TIBCO® Enterprise Administrator Developer's Guide

Software Release 1.1.0
February 2014

Two-Second Advantage®

Important Information

Contents

Figures

TIBCO Documentation and Support Services

All TIBCO documentation is available in the TIBCO Documentation Library, which can be found here:

https://docs.tibco.com

Product-Specific Documentation

The following documents can be found in the TIBCO Documentation Library for TIBCO Enterprise Administrator:

- *TIBCO® Enterprise Administrator User's Guide*
- *TIBCO® Enterprise Administrator Installation*

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support as follows:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to:

http://www.tibcommunity.com

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 artefacts 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 API**: 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 Collaborative Information Manager™ 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 uses on an agent-based architecture. TIBCO Enterprise Administrator SDK comes with two main components: the TIBCO Enterprise Administrator server and the agent library.

The TIBCO Enterprise Administrator provides two distinct user interfaces: a Web-based GUI and a command-line based shell interface. A product being managed using the TIBCO Enterprise Administrator must have a product agent registered with the TIBCO Enterprise Administrator server. You can develop your own agents for your products using the agent library. The TIBCO Enterprise Administrator SDK ships with samples that demonstrate the various aspects of developing product agents. To see your product on TIBCO Enterprise Administrator, perform the following steps:

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

Develop an agent

An agent represents your product in TIBCO Enterprise Administrator. An agent identifies the assets of the product that need to 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 need to 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. The steps are mentioned in Starting the Sample TIBCO Enterprise Administrator Agent.

Register an Agent with the Server

After starting the agent, register the agent with the TIBCO Enterprise Administrator server. The steps are listed in Step 6.

Components of TIBCO Enterprise Administrator

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

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.

Server 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.

Starting the Sample TIBCO Enterprise Administrator Agent

TIBCO Enterprise Administrator SDK ships with samples that demonstrate the various aspects of developing product agents. This procedure lists how to start the sample, `HelloWorldAgent.java` that is located under *TIBCO_HOME*`/tea/`*<version>*`/samples/helloworld`.

Procedure

1. Navigate to *TIBCO_HOME*`/tea/`*<version>*`/samples/helloworld`.

2. Open *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld/helloworld/src/com/tibco/tea/agent/HelloWorldAgent.java`. By default, the following code is displayed. You can make your changes to the code to suit your needs.

```
import com.tibco.tea.agent.TeaAgent;
import com.tibco.tea.agent.TeaAgentServer;
import com.tibco.tea.agent.api.TeaObjectType;
import com.tibco.tea.agent.api.TeaConcept;
import com.tibco.tea.agent.api.TeaGetInfo;
import com.tibco.tea.agent.api.TeaOperation;
import com.tibco.tea.agent.api.TeaParam;
import com.tibco.tea.agent.types.AgentObjectInfo;
import java.io.IOException;

@TeaObjectType(name = "HelloWorldAgent", concept = TeaConcept.TOP_LEVEL,
description = "Hello World Agent")

public class HelloWorldAgent {
    @TeaGetInfo
    public AgentObjectInfo getInfo() {
        AgentObjectInfo agentObject = new AgentObjectInfo();
        agentObject.setName("helloworld");
        agentObject.setDesc("Hellow World Agent");
        return agentObject;
    }
    @TeaOperation(name = "helloworld", description = "Send greetings")
    public String helloworld(
    @TeaParam(name = "greetings", description = "Greetins parameter")final
String greetings)
     throws IOException {
        return "Hello " + greetings;
    }
    public static void main(final String[] args) throws Exception {
        TeaAgent agent = new TeaAgent("HelloWorldAgent", "1.1", "Hellow World
Agent");
        agent.registerInstance(new HelloWorldAgent());
        TeaAgentServer server = new TeaAgentServer(agent, 1234, "/
helloworldagent");
        server.start();
    }
}
```

3. Open the *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld/build.xml`.By default, the following code is displayed. You can make your changes to the code to suit your needs.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="agent" default="compile">
  <target name="compile">
    <javac srcdir="src" destdir="dist">
      <classpath>
        <fileset dir="../../lib">
          <include name="*.jar"/>
        </fileset>
      </classpath>
    </javac>
  </target>

