TIBCO® Enterprise Administrator

Developer Guide

Version 2.4.2 | June 2024

Contents

TIBCO Enterprise Administrator Concepts

TIBCO® Enterprise Administrator provides a centralized administrative interface to manage and monitor multiple TIBCO products deployed in an enterprise.

You can perform common administrative tasks such as authenticating and configuring runtime artifacts across all TIBCO products within one administrative interface. You can also manage products that do not have a complete administrative interface, providing you a unified and simplified administrative experience.

The following are the salient features of TIBCO Enterprise Administrator:

- **Centralized Administration**: TIBCO Enterprise Administrator provides a single-point access to multiple products deployed across an enterprise. You can easily manage and monitor runtime artifacts.
- **Simple to use**: TIBCO Enterprise Administrator is simple to install, develop, use, and maintain.
- **Shared Services Model**: TIBCO Enterprise Administrator shares common administrative concepts across all products thereby promoting a consistent and reusable shared services model.
- **Pluggable and Extensible**: As your enterprise evolves, you can add new products to the TIBCO Enterprise Administrator.
- **Rich set of APIs**: With TIBCO Enterprise Administrator Agent Library, organizations can develop custom TIBCO Enterprise Administrator agents to manage TIBCO and non-TIBCO products and applications. TIBCO products such as TIBCO ActiveMatrix BusinessWorks™ and TIBCO® MDM provide agents for TIBCO Enterprise Administrator. If you have installed the TIBCO Enterprise Administrator SDK variant, you can develop your own agents to expose your product on TIBCO Enterprise Administrator. The SDK variant comes with a set of APIs that is both declarative and extensible. You can develop your own agents and decide what part of your product needs to be rendered on TIBCO Enterprise Administrator.
- **Support for Interactive Shell**: TIBCO Enterprise Administrator provides a command-line utility called TIBCO Enterprise Administrator Shell. You can use the shell to perform almost all the tasks offered by the web-based GUI.

TIBCO Enterprise Administrator SDK Architecture

TIBCO Enterprise Administrator SDK consists of two main components—the TIBCO Enterprise Administrator server and the agent library. An *agent* is a program that mediates between the TIBCO Enterprise Administrator server and a specific product. You can use the agent library to develop agents for any product.

The TIBCO Enterprise Administrator server has two distinct user interfaces—a web-based GUI and a command line shell interface.

To manage a product using TIBCO Enterprise Administrator, the corresponding agent must be registered with the TIBCO Enterprise Administrator server.

To monitor and manage your product with TIBCO Enterprise Administrator

1. Develop an agent.
2. Compile and start the agent.
3. Register the agent with the TIBCO Enterprise Administrator server.

You can develop agents for any product using the agent library. The TIBCO Enterprise Administrator SDK includes sample agents that demonstrate aspects of agent development.

Develop an agent

An agent represents your product in TIBCO Enterprise Administrator. An agent identifies the assets of the product that must be rendered on the TIBCO Enterprise Administrator. Use the agent library to model the set of assets available in your product. The agent library provides you with five basic concepts to represent your product on TIBCO Enterprise Administrator. The concepts are: Process, Application, Resource, Access_Point, and Top_Level. For example, in the ActiveMatrix world, a node is an operating system Process, the DAA file is an Application, an environment is a Group, an enterprise is a Top_Level concept, and a SOAP endpoint is the Access_Point. In this manner, assets available in your product can be modelled into a concept provided by the agent library.

Every asset can have attributes, actions, and relationships associated with it. For example, some *attributes* of an ActiveMatrix node are the name of a node, the default state, and the location of the node. Creating a node, starting or stopping a node are the *actions* that can be performed on the node. The correlation that the node has with its environment is the

relationship it shares with the environment. As an agent developer, you must start by identifying the assets that must be modelled using the agent library. You then must define the attributes, actions, and relationships of each asset.

Compile and Start the Agent

After developing the agent, compile and start the agent by running the appropriate ant scripts.

Register an Agent with the Server

After starting the agent, register the agent with the TIBCO Enterprise Administrator server.

Components of TIBCO Enterprise Administrator

The TIBCO Enterprise Administrator comprises a server, an agent corresponding to a product, a server UI, a shell interface, and python scripts.

The TIBCO Enterprise Administrator has the following components:

The Server

The server is the equivalent of a web server. The server is hosted within a web server and caters to the HTTP requests coming from the browser. The server manages the communication between the browser and agents. The server interacts with the agent to get data about the products registered on the TIBCO Enterprise Administrator. The server is responsible for:

- Collecting data on all the products registered with it
- Maintaining a cache of the data; thereby promoting faster searches
- Hosting all the TIBCO Enterprise Administrator server views
- Responding to auto-registration requests from agents
- Providing details about the machines on which the products are running
- Providing user management features such as granting and revoking a user's permissions

The Agent

An agent is a bridge between the TIBCO Enterprise Administrator server and a product. When an agent is registered with the TIBCO Enterprise Administrator, it discovers the product that must be exposed to the administrator. The agent creates a graph of objects specific to the product that needs to be rendered on the TIBCO Enterprise Administrator server UI. The agent interacts with the server using the REST API. TIBCO Enterprise Administrator agents can run in any of the following ways: standalone, embedded, or hosted. TIBCO Enterprise Administrator comes with an extensible API that helps you develop your own agents for your products. An agent provides the following basic concepts:

- Group: is a container of artifacts. For example, a cluster, domain, and ActiveMatrix environment.
- Process: is any operating system process. For example, a BusinessWorks engine, and ActiveMatrix node.
- Resource: is a shareable configuration or artifact. For example, a JMS connection, or a port number.
- Application: is any deployable archive. For example, a WAR and DAA.
- Access_Point: is a means of interacting with an application. For example, an ActiveMatrix service endpoint, or an EMS queue.
- Top_level: A special type that represents the root-level object in the tree. There can be only one such instance of the object per agent. This is the only object that cannot have a configuration or state. Note that methods that access objects of this type do not have the `key` argument that is otherwise required by other concepts.

Web UI

TIBCO Enterprise Administrator provides a default UI to manage and monitor products. You can customize labels and icons on the UI to match the object types of your product. You can add more views to suit your product requirements.

Shell

TIBCO Enterprise Administrator provides a command-line utility called the TIBCO Enterprise Administrator shell. It is a remote shell based on the SSH protocol. The Shell is accessible using any terminal program such as Putty. The scripting language is similar

to bash from UNIX, but has important differences. You can use the Shell to perform almost all the tasks offered by the server UI.

Python Scripting

You can use Python scripting to perform any activity you performed using the Web UI. Python scripting is especially useful when you have to repeat a task for multiple users or use control structures to work through some conditions in your environment. Although you can use the Shell utility to use the command-line UI, the Shell UI does not support conditional statements and control structures. Python scripting proves to be useful in such cases.

Running the HelloWorld Sample

You can begin to explore TIBCO Enterprise Administrator SDK by running the Hello world sample.

Before you begin

- JRE and Apache Ant are installed. For details on the versions, see `readme.txt`.
- The jar files that implement the TEA agent library are installed
- TIBCO Enterprise Administrator server is running, and your browser can connect to it

Procedure

1. In a command shell, navigate to `<TIBCO_HOME>/tea/tea_version/samples/helloworld`.
2. Run ant.

   The default ant target compiles and runs the sample program.
3. Register the HelloWorld agent with the TIBCO Enterprise Administrator server.
   a. In a web browser, navigate to the server home page.
   b. In the **Settings** pane, click the **Agents** icon.
   c. Click the **Register new** button.
   d. Enter a name for the agent; for example, HelloWorld Agent.
   e. Enter the URL where the agent accepts requests from the TIBCO Enterprise Administrator server: http://localhost:1234/helloworldagent
   f. Enter a description string for the sample agent; for example, My sample HelloWorld agent.
   g. Click the **Register** button to complete registration.
4. Test the HelloWorld agent.
   a. In the TIBCO Enterprise Administrator server GUI, navigate to the home page.
   b. In the **Products** pane, click the **HelloWorldTopLevelType** icon.

This action navigates to the HelloWorld product page.

c. Click the **hw** button, which invokes the agent's only operation.

d. In the dialog, select **Hello World Agent** as the operation target.

e. Enter a string in the **greetings** field.

f. Click the **hw** button to complete the operation.

The agent outputs a greeting in the alert box.

g. Click the **Dismiss** button to close the alert box.

What to do next

To learn more about the sample code, see The HelloWorld Sample: Lessons for Agent Developers.

The HelloWorld Sample: Lessons for Agent Developers

The HelloWorld sample is a minimum viable agent program. It illustrates a lower bound on code complexity, API support and operating environment. Careful examination of its code can yield valuable information for novice agent developers.

The HelloWorld sample is a stand-alone agent process, which runs in a Jetty server.

We have coded the sample's top-level object in the interface style. That is, we define the class `HelloWorldAgent` to implement the `TopLevelTeaObject` interface. (For background information, see Object Type Definition: Overview and Interface Style.)

We begin by defining the five required methods of that interface—`getName`, `getDescription`, `getMembers`, `getTypeName`, `getTypeDescription`. For the purposes of this sample, all five methods do the minimum work—`getMembers` returns an empty list, while the others each return a constant string.

```
public class HelloWorldAgent implements TopLevelTeaObject{
    private static final String NAME = "hw";
    private static final String DESC = "Hello World";

    @Override
```

```
    public String getDescription() {
        return DESC;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public Collection<BaseTeaObject> getMembers() {
        return Collections.EMPTY_LIST;
    }

    @Override
    public String getTypeDescription() {
        return "Top level type for HelloWorld";
    }

    @Override
    public String getTypeName() {
        return "HelloWorldTopLevelType";
    }
```

We also define a method named `helloworld`, and annotate it as a `@TeaOperation`. This method implements the one operation that this agent offers to users. If you ignore the annotations, you can see that the method itself is simple; it concatenates two strings (one of which is the method's argument) and returns the resulting string. The `@TeaOperation` annotation makes this method available in the TIBCO Enterprise Administrator server as an operation of the agent. Java introspection of the annotation gives the server the operation's name and description. The `@TeaParam` annotation tells the server to prompt the user for the operation's argument.

```
    @TeaOperation(name = NAME, description = "Send greetings")
    public String helloworld(
            @TeaParam(name = "greetings", description = "Greetings
parameter") final String greetings) throws IOException {
        return "Hello " + greetings;
    }
```

The `main` method initializes the agent process:

```
    public static void main(final String[] args) throws Exception {
        TeaAgentServer server = new TeaAgentServer("HelloWorldAgent",
"1.0", "Hello World Agent", 1234, "/helloworldagent", true);
        server.registerInstance(new HelloWorldAgent());
        server.start();
    }
```

1. It instantiates a `TeaAgentServer`—an object that wraps a Jetty instance. This instance, `server`, represents the dedicated Jetty server within which the agent runs.

   (Notice that we must distinguish between two distinct *servers*—the Jetty web server instance, and the TIBCO Enterprise Administrator server.)

   The fourth and fifth arguments to the constructor—`port` and `contextPath`—together specify the URL where agent listens for requests from the TIBCO Enterprise Administrator server.

2. It creates a singleton instance of the agent's top-level object, `HelloWorldAgent`.

3. `server.registerInstance` introspects that instance, gathering metadata from the annotations (`@TeaOperation` and `@TeaParam`, above).

   The TIBCO Enterprise Administrator server later requests this metadata from the Jetty server, and uses it to implement its user interface for the agent.

4. `server.start` starts the Jetty server and deploys the agent code within it.

Enabling Developer Mode

Developer mode offers tools for agent and GUI developers. To use these tools, you must enable developer mode in the server's configuration file.

Procedure

1. Configure developer mode.
    a. Open the file `<TIBCO_CONFIG_HOME>\tibco\cfgmgmt\tea\conf\tea.conf` in a text editor.
    b. Enable the property `tea.dev.developer-mode`.

    ```
    tea.dev.developer-mode=true
    ```

    c. Save the file.
2. Restart the TIBCO Enterprise Administrator server (to use the modified configuration).

Reloading an Agent during Iterative Development

When agent metadata and static resources change during the development cycle, you can reload the agent to the server.
When you register an agent, the server requests and caches the agent's metadata and static resources. This information is likely to change during iterative development. Instead of unregistering the agent and then re-registering it, it is faster and more convenient to *reload* the agent, which caches the new information in the server.

Before you begin

Development mode must be enabled for the server.

The agent must already be registered. The server must have connected to the agent at least once, before modification.

Four metadata items are integral to the agent. Reloading does *not* update these items. (If you change any of these items, then you must unregister and re-register the agent.)

- *type name* (For stand-alone agents, this is the `name` argument to the `TeaAgentServer` constructor. For servlet agents, this is the `agentName` argument to the `TeaAgentServlet.autoRegisterAgent` method.)
- *version* (For stand-alone agents, this is the `version` argument to the `TeaAgentServer` constructor. For servlet agents, this is an `init-param` in the agent's `web.xml` file.
- *agent ID* (This is a unique ID, which the agent code may set.)
- *product ID* (This is the type name of the top-level object.)

Reloading does *not* update role and permission metadata. (If you change roles or permissions, then you must unregister and re-register the agent.)

Procedure

1. Restart the agent.

   That is, after modifying the agent code, stop the old agent, and start the modified agent.
2. Navigate to the **Agent Management** page of the server GUI.
3. Select the agent to reload.
4. Click the **Reload** operation button.

Developer Documentation

When developer mode is enabled, the **Help** screen lets you access additional help pages specifically for developers.

Click the operation button **Go to Developer Documentation**.

Viewing the Structural Overview Diagram of an Agent

Developer mode lets GUI developers see a structural overview of an agent's managed object model. The overview includes agent type, object types, operations, parameters, return types, and developer notes. You can collapse and expand the branches of the overview diagram.

Before you begin

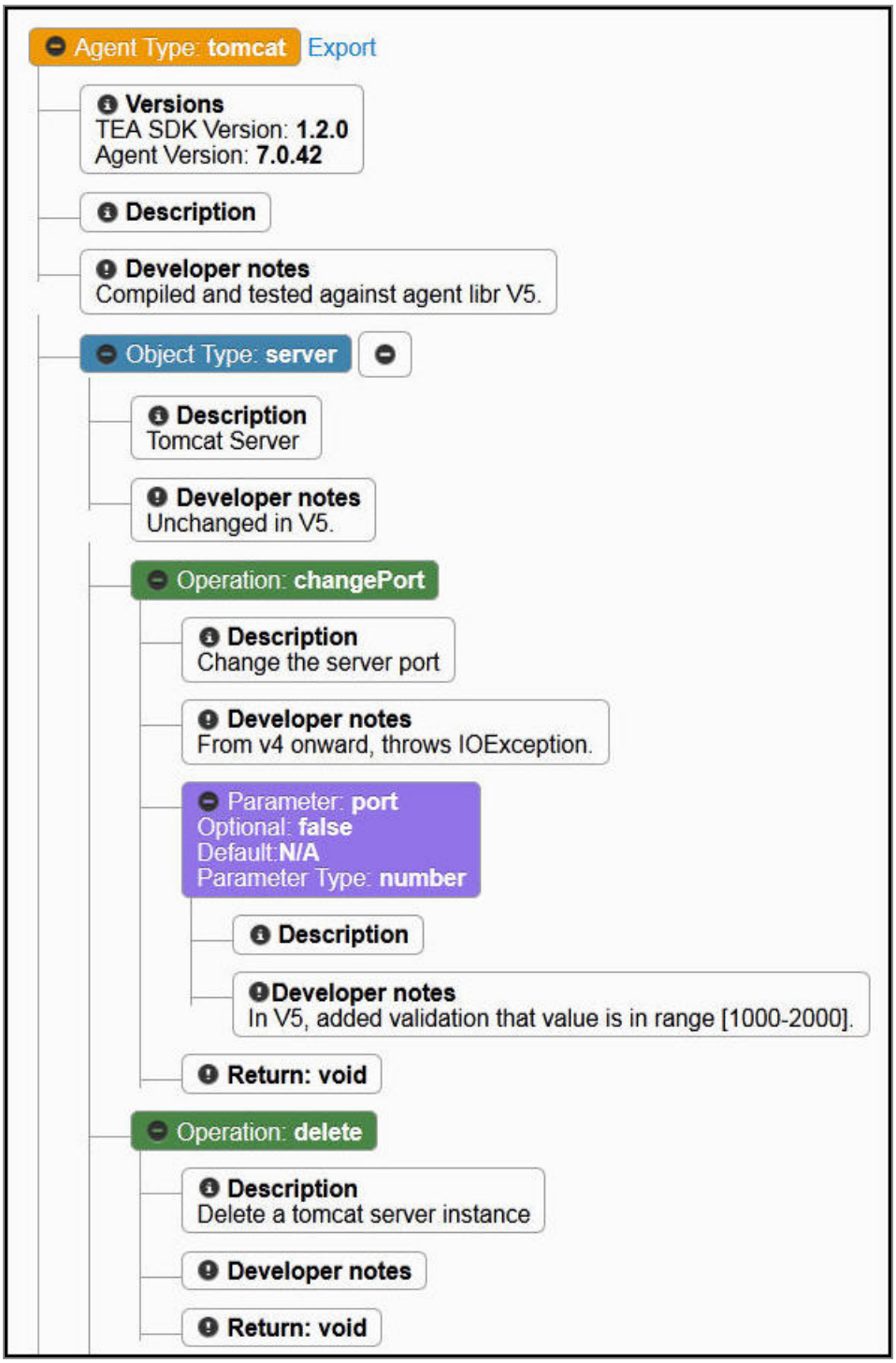
Enable developer mode.

Procedure

1. Navigate to an agent page.
    a. Navigate to the home page.
    b. In the **Settings** pane, click the **Agents** icon.

       The server GUI displays the **Agent Management** page.
    c. Click an agent row.

       An **Open API Documentation** button appears among the operations for the agent. This button is visible only when developer mode is enabled. It is visible to all users.
2. Click the **Open API Documentation** operation button to view the agent's structural overview diagram.



3. Use the diagram to do the following:
    - Collapse or expand subtrees of the diagram using the - or + icons.

- Collapse or expand *all* the *operations* of an object type with one click using the **-** or **+** button to the right of that **Object Type** item in the diagram.
- Download a JSON file representing the information in the diagram by clicking the **Export** link to the right of the **Agent Type** item at the top of the diagram.

Exposing the Agent API to Python Scripts

TIBCO Enterprise Administrator includes a Python module that can expose any agent as a collection of objects in Python, by generating Python classes that mimic the object types defined by your agent, and constructing Python objects as necessary to reflect your agent's object instances. This exposure to Python is not automatic. Your agent must specifically request the exposure.
To expose the agent API over Python binding use the following in your agent code:

`TeaAgentServer.setExposePythonAPI(true)` - in server mode agent. The product is accessible from `EnterpriseAdministrator.products` dictionary

or

`TeaAgentServlet.setExposePythonAPI(true)` - in servlet mode agent. The product is accessible from `EnterpriseAdministrator.products` dictionary.

The default value for `setExposePythonAPI()` is `false` in which case you don't have to call the method at all. When set to false, the product is accessible from `EnterpriseAdministrator.products_with_provisional_apis` dictionary. TIBCO Enterprise Administrator will still provide access to the "provisional" APIs, but you will see a message saying that the API is provisional when you try to access it using `EnterpriseAdministrator.products_with_provisional_apis` function.

For more details, refer to the *TIBCO Enterprise Administrator User Guide*.

If agents developed with TEA Agent Library version prior to 2.0.0 are registered with TIBCO Enterprise Administrator server having version above or equal to 2.0.0, the product is considered as "provisional product" and is accessible from `EnterpriseAdministrator.products_with_provisional_apis` dictionary. This is because the `setExposePythonAPI` is not available in the pre-2.0.0 agents library.

Setting SSL Properties on the Agent

To enable SSL, you must set the SSL system properties on both the TIBCO Enterprise Administrator server and the Agent.

Refer to the SSL Properties section for details on the system properties to be set.

Procedure

1. On the Agent, you can set the SSL system properties in **one** of the following ways:
    - Set the properties using the API.

      For example,

      ```
      server.setKeystorePath(
      “/tea/keystore/httpserversslkeys.jceks”
      server.setKeyStorePath
      ("/tea/keystore/httpserversslkeys.jceks");
      server.setKeyStorePassword("password");
      server.setCertAlias("httpserver");
      server.setTrustStorePath
      ("/tea/keystore/httpserverssltrusts.jceks");
      server.setTrustStorePassword("password");
      server.setKeyManagerPassword("password");
      server.setWantClientAuth(true);
      server.setNeedClientAuth(true);

      server.setHttpClientKeyStorePath
      ("/tea/keystore/httpclientsslkeys.jceks");
      server.setHttpClientKeyStorePassword("password");
      server.setHttpClientCertAlias("httpclient");
      server.setHttpClientTrustStorePath
      ("/tea/keystore/httpclientssltrusts.jceks");
      server.setHttpClientTrustStorePassword("password");
      server.setHttpClientKeyManagerPassword("password");
      ```

    - Create an SSLContext and inject it into the TIBCO Enterprise Administrator server using the Agent API.

