WxIspm

The IS Project Manager (or ispm, for short) is a utility, which aims to provide valuable help in managing a, possibly large, set of projects in the environment of the webMethods Integration Server.

The ispm grew out of a customer project, in which we had to develop, and maintain, hundreds of IS projects, so-called packages. By using a properly configured ispm, you will be able to clone an IS project from your hard drive, build, and install it into your local development IS within seconds.

Originally, the project was designed as a command line tool (with the intention to enable autoupdates, etc.) However, upon using an initial implementation, it turned out that the media disruption between Designer as the most important tool, and the command line was a problem. Besides, performance wasn't as good as expected, and a command line tool, that's only running on occasion, is not a good place for implementing a cache. So, the reimplementation as an Is package, seemed prudent.

In what follows, we'll be using the terms WxIspm, and ispm, synonymously.

Goals (What does it do?)

It provides services, that an IS developer maay use, for example:

- A service for reloading an IS package, that has been modified on disk.
  Of course, the same functionality is available from within the IS administration
  UI, or from the Designers "Package Explorer". However, the WxIspm service offers to
  do the same with dependencies disabled. This will greatly reduce the time, that the
  service requires for running.
- A service for compiling an IS packages Java services. This is supposed to work, even if running the
  equivalent IS internal functionality (**wx.server.package:packageCompile**) fails, and reports
  "Unable to locate or start compiler", an error message, which is usually, at best, misleading.
  The WxIspm service however, provides explicit diagnostics, which should help to quickly resolve
  any problems, like compiler errors, or the like.
- A service, that combines compile/reload into a single service invocation. (Typically, developers need
  to do an explicit reload after the compile.)
- **Testing stage**: Several services for managing IS sets of development projects (so-called local repositories),
  including checkout/clone, etc. from external sources (so-called remote repositories). WxIspm
  is heavily tied to the "local development mode".
- **Planning stage**: A service for running ABE/Deployer on a package, or a set of packages. This is not
  intended as a replacement for, or even as a tool for use within a CI/CD chain. Instead, it aims
  to help in a situation, where local testing is not possible, and you wish to ensure, that the
  tested version is identical with the one you are developing locally.

- It is a test bed for working on new ideas in the area of development, building, and the like. Example: The WxIspm services are documented in markdown files. At build time, these files are being incorporated into the services as comments.

Non-Goals (What it doesn't, and won't do?)

- WxIspm is not intended to provide anything beyond the development stage.
- It should never leave the developers workstation, unless for the purpose of distributing itself.
- It is not intended for deployment in environments like production, quality assessment, integration. Not even in testing!

Concepts

This section intends to introduce some concepts, that are used overall in WxIspm:

Instance

An instance is an IS server, that is actually, or (at least) potentially running. An instance is mainly identified by it's installation directory. For example, if the webMethods suite has been installed in *F:\SoftwareAG\webMethods10.3* with typical options, then an instance would be *F:\SoftwareAG\webMethods10.3\IntegrationServer\instances\default*.

A webMethods installation can contain multiple instances, typically other directories in *F:\SoftwareAG\webMethods10.3\IntegrationServer\instances*.

The "local" instance

A special instance is the so-called "local" instance. It is automatically created by WxIspm, and needs no explicit configuration. The "local" instance refers to the IS, in which WxIspm is currently running.

Other instances

Other instances are only present, if they have been configured in the WxIspm config file. For example, the following would configure another instance named "central":

```
    <instance id="central"
        baseDir="F:\SAG\wM10.3\IntegrationServer\inst\central"/>
```

So, in essence, an instance is given by it's Id, and it's base directory. It does have some other attributes, but these are usually covered by reasonable defaults:

Attribute	Default value
wmHomeDir	${baseDir}/../../..
packagesDir	${baseDir}/packages
configDir	${baseDir}/config
logsDir	${baseDir}/logs

Note: It is quite possible, to have two instances, which are referring to the same IS. For example, if you have an instance "wm103" with the base directory *F:\SoftwareAG\webMethods10.3\IntegrationServer\instances\default*, referring to the IS, in which WxIspm is running, then the instances "local", and "wm103" would both identify the same IS.

Instance properties

An instance can have properties. Basically, apart from the "local" instance, it should have properties:

Property	Example	Description
is.admin.url	http://127.0.0.1:10555/	URL of the instances IS administration UI
is.admin.user	Administrator	Admin user name
is.admin.pass	manage	Admin users password.

