adapter, it uses the **Adapter.invoke()** method to call the method on the adapter that sends the message to the external system. If your operation is forwarding the message to another production component, it uses the **SendRequestAsync()** or the **SendRequestSync()** method.

Abstract method: you must implement.

Parameters: (*request*)

- *request*—of type Object, this contains the incoming message for the business operation.

You must implement an OnMessage method with a single parameter of type Object. Within the method you can cast the parameter to the actual type passed by the caller.

Returns: Object

A.2.2 OnInit() Method

The **OnInit()** is called when the component is started. Use **OnInit()** to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.2.3 OnTearDown() Method

The **OnTearDown()** method is called before the component is terminated. Use **OnTeardown()** to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.2.4 SendRequestAsync() Method

The **SendRequestAsync()** method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *description*])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *description*—an optional string parameter that sets a description property in the message header.

Returns: void

A.2.5 SendRequestSync() Method

The **SendRequestSync()** method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *timeout*—an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.
- *description*—an optional string parameter that sets a description property in the message header.

A.2.6 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message*—a string that is written to the log.

A.3 Business Process

A Business Process class typically contains most of the logic in a production. A business process can receive messages from a business service, another business process, or a business operation. It can modify the message, convert it to a different format, or route it based on the message contents. The business process can route a message to a business operation or another business process. For information on making properties of a business process persistent, see Persistent Properties.

A.3.1 OnRequest() Method

The OnRequest() method handles requests sent to the business process. A production calls this method whenever an initial request for a specific business process arrives on the appropriate queue and is assigned a job in which to execute.

Abstract method: you must implement.

Parameters: (*request*)

- *request*—an object that contains the request message sent to the business process.

A.3.2 OnResponse() Method

The OnResponse() method handles responses sent to the business process in response to messages that it sent to the target. A production calls this method whenever a response for a specific business process arrives on the appropriate queue and is assigned a job in which to execute. Typically this is a response to an asynchronous request made by the business process where the request's responseRequired parameter has a true value.

Abstract method: you must implement.

Parameters: (*request*, *response*, *callRequest*, *callResponse*, *completionKey*)

- *request*—an object that contains the initial request message sent to business process.
- *response*—an object that contains the response message that this business process can return to the production component that sent the initial message.
- *callRequest*—an object that contains the request that the business process sent to its target.
- *callResponse*—an object that contains the incoming response.
- *completionKey*—a string that contains the completionKey specified in the completionKey parameter of the outgoing SendAsync() method.

A.3.3 OnComplete() Method

The OnComplete() method is called after the business process has received and handled all responses to requests it has sent to targets.

Abstract method: you must implement.

Parameters: (*request*, *response*)

- *request*—an object that contains the initial request message sent to business process.
- *response*—an object that contains the response message that this business process can return to the production component that sent the initial message.

A.3.4 OnInit() Method

The OnInit() method is called when the component is started.

Abstract method: you must implement.

Parameters: none

Returns: void

A.3.5 OnTearDown() Method

The OnTearDown() method is called before the component is terminated.

Abstract method: you must implement.

Parameters: none

Returns: void

A.3.6 Reply() Method

The Reply() method sends the specified response to the production component that sent the initial request to the business process.

Parameters: (response)

- response—an object that contains the response message.

A.3.7 SendRequestAsync() Method

The SendRequestAsync() method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *responseRequired* [, *completionKey* [, *description*]]])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *responseRequired*—a boolean value that specifies if the target must send a response message.
- *completionKey*—a string that will be sent with the response message.
- *description*—an optional string parameter that sets a description property in the message header.

A.3.8 SendRequestSync() Method

The SendRequestSync() method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *timeout*—an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.
- *description*—an optional string parameter that sets a description property in the message header.

A.3.9 SetTimer() Method

The SetTimer() method specifies the maximum time the business process will wait for all responses.