      To do so:

a. Create an SSLContext object. Follow the JDK documentation on the Oracle web site for instructions on how to do so.

b. Use the SSLContext API to set the configuration properties into the SSLContext instance. Follow the JDK documentation on the Oracle web site for instructions on how to do so.

c. Inject the SSLContext instance into the TEA Agent's HttpServer and HttpClient using one of the following APIs:

```
public TeaAgentServer(final String name, final String
version, final String agentinfo, final int port, final
String contextPath,
final Boolean enableMetrics, final SSLContext
sslContextForHttpServer, final SSLContext
sslContextForHttpClient)
```

or

```
public TeaAgentServer(final String name, final String
version, final String agentinfo, final String hostname,
final int port,
final String contextPath, final Boolean enableMetrics,
final SSLContext sslContextForHttpServer, final SSLContext
sslContextForHttpClient)
```

Note: If you choose not to specify the hostname parameter as shown in the first interface above, a default value of `localhost` is used for the hostname.

An example of using the first API above:

```
final TeaAgentServer server = new TeaAgentServer
("SSLTestAgent","1.1","Agent for SSL
test",port,"/ssltestagent",true,
sslContextForServer, sslContextForClient);
```

- Set the properties from the command line using these System.properties when running the Agent.

For example,

```
-
Dtea.agent.http.keystore="/Users/<username>/tea/keystore/httpse
rversslkeys.jceks"
-
Dtea.agent.http.truststore="/Users/<username>/tea/keystore/http
serverssltrusts.jceks"
-Dtea.agent.http.keystore.password="password"
-Dtea.agent.http.truststore.password="password"
-Dtea.agent.http.keymanager.password="password"
-Dtea.agent.http.cert-alias="httpserver"
-Dtea.agent.http.want.client.auth=true
-Dtea.agent.http.need.client.auth=true
-
Dtea.agent.http.client.keystore="/Users/<username>/tea/keystore
/httpclientsslkeys.jceks"
-
Dtea.agent.http.client.truststore="/Users/<username>/tea/keysto
re/httpclientssltrusts.jceks"
-Dtea.agent.http.client.keystore.password="password"
-Dtea.agent.http.client.truststore.password="password"
-Dtea.agent.http.client.keymanager.password="password"
-Dtea.agent.http.client.cert-alias="httpclient"
```

2. Start the Agent. If you did not set the system properties using the API or create and inject an SSLContext, then make sure to start the Agent in SSL mode by setting the properties through the command line as shown in the example in the last bullet item above.

SSL Properties

When configuring SSL on the TIBCO Enterprise Administrator, you must set some properties on both the TIBCO Enterprise Administrator server as well as the Agent.

Note: Setting the HttpClient properties on both the Agent and the TIBCO Enterprise Administrator server is mandatory **only** if you want to set up a two-way SSL configuration. You do not need to set the HttpClient properties if you want to set up a one-way SSL configuration or do not want to set up SSL at all. If you do not set the HttpClient properties on the Agent and the TIBCO Enterprise Administrator server, the HttpClients residing on both of them are configured to "Trust All".

To enable SSL on the TIBCO Enterprise Administrator server, set these properties for the HttpServer and HttpClient residing on the TIBCO Enterprise Administrator server:

TIBCO Enterprise Administrator Server Properties

Property	Description
Properties for the HttpServer on the TIBCO Enterprise Administrator server	
tea.http.keystore	The file name or URL of the key store location. For example: `tea.http.keystore = "/Users/<username>/tea/keystore/httpserversslkeys.jceks"`
tea.http.keystore-password	Password for the key store residing on the TIBCO Enterprise Administrator server. This is the password that was set when the key store was created. For example: `tea.http.keystore-password = "MyPassword"`
tea.http.cert-alias	Alias for the SSL certificate. The certificate can be identified by this alias in case there are multiple certificates in the trust store. For example: `tea.http.cert-alias = "httpserver"`
tea.http.key-manager-password	The password for the specific key within the key store. This is the password that was set when the key pair was created. For example:

Property	Description
	`tea.http.key-manager-password = "password"`
tea.http.truststore	The file name or URL of the trust store location. For example: `tea.http.truststore = "/Users/<username>/tea/keystore/httpserverssltrusts.jceks"`
tea.http.truststore-password	The password for the trust store. For example: `tea.http.truststore-password = "password"`
tea.http.want.client.auth	See section Guidelines to set the tea.http.want.client.auth and tea.http.need.client.auth Parameters below. This property is used for mutual authentication. For example: `tea.http.want.client.auth = true`
tea.http.need.client.auth	See section Guidelines to set the tea.http.want.client.auth and tea.http.need.client.auth Parameters below. This property is used for mutual authentication. For example: `tea.http.need.client.auth = true`
tea.http.exclude.protocols	The property to list the protocols to be excluded. To exclude multiple protocols, use comma as a delimiter. For example, `tea.http.exclude.protocols="SSLv3,TLS1"` If the property is *not* mentioned, the SSLV3 protocol is excluded. If TIBCO Enterprise Administrator server must support all protocols including SSLV3, set the property to be empty.

Property	Description
	For example, `tea.http.exclude.protocols=""` **Attention:** When connecting using HTTPS, some versions of the popular browsers may be configured to use SSLv3 as the protocol. If you have problems accessing secured TIBCO Enterprise Administrator server (by default the SSLv3 is disabled) using the browser, follow the browser's user guide to configure that browser to excludeSSLv3 protocol.
Properties for the HttpClient on the TIBCO Enterprise Administrator server Only required if you want to set up a two-way SSL configuration	
tea.http.client.keystore	The file name or URL of the key store location. For example: `tea.http.client.keystore = "/Users/<username>/tea/keystore/httpclientsslkeys.jceks"`
tea.http.client.keystore-password	The password for the key store residing on the client (Agent). For example: `tea.http.client.keystore-password = "password"`
tea.http.client.cert-alias	Alias for the SSL certificate. The certificate can be identified by this alias in case there are multiple certificates in the trust store. For example: `tea.http.client.cert-alias = "httpclient"`
tea.http.client.key-manager-password	The password for the specific key within the key store. For example: `tea.http.client.key-manager-password = "password"`
tea.http.client.truststore	The file name or URL of the trust store location.

Property	Description
	For example: `tea.http.client.truststore = "/Users/<username>/tea/keystore/httpclientssltrusts.jceks"`
tea.http.client.truststore-password	The password for the trust store. For example: `tea.http.client.truststore-password = "password"`
tea.http.client.exclude.protocols	The property to list the protocols to be excluded. To exclude multiple protocols, use comma as a delimiter. For example, `tea.http.exclude.protocols="SSLv3,TLS1"` If the property is *not* mentioned, the SSLV3 protocol is excluded. If TIBCO Enterprise Administrator server must support all protocols including SSLV3, set the property to be empty. For example, `tea.http.exclude.protocols=""` **Attention:** When connecting using HTTPS, some versions of the popular browsers may be configured to use SSLv3 as the protocol. If you have problems accessing secured TIBCO Enterprise Administrator server (by default the SSLv3 is disabled) using the browser, follow the browser's user guide to configure that browser to excludeSSLv3 protocol.

Agent PropertiesTo enable SSL on the Agent, set the following properties for the HttpServer and HttpClient residing on the Agent:

Property	Description
Properties for the HttpServer on the Agent	
tea.agent.http.keystore	The file name or URL of the key store location. For example: `tea.agent.http.keystore`

Property	Description
	= "/Users/ <*username*> /tea/keystore/httpserversslkeys.jceks"
tea.agent.http.keystore.password	Password for the key store residing on the Agent. This is the password that was set when the key store was created. For example: `tea.agent.http.keystore.password = "MyPassword"`
tea.agent.http.cert.alias	Alias for the SSL certificate. The certificate can be identified by this alias in case there are multiple certificates in the trust store. For example: `tea.agent.http.cert.alias = "httpserver"`
tea.agent.http.keymanager.password	The password for the specific key within the key store. This is the password that was set when the key pair was created. For example: `tea.agent.http.keymanager.password = "password"`
tea.agent.http.truststore	The file name or URL of the trust store location. For example: `tea.agent.http.truststore = "/Users/ <*username*> /tea/keystore/httpserverssltrusts.jceks"`
tea.agent.http.truststore.password	The password for the trust store. For example: `tea.agent.http.truststore.password =`

Property	Description
	`"password"`
tea.agent.http.want.client.auth	See section Guidelines to set the tea.http.want.client.auth and tea.http.need.client.auth Parameters below. This property is used for mutual authentication. For example: `tea.agent.http.want.client.auth = true`
tea.agent.http.need.client.auth	See section Guidelines to set the tea.http.want.client.auth and tea.http.need.client.auth Parameters below. This property is used for mutual authentication. For example: `tea.agent.http.need.client.auth = true`

tea.agent.http.exclude.protocols	The property to list the protocols to be excluded. To exclude multiple protocols, use comma as a delimiter.
	
	For example,
	`tea.http.exclude.protocols="SSLv3,TLS1"`
	If the property is *not* set either using system properties or using Agent Server API, the SSLV3 protocol is excluded. If TIBCO Enterprise Administrator Agent must support all protocols including SSLV3, set the property to be empty.
	
	For example,
	`tea.http.exclude.protocols=""`
	
	Attention: When connecting using HTTPS, some versions of the popular browsers may be configured to use SSLv3 as the protocol. If you have problems accessing secured TIBCO Enterprise Administrator server (by default the SSLv3 is disabled) using the browser, follow the browser's user guide to configure that browser

Property	Description
	to excludeSSLv3 protocol.
Properties for the HttpClient on the Agent Only required if you want to set up a two-way SSL configuration	
tea.agent.http.client.keystore	The file name or URL of the key store location. For example: `tea.agent.http.client.keystore = "/Users/<username>/tea/keystore/httpclientsslkeys.jceks"`
tea.agent.http.client.keystore.password	The password for the key store residing on the client (Agent). For example: `tea.agent.http.client.keystore.password = "password"`
tea.agent.http.client.cert.alias	Alias for the SSL certificate. The certificate can be identified by this alias in case there are multiple certificates in the trust store. For example: `tea.agent.http.client.cert.alias = "httpclient"`
tea.agent.http.client.keymanager.password	The password for the specific key within the key store. For example: `tea.agent.http.client.keymanager.password = "password"`
tea.agent.http.client.truststore	The file name or URL of the trust store location. For example: `tea.agent.http.client.truststore = "/Users/<username>`

Property	Description
	`/tea/keystore/httpclientssltrusts.jceks"`
tea.agent.http.client.truststore.password	The password for the trust store. For example: `tea.agent.http.client.truststore.password = "password"`

tea.agent.http.client.exclude.protocols	The property to list the protocols to be excluded. To exclude multiple protocols, use comma as a delimiter.
	
	For example,
	`tea.http.exclude.protocols="SSLv3,TLS1"`
	If the property is *not* set either using system properties or using Agent Server API, the SSLV3 protocol is excluded. If TIBCO Enterprise Administrator Agent must support all protocols including SSLV3, set the property to be empty.
	
	For example,
	`tea.http.exclude.protocols=""`
	
	Attention: When connecting using HTTPS, some versions of the popular browsers may be configured to use SSLv3 as the protocol. If you have problems accessing secured TIBCO Enterprise Administrator server (by default the SSLv3 is disabled) using the browser, follow the browser's user guide to configure that browser to excludeSSLv3 protocol.

Guidelines to set the tea.http.want.client.auth and tea.http.need.client.auth Parameters

Here are some guidelines for setting these parameters depending on the scenario you want to implement:

For this type of authentication...	setting the parameters in this combination...	results in...
Certification-based two-way authentication	http.want.client.auth = true http.need.client.auth = false	The TEA server asks the client (web browser or Agent) to provide its client certificate while handshaking. But the client chooses not to provide authentication information about itself, but the authentication process continues. So that would mean that the client certification is optional which in turn means that no certificate needs to be generated on the client. **End Result** The authentication process is successful.
	http.want.client.auth = false http.need.client.auth = true	The TEA server asks the client (web browser or Agent) to provide its client certificate while handshaking, but the client chooses not to provide authentication information about itself, the authentication process stops. So that would mean that the client certification is required which in turn means that a keypair and certificate must be generated on the client (Agent). **End Result** The authentication process fails
	http.want.client.auth = true http.need.client.auth = true	Same as the above case where the client certification is required and a keypair and certificate must be generated on the client (Agent). **End Result**

For this type of authentication...	setting the parameters in this combination...	results in...
		The authentication process fails
Certification-based one-way authentication	http.want.client.auth = false http.need.client.auth = false	Both of the parameters set to 'false' which means that it is a One-way Authentication, where only the client (web browser or Agent) verifies the TEA server but the TEA server trusts all the clients without verification. No need to generate any certificates at all. **End Result** The authentication process is successful, as long as the user name and password provided by the agent are both correct.

Support for IPv6 Addresses in TEA

Starting in version 2.2.0, TIBCO® Enterprise Adminsitrator (TEA) provides support for IPv6 address format.

By default, TEA uses IPv4 address format. To use IPv6 address format you must set the `java.net.preferIPv6addresses` system property to true. This property is set to false by default. Set this system property when running the agent:

```
-Djava.net.preferIPv6addresses=true
```

If you choose to use the IPv6 format, make a note of the following points:

- For TEA agents that are older than version 2.2.0 and are registered with TEA server version 2.2.0 or greater, the **IPv6 Address** column in the **Machines** view in the TEA Web user interface is blank. The IPv6 address for the agent's machine is not displayed in the IPv6 Address column.
- When registering an agent to a server, use square brackets around the IPv6 address. The following is an example of a URL that uses an IPv6 address format to register a Tomcat agent to the TEA server:

  `http://[FEDC:BA98:7654:3210:FEDC:BA98:7654:3210]:8082/tomcatagent`
- The syntax to use the `scp` command is as follows:

  ```
  scp -P 2222 admin@[10::4]:<filename>
  ```

  where 10::4 is the IPv6 address of the machine.

  The syntax to use the `sftp` command is as follows:

  ```
  sftp -P 2222 admin@[10::4]
  ```

  You can use the above command with or without the square brackets around the IP address.
- The syntax to use the `ssh` command is as follows:

  ```
  ssh -p 2222 admin@10::4
  ```

where 10::4 is the IPv6 address of the machine.

Setting up TIBCO Enterprise Administrator Agent Library

You can configure and setup the TIBCO Enterprise Administrator Agent library either in the server or the servlet modes.

Running TIBCO Enterprise Administrator Agent Library in the Server mode

In the server mode, the TIBCO Enterprise Administrator Agent runs as a standalone process. The TIBCO Enterprise Administrator Agent Library comes bundled with the Jetty server which is used for serving the agent service endpoints.

Procedure

1. To configure the TIBCO Enterprise Administrator Agent library to run in server mode, instantiate an object of the class `com.tibco.tea.agent.server.TeaAgentServer`. There are a few overloaded constructors available to instantiate the TeaAgentServer. The one used in this example takes the following arguments:

Option	Description
name	Name of the agent
version	Version of the agent
agentinfo	Description for the agent
hostname	Hostname for the jetty connector

Option	Description
port	Port for the jetty connector
context-path	Path for ServletContext
enable-metrics	Enables metrics for the TIBCO Enterprise Administrator SDK Agent Library

```
TeaAgentServer server = new TeaAgentServer("HelloWorldAgent",
"1.1", "Hello World Agent", 1234, "/helloworldagent", true);
```

2. To register Object Types with the TIBCO Enterprise Administrator Agent library, use any of the following: `com.tibco.tea.agent.server.TeaAgentServer.registerInstance()` and `com.tibco.tea.agent.server.TeaAgentServer.registerInstances()`. The `registerInstance()` method takes an instance of a TeaAgent as a parameter. The `registerInstances()` method takes a varargs parameter that receives a variable number of Object Types.

```
server.registerInstance(new HelloWorldAgent());
server.registerInstances(arg0);
```

3. You can configure the TIBCO Enterprise Administrator Agent library to customize the content specific to your requirements. The content includes HTML, CSS, javascript, images, and so on. Use `com.tibco.tea.agent.server.TeaAgentServer.registerResourceLocation()` to register the resource location that has these files.

```
server.registerResourceLocation(file);
```

4. (Optional) To disable the default search capability of an agent registered in the server, use the method `disableIndex()` on the server instance.

```
server.disableIndex();
```

5. After the TIBCO Enterprise Administrator agent server has been configured, use `com.tibco.tea.agent.server.TeaAgentServer.start()` to initiate and start the TIBCO Enterprise Administrator agent server.

```
server.start();
```

Starting the sample: HelloWorldAgent

```
TeaAgentServer server = new TeaAgentServer("HelloWorldAgent", "1.1",
"Hello World Agent", 1234, "/helloworldagent", true);
server.registerInstance(new HelloWorldAgent());
server.registerInstances(arg0)
server.registerResourceLocation(file);
server.start();
```

Running TIBCO Enterprise Administrator Agent Library in the Servlet mode

TIBCO Enterprise Administrator provides an abstract servlet that needs to be subclassed to register object instances.

Procedure

1. Extend the abstract servlet of TIBCO Enterprise Administrator to define the object that needs to be registered.

   The following example code defines only one object to register in the server.

Note: In the getObjectInstances method, instances of TopLevelTeaObject and TeaObjectProvider must be passed. For example, if you have a class implementing the TeaObject interface, and you write a class that implements a TeaObjectProvider to provide an instance of this class, while registering the instance in servlet mode, you must pass the TeaObjectProvider instance instead of the TeaObject instance itself in addition to the TopLevelTeaObject instance.

```
public class HelloWorldServlet extends TeaAgentServlet {

    /*
     * (non-Javadoc)
     *
     * @see
com.tibco.tea.agent.server.TeaAgentServlet#getObjectInstances()
     */
    @Override
    protected Object[] getObjectInstances() throws ServletException
{
        return new Object[]{new HelloWorldAgent()};
    }
}
```

2. Configure the servlet with proper parameters using `web.xml`. Map the agent servlet to `/*` as further dispatches are done by the servlet.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">

    <display-name>HelloWorld Agent</display-name>

    <servlet>
        <servlet-name>HelloWorldAgent</servlet-name>
        <servlet-class>HelloWorldServlet</servlet-class>
        <init-param>
            <param-name>name</param-name>
            <param-value>HelloWorldAgent</param-value>
        </init-param>
        <init-param>
            <param-name>version</param-name>
            <param-value>1.1</param-value>
```

```
        </init-param>
        <init-param>
            <param-name>agent-info</param-name>
            <param-value>HelloWorld Agent</param-value>
        </init-param>
        <init-param>
            <param-name>agent-id</param-name>
            <param-value>HelloWorldAgent</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldAgent</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

If you deploy the agent as a part of an existing web application, you can have a specific URL pattern matching instead of /* for your agent servlet. For example,

```
    <init-param>
        <param-name>agent-contextPath</param-name>
        <param-value>/path1/path2</param-value>
    </init-param>
    <servlet-mapping>
        <servlet-name>HelloWorldAgentServlet</servlet-name>
        <url-pattern>/path1/path2/*</url-pattern>
    </servlet-mapping>
```

3. Deploy the servlet using the standard servlet container mechanisms. To verify, you can start an embedded Jetty server, with the programmatically deployed servlet. For a standalone agent process, configuration through the server mode is a better approach.

```
public static void main(final String[] args) throws Exception {

    Server server = new Server(1234);
    ServletContextHandler servletContextHandler = new
ServletContextHandler(server, "/helloworldagent", false, false);

    ServletHolder servletHolder = servletContextHandler.addServlet
(HelloWorldServlet.class, "/*");
```

```
    servletHolder.setInitParameter("name", "HelloWorldAgent");
    servletHolder.setInitParameter("version", "1.1");
    servletHolder.setInitParameter("agent-info", "HelloWorld
Agent");
    server.start();

    server.join();
}
```

Note: It is recommended that you do not set the `load-on-startup` flag to `false` otherwise the servlet is not loaded on startup.

Agent ID

By default, TIBCO Enterprise Administrator uses the name provided during the registration of the agent to uniquely identify an agent. You can override that behavior by providing an agent identifier that will be used instead of the name.

Registering another agent with the same id automatically unregisters the previously registered agent. Ensure that the Agent identifier is set before agent server is started. For example:

```
server.setAgentId("uniqueAgentId");
server.start();
```

Configuring TIBCO Enterprise Administrator Agent for Auto-Registration

The TIBCO Enterprise Administrator agent library can be configured to auto-register itself with the TIBCO Enterprise Administrator server. When TIBCO Enterprise Administrator Agent comes up, the agent library connects to the server and registers itself. Ensure that the agent library is setup with the correct connection details for the server.