A configuration snipped specifying these properties could be:

```
<instance id="central"
    baseDir="F:\SAG\wM10.3\IntegrationServer\inst\central">
  <property key="is.admin.url" value="http://127.0.0.1:10555/"/>
  <property key="is.admin.user" value="Administrator"/>
  <property key="is.admin.pass" value="manage"/>
</instance>
```

Creating instances

To create a new instance, the following possibilities are available:

1. Change the Ispm config file accordingly, and reload the WxIspm package.
2. Invoke the service **wx.ispm.pub.config:addInstance** with the appropriate parameters.

Local Repositories

A local repository is a directory, that contains projects. Within WxIspm, a project is meant to be a directory, that contains one, or more packages, on which the developer is working.

In local development mode, the local repository is, where you checkout, or clone your source code to.

The "internal" local repository

Like the "local" instance, WxIspm creates a local repository automatically: The "internal" local repository is identical with the instances "packages" directory. Every directory in that directory is assumed to be a project, which contains exactly one package. This is intended to be useful in cases, where you are not using "local development mode".

Other local repositories

Apart from the "internal" local repository, the WxIspm config file can be used to specify local repositories, like this:

```
<localRepo id="wm103" dir="F:\GIT-DEV103">
  <!-- Configure values for git's user.name,
       and user.email settings. -->
  <property key="git.user.name"
         value="Jochen Wiedmann"/>
  <property key="git.user.email"
         value="jochen.wiedmann@gmail.com"
</localRepo>
```

Creating local repositories

Obviously, you don't need to create the "internal" repository.

To create another repository, there are the following options:

1. Create an entry in the ispm config file, and reload the WxIspm package.
2. Invoke the service **wx.ispm.pub.config:addLocalRepository**.

Remote repositories

A remote repository is the analogon of a source code repository, where you're projects are stored by the team.

No obvious idea of a default remote repository provides itself,, so there is no "local", or "internal" remote repository. In other words: All remote repositories are specified in the ispm config file, like so:

```
<remoteRepo id="azure"
           url="https://dev.azure.com/customer/projectGroup"
       handler="azure">
  <property key="azure.userName" value="jwi@softwareag.com"/>
  <property key="azure.password" value="552a....8fc5"/>
</remoteRepo>
```

Creating remote repositories

Like instances, or local repositories, there are two options for creating a remote repository:

1. Create a "remoteRepo" element in the ispm config file, and reload the WxIspm package.
2. Invoke the service **wx.ispm.pub.config:addRemoteRepository**.

Local repository layouts

Any project will typically have its own conventions on organizing projects. To deal with the differences, WxIspm has introduced the concept of a "local repository layout". The layout is a Java object, which implements the interface **com.github.jochenw.ispm.core.model.ILocalRepoLayout**. By default, a local repository has the "default" layout, which is based on the following assumptions:

1. Every subdirectory of the (not subdirectories of subdirectories) is a project.
2. Every project has a directory *IntegrationServer/packages*.
3. The projects packages are the subdirectories of *IntegrationServer/packages*. (Again, only direct subdirectories count, not indirect subdirectories.)

As an example, take the WxIspm project itself, which you can view here.

To use an alternative layout, set the attribute "layout" in the ispm config files "localRepo" element, like this

```
<localRepo id="wm103" dir="F:\GIT-DEV103"/>
<localRepo id="wm99" dir="F:\GIT-DEV99" layout="legacy">
```

In the example, the local repository "wm103" uses the layout "default", because it has no "layout" attribute. The local repository "wm99", however, uses a layout "legacy".

As of this writing, WxIspm includes only the "default" layout. The author is ready to implement other layouts on demand. Apart from that, custom layouts can be implemented as a plugin.

Remote repository handlers

Remote repository handlers are to remote repositories, what local repository layouts are to local repositories: They adapt WxIspm to the projects conventions. Additionally, they provide the interface to the respective version control system (Git, Subversion, etc.).

WxIspm comes with a default handler called "default": That handler is, in fact, an adapter to Azure DevOps. The same handler is available, if you use the Id "azure". Suggest the following remote repository definitions:

```
<remoteRepo id="azure"
           url="https://dev.azure.com/customer/projectGroup"/>
<remoteRepo id="azure2"
           url="https://dev.azure.com/otherCustomer/otherProjectGroup"
       handler="azure"/>
```

Both remote repositories would use the same handler "azure": The first one, because it doesn't have a "handler" attribute, thus uses the "default" handler, which is the same than the "azure" handler. The second one specifies the "azure" handler explicitly.

