Avaya Breeze™ External Database Access

The following document provides sample code and guidelines for how a Avaya Breeze Service can use Java Persistence API (JPA) to access data from an external database. The document is only meant as a "getting started" guide and only provides very high level concepts.
The document will use the WhiteList sample application to illustrate the points. For a design overview of the WhiteList sample application, please refer to the *Whitelist Sample Service Guide*. The source code, without the concepts discussed in this document, is located in the *Avaya Breeze SDK, Developer Preview Release*.
Both the Whitelist Sample Service Guide and the SDK are available on Avaya DevConnect.
This document will provide sample code in order to enhance the WhiteList service with the concepts described here.

There are six areas covered:
1. Standard JPA configuration file: persistence.xml
2. Stateless beans with local interfaces
3. Connection pooling
4. Entity
5. Entity Manager
6. Cache Specific Attributes

Standard JPA configuration file: persistence.xml

The persistence.xml configuration file, located at `META-INF/persistence.xml`, defines one or more persistence units. At a high level, a persistence unit contains information about how to contact a database, and entities associated with that unit. An entity represents a table in a relational database; an entity instance represents a row in that database.
The WhiteList sample service connects to a PostgreSQL database on the server itself. It uses an entity, `WhiteListEntry`, to read a row from the whitelist database. The WhiteList `persistence.xml` has one persistence unit. It is named `whiteListDataSource`, and contains one entity, `com.avaya.services.whitelist.db.WhiteListEntry`. It also contains properties about how to connect to a local PostgreSQL database, such as the database driver,`org.postgresql.Driver`, Url, `jdbc:postgresql://localhost/whitelist` and credentials.

Stateless beans with local interfaces

The main class for the WhiteList service is `WhiteList.class`. It makes use of a helper class, `DestinationFinderImpl`, which retrieves the ultimate destination for the call. The `DestinationFinderImpl` is a POJO and is created in the `WhiteList` constructor. We want to turn the `DestinationFinderImpl` into a stateless bean. That way, it can use dependency injection for its resources and enjoy all other benefits of being a container managed bean.

- In order to turn the class into a stateless local bean, add the bolded, **`@Local`**, text to the `DestinationFinder`:

```
@Local
public interface DestinationFinder {…}
```

- Add the bolded text, **`@Stateless`**, to the `DestinationFinderImpl`:

```
@Stateless
public class DestinationFinderImpl implements DestinationFinder {…}
```

The `WhiteList.class` is not managed by the container. Therefore it must use JNDI to lookup the `DestinationFinderImpl` bean. It cannot use injection.

There is only one implementation of the local `DestinationFinder` interface. The JNDI name for the bean becomes: `ejblocal:com.avaya.services.whitelist.DestinationFinder`.

The `DestinationFinderImpl` stateless bean also makes use of another stateless bean, `PermissionAgentImpl`. Since the `DestinationFinderImpl` is a bean managed by the container, it is not necessary to use JNDI to lookup that bean, it can use injection. The injection becomes:

```
@Stateless
public class DestinationFinderImpl implements DestinationFinder
{
        @EJB
        private PermissionAgent permissionAgent;
        …
}
```

The above describes the procedure of how to either lookup or inject a stateless bean.

Connection pooling

The WhiteList service makes a request to the database every time it is invoked. This is to determine whether the caller is allowed to directly call the called party.

Without connection pooling, each request requires a connection to be opened to the PostgreSQL database. That is a time intensive operation.

In order to avoid that time intensive operation, the application makes use of a connection pool. This, very simply, means that the database connections are maintained so future requests can reuse an existing connection.

One way to enable connection pooling for the application is to make use of the DBCP component of the Apache project (http://commons.apache.org/dbcp). Add the bolded line to the persistence unit in the persistence.xml:

```
<persistence-unit name="whiteListDataSource" transaction-type="JTA">
```

```
        <class>com.avaya.services.whitelist.db.WhiteListEntry</class>
        <properties>
                <property name="openjpa.ConnectionProperties"
                        value="DriverClassName=org.postgresql.Driver,
                        Url=jdbc:postgresql://135.9.182.116:5432/whitelist,
                        MaxActive=100,
                        MaxWait=10000,
                        TestOnBorrow=true,
                        Username=postgres,
                        Password=postgres" />
                <property name="openjpa.ConnectionDriverName"
                         value="org.apache.commons.dbcp.BasicDataSource"/>
        </properties>
    </persistence-unit>
</persistence>
```

Entity

The WhiteList service uses one entity, the `WhiteListEntry`. It contains two attributes: calling and called handles.