Note: If the TIBCO Enterprise Administrator server and the agent are on different machines, use the following constructor:

```
TeaAgentServer(String name, String version, String agentInfo,
String hostName, int port, String contextPath, boolean
enableMetrics)
```

Auto-Registering in the Server Mode

The following example shows how the TeaAgentServer can be configured to auto-register itself.

```
TeaAgentServer server =
new TeaAgentServer("HelloWorldAgent", "1.1", "Hello World Agent", 1234,
"/helloworldagent", true);
server.registerInstance(new HelloWorldAgent());
server.registerAgentAutoRegisterListener("http://localhost:8777/tea");
server.start();
```

`com.tibco.tea.agent.server.TeaAgentServer.registerAgentAutoRegisterListener()` is used to register the agent with the server. The method takes the server URL as the parameter.

Auto-Registering in the Servlet Mode

The following example shows how the TeaAgentServlet can be configured to auto-register itself.

```
public class HelloWorldAgentServlet extends TeaAgentServlet{
       private static final long serialVersionUID = 83270197184828944 67L;

       @Override
       public void init() throws ServletException {
               super.init();
this.autoRegisterAgent("http://localhost:8777",
"http://localhost:8080");
       }

       @Override
       protected Object[] getObjectInstances() throws ServletException {
               return new Object[] {new HelloWorld()};
       }
}
```

After making a call to the init method in the super class of TeaAgentServlet, call the`com.tibco.tea.agent.server.TeaAgentServlet.autoRegisterAgent()` method.

Note: The `com.tibco.tea.agent.server.TeaAgentServlet.autoRegisterAgent()` must be the last line in the code for the class. You cannot call any other method such as the `unregisterAgent()` method after this line.

Unregistering an Agent

The agent is unregistered by using the `TeaAgentServer.unregisterAgent()` method.

Note: Prior to version 1.1.0, when you unregistered an agent, the associated data got deleted automatically from `<TIBCO_HOME>\tea\1.0\data\sr\<Agent_Name>\<Version>\*.*`. Now, you can control this setting manually by setting a flag in the `<TIBCO_CONFIG_HOME>\tibco\cfgmgmt\tea\conf\tea.conf` file. Set `tea.dev.unregistration-cleanup=false` to delete the files manually. If you have set the property to false, remember to cleanup manually after unregistering an agent.

Developing an Agent

An agent lets administrative users monitor and manage a system of objects, which correspond to the moving parts of a software product. As an agent developer, your most important task is modeling that system of managed objects.

Procedure

1. Analyze the software product to understand its components and its administrative interactions. For detailed instructions, see Analyzing a System of Managed Objects.

   Your analysis produces a table and a diagram—artifacts that guide subsequent coding. For an example, see Example Analysis of Managed Objects for Tomcat.

2. Translate your model into a Java program using the API constructs in the SDK.

Managed Objects

Managed objects are entities in the world that users can monitor and manipulate using TIBCO Enterprise Administrator.

Administrative users interact with managed objects using the TIBCO Enterprise Administrator server.

Within the server, each agent models a set of interrelated managed objects. TIBCO Enterprise Administrator SDK lets you build agents that model managed objects.

When building an agent, the most obvious managed object is the *product* that you are modeling. For example, the sample Tomcat agent would surely include Tomcat server processes as managed objects.

You can also expose *components* of a managed object as managed objects in their own right. For example, the sample Tomcat agent exposes the web applications deployed within a Tomcat server.

Sometimes it is useful to model a *group* of managed objects as a managed object in its own right. For example, a Tomcat agent could model a group of Tomcat server processes as a Tomcat cluster.

Aspects of Managed Objects

Aspects of an object describe its internal structure and behavior, and its relationship to other managed objects. These aspects shape the way users view and interact with the object.

When you analyze a system of managed objects, you must describe the behavior of each object. Four aspects guide your analysis and contribute to the description. One or more of these aspects apply to every managed object:

- Configuration Aspect
- States
- Operations
- References

When coding an agent, these aspects translate into the building blocks of the SDK.

Configuration

The configuration of a managed object consists of name and value pairs that describe the object or affect its behavior.

Configuration can include any parameters of a managed object for which an administrator could supply values. For example, when creating a Tomcat server process, the administrator could supply a name for the server and an HTTP port number.

Configuration can also include attribute values that the administrator might need to know, but cannot modify. For example, the built-in model for a machine displays a computer's host name, IP address, operating system and hardware details.

The server GUI presents configuration parameters as name and value pairs. The default format lists the pairs in three columns, alphabetized by parameter name.

The value for a name can be either simple or complex (for example, the value could itself be a list of name and value pairs). However, the server GUI can display only simple values.

States

The states of a managed object reflect its operating states, from the administrator's point of view.

For example, a process could have states Running and Stopped. A data queue might have states DataAvailable, Empty and Full.

A managed object can be in only one state at a time.

The server GUI presents states with an icon and state name.

Operations

Operations include any commands that the administrator could use to manipulate a managed object.

For example, an administrator might start, stop, pause and resume a process; enable and disable communication on a port.

The server GUI presents operations as buttons.

References

References denote the relationships among the managed objects in a model.

For example, groups *contain* members, processes *listen* using HTTP connectors, file system directories *contain* files and other directories, and a web service might *depend* on a database.

The server GUI presents each relationship in a separate visual block.

When only one object stands in a relationship, the GUI presents its details.

When several objects stand in the same relationship, the GUI presents them in a table. Each table row represents one object in that relationship (for example, one group member). A column can display information about each object—either the object's current state, or one of its configurations (for configurations, the column header is the name, and the cells display the values).

Concept Types of Managed Objects

The *concept type* of an object describes its role within the larger system of managed objects.

When analyzing a system of managed objects, classify each object as one of six types. When coding an agent, these types translate into the enumerated constants of `TeaConcept`.

- Product (top level object)
- Application
- Process
- Access Point
- Resource
- Group

Product (Top Level Object)

The top level object represents a product or system as a whole.

In any system of managed objects, you must distinguish *exactly one* object as the top level object.

If the system is a product, then the top level object represents the product.

If the system is not a product, then the top level object represents the system as a whole. For example, the server's built-in User Management facility is a top level object.

A top level object usually has relationships to other objects. (The sample HelloWorld agent is a degenerate case, in which the model has no other objects, so there cannot be any relationships.)

A top level object can have operations. For example, in the sample Tomcat agent, creating Tomcat servers is an operation of the top level object.

A top level object cannot have configuration or state.

Top level objects appear as icons on the home page of the server GUI.

Application

An application object represents something that can execute, such as Java code, a shell script, a web app.

An application usually has configuration and state.

An application could have relationships to other objects.

An application could have operations, such as start and stop.

Process

A process object represents a process executing on a host computer.

A process usually has configuration and state.

A process could have relationships to other objects. For example, a process could refer to an application that runs within the process, or to access points.

An application could have operations, such as start and stop. In a Tomcat agent, a server process could have operations to deploy a web application, and to manage HTTP connectors.

Access Point

An access point object represents an endpoint or entry point that serves as the source or destination of a data stream. Examples include a network interface, HTTP connector, TCP port, WSDL endpoint, JMS topic or queue, or TIBCO Rendezvous transport.

An access point usually has configuration.

An access point usually has state; for example, it can be enabled or disabled. If the object had no state, then it might be simpler to think of it as a configuration of some other object.

An access point can have operations, for example to enable or disable it.

An access point can have relationships, for example, references back to the processes or applications that use it.

Resource

A resource object represents a shared resource, such as a file, thread pool, database connection pool, LDAP connection pool, or other objects defined using JNDI.

A resource usually has configuration.

A resource can have relationships, for example, references back to the objects that use it.

A resource can have state, for example, availability.

A resource can have operations. For example, administrators can adjust the parameters of thread pool, increasing its size.

Group

A group object represents a homogeneous set of objects.

For example, a Tomcat cluster could contain Tomcat servers.

A group usually has relationships to other objects (namely, the members of the group).

A group can have operations, for example to add and remove members. It is unusual for a group to have state and configuration.

Support for POJOs

TeaOperation supports Plain Old Java Object (POJO) as parameters and return types.

To enable POJO support, you must enable the `enable_class_generation` property on `EnterpriseAdministrator` as follows:

```
import tibco.tea
tea =tibco.tea.EnterpriseAdministrator(config={'enable_class_
generation':True})
```

The property is, by default, set to `false`, so ensure that you set this to `true` if you want the Python scripts to support POJO objects. POJO support comes with some supported scenarios and limitations.

Supported Scenarios

The following java objects are supported as a parameter or a return type:

1. A POJO
2. Array of POJO
3. List of POJOs
4. Map of POJOs
5. Nested POJOs
6. A list of POJOs passed as a parameter to a Map is supported as a return type.
7. If you have classes with same names in different packages, underscore separated fully qualified names are used to distinguish them. For example, if there is a class by the name, `TeaAgent` in `com.tibco.tea.agentA`, and `com.tibco.tea.agent.agentB`, the class in package `agentA` is identified by using `com_tibco_tea_agent_agentA_TeaAgent` and the class in `agentB` is identified by using `com_tibco_tea_agent_agentB_TeaAgent`.

Fully Qualified Name in Java	Fully Qualified Name in Python
`com.tibco.tea.agent.agentA.TeaAgent`	`com_tibco_tea_agent_agentA_TeaAgent`
`com.tibco.tea.agent.agentB.TeaAgent`	`com_tibco_tea_agent_agentB_TeaAgent`

Limitations of POJO

The POJO support comes with some limitations.

The following POJOs are not supported:

1. A Map cannot have a POJO as a key
2. Nested maps
3. Nested lists
4. A `List` that takes a `List` of POJOs as a parameter

Analyzing a System of Managed Objects

The first step in creating an agent for a system of managed objects is to analyze the system. Your analysis produces artifacts (a table and a diagram), to guide you as you code the agent.

Before you begin

To do this task, you must first understand managed objects:

- Managed Objects
- Aspects of Managed Objects
- Concept Types of Managed Objects

Note: Do the first three steps of this task in the order shown below. After that, you can do the remaining three steps in any order (you can even interleave them).

Procedure

1. Identify the entities in the system that users can monitor and manage.

   Organize this information as a table, with a row for each entity.

2. Classify the entities using concept types.

   See Concept Types of Managed Objects.

   Add the concept types as the second column of your table.

3. Identify relationships among the entities.

   a. Name each relationship.

   b. Draw a diagram of the entities (as nodes) and relationships (as edges)—separate from the table.

   c. Determine the cardinality of the relationship.

      Can this relationship include at most one other entity, or many entities? Add

the cardinality information to your diagram.

d. Copy the relationship information from your diagram into the third column of your table.

4. Determine the configuration for each entity—its parameters and attributes.

   Add this information as the fourth column of your table.

5. Determine the operations for each entity.

   What can users do with the entity?

   Add this information as the fifth column of your table.

6. Determine the states for each entity.

   Add this information as the sixth column of your table.

Example Analysis of Managed Objects for Tomcat

Analyzing a system of managed objects produces artifacts to guide you as you develop an agent. As an example, analysis of Tomcat might produce this table and this relationship diagram.

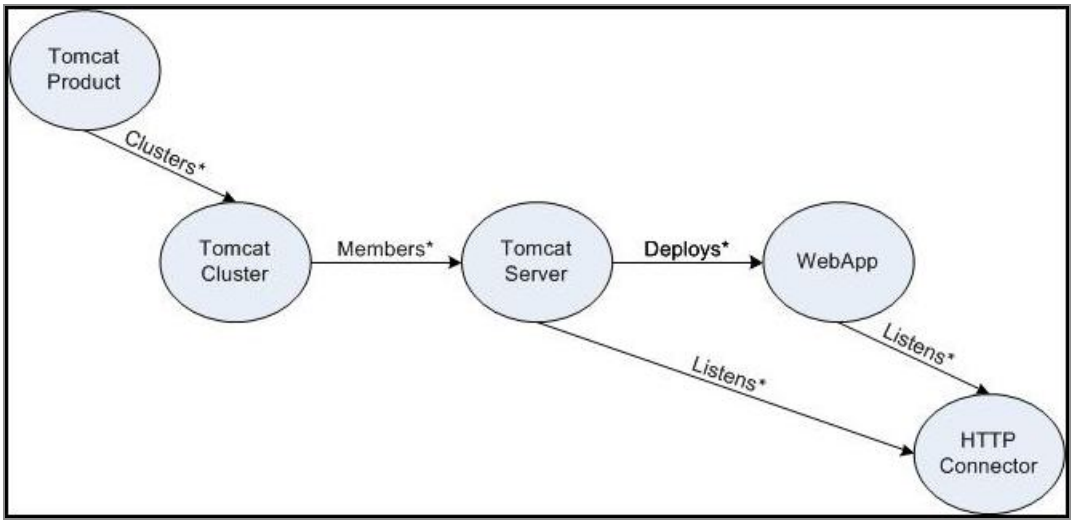
Managed Objects in Tomcat

Entity	Concept Type	Relationships	Configuration	States	Operations
Tomcat Product	Top Level Object	Manages* > Tomcat Cluster			Add Cluster Remove Cluster
Tomcat Cluster	Group	Members* > Tomcat Server			Add Server Remove Server
Tomcat Server	Process	Deploys* > WebApp	Installation Folder	Running	Start

Entity	Concept Type	Relationships	Configuration	States	Operations
		Listens* > HTTP Connector		Stopped	Stop Deploy App Add Connector Remove Connector
WebApp	Application	Listens* > HTTP Connector	Context Path	Running Stopped	Start Stop
HTTP Connector	Access Point		Port Number SSL Parameters	Enabled Disabled	Enable Disable

The third step might produce a diagram such as the following:

Relationships among Managed Objects in Tomcat



This example does not correspond exactly to the sample Tomcat agent. To illustrate a wider range of concept types, this example explicitly models the cluster and HTTP connector as managed objects. The sample code implements a simpler model, which omits these entities.

Sharing Data and Resources (TeaObjects) Between Agents

TIBCO Enterprise Administrator allows agents to share data or resources (TeaObjects) which can be used by other agents that are registered with the same TIBCO Enterprise Administrator server. For example, an LDAP resource that is created and shared by a TSS server could be used by the TIBCO Policy Director to create policies. TIBCO Enterprise Administrator provides some APIs that you can use to share data and resources.

To use this feature, you must set up the TIBCO Enterprise Administrator server to enable the data sharing API. Refer to the *TIBCO Enterprise Administrator SDK Installation* for details on how to set up the TIBCO Enterprise Administrator server for this feature.

Once the agent gets registered with the server, the proper Data Sharing configuration gets passed to the agent so TeaObjects can be shared and retrieved. No configuration is needed on the agent

The property `@objectId` contains the id of the TeaObject in the form of a `com.tibco.tea.agent.api.ObjectId`. This class is the recommended way to manage the `@objectId` property since it encodes and decodes the id accordingly. The objectId in the Data Sharing API agentId and agentTypeVersion are always empty.

Limitations

- All TeaObjects shared in the same AgentType must have an ObjectType name that is different when its non-alphanumeric values are converted to "_". The reason for that is that the ObjectType name is used as part of the Name of the DDBB where the shared object is stored. (For example: an invalid ObjectType name will be 2 objects with: "abc-def" and "abc.def" as object type names)
- There is currently no support in tea to change username and password for the DDBB. So, an external utility must be used for managing the DDBB.

How to Share a TeaObject

To share an object, use the `share()` method of the TeaPersistence class. Once an object is shared, it is available for other agents to use. When you share a TeaObject, you share all the references in that object too. For example, if you share a person object called Peter who has cousins (references in the object Peter) named Adam, Sarah, and Brian, when you share Peter, you effectively share objects Adam, Sarah and Brian too.

Here is a sample code which shares a TeaObject, John:

```
@TeaOperation(name = "sharePerson", description = "Share a Person
object")

public void sharePerson() throws IOException {
// Person implements TeaObject and has objectTypeName="Person"
//in this instance the key="John"
final Person person = new Person("John", "Public John Person");
//teaAgentServer is an instance of
//com.tibco.tea.agent.server.TeaAgentServer
//or//com.tibco.tea.agent.server.TeaAgentServletfinal TeaPersistence
persistence = teaAgentServer.getPersistence();
persistence.share(person);
}
```

How to Get Shared TeaObjects

Any TeaObjects that are shared by an agent can be used by other agents that are registered with the same TIBCO Enterprise Administrator server. The objects are returned in JSON format.

To get a single shared TeaObject use the `getObject()` method as shown in the example below:

```
@TeaOperation(name = "getJohn", description = "getJohn")
public String getJohn() throws IOException {
  //teaAgentServer is an instance of
com.tibco.tea.agent.server.TeaAgentServer or
    com.tibco.tea.agent.server.TeaAgentServlet
final TeaPersistence persistence = teaAgentServer.getPersistence();
  // We query the method getObject(String agentTypeName, String
agentTypeVersion, String objectTypeName,
```

```
  //   String key);
  // That returns a org.json.JSONObject, in this case the Json
Representation of Person with key John
final JSONObject object = persistence.getObject("agentTypeNameHere",
"Person", "john");
return object.toString();
}
```

Below is an example of how to get all shared TeaObjects for a specific TeaObjectType:

```
@TeaOperation(name = "getAllPersons", description = "getAllPersons")
public String getAllPersons() throws IOException {
//teaAgentServer is an instance of
com.tibco.tea.agent.server.TeaAgentServer or
  com.tibco.tea.agent.server.TeaAgentServlet
final TeaPersistence persistence = teaAgentServer.getPersistence();
final Map<String, JSONObject> allPersons = persistence.getObjects
("agentTypeNameHere", "Person");
return Joiner.on(",").join(allPersons.values());
}
```

How to Get Shared TeaObject Along with All the References

A TeaObject has references to all its members. When you get a TeaObject from the database, you have the option to get the high-level object or you can request the object with all the references included (Closure).

To retrieve a TeaObject Closure use the following code:

```
@TeaOperation(name = "getJohnClosure", description = "getJohnClosure")
public String getJohnClosure() throws IOException {
TeaPersistence persistence = ShareReaderAgent.server.getPersistence();

Map<String, JSONObject> johnClosure = persistence.getObjectClosure
("ShareWriterAgent", "Person", "john");
return Joiner.on(",").join(johnClosure.values());
}
```

The object returned by this method is in JSON format. Below is an example that is returned by the sample code above:

```
[
{
  "@objectId":"ShareWriterAgent::Person:Peter",
  "name":"Peter",
  "description":"Peter Person",
  "key":"Peter",
  "config":{
     "status":"Single",
     "address":"3303 Hillview Avenue"
},
  "members":[
     "ShareWriterAgent::Person:Adam",
     "ShareWriterAgent::Person:Sarah"
],
  "references":[
    {
     "name":"cousins",
     "elements":[
       "ShareWriterAgent::Person:Brian",
       "ShareWriterAgent::Person:Anna"
       ]
     },
     {
       "name":"parent",
       "elements":[
         "ShareWriterAgent::Person:John"
       ]
     },
     {
       "name":"cars",
       "elements":[
         "ShareWriterAgent::Car:BMW"
         ]
       }
      ]
     },
     {
       "@objectId":"ShareWriterAgent::Person:Adam",
       "name":"Adam",
       "description":"Adam Person",
       "key":"Adam",
       "config":{
          "status":"Single",
          "address":"3303 Hillview Avenue"
        }
     },
     {
```

```
      "@objectId":"ShareWriterAgent::Person:Sarah",
      "name":"Sarah",
      "description":"Sarah Person",
      "key":"Sarah",
      "config":{
         "status":"Single",
         "address":"3303 Hillview Avenue"
       }
    },
    {
      "@objectId":"ShareWriterAgent::Person:Brian",
      "name":"Brian",
      "description":"Brian Person",
      "key":"Brian",
      "config":{
         "status":"Single",
         "address":"3303 Hillview Avenue"
      }
    },
    {
      "@objectId":"ShareWriterAgent::Person:Anna",
      "name":"Anna",
      "description":"Anna Person",
      "key":"Anna",
      "config":{
         "status":"Single",
         "address":"3303 Hillview Avenue"
       }
    },
    {
      "@objectId":"ShareWriterAgent::Person:John",
      "name":"John",
      "description":"John Person",
      "key":"John",
      "config":{
         "status":"Single",
         "address":"3303 Hillview Avenue"
       }
    },
    {
      "@objectId":"ShareWriterAgent::Car:BMW",
      "name":"BMW",
      "description":"BMW Car",
      "key":"BMW",
      "config":{
         "color":"blue",
         "license":"6VHM435"
```

```
        }
      }
    ]
```

How to Unshare a TeaObject

It is a good practice to unshare a shared object after the purpose for sharing it is complete.