  <target name="start">
    <java classname="HelloWorldAgent" fork="yes">
      <classpath>
        <pathelement location="dist/"/>
        <fileset dir="../../lib">
          <include name="*.jar"/>
        </fileset>
      </classpath>
    </java>
  </target>
</project>
```

4. From the command prompt, navigate to *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld`. To build the java file, run `ant`.
5. To start the agent, run `ant start`.
6. To register the agent with the TIBCO Enterprise Administrator, perform the following steps:
   a) Login to TIBCO Enterprise Administrator.
   b) From the Agents panel, click **Register** .
   c) In the Register Agent window, enter the Agent Name as `helloworld`.
   d) Enter the Agent URL as `http://localhost:1234/helloworldagent`.
   e) Click **Register**.

   The Agent pane is displayed and the helloworld agent appears as a registered agent.
7. To access the agent, from the Agent pane, click **HelloWorldTopLevelType**.

Result

You can access the agent registered on the server.

After registering an agent with the server, objects exposed by the server may not be available for searching within the first 30 seconds after registration. To change this setting, open the `<TEA_CONFIG_HOME>\tibco\cfgmgmt\conf\tea.conf` file and change the value in the `tea.indexing.interval` property. The default value is `30000` milliseconds.

Configuring SSL between the Server and the Agent

To enable SSL, when you start the agent, you should set the SSL system properties. The SSL properties can be set in `<TIBCO_CONFIG_HOME>\tibco\cfgmgmt\conf\tea.conf`.

Procedure

- Set the following SSL properties to enable SSL between the server and the agent:

Property	Description
tea.agent.http.keystore	The file or URL of the SSL Key store location
tea.agent.http.keystore.password	The password for the key store
tea.agent.http.cert.alias	Alias of SSL certificate
tea.agent.http.keymanager.password	The password for the specific key within the key store
tea.agent.http.truststore	The file name or URL of the trust store location
tea.agent.http.truststore.password	The password for the trust store.
tea.agent.http.idletimeout	The maxIdleTime to set, which translates to the Socket.setSoTimeout(int) call.
tea.agent.http.threadpool.acceptors	The number of acceptor threads to set.

Starting the Sample TIBCO Enterprise Administrator Agent

TIBCO Enterprise Administrator SDK ships with samples that demonstrate the various aspects of developing product agents. This procedure lists how to start the sample, `HelloWorldAgent.java` that is located under *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld`.

Procedure

1. Navigate to *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld`.
2. Open *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld/helloworld/src/com/tibco/tea/agent/HelloWorldAgent.java`. By default, the following code is displayed. You can make your changes to the code to suit your needs.

```
import com.tibco.tea.agent.TeaAgent;
import com.tibco.tea.agent.TeaAgentServer;
import com.tibco.tea.agent.api.TeaObjectType;
import com.tibco.tea.agent.api.TeaConcept;
import com.tibco.tea.agent.api.TeaGetInfo;
import com.tibco.tea.agent.api.TeaOperation;
import com.tibco.tea.agent.api.TeaParam;
import com.tibco.tea.agent.types.AgentObjectInfo;
import java.io.IOException;

@TeaObjectType(name = "HelloWorldAgent", concept = TeaConcept.TOP_LEVEL,
description = "Hello World Agent")

public class HelloWorldAgent {
    @TeaGetInfo
    public AgentObjectInfo getInfo() {
        AgentObjectInfo agentObject = new AgentObjectInfo();
        agentObject.setName("helloworld");
        agentObject.setDesc("Hellow World Agent");
        return agentObject;
    }
    @TeaOperation(name = "helloworld", description = "Send greetings")
    public String helloworld(
    @TeaParam(name = "greetings", description = "Greetins parameter")final
String greetings)
     throws IOException {
        return "Hello " + greetings;
    }
    public static void main(final String[] args) throws Exception {
        TeaAgent agent = new TeaAgent("HelloWorldAgent", "1.1", "Hellow World
Agent");
        agent.registerInstance(new HelloWorldAgent());
        TeaAgentServer server = new TeaAgentServer(agent, 1234, "/
helloworldagent");
        server.start();
    }
}
```

3. Open the *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld/build.xml`.By default, the following code is displayed. You can make your changes to the code to suit your needs.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="agent" default="compile">
  <target name="compile">
    <javac srcdir="src" destdir="dist">
      <classpath>
        <fileset dir="../../lib">
          <include name="*.jar"/>
        </fileset>
      </classpath>
    </javac>
  </target>

  <target name="start">
    <java classname="HelloWorldAgent" fork="yes">
      <classpath>
        <pathelement location="dist/"/>
        <fileset dir="../../lib">
          <include name="*.jar"/>
        </fileset>
      </classpath>
    </java>
  </target>
</project>
```

4. From the command prompt, navigate to *TIBCO_HOME*`/tea/`*`<version>`*`/samples/helloworld`. To build the java file, run `ant`.
5. To start the agent, run `ant start`.
6. To register the agent with the TIBCO Enterprise Administrator, perform the following steps:
   a) Login to TIBCO Enterprise Administrator.
   b) From the Agents panel, click **Register** .
   c) In the Register Agent window, enter the Agent Name as `helloworld`.
   d) Enter the Agent URL as `http://localhost:1234/helloworldagent`.
   e) Click **Register**.

   The Agent pane is displayed and the helloworld agent appears as a registered agent.
7. To access the agent, from the Agent pane, click **HelloWorldTopLevelType**.

Result

You can access the agent registered on the server.

After registering an agent with the server, objects exposed by the server may not be available for searching within the first 30 seconds after registration. To change this setting, open the `<TEA_CONFIG_HOME>\tibco\cfgmgmt\conf\tea.conf` file and change the value in the `tea.indexing.interval` property. The default value is `30000` milliseconds.

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 will be 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
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 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.

```
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>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldAgent</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

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();
}
```

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 will automatically unregister 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.

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 = 8327019718482894467L;

    @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 thecom.tibco.tea.agent.server.TeaAgentServlet.autoRegisterAgent() method.

Unregistering an Agent

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

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 `<TEA_CONFIG_HOME>\tibco\cfgmgmt\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 example artifacts, 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
- States
- Operations Aspect
- 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.

