# 1. Project Overview

The connectivity project is composed of the following components:

- Driver management framework
- Connection profile framework
- Connection management layer
- Extension implementations supporting generic JDBC connections

# 2. Driver Management Framework

The driver management framework provides the user with a mechanism for defining driver definitions based on driver templates. Specific driver templates are supplied by developers through an Eclipse extension point.

Driver templates are designed to allow users to quickly and easily create driver definitions for specific server types. The template may specify a list of libraries required for connecting to the server, a list of default property values to be used when creating new connection profiles (e.g. driver class name, default connection port, etc.), and other information that may be used when opening a connection.

Driver definitions are created from driver templates by the user. The user has the option of overriding and/or supplementing the default values specified in the template (e.g. specifying different JAR files, a different default connection port, etc.). Driver definitions may be referenced from and used by connection profiles when opening connections to a server.

## *2.1. Extension Point*

The driver management extension point gives the developer the ability to define the following:

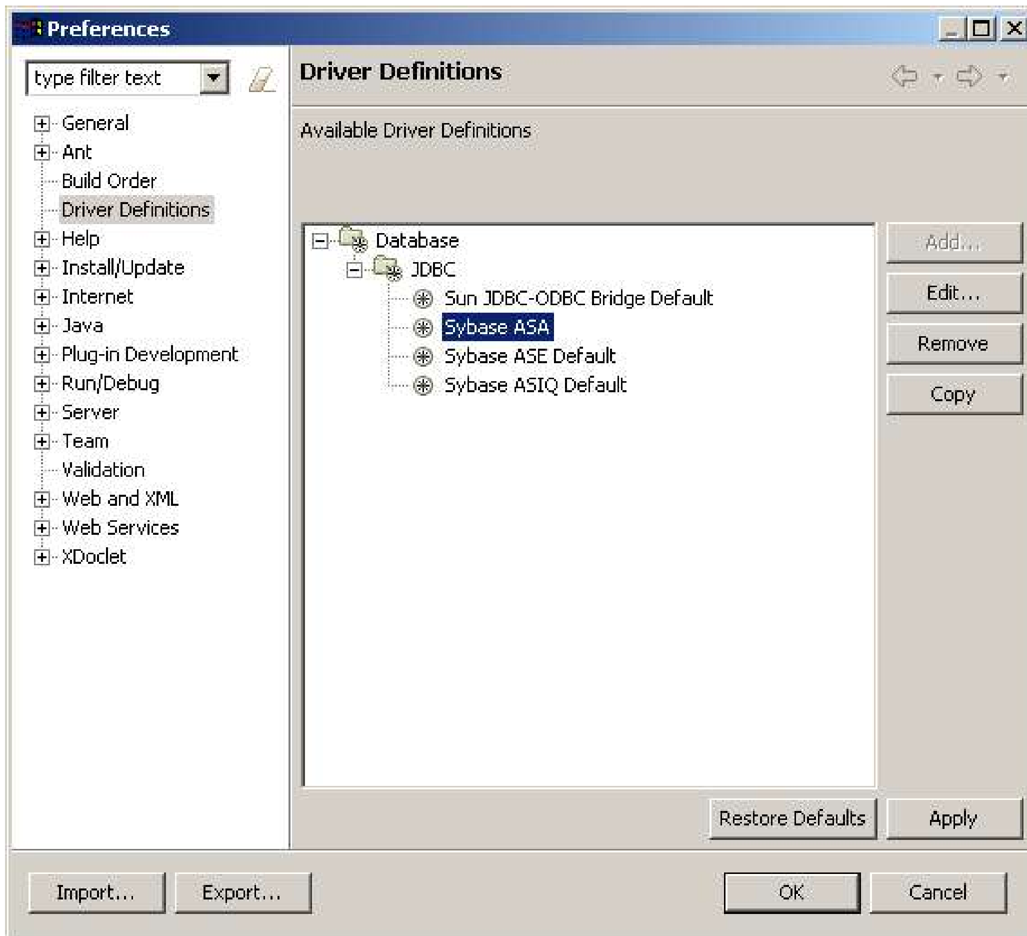
- Categories for driver templates/definitions
- Classpath/list of required JAR files
- User editable properties (e.g. port, uid)
- Hidden properties (e.g. driver class)

## *2.2. API*

The driver management API is intended to allow developers to create, modify or delete driver definition instances. It also gives developers access to the properties defined in driver definition instances.

## *2.3. User Interface*

A preference page will be provided allowing users to create driver definition instances from the driver templates defined through the extension point. The preference page should look something like this,


Preferences
type filter text
Driver Definitions
General
Ant
Build Order
Driver Definitions
Help
Install/Update
Internet
Java
Plug-in Development
Run/Debug
Server
Team
Validation
Web and XML
Web Services
XDoclet
Available Driver Definitions
Database
JDBC
Sun JDBC-ODBC Bridge Default
Sybase ASA
Sybase ASE Default
Sybase ASIQ Default
Add...
Edit...
Remove
Copy
Restore Defaults
Apply
Import...
Export...
OK
Cancel

# 3. Connection Profile Framework

The connection profile framework provides the user with a mechanism for defining connection profiles. Specific connection profile types (or classes) are supplied by developers through an Eclipse extension point.