Below is an example of how to unshare a TeaObject after it has been shared:

```
@TeaOperation(name = "unsharePerson", description = "Unshare a Person
object")
public void unsharePerson() throws IOException {
  // Person implements TeaObject and has objectTypeName="Person" in this
instance
  // the key="John"
final Person person = new Person("John", "Public John Person");
  //teaAgentServer is an instance of
com.tibco.tea.agent.server.TeaAgentServer or
  // com.tibco.tea.agent.server.TeaAgentServlet
final TeaPersistence persistence = teaAgentServer.getPersistence();
persistence.unshare(person);
}
```

Object Type Definition: Overview

Within your agent code, you can define an object type using either of two implementation styles—*interface style* or *annotation style*. Your choice of style depends on the characteristics of the product that the agent manages.

Aspect	Interface Style	Annotation Style
Defining Object Types	Define an object type by defining a class that implements interfaces such as `TopLevelTeaObject`, `TeaObject` or `SingletonTeaObject`.	Define a class, then annotate it as an object type using `TeaObjectType` or `TeaObjectTypes`.
Representing Product Objects	To represent an object in the product, your agent code creates a Java instance of a corresponding class.	To represent an object in the product, your agent code can use an object identifier that indexes into a database.
Objects in the Agent Library	Your agent code can pass these Java instances to the agent library through a *provider* object. The agent library calls the interface methods of these instances. **Note:** Your agent code and the agent library share access to these instances, which could affect memory management.	Your agent code passes only object identifiers to the agent library.
Advantages	Java coding style is straightforward.	Object identifiers are inexpensive strings. One database query can retrieve many objects.
Disadvantages	Instantiating many objects can be	Translating between object

Aspect	Interface Style	Annotation Style
	costly. If a product has many managed objects, then the agent instantiates many Java objects to represent them.	identifier strings and object information adds an additional layer of complexity to your agent code.
Choosing a Style	Appropriate for products with relatively few managed objects Appropriate for rapid prototyping	Appropriate for products with many managed objects Appropriate for products that already store objects in a database (including a flat-file database or a non-persistent database)

Interface Style

In *interface style*, agent code models a managed object type by defining a class that implements interfaces such as `TopLevelTeaObject`, `TeaObject` or `SingletonTeaObject`.

A separate class represents each object type that you identified in your analysis of the product.

For example, in an oversimplified analysis of a car, you might identify managed objects representing the car itself, its body, engine and wheels. Your agent would model these managed objects with four corresponding classes:

- The `Car` class would implement the interface `TopLevelTeaObject`, because this class represents the product itself.
- The `CarBody` and `CarEngine` classes would implement the interface `SingletonTeaObject`, because each car has only one body and one engine.
- The `CarWheel` class would implement the interface `TeaObject`, because each car has more than one wheel.

Your agent code creates Java instances of these classes to represent each managed object instance in the product. For example, the agent would instantiate one `Car`, one `CarBody`, one `CarEngine`, and four instances of `CarWheel`.

Defining an Object Type in Interface Style

To define an object type in interface style, your agent code models a managed object as a Java class that implements appropriate interfaces. Then it registers an instance, which lets the agent library call the interface methods.
We present the general task of defining an object type that can have more than one instance. Singleton object types and top-level object types are special cases; to see how they differ from this general case, see Defining a Singleton Object Type in Interface Style and Defining a Top-Level Object Type in Interface Style.

Procedure

1. Define a class that models the managed object type. Your class must implement the

interface `TeaObject`.

For example, when modeling a car, we could define a class to represent the wheels.

```
public class CarWheel implements TeaObject {

    public String getName() {
        // Return the name of the wheel instance,
       // for example, "Wheel 1"
    }

    public String getDescription() {
        // Return the description of the wheel instance,
        // for example, "Front-Left Wheel"
    }

    public String getKey() {
        // Return the key that corresponds to the wheel instance,
        // for example, "Wheel 1"
    }
}
```

2. Define a corresponding provider class that implements the interface `TeaObjectProvider`.

```
public class CarWheelProvider implements
TeaObjectProvider<CarWheel> {

    public String getTypeName() {
        // Return the name of the wheel object type.
        return "Wheel";
    }

    public String getTypeDescription() {
        // Return the description of the wheel object type.
        return "Car wheel";
    }

    public TeaConcept getConcept() {
        // Return the concept type of wheels.
        return TeaConcept.RESOURCE;
    }

    public CarWheel getInstance(final String key) {
```

```
        // Return the wheel instance corresponding to the key
argument.
        // For example, look-up the key in a hash table.
        ...
    }

}
```

3. Register an instance of the provider class.

```
CarAgentServer server = ...
// Register other instances, including the top-level object type.
CarWheelProvider myCarWheelProvider = new
CarWheelProvider;server.registerInstance(myCarWheelProvider);
 ...
server.start();
```

What to do next

To model aspects of object types in interface style, see these topics:

- Modeling State in Interface Style
- Modeling Configuration in Interface Style
- Modeling Members in Interface Style
- Defining Operations
- Defining References for Object Types with Interfaces

Defining a Singleton Object Type in Interface Style

Defining a singleton object type is a special case of defining an object type. The difference is that a singleton object type does *not* require a provider. Alternatively, we might say that its unique instance *is its own provider*.

Because a singleton is its own provider, the `SingletonTeaObject` interface subsumes the other provider methods (namely, `getTypeName`, `getTypeDescription` and `getConcept`).

Because it is a singleton, it needs no `getInstance` method.

Procedure

1. Define a class that models the managed object type. Your class must implement the interface `SingletonTeaObject`.

   For example, when modeling a car, we could define a singleton class to represent the engine.

```
public class CarEngine implements SingletonTeaObject {

    public String getName() {
        // Return the name of the engine instance;
        return "Engine";
    }

    public String getDescription() {
        // Return the description of the engine instance;
        return "Car Engine";
    }

    public String getKey() {
        // Return the key that corresponds to the engine instance;
        // for example, "Engine"
    }

      // These methods give the singleton functionality
      // that would otherwise be in the provider.

    public String getTypeName() {
        // Return the name of the engine object type.
        return "Engine";
    }

    public String getTypeDescription() {
        // Return the description of the engine object type.
        return "Car Engine";
    }

    public TeaConcept getConcept() {
        // Return the concept type of the driver.
        // In our model, it's a process that steers the car.
        return TeaConcept.RESOURCE;
    }

}
```

2. Register the singleton instance of the class.

```
CarAgentServer server = ...
// Register other instances, including the top-level object type.
CarEngine myCarEngine = new CarEngine;server.registerInstance
(myCarEngine);
 ...
server.start();
```

Notice that instead of registering the provider, we register the unique instance of the engine class.

Defining a Top-Level Object Type in Interface Style

Defining a top-level object type is a special case of defining a singleton type, with two differences: First, a top-level object type does *not* require a `getConcept` method, because its concept is self-evident (it is always `TeaConcept.TOP_LEVEL`). Second, `TopLevelTeaObject` extends the `WithMembers` interface, so you must implement the `getMemebers` method.

Procedure

1. Define a class that models the managed object type. Your class must implement the interface `TopLevelTeaObject`.

   For example, when modeling a car, the top-level is the car itself.

```
public class Car implements TopLevelTeaObject {

    public String getName() {
        // Return the name of the car instance;
        return "Car";
    }

    public String getDescription() {
        // Return the description of the car instance;
        return "Car";
    }

    public String getKey() {
        // Return the key that corresponds to the car instance;
        return "Car";
    }
```

```
      // These methods give the top-level object functionality
     // that would otherwise be in the provider.

    public String getTypeName() {
        // Return the name of the car object type.
        return "Car";
    }

    public String getTypeDescription() {
        // Return the description of the car object type.
        return "Car";
    }

    public Collection<BaseTeaObject> getMembers() {
                // Return the main parts of the car.
    }

}
```

2. Register the unique instance of the class.

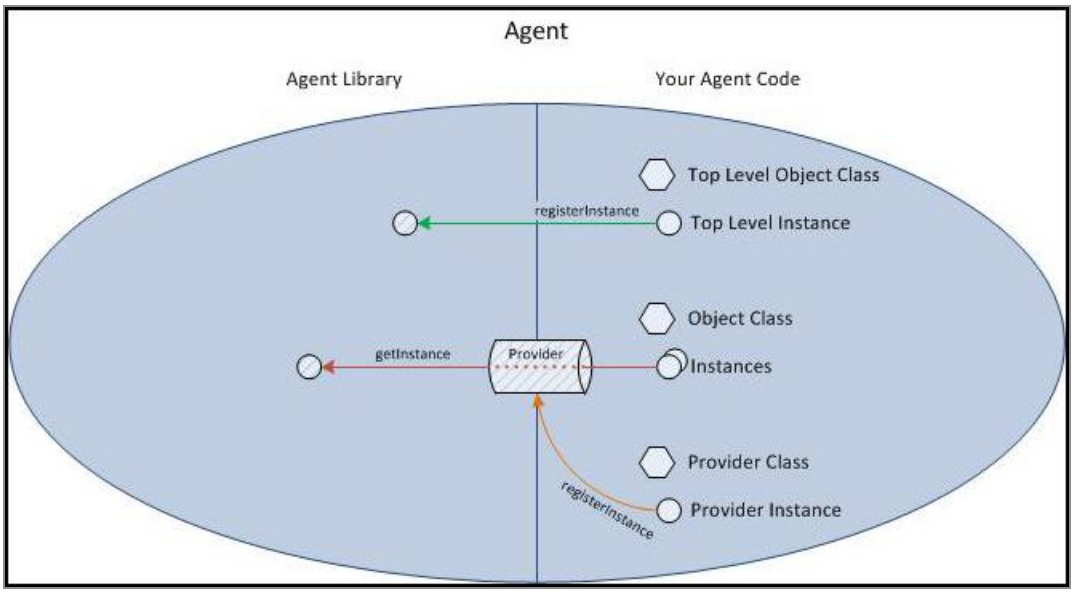
```
CarAgentServer server = ...
Car myCar = new Car;server.registerInstance(myCar);
// Register other instances.
 ...
server.start();
```

Notice that instead of registering the provider, we register the unique instance of the car class.

Access to Instances

Your agent code can give the agent library access to your object instances. The agent library can call methods of those instances. You can either register a single instance, or register a provider that lets the agent library request instances on demand.

Access to Instances



Instance Registration

Your *stand-alone* agent code can register a specific instance by calling `TeaAgentServer.registerInstance` (green arrow in the diagram above).

Your *servlet* agent code can register an instance by including it in the array that is the value of the servlet's `getObjectInstances` method.

On-Demand Access though Providers

Your agent code can also allow access to instances on demand by registering a provider instance. That is, your agent code:

1. defines an object type class (in interface style) that implements `TeaObject`
2. defines a corresponding provider class that implements `TeaObjectProvider<T extends TeaObject>`
3. instantiates a provider object
4. registers the provider object (orange arrow in the diagram)

After this preparation, the agent library can call `TeaObjectProvider<T>.getInstance` to gain access to any instance of that object type (red arrow in the diagram).

For more information, see Provider.

Note: Access to your instances within the agent library could affect memory management.

Provider Interface

A *provider* is an object within your agent code that translates instance keys into the corresponding Java objects.

When an agent defines an object type using the *interface style*, and that object type has more than one instance, then the agent must also define a corresponding provider.

Each such object type requires a separate provider class. (Top level object types and singleton object types do *not* require providers. Alternatively, one might say that for these types the single instance serves as its own provider.)

A provider class implements the interface `TeaObjectProvider`.

The `getInstance` method of this interface translates from instance keys to Java instances. Recall that the TIBCO Enterprise Administrator server refers to a managed object using an object ID; when the agent library receives an ID, it calls this provider method to obtain the actual Java instance that represents that managed object.

Other methods of the provider interface return information about the object type; for example, `getTypeName`. (For object types that have only one instance, the agent code defines these other methods on the object type class. For example, `getConcept` is part of the `SingletonTeaObject` interface.)

Your agent code must create an instance of the provider, and register it with the agent library.

If needed, you can register the Object provider for an interface/class that extends/implements the TeaObject interface. In this Object provider, you can choose to return instances of objects that implement this interface. For example, consider an object graph where clusters can have multiple machine, each machine can have multiple processes. Each process can have engines and an engine can have multiple cache/query/inference. In this example, the object provider for the engine can be registered and this object provider can return instances of cache/query/inference. The engine interface is implemented by cache, query, and inference.

Defining References for Object Types with Interfaces

References allow users to navigate from one object to related objects. To model relationships *other than membership*, you can expose a method as a reference. That is, define a method that returns the set of related objects, and annotate it as a `@TeaReference`.
When defining the source object type using interface style, define references as explained in this task.

To model a member relationship, implement the `WithMembers` interface.

Procedure

1. Declare the method signature.

   The method cannot have any parameters, and it must return an array or `java.util.Collection` of objects that implement `com.tibco.tea.agent.api.BaseTeaObject`.

   For example, in TIBCO ActiveMatrix, administrators can distribute an application to a set of nodes. In the agent, the `getNodes()` method returns an array or `java.util.Collection` of those nodes. The `Node` class implements the `BaseTeaObject` interface.

   ```
    public Node[] getNodes(){
       ...
   }
   ```

   ```
   public Collection<Node> getNodes(){
          ...
   }
   ```

2. Annotate the method as a reference using `@TeaReference`.

   ```
   @TeaReference(name = "nodes")
   public Node[] getNodes(){
       ...
   }
   ```

```
@TeaReference(name = "nodes")
public Collection<Node> getNodes(){
    ...
}
```

If the reference type class defines more than one object type, then include the `objectType` attribute, to distinguish among them.

Annotation Style

In *annotation style*, the agent models a managed object type with virtual objects rather than Java instances. Annotations such as `@TeaObjectType` encapsulate information about the object types. Annotations such as `@TeaGetInfo` indicate methods that interact with the server, providing information about the virtual objects.

Defining an Object Type in Annotation Style

In *annotation style*, agent code models a managed object type by defining a class, and marking it with annotations, such as `@TeaObjectType`.

Procedure

1. Annotate the object type class as `@TeaObjectType`. Supply the required attributes—`name`, `description` and `concept`.

```
@TeaObjectType(name="TOMCAT_SERVER",
    description="Tomcat Server"
    concept=TeaConcept.PROCESS,)
public class TomcatServerManager {
    ...
}
```

2. Define a method that translates from an object key to basic information about the object, and annotate that method as `@TeaGetInfo`.

   This method must return an instance of `AgentObjectInfo`.

```
@TeaObjectType(name="TOMCAT_SERVER",
    description="Tomcat Server"
    concept=TeaConcept.PROCESS,)
public class TomcatServerManager {

    @TeaGetInfo
    public AgentObjectInfo getInfo(@KeyParam final String key) {
```

```
        Server server = lookupTomcatServer(key);
        AgentObjectInfo result = new AgentObjectInfo();
        result.setName(server.getName());
        result.setDesc(server.getDescription);
        return result;
    }

    Server lookupTomcatServer(String key) {
        ... // domain specific implementation
    }
}
```

For top-level object types, this method does not accept any arguments. (An object key would be superfluous because a top-level type can have only one instance.)

3. If the object type has state, define a method that returns the current state of an instance, and annotate it as `@TeaGetStatus`.

   For details, see Modeling State in Annotation Style.

4. If the object type has configuration, define a method that returns the configuration, and annotate it as `@TeaGetConfig`.

   For details, see Modeling Configuration in Annotation Style.

5. If the object type can contain members, define a method that returns the members, and annotate it as `@TeaGetMembers`.

   For details, see Modeling Members in Annotation Style.

What to do next

Annotations:

To model aspects of object types in annotation style, see these topics:

- Modeling State in Annotation Style
- Modeling Configuration in Annotation Style
- Modeling Members in Annotation Style
- Defining Operations
- Defining References for Object Types with Annotations

Defining Multiple Object Types on One Class

In *annotation style*, you can use one Java class to model two or more managed object types.

Procedure

1. Annotate the class as `@TeaObjectTypes`, and include within it a separate `@TeaObjectType` annotation for each of the object types that the class can represent.

   For example,

   ```
   @TeaObjectTypes({
       @TeaObjectType(name = "TOMCAT_SERVER",
           concept = TeaConcept.PROCESS,
           description = "Tomcat Server"),
       @TeaObjectType(name = "TOMCAT_WEBAPP",
           concept = TeaConcept.APPLICATION,
           description = "Tomcat Web Application") })
   public class TomcatServerManager {
       ...
   }
   ```

2. For each object type, define a method that translates from an object key to basic information about the object, and annotate that method as `@TeaGetInfo`. You must include the `objectType` attribute in the annotation, to distinguish the object type to which each such method applies.

   ```
   @TeaObjectTypes({
       @TeaObjectType(name = "TOMCAT_SERVER",
           concept = TeaConcept.PROCESS,
           description = "Tomcat Server"),
       @TeaObjectType(name = "TOMCAT_WEBAPP",
           concept = TeaConcept.APPLICATION,
           description = "Tomcat Web Application") })
   public class TomcatServerManager {

       @TeaGetInfo(objectType = "TOMCAT_SERVER")
       AgentObjectInfo getServerInfo(@KeyParam final String key) {
           ...
       }

       @TeaGetInfo(objectType = "TOMCAT_WEBAPP")
   ```

```
    AgentObjectInfo getWebappInfo(@KeyParam final String key) {
        ...
    }
}
```

3. Similarly, include the `objectType` attribute when modeling aspects such as state, configuration, members, operations and references.

Defining References for Object Types with Annotations

References allow users to navigate from one object to related objects. To model relationships *other than membership*, you can expose a method as a reference. That is, define a method that returns the set of related objects, and annotate it as a `@TeaReference`.
When defining the source object type using annotation style, define references as explained in this task.

To model a member relationship, use the `@TeaGetMembers` annotation.

Procedure

1. Define a reference method that accepts an object key as its parameter, and returns an array or `java.util.Collection` of the related objects as instances of `com.tibco.tea.agent.types.AgentObjectIdentifier`.

   The agent library invokes this reference method in context of the source object, and supplies its object key as the parameter. Annotate that parameter using `@KeyParam`. Top-level and singleton object types can omit this parameter (and its annotation), because only one such object can exist.

   For example, in TIBCO Enterprise Messaging System, administrators can define a set of queues. The `getQueues` method returns an array or `java.util.Collection` of `AgentObjectIdentifier` references that represent those queues.

```
public AgentObjectIdentifier[] getQueues(
    @KeyParam final String key){
```

```
    ...
}
```

```
public Collection<AgentObjectIdentifier> getQueues(
    @KeyParam final String key){
    ...
}
```

2. Annotate the method as a reference using `@TeaReference`.

   For example, the `getQueues` method is marked as a reference using the TeaReference annotation.

```
@TeaReference(name = "queues",
    referenceType = "queueType",
    objectType = "serverType")
public AgentObjectIdentifier[] getQueues(@KeyParam final String
key){
    ...
}
```

```
@TeaReference(name = "queues",
    referenceType = "queueType",
    objectType = "serverType")
public Collection<AgentObjectIdentifier> getQueues(@KeyParam final
String key){
    ...
}
```

   Include the `referenceType` attribute to indicate the type of element that the object identifiers represent.

   If the source class defines more than one object type, then include the `objectType` attribute, to distinguish the `TeaObjectType` to which the reference method applies (as the source object).

Aspects for Object Types—Interfaces and Annotations

Supplemental interfaces and annotations let you model aspects of managed objects—such as states, configuration and references.

The results of your managed objects analysis indicate the aspects of each object type (see Aspects of Managed Objects). When defining an object type in interface style, implement these interfaces to model those aspects. When defining an object type in annotation style, use the corresponding annotations to model those aspects.

Aspect	Interface `com.tibco.tea.agent.api`	Annotation `com.tibco.tea.agent.annotations`
States	`WithStatus`	`TeaGetStatus`
Configuration	`WithConfig`	`TeaGetConfig`
Members (containment relationship)	`WithMembers`	`TeaGetMembers`

In contrast, to model operations and references you must use an annotation—even in interface style. See Defining Operations, Defining References for Object Types with Interfaces and Defining References for Object Types with Annotations.

Modeling State in Interface Style

To model a managed object that has states, implement the supplemental interface `com.tibco.tea.agent.api.WithStatus`. `WithStatus` requires that you implement one method, `getStatus`, which returns the current state of a managed object.

See also States Aspect.

Before you begin

The object is defined in interface style.

Procedure

Implement the `WithStatus` interface in your managed object type.
The status method must return an instance of the class `com.tibco.tea.agent.types.AgentObjectStatus`. `AgentObjectStatus` is a standard format for encapsulating state within agents. It has three fields:

`state`

Required. The state name.

`desc`

Optional. An interpretive description of the state.

`uptime`

Optional. The length of time that the managed object has been in this state.

```
public class TomcatServer implements TeaObject, WithStatus
    ...
    public AgentObjectStatus getStatus() {
        ...
    }
}
```

Modeling Configuration in Interface Style

To model a managed object that has configuration, implement the supplemental interface `com.tibco.tea.agent.api.WithConfig<CONFIG>`. `WithConfig` is a generic interface. It requires that you implement one method, `getConfig`, which returns a configuration object.

See also Configuration Aspect.

Before you begin

The container object is defined in interface style.