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

Operations Aspect

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.

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.

Prerequisites

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

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

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

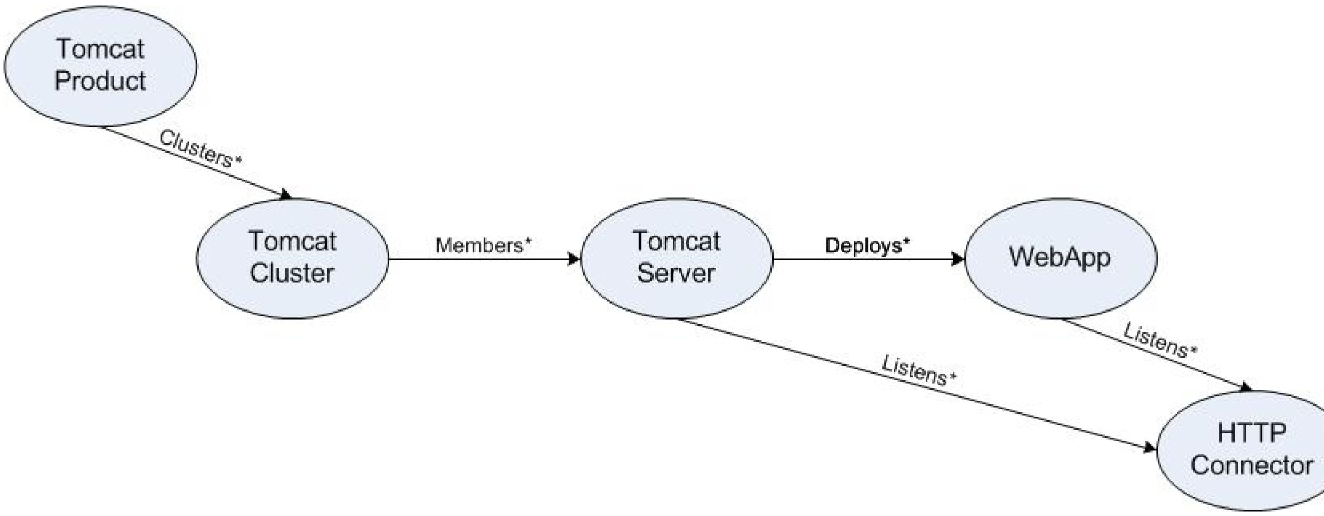
Analyzing a system of managed objects produces artifacts to guide you as you develop an agent. As an example, analysis of Tomcat might produce this table and this relationship diagram.

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 Listens* > HTTP Connecto r	Installation Folder	Running Stopped	Start Stop Deploy App Add Connector Remove Connector
WebApp	Application	Listens* > HTTP Connecto r	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.

Defining TIBCO Enterprise Administrator Object Types

TIBCO Enterprise Administrator Object Types are defined either by extending the interfaces or marking the classes with annotations.

Defining TIBCO Enterprise Administrator Object Types using Interfaces

You can define various Object Types by implementing interfaces provided as a part of the TIBCO Enterprise Administrator agent library.
There are three interfaces that can be used for defining object types:

- com.tibco.tea.agent.api.TopLevelTeaObject
- com.tibco.tea.agent.api.TeaObject
- com.tibco.tea.agent.api.SingletonTeaObject

You also can assign some aspects to these object types to retrieve additional information.

Defining a Top Level Object Type

A top level object type can be defined by creating a class that implements `com.tibco.tea.agent.api.TopLevelTeaObject`.

Procedure

1. Define a class that implements `com.tibco.tea.agent.api.TopLevelTeaObject`

   For example,

   ```
   public class MyProduct implements TopLevelTeaObject {

       public String getTypeName() {
           //..
       }

       public String getTypeDescription() {
           //..
       }

       public String getName() {
           //..
       }

       public String getDescription() {
           //..
       }

       public Collection(BaseTeaObject) getMembers() {
           //..
       }

       @TeaOperation(description = "get", methodType = MethodType.READ)
       public Result get() {
           //..
       }
   }
   ```

   Code explanation:

   - To define an object type which represents your product, create a class which implements `com.tibco.tea.agent.api.TopLevelTeaObject`.
   - To describe the top level object type, use `com.tibco.tea.agent.api.TopLevelTeaObject.getTypeName()` and `com.tibco.tea.agent.api.TopLevelTeaObject.getTypeDescription()`
   - As there is only one instance possible of this type, the instance specific methods are merged with object type specific methods. `com.tibco.tea.agent.api.TopLevelTeaObject.getName()` and `com.tibco.tea.agent.api.TopLevelTeaObject.getDescription()` should be used for describing the single instance of the top level object type.
   - `com.tibco.tea.agent.api.TopLevelTeaObject.getMembers()` should be used to return the members of the single instance of top level object type.
   - Any TeaOperation should be defined using the TeaOperation annotation.

2. Register the top level object type definition with the TIBCO Enterprise Administrator agent library.

   For example, create an instance of the top level object type and register it with the `TeaAgentServer`.