Parameters: (*timeout* [, *completionKey*])

- *timeout*—an integer that specifies a number of seconds or a string that specifies a time period, such as “PT15S”, which represents 15 seconds of processor time.
- *completionKey*—a string that will be returned with the response if the maximum time is exceeded.

A.3.10 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component’s Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message*—a string that is written to the log.

A.4 Business Service

The Business Service class is responsible for receiving the data from the external system and sending it to business processes or business operations in the production. The business service can use an adapter to access the external system.

A.4.1 OnProcessInput() Method

The OnProcessInput() method receives the message from the inbound adapter via the adapter’s ProcessInput() method and is responsible for forwarding it to target business processes or operations. If the business service does not specify an adapter, then the default adapter calls the OnProcessInput() method with no message and the business service is responsible for receiving the data from the external system and validating it.

Abstract method: you must implement.

Parameters: (*message*)

- *message*—an object containing the data that the inbound adapter sent to the business service. The message can have any structure agreed upon by the inbound adapter and the business service. The message does not have to be a subclass of Message or IRISObject and is typically not persisted in the database.

A.4.2 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.4.3 OnTearDown() Method

The OnTearDown() method is called before the business host is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.4.4 ProcessInput() Method

The inbound adapter calls the ProcessInput0 method of the business service and the ProcessInput() method in turn calls the OnProcessInput() method that is your custom code.

Parameters: (*input*)

- *input*—an object with an arbitrary structure by agreement with the inbound adapter. The parameters passed to the ProcessInput() method do not need to be persistent objects.

Returns: object

A.4.5 SendRequestAsync() Method

The SendRequestAsync() method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *description*])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *description*—an optional string parameter that sets a description property in the message header.

A.4.6 SendRequestSync() Method

The SendRequestSync() method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target*—a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request*—specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject

class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.

- *timeout*—an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.
- *description*—an optional string parameter that sets a description property in the message header.

A.4.7 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message*—a string that is written to the log.

A.5 Director

The Director class is used for nonpolling business services, that is, business services which are not automatically called by the production framework (through the inbound adapter) at the call interval. Instead these business services are created by a custom application by calling the Director.CreateBusinessService() method.

A.5.1 CreateBusinessService() Method

The CreateBusinessService() method initiates the specified business service.

Parameters: (*connection*, *target*)

- *connection*—an IRISConnection object that specifies the connection to the external language server for your language.
- *target*—a string that specifies the name of the business service in the production definition.

Return value: *businessService*—the return value contains the Business Service instance that has been created.

A.6 Inbound Adapter

The InboundAdapter is responsible for receiving the data from the external system, validating the data, and sending it to the business service by calling the ProcessInput() method.

A.6.1 OnTask() Method

The OnTask() method is called by the production framework at intervals determined by the business service CallInterval property. The OnTask() method is responsible for receiving the data from the external system, validating the data, and sending it in a message to the business service OnProcessInput() method. The message can have any structure agreed upon by the inbound adapter and the business service.

Abstract method: you must implement.

Parameters: none

A.6.2 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.6.3 OnTearDown() Method

The OnTearDown() method is called before the business host is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.6.4 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component’s Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message*—a string that is written to the log.

A.7 Outbound Adapter

The Outbound Adapter class is responsible for sending the data to the external system.

A.7.1 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.7.2 OnTearDown() Method

The OnTearDown() method is called before the business host is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.7.3 Invoke() Method

The Invoke() method allows the BusinessOperation to execute any public method defined in the OutboundAdapter.

Parameters: (*methodname*, *arguments*)

- *methodname* — specifies the name of the method in the outbound adapter to be executed.
- *arguments* — contains the arguments of the specified method.

A.7.4 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message*—a string that is written to the log.

A.8 Message

The Message class is the abstract class that is the superclass class for persistent messages sent from one component to another. The Message class has no properties or methods. Users subclass the Message class in order to add properties. The PEX framework provides the persistence to objects derived from the Message class.