Procedure

1. Define a configuration bean that encapsulates all the parameters of the managed object.

   For example, the configuration of a radio could include changeable settings (such as the frequency and volume) and unchanging attributes (such as model and serial number).

```
public class RadioConfig {

    private long frequency;
    private int volume;
    private string model;
    private string serial;

    // Constructors.  Get and set methods for each field.
    ...
}
```

2. Implement the `WithConfig<CONFIG>` interface in your managed object type. Substitute your configuration class as the generic type parameter `CONFIG`.

```
public class Radio implements TopLevelTeaObject,
WithConfig<RadioConfig> {
    ...
    public RadioConfig getConfig() {
        ...
    }
}
```

3. Declare that the `getConfig` method returns an instance of your configuration class.

```
public class Radio implements TopLevelTeaObject,
WithConfig<RadioConfig> {
    ...
    public RadioConfig getConfig() {
        ...
    }
}
```

Modeling Members in Interface Style

To model a managed object that contains other managed objects, implement the supplemental interface `com.tibco.tea.agent.api.WithMembers`.
Members indicates a containment relationship. Model any other type of relationship using @TeaReference. See also References Aspect.

Before you begin

The container object is defined in interface style.

Procedure

1. Implement the `WithMembers` interface in your managed object type.

```
public class TomcatServer implements TeaObject, WithMembers
    ...
    public Collection<BaseTeaObject> getMembers() {
    }
}
```

2. Implement the `getMembers` method to return the set of member objects.

   `getMembers` must return a generic collection of `BaseTeaObject` (or any compatible type).

   For example, the members of a Tomcat server are the web applications that run in the server.

```
public class TomcatServer implements TeaObject, WithMembers
    ...
    public Collection<BaseTeaObject> getMembers() {
        final Collection<BaseTeaObject> members = new
ArrayList<BaseTeaObject>();
        for (final WebApp wapp : this.getWebApps()) {
            members.add(wapp);
        }
        return members;
        ...
    }
}
```

   Implementing this method allows the Tomcat agent to supply the members when the TIBCO Enterprise Administrator server requests them.

Modeling State in Annotation Style

To model a managed object that has states, implement a method the returns the current state, and annotate that method as `@TeaGetStatus`.

See also States Aspect.

Before you begin

The object is defined in annotation style. Top-level object types *cannot* have state.

Procedure

Implement a method to get the state. Annotate it as `@TeaGetStatus`.
The status method must return an instance of the class `com.tibco.tea.agent.types.AgentObjectStatus`. `AgentObjectStatus` is a standard format for encapsulating state within agents. It has three fields:

`state`

Required. The state name.

`desc`

Optional. An interpretive description of the state.

`uptime`

Optional. The length of time that the managed object has been in this state.

```
@TeaGetStatus
AgentObjectStatus getStatus(@KeyParam final String key) {
    AgentObjectStatus result = new AgentObjectStatus();
    ...
    return result;
    }
```

Modeling Configuration in Annotation Style

To model a managed object that has configuration, implement a method that returns a configuration object, and annotate that method as `@TeaGetConfig`.

See also Configuration Aspect.

Before you begin

The object is defined in interface style.

Procedure

1. Define a configuration bean that encapsulates all the parameters of the managed object.

   For example, the configuration of a radio could include changeable settings (such as the frequency and volume) and unchanging attributes (such as model and serial number).

   ```
   public class RadioConfig {

       private long frequency;
       private int volume;
       private string model;
       private string serial;

       // Constructors.  Get and set methods for each field.
       ...
   }
   ```

2. Define a method that gets the current configuration, returning it as an instance of your configuration class.

   ```
       public RadioConfig getConfig() {
           ...
       }
   ```

3. Annotate that method as `@TeaGetConfig`.

   ```
   @TeaGetConfig
       public RadioConfig getConfig() {
           ...
       }
   ```

Modeling Members in Annotation Style

To model a managed object that contains other managed objects, implement a method that returns an array or `java.util.Collection` of members, and annotate that method as `@TeaGetMembers`.
Members indicates a containment relationship. Model any other type of relationship using @TeaReference. See also References Aspect.

Before you begin

The container object and the member object are both defined in annotation style.

Procedure

Implement a method to get the members. Annotate it as `@TeaGetMembers`.
The method must return either an array or `java.util.Collection` of `AgentObjectIdentifier` instances.

```
    @TeaGetMembers
    AgentObjectIdentifier[] getMembers(@KeyParam final String key)
{
        ...
        }
```

```
    @TeaGetMembers
    Collection<AgentObjectIdentifier> getMembers(@KeyParam final
String key) {
        ...
        }
```

`com.tibco.tea.agent.types.AgentObjectIdentifier is a standard format for virtual objects within agents.`

Defining Operations

To expose a method of an object type as an operation, annotate it as `@TeaOperation`. End users of TIBCO Enterprise Administrator server can invoke operations.

Annotations declare operations and their components. The TIBCO Enterprise Administrator server uses that information in its user interfaces.

Procedure

1. Declare the method signature.

   An operation method can take parameters. An operation can return either a simple type or an object that conforms to the Java bean specification.

   For example, a Tomcat agent could support a `create()` method to add a new server to the Tomcat cluster.

   ```
   public void create(
       final String name,
       final Integer port,
       Integer ajpPort,
       Integer shutdownPort)
       ...
   ```

   Note: If multiple agents manage the same object type, they must expose the same set of operations. Furthermore, corresponding operations must have identical names and method signatures. This restriction ensures that users can invoke an operation on either agent (selecting it from a drop-down menu in the GUI).

   Caution: If multiple versions of an agent must coexist simultaneously, do not modify the method signature of an existing operation in the new version of the agent.

2. Annotate the method as a `@TeaOperation`.

   In our continuing example, we use the annotation to rename the operation and to

supply a description string—both become part of the product's GUI in the TIBCO Enterprise Administrator server. We also supply a method type, indicating that this operation effects changes in the product.

```
@TeaOperation(name = "create server",
    description = "Create a Tomcat server instance",
    methodType = MethodType.UPDATE)
public void create(
    final String name,
    final Integer port,
    Integer ajpPort,
    Integer shutdownPort)
    ...
```

3. Optional. Annotate the permissions that this operation requires.

```
@TeaOperation(name = "create server",
    description = "Create a Tomcat server instance",
    methodType = MethodType.UPDATE)
@TeaRequires(value = TomcatAgent.LIFECYCLE_PERMISSION)
public void create(
    ...
```

Users that have any of the required permissions for the object can execute the operation.

If you omit the `@TeaRequires` annotation, the default requirement is for the minimum permission, which is `read`.

Define permissions using the `@TeaPermission` annotation.

4. Annotate the method's parameters as parameters of the operation.

Annotation	Description
`@KeyParam`	If one of the parameters receives an object key, then annotate that parameter with `@KeyParam`. The server automatically supplies the target object as the argument of this parameter. Top-level and singleton object types can omit this annotation, because they have only one instance.

Annotation	Description
`@TeaParam`	Annotate each parameter with a parameter `name`, an optional `description` and an optional `defaultValue`. The `name` attribute is required because Java does not preserve parameter names at run time. The defaultValue attribute of `TeaParam` is optional. If you choose to provide it, its value must be provided in the JSON format.

```
@TeaOperation(name = "create server",
    description = "Create a Tomcat server instance",
    methodType = MethodType.UPDATE)
public void create(
    @TeaParam(name = "name",
         description = "Name of the Tomcat instance")
         final String name,
    @TeaParam(name = "port",
         description = "Port for HTTP connector")
         final Integer port,
    @TeaParam(name = "ajpport",
         defaultValue = "-1",
         description =
           "Port for AJP connector.
            If value is -1, agent randomly picks a port.")
         Integer ajpPort,
    @TeaParam(name = "shutdownport",
         defaultValue = "-1",
         description =
           "Port for shutting down tomcat.
            If value is -1, agent randomly picks a port.")
         Integer shutdownPort)
    ...

@TeaOperation(name = "changePort", objectType = "server",
    description = "Change the server port")
@TeaRequires(value = { TomcatAgent.UPDATE_PERMISSION })
public void changePort(
    @KeyParam final String key,
    @TeaParam(name = "port", description = "Change to this port.")
        final int port)
```

```
    ...
```

At this stage, these annotations suffice to expose these operations in the default GUI and the shell UI. The operation bar on the object displays a button that creates a server. That button opens a default form so the user can enter argument values.

5. Prepare the return value.

   An operation method can return a value to the TIBCO Enterprise Administration server. The form of that value could require preparation within the agent code.

Return Type	Description
`TeaObject`	No further preparation. The agent library automatically translates the TeaObject instance to JSON format before returning it to the server.
`Map<String, Object>`	No further preparation. If the operation returns a map of maps, the agent library automatically converts it to JSON format before returning it to the server.
Java Object	The agent library uses Jackson to serialize the Java object to JSON format before returning it to the server. Two options are available: • Annotate the Java class using Jackson or JAXB annotation. • Arrange translation using the Jackson databinding API. **Note:** The agent library does not support Jackson's org.json module.

Adding Developer Notes

Within agent code, agent developers can include *developer notes* for GUI developers. For example, when changing the agent, the agent developer can include a developer note so the GUI developer can change the GUI accordingly. We recommend mentioning the version in which each such change occurs.
Developer notes appear when viewing API documentation in developer mode.

Procedure

Add developer notes at any level within the agent code.
The form for adding a note depends on the level to which the note applies.

Level	Description
`TeaAgentServer`	Call the method `withDocumentation` on your agent's server instance (before registering the instance). Supply your notes as a string argument.
`TeaAgentServlet`	In your class that extends `TeaAgentServlet`, override the `getDeveloperNotes` method to return your notes as a string.
`TeaObjectType` Interface Style	In your class that extends `BaseTeaObject`, implement the `withDocumentation` interface. Code the `getDeveloperNotes` method to return your notes as a string.
`TeaObjectType` Annotation Style	In the annotation `@TeaObjectType`, supply the attribute `developerNotes="`*`my_notes`*`"`.
`TeaOperation`	In the annotation `@TeaOperation`, supply the attribute `developerNotes="`*`my_notes`*`"`.
`TeaParam`	In the annotation `@TeaParam`, supply the attribute `developerNotes="`*`my_notes`*`"`.

```
    private static void setupTomcatAgent(final TomcatAgentConfig
tomcatAgentConfig)
            throws Exception {
        final TeaAgentServer server = new TeaAgentServer("tomcat",
"7.0.42",
                tomcatAgentConfig.getAgentInfo(),
tomcatAgentConfig.getPort(),
                "/tomcatagent", true);
        server.withDocumentation("Compiled and tested against agent
libr V5.");
        server.registerInstance(new TomcatAgent(tomcatAgentConfig,
server));
        server.registerInstance(new TomcatServer(tomcatAgentConfig,
```

```
server));
        ...
```

```
@TeaObjectType(name = TomcatAgentUtil.SERVER,
    concept = TeaConcept.PROCESS, description = "Tomcat Server",
    developerNotes="Unchanged in V5.")
public class TomcatServer {

    private final TomcatAgentConfig tomcatAgentConfig;
    private final TeaAgentServer teaAgentServer;

    public TomcatServer(final TomcatAgentConfig tomcatAgentConfig,
                        final TeaAgentServer server) {
        this.tomcatAgentConfig = tomcatAgentConfig;
        this.teaAgentServer = server;
    }

    // http://host:port/tomcatagent/server/
{key}/changeport?port=8080
    @Customize(value = "label:Change port;icon:edit_16x16.png")
        @TeaOperation(name = "changePort", objectType = "server",
                description = "Change the server port",
             developerNotes="From v4 onward, throws IOException.")
    @TeaRequires(value = { TomcatAgent.UPDATE_PERMISSION })
    public void changePort(@KeyParam final String key,
        @TeaParam(name = "port", description = "",
            developerNotes="In V5, added validation that value is
in range [1000-2000].")
        @Customize(value = "label=Port") final int port)
```

Specifying the Availability of TeaOperations in TIBCO Enterprise Administrator Clients

Users can connect to the TEA server using one of three clients - the TEA Web User Interface, the Shell, and Python scripts. However, not all TEA operations can be supported in all clients. When coding a method that is annotated with TeaOperation, the agent developer can specify which clients expose that operation.

To do so, in the TeaOperations annotation, supply the attribute `hideFromClients`, which can have one or more of the following values:

- ANY - operation is not available to any client
- PYTHON - operation not available from the Python client
- SHELL - not available from the Shell
- WEB_UI - not available in the Web User Interface
- If you do not specify this attribute, the operation is exposed in all clients.

Example - using a single value

```
@TeaOperation(name = NAME, description = "Ping agent", hideFromClients=
{ClientType.ANY})
```

Example - using multiple values

```
@TeaOperation(name = NAME, description = "Ping agent", hideFromClients=
{ClientType.SHELL, ClientType.PYTHON})
```

Note: The `internal` attribute has been deprecated. Use the `hideFromClients` attribute instead.

Note: If the TeaOperations name contains an invalid character, the operation are not be available in the Python binding. You can retrieve a list of omitted operations at the respective object types. For example, you can retrieve the list of omitted operations for a provisional product and its members as follows:

```
>>> import tibco.tea
>>> tea = tibco.tea.EnterpriseAdministrator()
>>> prod = tea.product_with_provisional_api(<product_name>)
>>> prod.omitted_operations
>>> prod.members[<member_name>].omitted_operations
```

Passing Data Streams to Operation Methods

`TeaOperation` methods can accept data streams (such as file data) from the server. The default GUI makes it easy to select a file and pass its data content to an agent method.

Procedure

1. Define an operation with a parameter of type `javax.activation.DataSource`, and annotate it as a `@TeaParam`.

```
@TeaOperation(name = "upload", description = "Upload File")
public void uploadFile(
    @TeaParam(name = "filepath", description = "File to
Upload")final DataSource fileDataSource)
  throws IOException {
  ...
}
```

   The default GUI lets a user select a file from the local file system. The server passes the file's contents to the agent as an argument to the operation method. The agent specifies this behavior with the example code shown in bold above—specifically, the parameter type.

2. Use the data stream in the method body.

   In this example, the method writes the data stream to a temporary file, effectively transferring the data from the user's host computer to the agent's host computer.

```
{
  byte[] buffer = new byte[8 * 1024];
  String name = fileDataSource.getName();
  FileOutputStream writerStream = new FileOutputStream(name, true);
  final InputStream inputStream = fileDataSource.getInputStream();
  int read;
  while((read = inputStream.read(buffer)) > 0)
  {
     writerStream.write(buffer, 0, read);
  }
  inputStream.close();
  writerStream.flush();
  writerStream.close();
}
```

Customizing Parameters of an Operation in the Shell Interface

You can modify an operation's usage syntax in the shell interface. Operations can accept named or positional parameters. You can shorten a named parameter by defining an alias. In the shell interface, an operation can accept two kinds of parameters:

- *Named* parameters. Users specify each parameter as a flag (such as `-aa`*value*). Users can supply named parameters in any order relative to one another. Named parameters precede all positional parameters.
- *Positional* parameters. Users supply positional parameters in a specific order, parallel to the order of parameters in the operation method signature.

Before you begin

You have already annotated the operation's parameters with `@TeaParam`. You have specified the `name` attribute, as required.

Procedure

1. Determine the syntax of each parameter.

   Specify that syntax as the value of the `usage` attribute within the `@TeaParam` annotation. Supply a value enumerated by `AgentParamUsage`.

Value	Description
NAMED	Named parameter.
POSITIONAL	Positional parameter.
LEGACY	Backward compatibility. When `usage` is absent, this is the default behavior. The `usage` attribute is new in release 1.2.0. In earlier releases the combination of `name` and `alias` attributes determined whether the parameter syntax was positional or named. To preserve that behavior in later releases, supply this value.

Definition

```
public String testA(
  @TeaParam(name = "aa", description = "AA parameter",
            usage = AgentParamUsage.POSTITIONAL)
    final long[] greetings)
```

Resulting Usage

```
shell> testA [12 15]
```

2. Optional. Specify a parameter `alias`.

   Sometimes the name of a parameter is too long to use as an option flag. You can supply a shorter flag as the value of the `alias` attribute within the `@TeaParam` annotation.

   Alias is available only for named parameters; it has no effect on positional parameters.

   Definition

```
public String testB(
  @TeaParam(name = "awkwardly_long_parameter",
            description = "A parameter with a long name",
            usage = AgentParamUsage.NAMED,
            alias="bb")
    final long[] greetings)
```

   Resulting Usage

```
shell> testB -awkwardly_long_parameter [12 15]
shell> testB --awkwardly_long_parameter [12 15]
shell> testB -bb [12 15]
shell> testB --bb [12 15]
```

Getting the User Name of the Current User

In the TIBCO Enterprise Administrator, when the current user invokes an operation on the product through its agent, the name of the current user who is invoking this operation can be made available to the Agent.

This can be done by adding a parameter whose type is `com.tibco.tea.agent.api.TeaPrincipal`. No `TeaParam` annotation is needed for this method parameter. In the method code, the `getName()` method of the `TeaPrincipal` class can be invoked to get the name of the user who invoked the operation.

Procedure

Pass the `TeaPrincipal` as a parameter in the `uploadFile()` method:

```
@TeaOperation(name = "upload", description = "Upload File")
public void uploadFile(
@TeaParam(name = "filepath", description = "File to Upload")
final DataSource fileDataSource, TeaPrincipal teaPrincipal)
throws IOException
{ String userName = teaPrincipal.getName(); ... }
```

Object ID

Every object addressable by the TIBCO Enterprise Administrator server must have a unique object ID. The object ID must identify an agent and an object managed by that agent.

All TIBCO Enterprise Administrator object IDs have the following structure:

```
<agentID>:<agentType>:<agentVersion>:<objectType>:<objectKey>
```

The agentId, agentType, agentVersion, objectType and objectKey tokens are URL encoded and the character colon ':' is allowed in those tokens.

Object ID tokens

agentID

Specifies the agent that owns a requested object or collection.

Ensure that the Agent IDs are reproducible using only the sort of information that external applications use to identify related objects. For example, an EMS server would base its Object id either on the JNDI name of the connection factory or on the connection URL (possibly with a well-known, hard-coded prefix such as "jndi:" or "url:"). Avoid using Agent IDs that contain random numbers, internal-use-only keys or other difficult-to-reproduce information. Agent developers must address this issue to support pivoting.

An effective agent ID must not begin with "_". All strings beginning with "_" are reserved for TIBCO Enterprise Administrator.

agentType

Name of the Agent type.

Agent type names must not begin with "_". All strings beginning with "_" are reserved for TIBCO Enterprise Administrator.

agentTypeVersion

Version of the Agent type.

objectType

Name of the Object type.

Object type names must not begin with "_". All strings beginning with "_" are reserved for TIBCO Enterprise Administrator. The name "agent" is reserver for use by TIBCO Enterprise Administrator.

objectKey

A key to show details about a specific object instance.

The object key is an opaque string. The pair (agentID, objKey) must be unique among all objects that share the same pair (agent type, object type).

An effective object key must not begin with "_". All strings beginning with "_" are reserved for TIBCO Enterprise Administrator.

Solution

A *solution* defines a set of managed objects that can be managed by agents other than the one defining the solution.

A solution can contain objects defined by the agent and can have links to other objects. A Solution must be registered before an agent is started.

For example,

```
final TeaSolution solution = new TeaSolution("sampleSolution", "This is
my sample solution");
        // Add tomcat reference
        final TeaObjectHardLink hl = new TeaObjectHardLink() {

        @Override
        public String getName() {
        return "Tomcat";
        }

        @Override
        public String getDescription() {
        return "Link to tomcat";
        }

        @Override
        public String getObjectID() {
        return "Tomcat:::server:t1";
        }
        };
        solution.addMembers(devNode, platformapp, hl);
        server.registerSolution(solution);
        server.start();
```

Customizing the Solution

The `setCustomization` method takes a String parameter to customize the UI for the solution. The following snippet shows an example of customizing the solution:

```
solution.setCustomization("{ " +
"\"solutionName\": \"SampleProductAgent Solution\"," +
"\"title\": \"SampleProductAgent\"," +
```

```
"\"subtitle\": \"SampleSolution\"," +
"\"columns\": [ " +
"{ \"label\": \"name\", \"expr\": \"name\", \"entityLink\": true }," +
"{ \"label\": \"agent name\", \"expr\": \"agentId\"}," +
"{ \"label\": \"type\", \"expr\": \"type.name\" }," +
"{ \"label\": \"description\", \"expr\": \"desc\" }," +
"{ \"label\": \"status\", \"expr\": \"status.state\" }" +
"]" +
"}");
```

The String passed to the method can customize the following:

- Solution Name
- Title
- Subtitle
- Columns of the table

Permissions

TIBCO Enterprise Administrator permits access to objects and operations based on permissions, privileges, roles and requirements.

Key terms

User

Users are entities that need access to the system. Each user might need a different level of access. Users can be assigned to one or more roles. TIBCO Enterprise Administrator does not manage users; instead, it maps user information from external systems (such as an LDAP).