   ```
   TeaAgentServer server = //..
   // register other instances including the top level object type
   MyProduct myProduct = //..
   server.registerInstance(myProduct);
   server.start();
   ```

   When two different versions of agents co-exist on the same TIBCO Enterprise Administrator server, ensure that you do not change the method signature of TopLevelTeaObject between the two versions.

Defining an Object Type

Any object type other than the top level object can be defined using the `com.tibco.tea.agent.api.TeaObject`. The instances of these object types are represented by the object instanciated by the class implementing `com.tibco.tea.agent.api.TeaObject`.

Procedure

1. To define an object type using the `com.tibco.tea.agent.api.TeaObject`, provide a class which implements `com.tibco.tea.agent.api.TeaObjectProvider`.

   For example,

   ```
   public class MyObjectTypeProvider implements TeaObjectProvider<MyObjectType> {

       public String getTypeName() {
           //..
       }

       public String getTypeDescription() {
           //..
       }

       public TeaConcept getConcept() {
           //..
       }

       public MyObjectType getInstance(final String key) {
           //..
       }
   }
   ```

   code explanation:

   - Create a class that implements `com.tibco.tea.agent.api.TeaObjectProvider`.
   - `com.tibco.tea.agent.api.TeaObjectProvider.getTypeName()` and `com.tibco.tea.agent.api.TeaObjectProvider.getTypeDescription()` should be used for describing the object type.
   - `com.tibco.tea.agent.api.TeaObjectProvider.getConcept()` should be used for defining the concept type for this object type. The various concept types are explained later in the document.
   - `com.tibco.tea.agent.api.TeaObjectProvider.getInstance()` should be used for key to instance mapping. When the TIBCO Enterprise Administrator agent library needs to obtain an instance corresponding to a given key for an object type, it will depend on the getInstance() method. For example, when an operation needs to be invoked on a particular instance of an object type, the TIBCO Enterprise Administrator agent library depends on this method for getting the instance by passing the `key` as a parameter.

2. Define the object type by implementing the `com.tibco.tea.agent.api.TeaObject`.

   For example,

```
public class MyObjectType implements TeaObject {

    public String getName() {
        //..
    }

    public String getDescription() {
        //..
    }

    public String getKey() {
        //..
    }

    @TeaOperation(description = "get", methodType = MethodType.READ)
    public Result get() {
        //..
    }
}
```

   code explanation:

   - Create a class which implements `com.tibco.tea.agent.api.TeaObject`.
   - `com.tibco.tea.agent.api.TeaObject.getName()` and `com.tibco.tea.agent.api.TeaObject.getDescription()` should be used for describing the instance of the object type.
   - `com.tibco.tea.agent.api.TeaObject.getKey()` should be used for returning the key of the instance of the object type.
   - Any TeaOperation should be defined using the TeaOperation annotation.

3. Register the object type definition with the TIBCO Enterprise Administrator Agent library.

   For example, create an instance of the object type provider that you created and register that with the `TeaAgentServer`.

```
TeaAgentServer server = //..
// register other instances including the top level object type
MyObjectTypeProvider myObjectTypeProvider = //..
server.registerInstance(myObjectTypeProvider);
server.start();
```

Defining an Object Type with Only One Instance

Procedure

1. To define an object type which only has a single instance, create a class that implements `com.tibco.tea.agent.api.SingletonTeaObject`.

   For example,

```
public class MySingleton implements SingletonTeaObject {

    public String getTypeName() {
        //..
    }

    public String getTypeDescription() {
        //..
    }

    public String getName() {
        //..
    }

    public String getDescription() {
        //..
    }

    public TeaConcept getConcept() {
        //..
    }

    @TeaOperation(description = "get", methodType = MethodType.READ)
    public Result get() {
        //..
    }
}
```

   Code explanation:

   - To define an object type which only has a single instance, create a class which implements `com.tibco.tea.agent.api.SingletonTeaObject`.
   - `com.tibco.tea.agent.api.SingletonTeaObject.getTypeName()` and `com.tibco.tea.agent.api.SingletonTeaObject.getTypeDescription()` should be used for describing the singleton object type.
   - Since there is only one instance possible for this type, the instance specific methods are merged with object type specific methods. `com.tibco.tea.agent.api.SingletonTeaObject.getName()` and `com.tibco.tea.agent.api.SingletonTeaObject.getDescription()` should be used for describing the single instance of the singleton object type.
   - `com.tibco.tea.agent.api.SingletonTeaObject.getConcept()` should be used to return the type of concept of the object type.
   - Any TeaOperation should be defined using the TeaOperation annotation(explained later in the developer's guide).

2. Register the singleton object type definition with the TIBCO Enterprise Administrator Agent library.

   For example, create an instance of the singleton object type and register it with the `TeaAgentServer`.

   ```
   TeaAgentServer server = //..
   // register other instances including the top level object type
   MySingleton mySingleton = //..
   server.registerInstance(mySingleton);
   server.start();
   ```

Object Types Optional Aspects

The TIBCO Enterprise Administrator Agent library supports optional aspects for defining the object type: members, status, and configuration.

To control the definition of an object type, the following interfaces can be added to the interfaces that are used to create object types.