As of this writing, WxIspm includes only the "azure" layout, which is available under the id's "azure", and/or "default". This handler assumes, that you are using the Git vcs (version control system). The author is ready to implement other handlers on demand. Apart from that, custom handlers can be implemented as a plugin.

Plugins

A plugin is a WxIspm extension, that provides additional, custom components, for example local repository layouts, or remote repository handlers. Plugins can either be implemented as Java classes, or as Groovy scripts.

Java-Plugins

A Java plugin is a Java class, that implements a module. The class must have a public default constructor.

The plugin class is typically located in a Jar file, that's placed in *<IS_INST_DIR>/packages/WxIspm/config/plugins/lib*, or *<IS_INST_DIR>/config/packages/WxIspm/plugins/lib*. (The former location is preferred, if you are distributing a custom version of WxIspm, in which your plugins should be integrated. The latter location has the advantage, that it survives a redeployment of WxIspm. It is therefore better suited, if you use the standard version of WxIspm.)

A Java plugin is specified in the ispm config file, like this:4

```
<plugin class="fully.qualified.plugin.className"/>
```

Groovy-Plugins

A Groovy-Plugin is a Groovy-Script, that returns a module. Groovy scripts have the obvious advantage, that they are easier to maintain, and change. The WxIspm author expects, that custom plugins will be mostly written in Groovy. To help in the implementation of plugins, and as a test bed for the Groovy plugins, WxIspm comes with a set of plugin implementations:

1. The local repository handler "default" is implemented as a Groovy plugin.
2. The remote repository handler "azure" is implemented as another Groovy plugin

So, feel free to write your own plugins. Additionally, feel free to contact the WxIspm author for help.

Modules

When writing your own plugins, you should understand, what a Module is. In a few words:

Internally, WxIspm is using a dependency injection framework.

There is a thing, called component factory Whenever you need to do something, and may reasonably assume, that there's a preexisting piece of code, that you should be able to use, then ask the component factory for it. For example, if you need to invoke Git, then you'd ask for the "GitHandler".

That works, because initially, there is a module, that tells the component factory "If anyone's asking for a GitHandler, then give him..."

The beauty of the system is, that it makes some things extremely easy, like mocking (just tell the component factory to return a mock. rather than the real thing), or implementing minor changes (tell the component factory to return a proxy object, that internally uses the real component).

For more details on dependency injection, and modules, I very much recommend that you study the docs for Google Guice.

Services

In this section, we provide a list of the public services, that WxIspm provides. Or, in other words: A list of services in the namespace **wx.ispm.pub**.

Service	Short description
addInstance	Adds a new instance to the WxIspm configuration.
compilePackages	Compiles the Java services in one, or more, packages
compileAndReloadPackages	Compiles the Java services in one, or more, packages, and reloads those packages afterwards.
reloadPackages	Reloads one, or more, packages.

addInstance

The service **wx.ispm.pub.config:addInstance** creates a new instance in the WxIspm configuration. After creating the instance, WxIspm reinitializes itself.

The behaviour is equivalent to entering the instance manually into the ispm config file, and reloading the package WxIspm.

Input Parameters

- *id* The instance id (Required). Note, that the instance id must be unique with regard to all other instances. (However, it is perfectly valid to use the same id for an instance, and a local repository.)
- *baseDir* The instance directory, for example *F:\SoftwareAG\webMethods103\IntegrationServer\instances\default*. (Required).
- *wmHomeDir* The instances webMethods installation directory. Optional, defaults to *${baseDir}/../../..*.
- *packagesDir* The instances "packages" directory. Optional, defaults to *${baseDir}/packages*.
- *configDir* The instances "config" directory. Optional, defaults to *${baseDir}/config*.
- *logsDir* The instances "logs" directory. Optional, defaults to *${baseDir}/logs*.
- *properties* An optional document with instance properties. Suggested property keys for enabling remote access to the instance are
    - is.admin.url (Example: **http://127.0.0.1:10555/**)
    - is.admin.user (Example: **Administrator**)
    - is.admin.pass (Example: **manage**)

Output Parameters

None

See also

- addLocalRepository
- addRemoteRepository
- addPlugin

addLocalRepository

The service **wx.ispm.pub.config:addLocalRepository** creates a new local repository in the WxIspm configuration. After creating the local repository, WxIspm reinitializes itself.