Group

A group is a subset of users within an organization. A user can belong to multiple groups and a group can contain multiple users. Groups simplify the administration of access. Instead of specifying the access permissions for each user, administrators specify access permissions for the groups to which users belong.

Realm

A security realm comprises mechanisms for protecting TIBCO Enterprise Administrator resources. A realm contains users and groups, and their security credentials. TIBCO Enterprise Administrator supports two kinds of realms: an internal database within the server (which is the factory default) and an LDAP. In an *internal database realm*, information about users and groups is stored in a file. In an *LDAP realm*, the information exists on an LDAP server, and the TIBCO Enterprise Administrator server requests that information from the LDAP server.

Permission

A permission is a string that TIBCO Enterprise Administrator uses to enforce access control. The agent determines the granularity of the permissions that it defines. For example, you could define a permission `UpdateConfig` which applies to only one operation. In contrast, the built-in permission `full_control` applies to all agents.

Privilege

A privilege is a collection of permissions. Define privileges to simplify the administrator's task of assigning permissions to users and groups.

Role

A role is a collection of privileges. Administrators assign roles to users or groups. A user receives all the permissions in all the privileges in all its roles.

Defining Permissions and Requirements

You can define permissions with the `@TeaPermission` and `@TeaPermissions` annotations.

For example:

```
@TeaObjectType(name = TomcatAgentUtil.TOMCAT, concept = TeaConcept.TOP_
LEVEL,
         description = "Tomcat TIBCO Enterprise Administrator SDK
Agent")
        @TeaPermissions({
        @TeaPermission(name = TomcatAgent.LIFECYCLE_PERMISSION,
         desc = "Permission to create/start/stop server, webapp"),
        @TeaPermission(name = TomcatAgent.UPDATE_PERMISSION,
         desc = "Permission to update configurations of server, webapp")
})

        public class TomcatAgent {
        ...
        }
```

Use the `@TeaRequires` annotation to specify the permissions that a user needs to execute each operation. If an operation method does not require any permissions, then any user can invoke that operation.

Effective Permissions

The server uses this algorithm to compute the set of privileges that apply to a user:

1. Gather all the roles assigned to the user.
2. Gather all the roles assigned to groups to which the user belongs.
3. Gather all the privileges from all those roles.

4. Gather all the permissions from all those privileges.

Built-In Permissions

These permissions are built into the server, and are always available:

TeaPermission.READ

Read-only permission.

TeaPermission.FULL_CONTROL

Full access to all objects and all operations.

Roles

Roles are the central mechanism that administrators use to allot permissions to users. A *role* is a collection of privileges. When an administrator assigns a role to a user or group, the user or group receives all the permissions in the role.

Agent code can define roles using annotations. Registering an agent with the server makes all the roles that it defines available on the server. A role remains available until the administrator unregisters the last agent that defines the role.

If the server already contains a role with a given name, then any subsequent definition of a role with the same name has no effect.

(Administrators can also define roles directly on the server.)

This example defines two roles—one for Tomcat administrators and one for regular users:

```
@TeaRoles({
        @TeaRole(name = "Tomcat Admin", desc = "Manage all tomcat
servers",
         privileges = { @TeaPrivilege(permissions =
                        { TeaPermission.FULL_CONTROL }) }),
        @TeaRole(name = "Tomcat User", desc = "Read only access to all
tomcat
         servers", privileges = { @TeaPrivilege(permissions = {
                TeaPermission.READ, TomcatAgent.UPDATE_PERMISSION }) })
         })

public class TomcatServer {
```

```
    @TeaRequires(TeaPermission.FULL_CONTROL)
    public void changePort(@KeyParam final String key,
            @TeaParam(name = "port", description = "New port number to
use")
            @Customize(value = "label=Port")
            final int port) throws TeaIllegalArgumentException {
            // code
    }
}
```

TeaRole

`@TeaRole` defines a role. A role becomes available in the TIBCO Enterprise Administrator server only after the administrator registers an agent of a specific agent type for the first time. The role remains until the administrator unregisters the last agent of that agent type. If the role is already available on the server, the server ignores the redundant definition.

TeaRoles

`@TeaRoles` groups multiple roles that apply to the same object type class.

TeaPrivilege

`@TeaPrivililge` defines a privilege within a role, specifying its set of permissions.

You can specify these elements within `@TeaPrivilege`:

permissions

A list of permissions that are applicable to this role.

objectType

The object type to which a privilege applies. When absent, the default value is `all`.

Enabling Instance-Based Permissions on an Agent

By using instance-based permissions, users can now enforce permissions on a particular instance of an entity type.

When you assign instance-based permission to a given agent, you can control whether or not the permission is applicable to the user, group, or role on one or more instances of an entity type. In addition to that, you can also control whether the permission must be assigned to one or multiple instances of an entity type. .

Before you begin

The feature is available only on those agents that use the annotation style of development. `TOP_LEVEL_TEA_OBJECT` does not have instances. Therefore, it does not support instance-based permissions.

The following points are the prerequisites to enable instance-based permissions:

- The separator character used in `ObjectKey` has to be unique and consistent across the agent.
- An `ObjectType` can have only one parent `ObjectType`.
- Cyclic parent-child relationship among the ObjectType instances is not supported.
- In a given `ObjectKey` for an `ObjectType`, the `ObjectKey` of the parent must have a prefix of the current `ObjectKey`.
- A response from `TeaOperation` can be filtered only if it is an instance of `AgentObjectIdentifier` (AOI). Plain Java objects (POJO) cannot be filtered by the TIBCO Enterprise Administrator server.
- An `ObjectKey` cannot contain a separator character unless it is separating the key of the current object from that of its parent.
- `ObjectKey` must not contain the * (star) character. Also make sure that the agent separator is not a *(star) character.

To enable instance-based permissions, the TIBCO Enterprise Adminstrator server must recognize the hierarch of entity types. Perform the following steps to ensure that the TIBCO Enterprise Adminstrator server recognizes the hierarchy of entity types.

Procedure

1. Define a separator character on the `TeaAgentServer` object using the `setSeparator()` method. A *separatorcharacter* is of type `char` and is used by the agent in the instance keys to separate the current instance from the parent instance key. For example:

```
final TeaAgentServer server = new TeaAgentServer("tomcat",
"7.0.42",
tomcatAgentConfig.getAgentInfo(), tomcatAgentConfig.getPort(),
 "/tomcatagent", true);
.
.
server.setSeparator("/");
```

2. On every `ObjectType` that has a parent `ObjectType`, add a new attribute in `@TeaObjectType` annotation "parentObjectType" as shown in the following example:

```
@TeaObjectType(name = TomcatAgentUtil.WEBAPP,
concept = TeaConcept.APPLICATION, description = "Tomcat Webapp",
 parentObjectType = TomcatAgentUtil.SERVER)
public class TomcatWebApp{
    .
    .
}
```

3. To filter the response from `TeaOperation` by instance-based permissions, ensure that `TeaOperation` sends back `AgentObjectIdentifier` or an `AgentObjectIdentifier[]`. For example:

```
@TeaOperation(name = "getWebAppsAsAOI",
description = "Returns array of web apps in this tomcat server
 instance", methodType = MethodType.READ)
    public AgentObjectIdentifier[]
    getWebAppsAsAOI(@KeyParam final String key) {
    .
    .
    .
}
```

4. As an agent developer, you can hide instances on the permission assignment page of the TIBCO Enterprise Administrator UI by adding the `showInstancesInUI` attribute to the `TeaObjectType` annotation and setting its value to `false`. For example:

```
@TeaObjectType(name = TomcatAgentUtil.WEBAPP,
 concept = TeaConcept.APPLICATION, description = "Tomcat Webapp",
 parentObjectType = TomcatAgentUtil.SERVER,
 showInstancesInUI = false)
```

```
public class TomcatWebApp implements WithNotifications {
.
.
}
```

User Interface Customization

The UI can be customized using an external file.

The file format of the customization file is JSON. If the customization file is provided as a part of the custom static resource, the file must be named customization.json and placed in the top-level folder. The file must contain a single object. The object members are individual customization rules, where the member name is the selected attribute and the member value is the customization value. See the section, Running TIBCO Enterprise Administrator SDK Agent Library in Server mode.

Caution: If there are errors in the `customization.json` file, you cannot register agents. In such cases, stop the TIBCO Enterprise Administrator server, delete the data folder, restart the server, and then register the agent.

Note: Java/C++ style comments (both '/'+'*' and '//') are supported in `customization.json` file.

Syntax for the customization rules:

```
<Object Type Name>#<Operation Name>#<Method Type>.<Parameter Name>
```

Syntax for the customization of references:

```
<Object Type Name>@<TeaReference Name>
```

The following is an example for customizing the references in the EMS sample provided with the product:

```
 "EMS" : {
        "label": "TIBCO Enterprise Messaging Service",
        "app" : {
            "name": "ems-app",
            "modules": {
                "$strap.directives": "/tea/vendor/angularstrap/angular-
strap.min.js",
```

```
                "ems": "scripts/ems.js"
            }
        },
        "views": {
            "default": {
                "template": "partials/ems.html",
                "app": "ems-app"
            }
        }
    },

    "EMS#registerEmsServer#UPDATE": {
           "label": "Register EMS Server"
    },

     "EMS#registerEmsServer#UPDATE.serverName": {
           "label": "EMS Server Name"
    },


    "EMS@members": {
        "title": "Servers",
        "columns": [
            { "label": "name", "expr": "name", "entityLink": true },
            { "label": "version",  "expr": "config.versionInfo" },
            { "label": "status",  "expr": "status.state" }
        ]
    },
    "server@queues": {
        "title": "Queues",
        "columns": [
            { "label": "name", "expr": "name", "entityLink": true },
            { "label": "pending message count",  "expr":
"config.pendingMessageCount" },
            { "label": "pending message size",  "expr":
"config.pendingMessageSize" }
        ]
    },
    "server@topics": {
        "title": "Topics",
        "columns": [
            { "label": "name", "expr": "name", "entityLink": true },
            { "label": "pending message count",  "expr":
"config.pendingMessageCount" },
            { "label": "pending message size",  "expr":
"config.pendingMessageSize" }
        ]
```

```
    }
}
```

code explanation:

- `"EMS@members":` is the reference for the object type `server`
- `"title": "Servers"`: is the custom title which can also be defined using the `"label"` flag
- `"columns"`: indicates the number of columns. In this case, it is 3.
    - `"label"`: is the title of the column.
    - `"expr"`: is the content in the column.
    - `"entityLink"` is a toggle that can be either `true` or `false`. Indicates that a cell can be displayed as a link to an entity page.

The following additional properties can be set:

- `"collapsed"`: this property can be set on a reference. If set, the panel containing the reference is collapsed, on display.
- `"referenceOrderIs"`: is the order in which a reference is displayed. For example, `"referenceOrderIs": ["foo", "bar"]` indicates that `"foo"` is displayed before `"bar"`.
- `"confirm"`: this property can be set on an operation. If set to `false`, the confirmation dialog is skipped.

Note: `@Customize(value="confirm:false")` is not applicable for operations on `TopLevelObject` type.

Usage

```
{
    "CustomUI" : {
        "app" : {
            "name": "custom",
            "modules": {
                "dependency" : "scripts/dependency.module.js",
```

```
                "test" : "scripts/test.module.js"
            },
            "libraries" : ["scripts/test.library.js"],
            "stylesheets" : ["css/test.css", "css/test1.css"]
        },
        "views" : {
            "default": {
                "template": "mypage.html",
                "app": "custom"
            }
        }
    },
    "CustomApp" : {
        "views" : {
            "default": {
                "template": "mygrouppage.html",
                "app": "custom"
            }
        }
    }
}
```

where the customization properties are as follows:

Property	Description
app	Configures an AngularJS application. The `app` property is only supported by an instance of `TopLevelObject`.The property takes the following values: 1. name: defines the name of the application, defaults to the name of the TopLevelObject. 2. libraries: an array of paths of javascript files. Paths are relative to the static resource folder. Libraries are loaded before the modules get loaded. They are loaded in the order they occur in the array. 3. modules: is a map that maps module names to the path of the javascript file that point to the module. Paths are relative to the static resource folder. Modules are loaded after libraries. 4. stylesheets: an array of paths of css files. Paths are relative to the static resource folder. Stylesheets are loaded before the `app` is initialized and

Property	Description
	they maintain the order of the array.
views	Applies to all object types. The value taken by this property is a Map. The map maps the view to the view definition as an object. The view definition contains the following properties: 1. template: path to the HTML partial to load in the page. Paths are relative to the static resource folder. 2. app: name of the `app` defined by the agent.
label	Applies to all object types. The value is the name of the object to be displayed on the UI. The label of the `TopLevelObject` is used as the product name.

Configuring multiple .js files in customization.js

Use an array to configure more than one .js file in the customization.js file. For example, in the EMS sample provided above, you would configure multiple files as follows:

```
"ems": ["scripts/ems.js", "scripts/ems1.js", "scripts/ems2.js"]
```

The files must be listed in the correct order of dependency. In the example above, `ems.js` is the main file. So, it must be mentioned first in the array. `ems2.js` is dependent on `ems1.js`. So, `ems2.js` must be placed after `ems1.js`. If there are no dependencies, the files can be listed in any order.

Selecting a Specific Version of the TIBCO Enterprise Administrator UI Library

TIBCO Enterprise Administrator allows you to set the `tea_version` attribute in the `customization.json` file which automatically allows you to use the versions of AngularJS, AngularStrap, and Bootstrap supported in TIBCO Enterprise Administrator.

Before you begin

The `tea_version` attribute is available in TIBCO Enterprise Administrator 2.0 and later.

Note: The `angular_version` attribute has been deprecated in TEA 2.0. The agents developed using the 1.3 or earlier version of the agent library can be used with TIBCO Enterprise Administrator 2.0 server without any modifications.

Procedure

In the static resources folder of the agent, edit the file `customization.json`. Within the `app` definition, set the attribute `tea_version`.
For example, the `customization.json` file for the Tomcat sample would be:

```
{
  "tomcat": {
    "app": {
      "name": "tomcat-app",
      "modules": {
        "$strap.directives": "/tea/vendor/angularstrap/angular-
strap.min.js",
        "tomcat": "scripts/tomcat.js"
        },
      "tea_version": "2.0"
      },
    }
  }
```

Setting the `tea_version` attribute to 2.0 uses the following versions of these software:

Software	Version
AngularJS	1.2.14
AngularStrap	2.1.0
Bootstrap	3.2.0

Keep the following in mind:

- If the user interface has not been customized, the latest versions of AngularJS, AngularStrap, and Bootstrap supported by TEA get loaded.
- If the user interface has been customized and you set the attribute `tea_version=2.0`, the latest versions of AngularJS, AngularStrap, and Bootstrap supported by TEA get loaded.

- If the user interface has been customized and you have used the `angular_version` attribute, the specified version of AngularJS(supported by TEA), AngularStrap v0.7.8, and Bootstrap v2.3.2 get loaded.
- If the user interface has been customized and neither the tea_version nor the angular_version attribute is set, the AngularJS v1.0 , AngularStrap v0.7.8, and Bootstrap v2.3.2 get loaded.

Linking Across Two Products

You can allow the agent of one product to cross-link to an object contributed by another agent. For example, assuming that TIBCO ActiveMatrix and TIBCO Enterprise Message Service Agents are registered with the TIBCO Enterprise Administrator server, when looking at an TIBCO ActiveMatrix configuration, a TIBCO Enterprise Administrator operator can click the configured TIBCO Enterprise Message Service link and see the Enterprise Message Service management screens. In other words, TIBCO Enterprise Administrator operators can seamlessly access an asset by seamlessly traversing different products whose Agents are registered with the TIBCO Enterprise Administrator server.

You can achieve cross product linking using the `teaObjectService.load` javascript API (through customized User Interface from the agent code and `TeaObjectHardLink` API (`TeaReference` returning instances of `TeaObjectHardLink` object) provided by the TIBCO Enterprise Administrator Agent library. This causes a link (to the object being linked to) to appear in the TIBCO Enterprise Administrator User Interface. If the agent whose object is being linked to is not registered or the object that is being linked to is unreachable, the link for the object behaves according to how you have customized it to behave in your Agent code.
The `teaObjectService.isProductRegistered` javascript API can be used to figure out if a particular product is registered with the TIBCO Enterprise Administrator server or not. This can be used to solve a use case where an agent wants to leverage an operation contributed by another product. If this product is not registered then the link for the object behaves according to how you have customized it to behave in your Agent code.

Before you begin

The following procedure is an example that assumes that the Agents of both TIBCO ActiveMatrix and the TIBCO Enterprise Message Service are registered with the TIBCO Enterprise Administrator server.

Procedure

1. Use the `teaObjectService.load` (in the agent code where you customize the TIBCO Enterprise Administrator User Interface) function to retrieve `TeaObject` to be linked by passing `ObjectId`. For example,

```
teaObjectService.load(
{agentId: "ems", agentType: "EMSAgent", objectType: "server",
objectKey: "s1"}).then(function (object)
                                {
      $scope.emsServer1.teaObject=object;
    },function (object)
    {
      $scope.emsServer1.errorMessage=object.message;
    });
```

2. Use the `TeaObjectHardLink` interface (java-side of TIBCO Enterprise Administrator Agent) to create the link (to the object being linked to) to appear in the TIBCO Enterprise Administrator User Interface. For example,

```
@Customize("label:EMS Queues")
       @TeaReference(name = "queues")
       public BaseTeaObject[] getEmsQueues() {
               final TeaObjectHardLink hl1 = new TeaObjectHardLink() {

                       @Override
                       public String getName() {
                               return "sample";
                       }

                       @Override
                       public String getDescription() {
                               return "Sample queue";
                       }

                       @Override
                       public String getObjectID() {
                       return "ems:EMSAgent::queue:s1%3Asample";
                       }
               };
               final TeaObjectHardLink hl2 = new TeaObjectHardLink() {

                       @Override
```

```
                        public String getName() {
                                return "doesNotExist";
                        }

                        @Override
                        public String getDescription() {
                                return "This queue does not exist";
                        }

                        @Override
                        public String getObjectID() {
                        return "ems:EMSAgent::queue:s1%3AdoesNotExist";
                        }
                };
                return new BaseTeaObject[] {hl1, hl2};
        }
```

3. Use the `teaObjectService.isProductRegistered` javascript API (in the agent code where you customize the TIBCO Enterprise Administrator User Interface) to figure out if the product is registered with the TIBCO Enterprise Administrator server or not. For example,

```
teaObjectService.isProductRegistered(
{agentTypeName: "EMSAgent", agentTypeVersion: "0.13"}).then
(function (object) {
        $scope.emsProduct=true;
    }, function(object) {
       $scope.emsProduct=false;
                                                                    ...
                                                                    ...
```

Reference to Customize the User Interface

You can use the AngularJS concepts to customize the user interface. AngularJS offers services, filters, and directives to customize the UI.

Services

teaLocation, teaObjectService, teaScopeDecorator, and teaAuthService are some of the services that can help with UI customization.

teaLocation

This service is available under tea. services. The teaLocation service parses the URL in the browser address bar (based on AngularJS $location) and makes the URL available to your application. Changes to the URL in the address bar are reflected into teaLocation service. Use teaLocation to navigate from one page or view to another.

> **Note:** Do not to directly edit AngularJS $location in your application.

URLs in TIBCO Enterprise Administrator have the following form:
`http://localhost:8777/tea/view/{agentType}/{objectType}/{viewName}[?{query}]`

Dependencies

- $rootScope
- $window
- $location
- teaObjectService
- $log

Methods

goto

The goto function causes the browser to navigate to a TIBCO Enterprise Administrator screen that is registered by the specified agent and which displays a particular view of a particular type of object.

Parameters

`params – {object}` – Parameters object containing the following properties ([propKey] indicates that a property is optional):

- `agentType – {string}` - The agent name. Used to set agent token in the URL.
- `objectType – {string}` - The object type. Used to set objType token in the URL.
- `[viewName] – {string}` - The view name. Used to set objView token in the URL. If no view is specified, the view name is '_'. This property is optional.
- `[query] – {object}` - Query object. Specifies name-value pairs that is added as query parameters within the URL. This property is optional.

Properties

info

This object allows the implementers of statelessly navigable TIBCO Enterprise Administrator screens to bind to the information that was provided in the URL, without having to hard-code the URL structure everywhere.

Parameters appearing in teaLocation.info are automatically URL-decoded. TIBCO Enterprise Administrator does not impose any limitations on the characters being used in object keys, query parameters, and so on. The query string may contain any valid UTF-8 character. As a result, screen developers do not need to implement URL encoding or decoding for these parameters, as this is provided by teaLocation.

Returns

`{object}` – object with the following properties:

- `agentType – {string}` - The agent type token in the URL.
- `objectType – {string}` - The object type token in the URL.
- `viewName – {string}` - The view name token in the URL. If no view is specified, the

view name is '_'. This property is optional.