- com.tibco.tea.agent.api.WithMembers
- com.tibco.tea.agent.api.WithStatus
- com.tibco.tea.agent.api.WithConfig

Define an object type with members

An object type can have members or subclasses that the object type contains or refers to. For example, an enterprise might have environments, a machine might have processes, a domain might have applications and so on. Such relationships can be defined using com.tibco.tea.agent.api.WithMembers. TopLevelTeaObject is expected to support this interface.

Define an object type with status

An object type can have a status. For example, a process might be running or stopped. Such object types can be defined using the com.tibco.tea.agent.api.WithStatus.

Define an object type with configuration

An object type can have a configuration. For example, an environment might have virtualization layer configuration or the node might have port configuration. These can be defined using the com.tibco.tea.agent.api.WithConfig.

Defining TIBCO Enterprise Administrator Object Types Using Annotations

User defined classes can be annotated to define TIBCO Enterprise Administrator Object Types. One Java class can be used to define one or more TIBCO Enterprise Administrator Object Types.

The main annotation that defines the TEA Object Type is `@TeaObjectType`. While defining it in the java class, specify the name of the type, concept, and description. The object type should at least have a method with `@TeaGetInfo` annotation. When a key is passed to the method, it should return basic information about the object. An exception to this rule is the TOP_LEVEL type. Since there is only one instance of the TOP_LEVEL type per agent, the getInfo() method does not take a `key` as an argument. You can define a *key* that helps you identify the instances of an object type. A key is a String that is not processed by the server and not visible on the UI.

Procedure

1. To define TIBCO Enterprise Administrator Object Type, mark the java class with the `@TeaObjectType` annotation. Specify the name of the type, concept, and description. You can use the `@TeaObjectTypes` annotation to define multiple types on the same class. The getInfo() method is mandatory for all types, except for top level objects where the method does not take a key as an argument. You can also have domain-specific implementation of obtaining the actual object. For example:

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

    @TeaGetInfo
    public AgentObjectInfo getInfo(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
    }
}
```

2. For an Object Type with many types on the same java class, the minimal implementation is that there is a method marked with the `@TeaGetInfo` annotation. The annotation on the methods has the type they belong to using the `objectType` attribute. This attribute is mandatory when more than one type is defined on a class. Mark the method that takes a `key` as a parameter and returns basic information about an object, with the `@TeaGetInfo` annotation.

   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 {

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

    @TeaGetInfo(objectType = "TOMCAT_WEBAPP")
    AgentObjectInfo getWebappInfo(String key) {
    // ...
    }
}
```

3. (Optional) Mark the method with the `@TeaGetConfig` annotation to retrieve the configuration.

   Configuration is a Java Bean object that can be displayed on the TIBCO Enterprise Administrator server.

   For example,

```
@TeaGetConfig
MyJavaBean getConfiguration(String key) {
    // ...
}
```

4. (Optional) Mark the method with the `@TeaGetStatus` annotation to retrieve the current state of the object.

   State is a custom label that is associated with an object. The label can be defined by the agent and it should represent a state in the life-cycle of the object. Some operations can be linked to particular states of the object. For example, you should not start an application that is already running.

   The getStatus()method returns the `AgentObjectStatus` instance with the state name and description.

   ```
   @TeaGetStatus
   AgentObjectStatus getStatus(String key) {
       // sample code, it should depend on the state of actual object
       AgentObjectStatus result = new AgentObjectStatus();
       result.setState("DEPLOYED");
       result.setDesc("Application is deployed successfully");
       return result;
       }
   ```

5. (Optional) Mark the method with the `@TeaGetMembers` annotation to retrieve the members list. The getMembers() method returns an array of `AgentObjectIdentifier` instances.

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

Defining Operations for Objects

Any registered object type can expose a method as an operation available to TIBCO Enterprise Administrator server and the user interface.

Procedure

1. Define a method signature. The method can have parameters or return simple types or objects that conform to the java bean specification.
   For example, the `addUser()` method maps a user from the given LDAP realm to a set of roles.
   ```
   public void addUser(String realm, LdapUserMapping config) {
           // the code
           }
   ```
2. Add annotations to mark the method as an operation and define its parameters.

Annotations	Description
TeaOperation	Marks the method as an operation. The annotation defines the name and description of the operation.
KeyParam	A key for the object that is the target for this operation. This attribute can be skipped for an operation on top-level types, as there can be only one instance of top level object per agent.
TeaParam	Each parameter is annotated with a parameter name, an optional description and a default value. This is required as Java does not preserve names of the parameters at runtime.
TeaRequires (optional)	The annotation should go together with the TeaOperation annotation. It defines list of permissions required to invoke given operation. If the operations does not have permission annotation, it is considered 'Read-only', and it will require only 'Read' permission.

   For example, the `addUser()` method is marked as an operation using the TeaOperation annotation.
   ```
   @TeaRequires(value={"TEA_ADMIN","TEA_USER", "TEA_SUPER_USER"} )
   @TeaOperation(name ="add-user", description = "Create new user")
   public void addUser(
          @KeyParam final String key,
          @TeaParam(name = "realm") final String realm,
          @TeaParam(name = "user") final Object config) {
                // code
   }
   ```
   The annotation is sufficient to expose this operation in the default UI. The operation bar on the object has a button for adding a user which opens a default form. The operation is available in the shell as well.