The behaviour is equivalent to entering the local repository manually into the ispm config file, and reloading the package WxIspm.

Input Parameters

- *id* The local repository id (Required). Note, that the instance id must be unique with regard to all other local repositories. (However, it is perfectly valid to use the same id for an instance, and a local repository.)
- *dir* The repositories directory, for example *F:\GIT-DEV103*. (Required).
- *layout* The id of the local repository layour. Optional, defaults to "default".
- *properties* An optional document with properties, that the local repository should have.

Output Parameters

None

See also

- addInstance
- addRemoteRepository
- addPlugin

addRemoteRepository

The service **wx.ispm.pub.config:addRemoteRepository** creates a new instance in the WxIspm configuration. After creating the remote repository, WxIspm reinitializes itself.

The behaviour is equivalent to entering the remote repository manually into the ispm config file, and reloading the package WxIspm.

Input Parameters

- *id* The remote repository id (Required). Note, that the instance id must be unique with regard to all other remote repositories. (However, it is perfectly valid to use the same id for an instance, or a local repository, and a remote repository.)
- *url* The remote repositories URL, for example **https://dev.azure.com/customer/projectGroup**.

compilePackages

The service **wx.ispm.pub.admin:compilePackages** invokes the Java compiler to convert the Java service descriptors of one, or more packages, into class files. This service is intended as a replacement for the service **wm.server.packages:compilePackage**. The latter service is generally fine, except that, whenever there is a problem, it simply reports "Unable to locate compiler", leaving absolutely no clue about the cause. The WxIspm service is designed to work in a number of cases, when the IS service doesn't. If there's no other help, it should at least provide a clear explanation, what's wrong.

Input Parameters

- packageNames A string list of package names, that are being compiled.

Output Parameters

- messages A list of messages, that indicate what has (or hasn't been done).

compileAndReloadPackages

The service **wx.ispm.pub.admin:compileAndReloadPackages** invokes the Java compiler to convert the Java service descriptors of one, or more packages, into class files. This service is intended as a replacement for the service **wm.server.packages:compilePackage**. The latter service is generally fine, except that, whenever there is a problem, it simply reports "Unable to locate compiler", leaving absolutely no clue about the cause. The WxIspm service is designed to work in a number of cases, when the IS service doesn't. If there's no other help, it should at least provide a clear explanation, what's wrong.
After compiling, the service will also reload those packages. This is useful, because you'd need to do that anyways, afterwards.

Input Parameters

- packageNames A string list of package names, that are being compiled, and reloaded.

Output Parameters

- messages A list of messages, that indicate what has (or hasn't been done).

deployPackages

The service **wx.ispm.pub.admin:deployPackages** deploys one, or more IS packages from the local IS to a remote system. It does so by running *ABE* on those packages, thus creating a set of deployable files, which are then handed over to *Deployer*.

Basically, this is what a CI/CD system typically does, so nothing that a developer would typically need, at least not in a well structured project. The use case are those rare occasions, where you wish to ignore, or circumvent the CI/CD system. Typical examples would be:

1. The CI/CD server (or important parts of it) is down.
2. The target server is not in the scope of the CI/CD system.
3. Local testing is not possibleon your instance,, and you need a deployment after every commit in order to have your local development instance, and the local test instance in sync.

Note: This service is not yet implemented. It's only in the planning stage, and expected to be available in WxIspm, version 1.1.

Input Parameters

- *packageNames* A list of package names, which are being deployed. (Required)
- *instanceId* ID of an IS instance, which acts as the target system. (Optional)
- *isAdminUrl* URL of the target servers IS Administration UI. (Optional)
- *isAdminUser* Administrative user on the target server. (Optional)
- *isAdminPass* Password of the administrative user. (Optional)

For access to the target server, you must either

- Give the *instanceId* parameter (in which case, the properties *is.admin.url*, *is.admin.user*, and *is.admin.pass* must be present on that instance, or

- give the *isAdminUrl*, *isAdminUser*, and *isAdminPass* parameters.