- `query – {object}` - the query token from the URL, as name-value pairs.

Events

teaLocationChangedSuccess

Broadcasted when the change of URL has been processed by teaLocation.

Type: broadcast

Target: root scope

Parameters

- `event – {object}` - Synthetic event object.
- `newInfo – {object}` - info property that represents information about the new URL. See teaLocation.info for more information about the attributes of this object.
- `oldInfo – {object}` - info property that represents information about the old URL. See `teaLocation.info` for more information about the attributes of this object.

Usage

The `goto` method facilitates linking from one screen to another. It allows the implementers of TIBCO Enterprise Administrator screens to support pivoting to related objects, without having to hard-code the URL structure everywhere and without needing to know details of how TIBCO Enterprise Administrator accomplishes the link.

`teaLocation.goto()` automatically encodes the URLof the query string parameters. TIBCO Enterprise Administrator does not impose any limitations on the characters being used in object keys, query parameters, and so on. The query string may contain any valid UTF-8 character. As a result, screen developers do not need to implement URL encoding or decoding for these parameters, as this is provided by teaLocation.

Example

One of the BusinessWorks screens within TIBCO Enterprise Administrator displays details about an AppNode. This screen contains a table of Apps that are associated with the AppNode. Each row in the table shows a very brief summary view of one App. The first column in the table shows the App's unique ID from the BusinessWorks object model. This value is rendered as a link. If the user clicks on the link, TIBCO Enterprise Administrator

navigates to the TIBCO Enterprise Administrator screen that displays details about a single App.

One of the BusinessWorks screens within TIBCO Enterprise Administrator displays details about a process. This screen contains a table of SAP Adapter Endpoints that are associated with the process. Each row in the table shows a very brief summary view of one SAP Adapter Endpoint, with information retrieved from BusinessWorks's own process model. The first column in the table shows the SAP Adapter Endpoint's ID. This value is rendered as a link. If the user clicks on the link, TIBCO Enterprise Administrator navigates to the screen that displays details about a single SAP Adapter Endpoint. Note that this screen is not contributed to TIBCO Enterprise Administrator by the BusinessWorks agent, but rather by the SAP Adapter agent. The information about this agent - agent name, agent version, available object types, available views, and so on - is published as a spec by the Adapters team and used by the BusinessWorks team when developing the BusinessWorks UI.

teaObjectService

This is a service available under tea.services. The `objectService` is a factory which creates TeaObjects providing data from TIBCO Enterprise Administrator Server and agents.

Dependencies

- $q
- $http
- $rootScope

Methods

load

The load function causes the browser to emit an AJAX call to the server to retrieve a TIBCO Enterprise Administrator object.

Parameters

`paramsObject – {object}` – Parameters object containing the following properties ([propKey] indicates that a property is optional):

- `agentType – {string}` - The agent type name. Used to set agent type token in the

Object ID.

- `objectType – {string}` - The object type. Used to set objType token in the Object ID.
- `[agentID] – {string}` – The agent ID. Used to set agentID in the Object ID. Not needed if there is only one agent of a given agentType.
- `[objectKey] – {string}` – The object key. Used to set objectKey in the Object ID. Not needed if the object is a singleton.

Returns

`{Promise}` – Returns a promise that is resolved with the object requested as TeaObject

loadPath

The loadPath function causes the browser to emit an AJAX call to the server to retrieve a TeaObject.

Parameters

`path – {string}` – path of the object following members relation from the root object.

Returns

`{Promise}` – Returns a promise that is resolved with the object requested as TeaObject

loadType

The loadType function causes the browser to emit an AJAX call to the server to retrieve a TeaObject type.

Parameters

`typeName – {string}` – typeName string following the format agentType:agentVersion:objectType.

Returns

`{Promise}` – Returns a promise that is resolved with the object requested as TeaObjectType

invoke

The invoke method invokes an operation on an object. An AJAX call is made to the server that uses object meta-data to invoke the object operation on the agent managing this object.

Parameters

`paramsObject - {object}` – Parameters object containing the following properties ([propKey] indicates that a property is optional):

- `agentType - {string}` – The agent type name. Used to set agent type token in the Object ID.
- `objectType - {string}` – The object type. Used to set objType token in the Object ID.
- `[agentID] - {string}` – The agent ID. Used to set agentID in the Object ID. Not needed if there is only one agent of a given agentType.
- `[objectKey] - {string}` – The object key. Used to set objectKey in the Object ID. Not needed if the object is a singleton.
- `operation - {string}` – String parameter defining the operation name to invoke
- `methodType - {string}` – String parameter defining the method type i.e. READ, UPDATE or DELETE
- `params - {object}` – Object parameter defining arguments of the operation invocation.

`requireHeaders - {boolean}` - If this parameter is true, then the returned Promise will be resolved with object containing response and http headers.

Returns

`{Promise}` – Returns a promise that is resolved with the response object of the operation.

query

The query method queries the agent and retrieves an array of TeaObject instances. An AJAX call is made to the server that uses object meta-data to retrieve the object data from the agent managing the specified type of object.

The intended use case for this query is to support lists.

The `query` operation invokes the agent READ operation named query on the given object. That operation can be defined using standard mechanism to define object operations inside TIBCO Enterprise Administrator agents.

Parameters

`paramsObject – {object}` – Parameters object containing the following properties ([propKey] indicates that a property is optional):

- `agentType – {string}` – The agent type name. Used to set agent type token in the Object ID.
- `objectType – {string}` – The object type. Used to set objType token in the Object ID.
- `[agentID] – {string}` – The agent ID. Used to set agentID in the Object ID. Not needed if there is only one agent of a given agentType.
- `[objectKey] – {string}` – The object key. Used to set objectKey in the Object ID. Not needed if the object is a singleton.
- `params – {object}` – Object parameter defining arguments of the operation invocation.

Returns

`{Promise}` – Returns a promise that is resolved with the objects requested as Array<TeaObject>

reference

The `reference` operation queries the agent and retrieves an array of TeaAgent instances. An AJAX call is made to the server that uses object meta-data to retrieve the object data from the agent managing the specified type of object.

The intended use case for this reference is to support relationship lists. query needs parameters while reference is named. A reference is navigable which means that the TIBCO Enterprise Administrator server can retrieve the list of objects based only on meta-data. This allows the reference results to be indexed.

Parameters

`paramsObject – {object}` – Parameters object containing the following properties ([propKey] indicates that a property is optional):

- `agentType – {string}` – The agent type name. Used to set agent type token in the Object ID.
- `objectType – {string}` – The object type. Used to set objType token in the Object ID.

- `[agentID] – {string}` – The agent ID. Used to set agentID in the Object ID. Not needed if there is only one agent of a given agentType.
- `[objectKey] – {string}` – The object key. Used to set objectKey in the Object ID. Not needed if the object is a singleton.
- `reference – {string}` – String parameter defining the name of the reference to retrieve.

Returns

`{Promise}` – Returns a promise that is resolved with the objects requested as Array<TeaObject>

isProductRegistered

The isProductRegistered function causes the browser to send an AJAX call to the server to check whether the product is registered with the TIBCO Enterprise Administrator server.

Parameters

`paramsObject – {object}` – Parameters object containing the following properties:

- `agentTypeName – {string}` - The agent type name of the product.
- `agentTypeVersion – {string}` - The agentTypeVersion of the product.

Returns

`{Promise}` – Returns a promise that is resolved with a boolean value indicating whether the product is registered or not.

Usage

The typical usage of the object service is to retrieve objects needed to render the view.

teaAuthService

This is a service available under tea.services. TIBCO Enterprise Administrator provides `teaAuthService`, which is an Angular service that can retrieve a list of privileges associated with a specific user.

`teaAuthService` provides authorization services. Currently, the only public API is the`listPrivileges` method.

Dependencies:

1. $q
2. $http

Methods

The only method currently available in this service is the `listPrivileges` method.

listPrivileges

Returns the list of privileges associated with the current user.

Parameters

None.

Returns

{Promise} – Returns a `promise` that is resolved with the response object of the `listPrivileges` operation. The response object is a list. Each entry in the list has the following properties:

Property	Type	Description
product	String	The name of the product that defined this permission.
objectType	String	The object type
permissions	Object	A Map object with the name of the property as the key. The value object has the following properties: 1. productName: String. The name of the product that defined this permission 2. name: String. The name of the permission 3. desc: String. The description of the permission

Usage

```
myModule.controller('MyController', function ($scope, teaAuthService) {
     teaAuthService.listPrivileges().then(function(data) {
       // this callback will be called asynchronously
       // when the response is available.

      }, function(error) {
      // called asynchronously if an error occurs.
      });
});
```

teaScopeDecorator

This is a service available in the module tea.services. The teaScopeDecorator service decorates the scope with a TeaObject based on the current URL. It also attaches several utility methods that are useful in the teaMasthead directive and elsewhere.

Dependencies

- teaLocation
- teaObjectService
- $q
- $http
- $modal

Methods

goto

The `goto` function offers a simplified way to call `teaLocation.goto`.

Parameters

`object {TeaObject}` – The object to navigate to.

reload

The reload function causes the scope's TeaObject to be reloaded from scratch. This can be important in an "onSuccess" function as passed to openOperation.

Parameters

`object {TeaObject}` – The object to navigate to.

goToProduct

The goToProduct function navigates to the current product's top-level object.

getStatusClass

The getStatusClass function must be used to set the class for styling object.status. If the state is "running" or "started" (case insensitive), the function returns tea-status-ok, otherwise it returns tea-status-error.

Parameters

`object {TeaObject}` – The object whose status is to be displayed

Returns

`{String}` – tea-status-ok or tea-status-error

openOperation

The openOperation function opens a modal dialog with a form for invoking an operation on the current object.

Parameters

- `operationName {String}` – Name of the operation
- `defaultValues {String[]}` – (optional) default values for the operation parameters
- `onSuccess {Function}` – A function to call upon successful completion of the operation call. By default, scope.reload is called.
- `validatorfn {Function(paramName, paramValue)}` – (optional) A function which validates the form's parameter values. This function takes two parameters, name and value, to be validated. The function returns an error message in case the validation fails. If the validation passes, it does not return anything.

openOperationWithResponseHeaders

The openOperationWithResponseHeaders function opens a modal dialog with a form for invoking an operation on the current object.

Parameters

- `operationName {String}` – Name of the operation
- `defaultValues {String[]}` – (optional) default values for the operation parameters
- `onSuccess {Function}` – A function to call upon successful completion of the operation call. By default, scope.reload is called.

getReference

The getReference function offers a simplified way to call teaObjectService.getReference.

Parameters

- `object {TeaObject}` – The object containing the reference
- `refName {String}` – The name of the reference
- `onReferenceDo {Function}` – The function that will be executed as a promise, passed the Array<TeaObject> returned by teaObjectService.getReference

Usage

```
myModule.controller('MyController', function ($scope, teaScopeDecorator)
{
    teaScopeDecorator($scope);

    // TeaObject now available under $scope.object

    // functions $scope.openOperation, $scope.goto, $scope.goToProduct,
    //     $scope.getReference, and $scope.getStatusClass available
});
```

Directives

Placement of objects in a window can be customized with the use of directives such as teaPanel, teaMastHead, teaAttribute, teaLongAttribute, and teaConstraint.

teaPanel

This is a directive available under tea.directives. The `teaPanel` builds a custom panel using the nested elements as panel content.

Usage

```
<tea-panel class="span7" title="'Tomcat'" name="'Web Application'">
    <table class="tea-table">
        <thead>
            <tr>
                <th>Name</th>
                <th>Url</th>
                <th>Status</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="webapp in webapps">
                <td class="highlight"><a ng-click="goto(webapp)">
{{webapp.name}}</a></td>
                <td>{{webapp.config.path}}</td>
                <td class="status"
                ng-class="webapp.status.state == 'RUNNING'
|conditional:'tea-status-ok':'tea-status-error'">
                {{webapp.status.state|lowercase}}
                </td>
            </tr>
         </tbody>
    </table>
</tea-panel>
```

Directive Details

The directive creates a new scope.

Customization

The following types are supported:

- **fixed**: If a directive is set to be fixed, you cannot resize or collapse the panel.
- **titleLink**: This attribute makes the title of a panel a hyperlink. You can provide the link in the value of the attribute.

Nested Directives

This directive supports the nested directive `teaPanelActions`.

```
<tea-panel-actions>
    <a class="tea-action-btn" ng-click="openOperation
('registerAgent')">Register</a>
</tea-panel-actions>
```

teaMasthead

This is a directive available under tea.directives. The `teaMasthead` builds a masthead for the page. The *masthead* is used to define entities.

Usage

```
<tea-masthead title="'Tomcat Server'"
              subtitle="object.name">

    <tea-masthead-attributes>
        <tea-attribute ng-repeat="(name, value) in object.config"
label="{{name}}">{{value}}</tea-attribute>
    </tea-masthead-attributes>

    <tea-masthead-long-attributes>
        <tea-long-attribute label="Description">{{object.desc}}</tea-
long-attribute>
    </tea-masthead-long-attributes>

    <tea-masthead-actions>
        <a ng-repeat="operation in object.type.operations |
filter:isPublic" class="tea-action-btn"
           ng-click="openOperation(operation.name)">
{{operation.label}}</a>
    </tea-masthead-actions>
</tea-masthead>
```

Parameters

- `title - {string}`: Title of the page.
- `subtitle - {string}`: Subtitle of the page.

Customization

- **teaMastheadAttributes**: Attributes that are rendered in the first column of the

masthead. This must be used with `teaAttribute` directive only.

- **teaMastheadLongAttributes**: Long attributes that are rendered in the second column of the masthead. This must be used with `teaLongAttribute` directive only.
- **teaMastheadActions**: Action buttons that are rendered in the masthead. This must be used with HTML anchors using the class `tea-action-btn` only. Operations on the current `TeaObject` can be performed using the `openOperation` function provided by `teaScopeDecorator`.
- **teaMastheadConstraints**: Constraints that are rendered in the masthead. This must be used with `teaConstraint` directive only.
- **teaMastheadStatus**: Status that is rendered just below the title and name.

```
<tea-masthead-status>
    <span ng-show="object.status" ng-class="getStatusClass
(object)">
        {{object.status.state|lowercase}}
    </span>
</tea-masthead-status>
```

Directive Details

This directive creates a new scope.

teaAttribute

This is a directive available under tea.directives. The `teaAttribute` formats an attribute with a label and content.

Usage

```
<tea-attribute label="Version">{{object.config.version}}</tea-attribute>
```

Parameters

`label - {string}`: Label of the attribute.

Directive

This directive creates a new scope.

teaLongAttribute

This is a directive available under tea.directives. The `teaLongAttribute` formats an attribute with a label and content. This directive includes support for long text.

Usage

```
<tea-long-attribute label="Description">{{object.desc}}</tea-long-
attribute>
```

Parameters

`label - {String}` Label of the attribute.

Directive Details

This directive creates a new scope.

teaConstraint

This is a directive available under tea.directives. The `teaConstraint` formats constraints on a label.

Usage

```
<tea-constraint label="Contraint1"></tea-constraint>
```

Parameters

`label - {string}` label of the constraint.

Directive Details

This directive creates a new scope.

Types

The types of objects used for UI customization include TeaObject, TeaObjectType, Typewritten, TeaParam, and TeaReference.

TeaObject

This is a type available in the module tea. `TeaObject` is an object managed by TIBCO Enterprise Administrator. Instances of this class are retrieved using `teaObjectService`.

Properties

Property	Returns
path	{String} - Path to access the object.
name	{String} - Name of the object
desc	{String} - Detailed description of the object.
status	{Object} - An object representing the status of this object. The status object always has a `state` property defined.
config	{Object} - An object representing the configuration of this object.
type	{TeaObjectType} – The type of this object.
agentID	{String} – ID of the agent that manages this object.
key	{String} – Key of the object.
objectId	{String} – Identifier of the object

TeaObjectType

This is a type available in the module tea. `TeaObjectType` is a type of `TeaObject.`

Properties

Property	Returns
type	{String} - Fully qualified type name that is `<agent identifier>:<agent version>:type name`.
name	{String} - Name of the type.
concept	{String} - The kind of concept this type represents.
desc	{String} - A detailed description of the type.
operations	{Array.<TeaOperation>} – An array of operations.
references	{Array.<TeaReference>} – An array of references.

Methods

Name of the Method	Parameters	Returns
getOperation	`name - {String}` : Name of the attribute.	`{TeaOperation}` : Operation with the given name.
getReference	`name - {String}` : Name of the reference.	`{TeaReference}` : Reference with the given name.

TeaOperation

This is a type available in the module tea. This is an operation applicable to a `TeaObject`.

Properties

Property	Returns
name	{String} - Name of the operation.
desc	{String} - A detailed description of the operation.
params	{Array.<TeaParam>} – An array of operation parameters.
type	{String} - One of the following operation type : `READ, UPDATE, DELETE.`
returnType	{String} – One of the following type of return value: `'string', 'number', 'boolean', 'object', 'reference'.`

TeaParam

This is a type available in the module tea. A parameter defined on `TeaOperation`.

Properties

Property	Returns
name	{String} - Name of the parameter.
desc	{String} - A detailed description of the parameter.
type	{String} – Type of the parameter. Supported values are: `string, number, boolean, object and agentId`. Specify the `agentId` to select the agent to be invoked when the target object is managed by multiple agents.
optional	{boolean} – Is set to `true` if the parameter is required for the operation; `false` otherwise.

TeaReference

This is a type available in the module tea. A reference defined by `TeaObjectType`.

Property	Returns
name	{String} - Name of the reference.
type	{String} - The type ID of the elements of the reference.
multiplicity	{boolean} – Is set to `true` if the reference is defined to have multiple targets; `false` otherwise.

Error Handling in Agents and Custom User Interfaces

When an agent cannot complete an operation, the operation method throws an exception. The error result propagates back through the server to the browser GUI. Code your agent and custom GUI so that errors help the user understand and correct the problem.

Consider these recommendations as you design and code your agent's handling of errors in operation calls:

- Code operation methods to throw appropriate and helpful exceptions.
    - Choose the exception class that best characterizes the situation.
    - Choose an HTTP status code that gives useful information.
    - In the error string, explain the reason the operation could not complete correctly, and suggest recovery strategies for the end user.
    - Use wording that end users understand.
- Code custom GUI elements to extract error information:
    - Extract the HTTP status code. The GUI can use it to select an appropriate response action.
    - Extract the error string. The GUI can display it to the user.
- Code custom GUI elements to respond appropriately to errors. Consider these possible responses:
    - Display the error string.
    - Display other information.
    - Navigate to a different GUI page. For example, if the managed object that the current page represents no longer exists, consider navigating to a page that lists all objects of its type.
    - Let the user choose from a set of recovery options.

Coding Exceptions in Agent Operations

When an operation method throws an exception, you can either use the implicit status code associated with the exception class, or encode the HTTP status code explicitly.

Procedure

1. Create and throw an exception object.

```
final String errorMessage = MessageFormat.format(
    "Cannot start web application ''{0}'' because it is already
running.",
       tomcatWebAppConfig.getName());
throw new TeaIllegalStateException(errorMessage);
```

2. Optional. Supply an explicit status code (which overwrites the implicit status code).

   You must not supply 303 as an explicit status code. Status 303 has special meaning for the server.

```
throw new TeaIllegalStateException("Too many
requests").withHttpStatusCode(429);
```

3. Optional. Supply an explicit exit code for the shell interface.

   In the shell interface an operation returns an exit code when it returns in error. When constructing the exception in your operation method, you may supply a positive integer as the shell exit code.

   If you do not supply an explicit exit code, the shell command returns an implicit exit code that is appropriate to the return status of the operation.

Exceptions and Implicit HTTP Status Codes

When your agent code throws an exception, the agent library catches it. The library implicitly maps certain exception classes to corresponding status codes.

Exception & Usage	HTTP Status Code
`TeaIllegalArgumentException` For example, a numeric value to an operation is out of range.	400
`TeaIllegalStateException` For example, the operation attempts to stop a process that is not running.	409
`TeaException` Throw this exception when an operation cannot complete for other reasons.	550
`javax.ws.rs.core.Response.Status.NOT_FOUND` The agent library throws this exception when the GUI passes an object key for which a referent no longer exists in the agent.	404
`javax.ws.rs.core.Response.Status.INTERNAL_SERVER_ERROR` The agent library throws this exception when your agent throws any unexpected exception (for example, a runtime exception).	500

Extracting Error Information in the GUI

Code your custom UI error function to extract information from the error response. When an operation method throws an exception, the agent library sends error information to the server, which in turn sends it back to the browser that invoked the operation. Your JavaScript code can extract fields from the error information.