   To return a JSON object, the agent library supports Jackson databinding through Jackson or JAXB annotations. The agent library does not support org.json module for Jackson. You can write plain java objects with annotation, or if you want to use generic types, you can use the Jackson databind API or use Map<String, Object>, which is a a map of maps representing the JSON structure.

3. Add parameters that represent file artifacts. Use the TeaParam annotation on a method parameter with type `javax.activation.DataSource`. For example:

```
@TeaOperation(name = "upload", description = "Upload File")
public String uploadFile(
    @TeaParam(name = "file", description = "File to Upload") DataSource upload)
throws IOException {

  InputStream inputStream = upload.getInputStream();
  ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
  byte[] buffer = new byte[1024];
  int read;
  while ((read = inputStream.read(buffer)) != -1) {
    outputStream.write(buffer, 0, read);
  }
  inputStream.close();
  outputStream.close();
  File file = File.createTempFile(upload.getName(), null);
  FileOutputStream fileOutputStream = new FileOutputStream(file);
  outputStream.writeTo(fileOutputStream);
  fileOutputStream.close();
  return upload.getName();
}
```

4. Specify the permissions required to invoke the operation. The TeaRequires annotation lists all the permissions required to invoke the operation in addition to the Read permission for the parent object. If the annotation is not present, anyone with the Read permission for the parent object can invoke it. For example:

```
@TeaRequires(value={"TEA_ADMIN","TEA_USER", "TEA_SUPER_USER"} )
@TeaOperation(name ="add-user", description = "Create new user")
public void addUser(
    @KeyParam final String key,
    @TeaParam(name = "realm") final String realm,
    @TeaParam(name = "user") final Object config) {
        // code
}
```

You must define the Permission names (in this case, "TEA_ADMIN", "TEA_USER", "TEA_SUPER_USER") which is displayed in the UI for creating privileges and roles.

If there are two agents for the same object type, ensure that they have the same operation name and number. This is to ensure that when you invoke an operation, you can select the agent on which you want to execute the operation from the drop-down list on the Web-based GUI.

Caution: When multiple agents coexist with each other, you should not modify an existing operation in the new version of the agent. Do not add or delete parameters to an existing operation. Also, ensure that you do not modify the signature of an existing TeaOperation.

Downloading a File

To download a file, you need a TeaOperation that returns a Datasource pointing to the file you want to download. While developing the UI, you must invoke the TeaOperation using the teaObjectService. Take the example of the Tomcat sample provided with the product.

Procedure

1. The downloadLog() method defines a TeaOperation that returns a Datasource that points to the file you want to download. The sample downloads the license file shipped with the product.

```
@Customize(value = "label:DownloadLicense;internal:true")
@TeaOperation(name = "downloadLicense", description = "Download Tomcat license
file",
internal = true, methodType = MethodType.UPDATE)
public DataSource downloadLog(){
   return new FileDataSource(this.tomcatAgentConfig.getInstallationDir()+ "/
LICENSE");
}
```

2. While developing the UI, invoke the TeaOperation using the teaObjectService.

```
$scope.downloadFile = function(view) {
              teaObjectService.invoke({
                     agentType : $scope.object.type.agentType,
                     objectType : $scope.object.type.name,
                     agentId : $scope.object.agentId,
                     objectKey : $scope.object.key,
                     operation : "downloadLicense",
                     methodType : "UPDATE",
                     params : {}
              }).then(function(data) {
                     var url = document.location.origin + '/tea/' + data;
                     if(view){
                           document.location.href = url;
                     } else {
                           document.location.href = url +"?download=true";
                     }
              }, function(error) {
                     $scope.errorMessage = error.message;
                     if (error && error.params) {
                           for ( var param in $scope.params) {
                                  if (typeof $scope.params[param] === 'object') {
                                         $scope.params[param].error =
error.params[param];
                                  }
                           }
                     }
              });
       }
```

Asynchronous Operations Support

For defining an asynchronous operation, you should return an instance of `com.tibco.tea.agent.api.SettableFuture`.

SettableFuture is a custom Future object which the TIBCO Enterprise Administrator agent library uses for tracking asynchronous operations.

For example,

```
public SettableFuture<Response> startProcess {
     SettableFuture<Response> future = SettableFutureFactory.create();
     // pass future to the code which will handle the operation execution
     return future;
}
```

Where,

- startProcess is an asynchronous operation which has to return an object of type Response. To use the asynchronous support provided by the TIBCO Enterprise Administrator agent library, startProcess sets the return type as `SettableFuture<Response>`.
- startProcess will start by creating an instance of SettableFuture using the SettableFutureFactory.
- The SettableFuture instance is passed to the code which handles the operation execution. For example, the SettableFuture instance might be passed on to an instance of a class which implements `java.lang.Runnable` and that instance is passed to an instance of `java.util.concurrent.Executor`. Hence the starting of the process is handled by a thread pool.
- Finally, the instance of SettableFuture is returned to the TIBCO Enterprise Administrator agent library. The TIBCO Enterprise Administrator agent library tracks the asynchronous operation by following this instance.