A connection profile is used to define all the information required to connect to a server. This information may include a reference to a driver definition instance specifying the classpath to be used for loading the driver, a user ID, port, host, etc. This information is used when opening connections to a server.

## *3.1. Extension Point*

The connection profile extension point gives the developer the ability to define the following:

- Categories for connection profiles
- Connection profile types
- Connection factories for opening connections
- Wizards for creating connection profiles

A connection profile may have one or more connection factories associated with it (for example, an application server may have a connection factories that create connections for accessing/manipulating J2EE objects (EARs, EJBs, WARs, etc.), JMS functionality on the server, etc. By convention, connection factories should be IDed by the interface of the connection object they create (e.g. java.sql.Connection for databases). This allows a general UI feature set to be developed for working with any type of connection profile, provided a connection factory of the appropriate type is defined for that connection profile type.

## *3.2. API*

The connection profile API is intended to allow developers to create, modify and delete connection profile instances; to open connections to servers (through provided connection factories); to access properties defined on connection profile instances; to be notified of changes to connection profiles (e.g. create, modify, delete).

## *3.3. User Interface*

A generic new wizard will be provided. This wizard will allow the user to select from a list of new wizards provided for each connection profile type (similar to the existing "File->New…" wizard).

For DTP, a "Data Source Explorer" view will be provided. This view will display connection profiles that can be used with other DTP components (e.g. SQL editor). Navigator extensions should be provided for displaying the contents of the DB (as the current servers view in WTP does).

# 4. Connection Management Layer

The connection management layer is intended to provide functionality enabling connection sharing between tooling components.[1]

Some issues that need to be addressed:

- Transactions[2]
- Investigate using J2EE connectors[3]
- Connection events (e.g. connect, disconnect, reconnect)
- Offline capability[4]
- Others?

## *4.1. API*

## *4.2. User Interface*

A generic login dialog should be provided.[5]

# 5. Generic JDBC Extension Implementations

Driver definition and connection profile extensions should be implemented that allow the user to work with any type of JDBC accessible data source. Specific vendors may provide additional driver definition and connection profile extensions that are tailored to their specific DB.

## *5.1. Driver Definition Extension*

The driver definition extension should allow the user to specify a list of required JAR files (i.e. classpath) and the driver class name.

## *5.2. Connection Profile Extension*

The connection profile extension should allow the user to specify a driver definition, connection URL, UID and PWD for connecting to a DB.

Connection factories should be implemented for creating java.sql.Connection objects and SQL model objects.

---

[1] Currently, only a simple connection management mechanism is implemented (i.e. more design work is necessary for this component). We might consider generalizing the functionality already present in the RDB implementation.

[2] One idea that came up during the consolidation meetings was to have a globally shared read-only connection and any tooling requiring write-access would need to use a clone of the shared connection. (The issue here was how to address sharing between components that may make updates to the DB, e.g. multiple editors can be acting on a single DB).

[3] This was another idea that came up during the consolidation meetings. This would require changes to the existing connection profile management code. This might be a good way to go if there is an existing open source solution compatible with the EPL. We'd also need to look at how the driver management framework would be impacted.

[4] This is supported through the SQL model (or is it the DB definition model).

[5] The current implementation stores both UID and PWD in the connection profile. This is a deviation from the existing RDB implementation. If this is provided, we should probably standardize property keys for commonly used properties (e.g. UID, URL, host, port, etc.; we might want to do this anyway).

A property page should also be supplied for editing existing generic JDBC connection profiles.

# 6. Project Integration

This section describes how the connectivity layer might be used by other components within Eclipse.

## *6.1. DTP – Model Base*

### 6.1.1. Database Definition Model

The database definition model should probably be associated with a driver definition as a hidden property.[6]

### 6.1.2. SQL Model

The SQL model instance for the DB should be accessible from a connection profile through a connection factory. This will allow any consumer access to the SQL model instance in a generic way (assuming the ID convention noted above is used). This should also be integrated with the connection management layer to support offline capabilities.

## *6.2. DTP – SQL Development Tools*

## *6.3. BIRT*

ODA data sources may be represented using a driver definition/connection profile pair. A specific connection factory type could be defined for supplying connection types known by BIRT tooling.

### 6.3.1. JDBC Data Sources

It may be possible to implement the appropriate ODA interfaces using an underlying JDBC or SQL model connection. If so, existing DB connection profiles could be extended by providing an ODA connection factory using the default implementation. (The generic implementation could have a protected method used for creating the JDBC connection. This would be implemented by the vendor providing support for BIRT. Presumably, there would be a generic implementation extending the generic JDBC connection profile supplied by DTP.)

[6] The assumption here is that the DB definition model being referenced is probably specific to the driver being used and is at least specific to the physical DB being supported by the driver definition.