Procedure

Code the error response to extract the `message`, `details` and `status` fields from the error, as needed.

```
function(data) {
  // Success response
  ...
```

```
  },
function(error) {
// Error response
  // Get the error message
  var errorMessage = error.message;
  // Get the error details
  var errorDetails = error.details;
  // Get the error status code
  var errorStatusCode = error.status;

  // Use this information
  ...
  });
}
```

Receive Agent Registration Notifications

An agent can store the server URL which is used to register itself with the TIBCO Enterprise Administrator server. This is especially useful when you have a failover mechanism in place. When the secondary agent comes up, the agent can use the URL information that is stored to register with TIBCO Enterprise Administrator.

The agent must implement the TeaAgentRegistrationListener interface to send the URL information from the TIBCO Enterprise Administrator server to the agent.

```
public interface TeaAgentRegistrationListener
extends EventListener{
void onAgentRegistered(TeaServerInfo event);
}
```

Sample Code that Implements TeaAgentRegistrationListener

The following is a sample code snippet that shows how a standalone agent API adds the AgentRegistrationListener. It is assumed that you have created an instance of TeaAgentServer as follows:

```
TeaAgentServer server = new TeaAgentServer("HelloWorldAgent", "1.1",
"Hello World Agent", 1234, "/helloworldagent", true);

//Add AgentRegistrationListener to the server.
server.addEventListener(new SampleAgentRegistrationListener());
```

The following is a sample code snippet that shows how a servlet API adds the AgentRegistrationListener. It is assumed that SampleTeaAgentServlet extends TeaAgentServlet and you have an instance of it.

```
SampleTeaAgentServlet servlet = new SampleTeaAgentServlet();

//Add AgentRegistrationListener to servlet
servlet.addEventListener(new SampleAgentRegistrationListener());
```

The following class implements TeaAgentRegistrationListener:

```
import com.tibco.tea.agent.api.TeaAgentRegistrationListener;
import com.tibco.tea.agent.api.TeaServerInfo;

public class SampleAgentRegistrationListener implements
TeaAgentRegistrationListener {

@Override
public void onAgentRegistered(TeaServerInfo serverInfo) {
 System.out.println("TeaServerInfo Event Tea Server URL :" +
serverInfo.getServerUrl()); }

}
```

Notify Agents about Session Timeouts

TIBCO Enterprise Administrator can notify agents about HTTP session timeouts when a session expires or when a user logs out. To receive session timeout notifications, an agent must implement the `TeaSessionListener` interface.

The agent must implement the `TeaSessionListener`, add the listener, and pass the session ID to a `TeaOperation`.

TeaSessionListener

```
public interface TeaSessionListener
extends EventListener{
void onSessionDestroyed(TeaSessionEvent event);
}
```

You can access the session id using an instance of `TeaSessionEvent`.

Sample code that Implements TeaSessionListener

```
package com.tibco.tea.agent;
import java.io.IOException;
import java.util.Collection;
import java.util.Collections;

import com.tibco.tea.agent.annotations.TeaOperation;
import com.tibco.tea.agent.annotations.TeaParam;
import com.tibco.tea.agent.api.BaseTeaObject;
import com.tibco.tea.agent.api.TopLevelTeaObject;
import com.tibco.tea.agent.server.TeaAgentServer;
import com.tibco.tea.agent.api.TeaSession;
import com.tibco.tea.agent.api.*;


public class HelloWorldAgent implements TopLevelTeaObject{
    private static final String NAME = "hw";
    private static final String DESC = "Hello World";

    @Override
```

```
    public String getDescription() {
        return DESC;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public Collection<BaseTeaObject> getMembers() {
        return Collections.EMPTY_LIST;
    }

    @Override
    public String getTypeDescription() {
        return "Top level type for HelloWorld";
    }

    @Override
    public String getTypeName() {
        return "HelloWorldTopLevelType";
    }

    @TeaOperation(name = NAME, description = "Send greetings")
    public String helloworld(
            @TeaParam(name = "greetings", description = "Greeting
parameter")
            final String greetings, final TeaSession session) throws
IOException {
               System.out.println("Input====="+greetings);
               System.out.println("Session Id ====="+session.id());
               return "Hello " + greetings;
    }

    public static void main(final String[] args) throws Exception {
        TeaAgentServer server = new TeaAgentServer("HelloWorldAgent",
"1.1","Hello World Agent", "localhost", 1234,
                                    "/helloworldagent", true);
        server.setAgentId("test");
        server.registerInstance(new HelloWorldAgent());
        server.addEventListener(new HelloSessionListener());
        server.start();
        server.autoRegisterAgent("http://localhost:8777/tea");
        System.out.println("Agent Started at with changes:
http://localhost:1234/helloworldagent");
```

```
    }

    public static class HelloSessionListener implements
TeaSessionListener {

        public void onSessionDestroyed(final TeaSessionEvent event) {
            String id = event.getTeaSession().id();
            System.out.println("Session destroyed " + id);
        }

    }

}
```

```
package com.tibco.tea.agent;
import java.io.IOException;
import java.util.Collection;
import java.util.Collections;

import com.tibco.tea.agent.annotations.TeaOperation;
import com.tibco.tea.agent.annotations.TeaParam;
import com.tibco.tea.agent.api.BaseTeaObject;
import com.tibco.tea.agent.api.TopLevelTeaObject;
import com.tibco.tea.agent.server.TeaAgentServer;
import com.tibco.tea.agent.api.TeaSession;
import com.tibco.tea.agent.api.*;


public class HelloWorldAgent implements TopLevelTeaObject{
    private static final String NAME = "hw";
    private static final String DESC = "Hello World";

    @Override
    public String getDescription() {
        return DESC;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public Collection<BaseTeaObject> getMembers() {
        return Collections.EMPTY_LIST;
```

```
    }

    @Override
    public String getTypeDescription() {
        return "Top level type for HelloWorld";
    }

    @Override
    public String getTypeName() {
        return "HelloWorldTopLevelType";
    }

    @TeaOperation(name = NAME, description = "Send greetings")
    public String helloworld(
            @TeaParam(name = "greetings", description = "Greeting
parameter")
            final String greetings, final TeaSession session) throws
IOException {
         System.out.println("Input====="+greetings);
         System.out.println("Session Id ====="+session.id());
        return "Hello " + greetings;
    }

    public static void main(final String[] args) throws Exception {
        TeaAgentServer server = new TeaAgentServer("HelloWorldAgent",
"1.1","Hello World Agent", "localhost", 1234,
                                    "/helloworldagent", true);
        server.setAgentId("test");
        server.registerInstance(new HelloWorldAgent());
        server.addEventListener(new HelloSessionListener());
        server.start();
        server.autoRegisterAgent("http://localhost:8777/tea");
        System.out.println("Agent Started at with changes:
http://localhost:1234/helloworldagent");

    }

    public static class HelloSessionListener implements
TeaSessionListener {

        public void onSessionDestroyed(final TeaSessionEvent event) {
            String id = event.getTeaSession().id();
            System.out.println("Session destroyed " + id);
        }

    }
```

```
}
```

```
package com.tibco.tea.agent;
import java.io.IOException;
import java.util.Collection;
import java.util.Collections;

import com.tibco.tea.agent.annotations.TeaOperation;
import com.tibco.tea.agent.annotations.TeaParam;
import com.tibco.tea.agent.api.BaseTeaObject;
import com.tibco.tea.agent.api.TopLevelTeaObject;
import com.tibco.tea.agent.server.TeaAgentServer;
import com.tibco.tea.agent.api.TeaSession;
import com.tibco.tea.agent.api.*;


public class HelloWorldAgent implements TopLevelTeaObject{
    private static final String NAME = "hw";
    private static final String DESC = "Hello World";

    @Override
    public String getDescription() {
        return DESC;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public Collection<BaseTeaObject> getMembers() {
        return Collections.EMPTY_LIST;
    }

    @Override
    public String getTypeDescription() {
        return "Top level type for HelloWorld";
    }

    @Override
    public String getTypeName() {
        return "HelloWorldTopLevelType";
    }
```

```
    @TeaOperation(name = NAME, description = "Send greetings")
    public String helloworld(
            @TeaParam(name = "greetings", description = "Greeting
parameter")
            final String greetings, final TeaSession session) throws
IOException {
                        System.out.println("Input====="+greetings);
                        System.out.println("Session Id ====="+session.id());
                        return "Hello " + greetings;
    }

    public static void main(final String[] args) throws Exception {
        TeaAgentServer server = new TeaAgentServer("HelloWorldAgent",
"1.1","Hello World Agent", "localhost", 1234,
                                    "/helloworldagent", true);
        server.setAgentId("test");
        server.registerInstance(new HelloWorldAgent());
        server.addEventListener(new HelloSessionListener());
        server.start();
        server.autoRegisterAgent("http://localhost:8777/tea");
        System.out.println("Agent Started at with changes:
http://localhost:1234/helloworldagent");

    }

    public static class HelloSessionListener implements
TeaSessionListener {

        public void onSessionDestroyed(final TeaSessionEvent event) {
            String id = event.getTeaSession().id();
            System.out.println("Session destroyed " + id);
        }

    }

}
```

TIBCO Enterprise Administrator Server Services

With the 1.1.0 version, the TIBCO Enterprise Administrator server exposes HTTP services that can be used by a client to communicate with TIBCO Enterprise Administrator server through HTTP.

LDAP Realm Configurations

The HTTP services offered by the TIBCO Enterprise Administrator server can be used to retrieve LDAP realm configurations, query them, and notify agents about the changes in the configurations.

TIBCO Enterprise Administrator can query for LDAP realm configurations using the HTTP services. However, the agent must provide the following details:

1. The administrator credentials to handle `Basic authentication`
2. HTTP URL: `http://[tea-server-hostname]:[port]/teas/task`
3. HTTP Method: `PUT`
4. HTTP Request Headers: `Content-Type: application/json`

Retrieve All LDAP Realm Configurations

The example shows the sample responses obtained based on the realms configured on the LDAP server.

Before proceeding with the example, ensure that the following conditions are met:

1. The TIBCO Enterprise Administrator server is running.
2. Provide the administrator credentials to handle `Basic authentication`.
3. HTTP URL: `http://[tea-server-hostname]:[port]/teas/task`

4. HTTP Method: `PUT`
5. HTTP Request Headers: `Content-Type: application/json`

The following is the sample request body to retrieve all LDAP realm configurations:

```
{
 "operation":"getLdapRealmConfigs",
 "methodType":"READ",
 "objectId":"tea:tea::realms:"
}
```

When there are LDAP realms available, they are returned as an `Array` of `Maps` against the `result` key. The following is the sample response when there are realms available on the LDAP server:

```
{ "agentId" : "tea",
  "error" : null,
  "id" : "21cc6a2a66-14399518403-21",
  "operation" : "getldaprealmconfigs",
  "progress" : 100,
  "progressStatus" : "DONE",
  "result" : [ { "bindPassword" : "secret",
        "bindUserDN" : "uid=admin,ou=system",
        "description" : "acme description",
        "groupIdAttribute" : "cn",
        "groupSearchBaseDN" : "ou=Special Groups,dc=example,dc=com",
        "groupSearchExpression" : "cn={0}",
        "groupUsersAttribute" : "uniquemember",
        "groups" : [  ],
        "realmName" : "acme",
        "searchTimeOut" : 10005,
        "serverUrl" : "ldap://acme.na.tibco.com:10389/",
        "subGroupsAttribute" : "uniquemember",
        "userIdAttribute" : "uid",
        "userPasswordAttribute" : "userPassword",
        "userSearchBaseDN" : "ou=Special Users,dc=example,dc=com",
        "userSearchExpression" : "(&(uid={0})(objectclass=*))",
        "users" : [  ]
      },
      { "bindPassword" : "password",
        "bindUserDN" : "cn=Directory Manager",
        "description" : "SunONE description",
        "groupIdAttribute" : "cn",
        "groupSearchBaseDN" : "ou=groups,dc=policy,dc=tibco,dc=com",
        "groupSearchExpression" : "cn={0}",
```

```
        "groupUsersAttribute" : "uniquemember",
        "groups" : [  ],
        "realmName" : "SunONE",
        "searchTimeOut" : 10000,
        "serverUrl" : "ldap://10.97.107.23:389",
        "subGroupsAttribute" : "uniquemember",
        "userIdAttribute" : "uid",
        "userPasswordAttribute" : "userPassword",
        "userSearchBaseDN" : "ou=people,dc=policy,dc=tibco,dc=com",
        "userSearchExpression" : "(&(uid={0})(objectclass=*))",
        "users" : [  ]
      }
    ],
  "returnType" : null,
  "status" : "DONE",
  "timeCreated" : 622351252380946,
  "timeFinished" : 622351333309405,
  "userId" : "admin"
}
```


The following is the sample response when there are no realms configured on the LDAP server:


```
{ "agentId" : "tea",
  "error" : null,
  "id" : "21cc6a2a66-14399518403-25",
  "operation" : "getldaprealmconfigs",
  "progress" : 100,
  "progressStatus" : "DONE",
  "result" : [  ],
  "returnType" : null,
  "status" : "DONE",
  "timeCreated" : 622707877203368,
  "timeFinished" : 622707877656611,
  "userId" : "admin"
}
```


Retrieve LDAP Configurations by Providing the Realm Name

The example shows the sample responses when provided with a valid and an invalid realm name.

Before proceeding with the example, ensure that the following conditions are met:

1. The TIBCO Enterprise Administrator server is running.
2. Provide the administrator credentials to handle `Basic authentication`.
3. HTTP URL: `http://[tea-server-hostname]:[port]/teas/task`
4. HTTP Method: `PUT`
5. HTTP Request Headers: `Content-Type: application/json`

This example considers 'acme' to be the realm name that is passed to retrieve the LDAP configuration. In this case the sample request body is:

```
{
"operation":"getRealmConfig",
"methodType":"READ",
"objectId":"tea:tea::realm:acme"
}
```

The following is the sample response if the realm name is valid:

```
{ "agentId" : "tea",
  "error" : null,
  "id" : "21cc6a2a66-14399518403-20",
  "operation" : "getrealmconfig",
  "progress" : 100,
  "progressStatus" : "DONE",
  "result" : { "bindPassword" : "secret",
      "bindUserDN" : "uid=admin,ou=system",
      "description" : "acme description",
      "groupIdAttribute" : "cn",
      "groupSearchBaseDN" : "ou=Special Groups,dc=example,dc=com",
      "groupSearchExpression" : "cn={0}",
      "groupUsersAttribute" : "uniquemember",
      "groups" : [  ],
      "realmName" : "acme",
      "searchTimeOut" : 10005,
      "serverUrl" : "ldap://acme.na.tibco.com:10389/",
      "subGroupsAttribute" : "uniquemember",
      "userIdAttribute" : "uid",
      "userPasswordAttribute" : "userPassword",
      "userSearchBaseDN" : "ou=Special Users,dc=example,dc=com",
      "userSearchExpression" : "(&(uid={0})(objectclass=*))",
      "users" : [  ]
      },
  "returnType" : null,
```

```
  "status" : "DONE",
  "timeCreated" : 621835579957612,
  "timeFinished" : 621835580561226,
  "userId" : "admin"
}
```

The following is the sample response if the realm name is invalid:

```
"Managed object with ID 'tea:tea:1.1.0:realm:acme' not found"
```

Notify Changes Related to LDAP Realm Configurations

To listen to changes made on the LDAP configurations on the TIBCO Enterprise Administrator server, the agent must implement the `com.tibco.tea.agent.api.TeaLdapConfigurationListener` interface. On implementing the interface, the agent gets a notification when there is an addition, modification, or deletion of an LDAP configuration on the TIBCO Enterprise Administrator server.

The following steps ensure that changes to the LDAP realm configurations on the TIBCO Enterprise Administrator server are notified to the agent:

1. The TIBCO Enterprise Administrator server is running.
2. At least one LDAP realm configuration is created on the TIBCO Enterprise Administrator server.
3. The agent has implemented the `com.tibco.tea.agent.api.TeaLdapConfigurationListener` interface.
4. The agent is registered with the TIBCO Enterprise Administrator server.

Note: In the steps mentioned earlier, if the TIBCO Enterprise Administrator server is not running, whenever the server starts, every agent in the `Running` mode receives the LDAP realm configurations.

When an agent is reconnected to the server or when the agent is in the Running mode, the agent receives notifications about the updates on the LDAP realm configurations.

The `TeaLdapConfigurationListener` Interface

The interface implementation must be registered with `com.tibco.tea.agent.server.TeaAgentServer` using the `addEventListener()` method. This is usually included as a part of the agent setup code block. The following is a snippet from `TeaLdapConfigurationListener.java`:

```
public interface TeaLdapConfigurationListener extends
java.util.EventListener {

    public void onRealmAdded(final TeaLdapConfigurationEvent
    ldapConfigurationEvent);

    public void onRealmUpdated(final TeaLdapConfigurationEvent
    ldapConfigurationEvent);

    public void onRealmDeleted(final TeaLdapConfigurationEvent
    ldapConfigurationEvent);

}
```

As a consumer of such an event, you can examine the `com.tibco.tea.agent.api.TeaLdapConfigurationEvent.TYPE` to retrieve the `com.tibco.tea.agent.api.LdapRealmConfig` instance from the event object.

Note: `com.tibco.tea.agent.api.LdapRealmConfig` is available only for `com.tibco.tea.agent.api.TeaLdapConfigurationEvent.TYPE.ADD` and `com.tibco.tea.agent.api.TeaLdapConfigurationEvent.TYPE.UPDATE` types and not for the `com.tibco.tea.agent.api.TeaLdapConfigurationEvent.TYPE.DELETE` type of event. Refer to API documents of these classes for additional details.

The sample Tomcat Agent shipped with TIBCO Enterprise Administrator server contains a sample implementation of the interface in `TomcatAgentLdapConfigListener` and its registration using the `addEventListener()` method in `TomcatAgentLauncher`.

Upgrading Agents and Agent Coexistence

To upgrade an agent that was built using an older version of the agent library, you need to stop the agent, point its CLASSPATH to the new version of the agent library, and restart the agent.
There are a few limitations that you must keep in mind when upgrading an existing agent to a newer version. Consider a scenario where you have a version of an agent installed. Later, you upgrade to a newer version of the agent. Assuming the Agent ID and the Agent URL do not change, the following holds true:

- The UI rendered depends on the newer version of the agent.
- The name of the Agent Type does not change.
- When upgrading an agent, the Permissions, ObjectTypes, and Roles that were defined by the older version of the agent must be carried forward. This ensures agent backward compatibility. If you do not carry forward the Permissions, ObjectTypes, and Roles, the agent fails to register with the server. However, you can remove operations from child objects and also remove solutions without causing disruptions in agent backward compatibility.
- You can add child objects, solutions, roles, or operations to Top Level objects.
- Old agents are not running.
- Downgrade is supported.

Agent Type and Version Coexistence

Consider a scenario when two different versions of agents are managing two different versions of the product. Then the following holds true:

- The Agent Type name is the same.
- The Agent ID or Agent URL must be different.
- The information must be merged from different agents.
- The Permissions, ObjectTypes, and Roles that were defined by the older version of the agent must be carried forward. This ensures agent backward compatibility. If you do not carry forward the Permissions, ObjectTypes, and Roles, the agent fails to

register with the server.

- Operations on shared objects do not change.
- The UI displays the latest TopLevelTeaObject.
- Only nondestructive changes are allowed on Operations of the TopLevelTeaObject.
- If your first agent has four solutions and the second one has only three, the second agent displays all four solutions.
- If you delete some operations in the second agent, the deleted operations continue to display in the second agent but cannot be invoked from the second agent.
- When invoking the operations for the TopLevel object, you must select the agent version from where you want to invoke it. When you click on the link for the operation the resulting dialog prompts you to select the agent version from where you want the operation invoked.
- The GroupBy Type operation on the server goes to the correct version of the object.

Troubleshooting

Problem

Given an agent URL, the TIBCO Enterprise Administrator server resolves the URL to a specific IP address. When you run TIBCO Enterprise Administrator on laptops, they tend to switch between networks based on your location. In such cases, the TIBCO Enterprise Administrator server is unable to reach the agent.

Solution

Avoid using hostName of the machine when you construct an instance of `TeaAgentServer`. Use 0.0.0.0 or localhost if you anticipate agent machine or the TIBCO Enterprise Administrator server machine to switch between networks.

API Reference Pages

The Java API Reference includes detailed information for each class and method of the agent developer Java API.

Java API Reference pages

Note: Javadocs is accessible from the following path: *TEA_HOME*/<*version*>doc/developer/index.html.

Note: To access the documents locally, download the `tibco-enterprise-administrator_documentation.zip` from TIBCO Enterprise Administrator documentation website.

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the TIBCO® Enterprise Administrator Product Documentation page.

To directly access documentation for this product, double-click the following file:

TIBCO_HOME`/release_notes/TIB_tea_2.4.2_docinfo.html` where *TIBCO_HOME* is the top-level directory in which TIBCO products are installed. On Windows, the default *TIBCO_HOME* is `C:\tibco`. On UNIX systems, the default *TIBCO_HOME* is `/opt/tibco`.

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

TIBCO, the TIBCO logo, the TIBCO O logo are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.tibco.com/patents.