Output Parameters

Action Output Parameters

See also

- addInstance

importProjectFromLocalRepository

The service **wx.ispm.pub.admin:importProjectFromLocalRepository**

is invoked to make a project's packages available in the currently running IS for local development mode. More precisely, the service performs the following steps:

1. The selected project is scanned for Is packages. (In the case of the default local repository layout, this means, that subdirectories in the folder **<PROJECT_DIR>/IntegrationServer/packages** are being scanned for subdirectories.)
2. For every package in the project:
    1. If not already present: A symbolic link (Linux, Unix, MacOS), or a directory junction (Windows) is being created in the instances "packages" directory, pointing to the projects package directory.
    2. If the symbolic link had to be created: Activates the package in the currently running IS. Otherwise, reloads the package.

Input Parameters

- *localRepoId* The id of the local repository, from which a project is being imported.
- *projectId* The imported projects Id.

Output Parameters

Action Output Parameters

See Also

- importFromRemoteRepository

importProjectFromRemoteRepository

The service **wx.ispm.pub.admin:importProjectFromRemoteRepository** is invoked to clone a project from a remote repository to a local repository, and then make the project's packages available in the currently running IS for local development mode. More precisely, the service performs the following steps:

1. Checks, if a project with the given Id is already present in the local repository. If that is the case, refuses operation, unless the parameter *overwriteExisting* is given (in which case the

project in the local repository would be deleted).
2. Clones the project from the remote repository to the local repository by doing a "git clone", a "svn checkout", or whatever the remote repository handler deems appropriate.
3. Performs necessary preparations on the cloned repository (Examples: Configuring git's *user.name*, and *user.email* settings, switching to the correct branch, etc.)
4. Invokes importProjectFromLocalRepository to import the cloned project into the currently running Is.

Input Parameters

- *remoteRepoId* The remote repository, from which to clone the project.
- *localRepoId* The local repository, to which the project is being cloned.
- *projectID* The project, which is being imported.

Output Parameters

Action Output Parameters

See also

- importProjectFromLocalRepository

reloadPackages

The service **wx.ispm.pub.admin:reloadPackages** is a replacement for **wx.server.packages:packageReload**. In fact, the former service invokes the latter internally. The neat thing is, that the WxIspm service is generally quicker, than the latter, because it applies a simple trick:

1. Before reloading package A, it temporarily removes all dependencies that other packages have on A.
2. It reloads package A. This is running quicker as usual, because, right now, no package depends on A. So, reloading A is sufficient, and no other packages need reloading. (Keep in mind, that WxIspm is running on a developers IS, so there's no harm in that.)
3. After package A is reloaded, the dependencies on A are restored, and everything's back to normal.

Input Parameters

- packageNames A string list of package names, that are being reloaded.

Output Parameters

- messages A list of messages, that indicate what has (or hasn't been done).

Configuration

WxIspm is configured by a set of files, that we'll cover in the following section:

Locations

For all files, a default version exists, that is present in the directory **<INST_DIR>/packages/WxIspm/config**. If you intend to change the defaults, then it is recommended to begin by copying the default version to **<INST_DIR>/config/packages/WxIspm**, and editing the copy. By doing so, you can redeploy a new version without loosing your changes.

Logging

WxIspm uses Apache Log4J, version 2, for internal logging purposes. Configuration is done by reading the file log4j2.xml, either from **<INST_DIR>/config/packages/WxIspm** (preferred), or from **<INST_DIR>/packages/WxIspm/config**.

Main configuration

The main configuration file is the file **ispm-configuration.xml**, again located in either of the standard locations. The purpose of the file is to specify the instances, local repositories, remote repositories, and plugins, plus object specific properties. This file may be validated against the schema in **<INST_DIR>/packages/WxIspm/config/ispm-config.xml**.

Global properties

Finally, there is a file with default properties (also called global properties). Actions, repository layouts, and handlers, are supposed to prefer their object specific properties, but should fall back to the default properties, if necessary. For example, the remote repository handler "azure" will read the properties
git.user.name, or *git.user.email* from the remote repository properties, if they are present there. If not, the handler will read the global properties.

The global properties are being read from a file, which is named **ispm.properties**, which is searched in the standard locations.

Additionally, there is a file named **ispm-factory.properties** with properties, that aren't supposed to be changed. You aren't supposed to change this file. If you need to customize properties, use the **ispm.properties**, which take precedence over the factory properties.