Using SettableFuture instance to set the progress

You can also use the instance of SettableFuture for setting the progress, progress status, and returning the object. For example,

```
// inside the code handling operation execution
future.setProgressStatus("Connecting to host to start the process");
// ...
future.setProgress(25);
future.setProgressStatus("Spawning the process");
// ...
future.setProgress(50);
future.setProgressStatus("Waiting for the process to come up");
// ...
future.setProgress(100);
future.set(response);
```

Where,

- `SettableFuture.setProgressStatus()`is used to update the message which is displayed to the user to indicate the status of the asynchronous operation.
- `SettableFuture.setProgress()` is used to update the percentage of work that is completed so far. This number should be between 0 and 100.
- `SettableFuture.set()` is used for setting the response object which is returned to the TIBCO Enterprise Administrator server. This indicates the completion of the asynchronous operation.

Using SettableFuture instance for exception

You can also use the instance of SettableFuture in case of an exception.

```
// inside the code handling operation execution
future.setException(ex);
```

Where, `SettableFuture.setException()` should be used for setting the exception object that is returned to the TIBCO Enterprise Administrator server. This indicates the completion of the asynchronous operation.

Defining References for Object Types with Interfaces

A method on one of the registered object types can be exposed as a reference available to TIBCO Enterprise Administrator server and the user interface. In this manner, you can refer a TeaObject from another TeaObject.

Procedure

1. Define the method signature. The method cannot have any parameters and must return an array of objects that implement `com.tibco.tea.agent.api.BaseTeaObject`.
   For example, the `getNodes()` method returns an array of nodes, and the Node class implements the class `com.tibco.tea.agent.api.BaseTeaObject`.
   ```
    public Node[] getNodes(){
           // The code...
           }
   ```
2. Mark the method as a reference using the `@TeaReference` annotation. It provides the following information:

Parameters	Description
name	Name of the reference
objectType (optional)	Needed when more than one object types are defined on the referenced class

   For example:
   ```
   @TeaReference(name = "nodes")
           public Node[] getNodes(){
           // The code...
           }
   ```

Defining References for Object Types with Annotations

A method on one of the registered objects types, created using annotations, can be exposed as a reference available to TIBCO Enterprise Administrator server and the user interface. In this manner, you can refer to a TeaObject from another TeaObject.

Procedure

1. Define a method, which has an object key as a parameter and returns an array of `com.tibco.tea.agent.types.AgentObjectIdentifier` objects.

   Each reference is invoked in context of the object and the parameter `key` identifies the source object. This attribute can be skipped for references on top-level types, as there can be only one instance of a top level object per agent.

   For example, the `getApplications()` method returns an array of `com.tibco.tea.agent.types.AgentObjectIdentifier` references.

   ```
    public AgentObjectIdentifier[] getApplications(final String key){
               //The code...
           }
   ```

2. Add the TeaReference annotation.
   For example, the `getApplications()` method is marked as a reference using the TeaReference annotation.

   ```
   @TeaReference(name = "applications", referenceType = "applicationType",
   objectType = "applicationObjectType")
   public AgentObjectIdentifier[] getApplications( final String key){
       //The code...
   }
   ```

   Code explanation:

   - The TeaReference annotation marks the method as a reference. It gives you the following information:

Parameters	Description
name	Name of the reference
referenceType	Type of the element
objectType (optional)	Needed when more than one object type is defined on the referenced class

   - AgentObjectIdentifier[] -The getApplications() method returns an array of AgentObjectIdentifier references.
   - @keyParam - Each reference is invoked in context of the object, KeyParam, which is a key of the source object. This attribute can be skipped for references on top-level types, as there can be only one instance of top level object per agent.

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 should 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

Roles

A *role* is a collection of privileges that are assigned to users or groups using class-level annotations.

Before defining roles on an agent, nsure that the agent is registered with the server. While defining roles, if there is a conflict in the names, the roles available on the server are used. You can change or delete roles imported to the server. When the last agent of a specific type is unregistered, associated roles are removed.

Roles can be assigned using class-level annotations. For example,

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

public class TomcatServer {

    @TeaRequires("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
    }
}
```

As shown in the example, two roles are assigned: one for the Tomcat Administrator and one for a regular user.

TeaRole

TeaRole annotation is used to assign the default roles provided by the agent. The roles are available in the TIBCO Enterprise Administrator server only after registering an agent of a specific agent type for the first time. They are removed when the last agent of a specific type is unregistered. After creation, roles can be changed by the TIBCO Enterprise Administrator SDK Server administrator.

TeaRoles

If a specific role already exists on the server, it will be ignored. To assign multiple roles by the same class, use the grouping annotation, TeaRoles.

TeaPrivilege

Defines the privileges of a role. A *privilege* describes a set of permissions that are granted to objects that match the specified path pattern. A role can have one or more privileges associated with it.

The following elements are available on the TeaPrivilege annotation :

permissions

A list of permissions that are applicable to this role. Some default permissions are available on the server. Agents can add more to this list. By default, the `Full Control` (full access to all objects and operations) and `Read` (read-only access) permissions are available on the server.

objectType

Is the objectType to which a privilege is applied. The default value is `all`.

Caution: If you upgrade an agent, you can only have additional roles on the same agent. You cannot delete or change the existing role definitions.

Permissions

TIBCO Enterprise Administrator implements permission checking based on the privileges and the roles defined on an object.

Key terms

User

Users are entities that need access to the system. Each user might need a different level of access to the system. Users can be assigned to multiple Roles. TIBCO Enterprise Administrator does not manage users by itself. Users from external systems are mapped into TIBCO Enterprise Administrator to allow access to the system.

Group

Groups are logical groupings of the users within an organization. A user can belong to multiple groups and a group can contain multiple users. Groups provide an easier way to control access to users. Instead of specifying the access permissions for each user, it is easier and practical to specify access permissions to the groups to which they belong to. Groups can contain sub-groups.

Realm

A security realm comprises mechanisms for protecting TIBCO Enterprise Administrator resources. It contains users, groups, and their security credentials. The realm provides information about users and the groups they belong to. TIBCO Enterprise Administrator supports two kinds of realms : File and LDAP. In a File realm, the user and group information is stored in a file. In an LDAP realm, the user or group information exists on an LDAP server and is accessed from the server.

Permission

A string on the basis of which access control is enforced. It is upto the agent to decide the granularity of the permissions that it provides. For example, a permission could be as fine-grained as 'UpdateConfig' which is applicable to only one operation, or it could be as coarse-grained as 'Full Control' which applies to the entire system.

Caution: If you upgrade an agent, you can only have additional permissions on the same agent. You cannot delete or change the existing permission definitions.

Privilege

Privilege is a collection of permissions that are applicable to an object or a collection of objects.

Role

Role is a mechanism to grant or revoke access to users. A Role is a collection of privileges and are assigned to users and groups. All the privileges in a role get associated to the user or group to which it is assigned.

Custom Permission

You can assign custom permissions by using the TeaPermission and TeaPermissions annotation.

For example, Lifecycle and Update permission are grouped using the TeaPermissions annotation.

```
@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 {
        // code
        }
```

An agent can define the permissions needed to execute each of the operations that it provides. If a method does not have any TeaRequires annotation on it, then that method can be executed by anyone.

Effective Permissions

The collection of privileges that are applicable to a user are obtained as follows:

- Gather the privileges from all the roles assigned to this user directly.
- Gather the privileges from the roles assigned to all the groups to which the user belongs.

User Interface Customization

The UI can be customized using an external file. The file can be uploaded through the `customize` operation on the agent type.

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 should 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.

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

Syntax for the customization rules:

```
<Object Type Name>#<Operation Name>#<Method Type>.<Parameter 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 will be collapsed, on display.
- `referenceOrder`Is the order in which a reference is displayed. For example, `"referenceOrder": ["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.

`@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, will default 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 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.

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

The 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/view to another.

URLs in TEA 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 TEA 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 will be '_'.
  - **`[query] - {object}`** - Query object. Specifies name-value pairs that will be added as query parameters within the URL.

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 TEA screens to support pivoting to related objects, without having to hard-code the URL structure everywhere and without needing to know details of how TEA accomplishes the link.

teaLocation.goto() will automatically URL-encode the query string parameters. TEA does not impose any limitations on the characters being used in object keys, query parameters, etc. The query string may contain any valid UTF-8 character. As a result, screen developers should not need to implement URL encoding or decoding for these parameters, as this is provided by teaLocation.

Example

One of the BusinessWorks screens within TEA 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, TEA navigates to the TEA screen that displays details about a single App.

One of the BusinessWorks screens within TEA 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, TEA navigates to the TEA screen that displays details about a single SAP Adapter Endpoint. Note that this screen is not contributed to TEA by the BW agent, but rather by the SAP Adapter agent. The information about this agent - agent name, agent version, available object types, available views, etc - is published as a spec by the Adapters team and used by the BW team when developing the BusinessWorks UI.

teaObjectService

This is a service available under tea.services. The objectService is a factory which creates TEA objects providing data from TEA 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 TEA 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 be 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 be needed if the object is a singleton.

Returns

`{Promise}` – Returns a promise that will be 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 TEA object.

Parameters

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

Returns

`{Promise}` – Returns a promise that will be 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 TEA object type.

Parameters

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

Returns

`{Promise}` – Returns a promise that will be 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 be 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 be 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.

Returns

`{Promise}` – Returns a promise that will be 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 TEA 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 be 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 be needed if the object is a singleton.
- `params – {object}` – Object parameter defining arguments of the operation invocation.

Returns

`{Promise}` – Returns a promise that will be 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 i.e. TEA 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 be 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 be 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 will be resolved with the objects requested as Array<Teaobject>

Usage

Typical usage of the objectService 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 thelistPrivileges method.

Dependencies:

1. $q
2. $http

Methods

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

listPrivileges

Returns the 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 should 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.

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 should be used with teaAttribute directive only.
- **teaMastheadLongAttributes**: Long attributes that are rendered in the second column of the masthead. This should be used with teaLongAttribute directive only.
- **teaMastheadActions**: Action buttons that are rendered in the masthead. This should 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 should 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} - 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} - The type ID of the elements 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.

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 = "Greetins 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.

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.

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.

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.