The class is annotated as an entity, using the `@Entity` annotation. It is also annotated with name of the database, `@Table(name = "WHITELIST")`.

Entity Manager

An Entity Manager is associated with a Persistence Context. A Persistence Context is essentially a cache.

The `WhiteList` service uses the `WhiteListDaoImpl` to interact with the PostgrSQL database. The `WhiteListDaoImpl` is created for every request along with an Entity Manager. This means that any cached data is flushed for each request.

Using the same technique described in the "Stateless beans with local interfaces" section above we can turn the `WhiteListDaoImpl` into a stateless bean. This allows us to inject an Entity Manager into the bean, thus removing the need to get an Entity Manager for each request.

The code for the `WhiteListDaoImpl` with the Entity Manager injected looks like this:

```
@Stateless
public class WhiteListDaoImpl implements WhiteListDao
{
    private static final String PERSISTENCE_UNIT_NAME = "whiteListDataSource";

    @PersistenceContext(unitName = PERSISTENCE_UNIT_NAME)
    private EntityManager entityManager;
    …
}
```

In order to inject a `@PersistenceContext`, the Persistence Unit must use a transaction-type of JTA. The persistence.xml changes the name

- From: `<persistence-unit name="whitelist" transaction-type="RESOURCE_LOCAL">`
- To: **`<persistence-unit name="whiteListDataSource" transaction-type="JTA">`**

Although not necessary, the name change better represents that we're going to use a data source.

Cache Specific Attributes

The above information describes a very simple configuration and java code to read information from a PostgreSQL database.

The following information will explain what additional attributes are needed to enable basic JPA caching and a time-based eviction scheme for queries.

One way to turn on caching is to add the following line to the persistence unit in persistence.xml:

```
<property name="openjpa.DataCache" value="true"/>
```

That will cache a default number of queries. The default number of queries can be changed by adding the following attribute:

```
<property name="openjpa.QueryCache" value="true(CacheSize=10000, SoftReferenceSize=0)"/>
```

The above line will make the cache size 10,000. The `SoftReferenceSize` parameter indicates how many evicted queries will still have soft references to them.

With the two above lines added, the persistence.xml becomes:

```
<persistence-unit name="whiteListDataSource" transaction-type="JTA">
               <class>com.avaya.services.whitelist.db.WhiteListEntry</class>
               <properties>
                      <property name="openjpa.ConnectionProperties"
                             value="DriverClassName=org.postgresql.Driver,
                      Url=jdbc:postgresql://135.9.182.116:5432/whitelist,
                      MaxActive=100,
                      MaxWait=10000,
                      TestOnBorrow=true,
                      Username=postgres,
                      Password=postgres" />
                      <property name="openjpa.ConnectionDriverName"
                      value="org.apache.commons.dbcp.BasicDataSource"/>
<property name="openjpa.DataCache" value="true"/>
<property name="openjpa.QueryCache" value="true(CacheSize=10000, SoftReferenceSize=0)"/>
               </properties>
       </persistence-unit>
</persistence>
```

The WhiteList service only reads from the external database. It does not use JPA to write data. Updates to the database are performed by logging into PostgresSQL and affecting the data directly. Because updates are not reflected in the persistence context, such data is stale from the service's perspective.

For instance, say Alice is allowed to call Bob directly. The element manager caches this query. Then the database is modified such that Alice is not allowed to call Bob directly. The persistence context still has the old query cached and until the cache changes the service will allow calls directly between Alice and Bob. That's not good.

One simple way around the above issue is to evict queries from the cache periodically. This can be achieved by simply adding the following line to the entity:

```
@DataCache(timeout = 600000) // This would evict the cached entry after 600 seconds (10 minutes)
```

The above strategy would guarantee that a query is at most 10 minutes out of synch with the data source.

With the above annotation added in bold, the entity becomes:

```
@Entity(name = "WHITELIST")
@DataCache(timeout = CacheValues.DEFAULT_EVICTION_TIMEOUT_IN_MILLISECONDS)
@Table(name = "WHITELIST",
        uniqueConstraints =
        { @UniqueConstraint(columnNames =
        { "CALLED_HANDLE", "CALLING_HANDLE" }) })
@NamedQuery(
        name = NamedQueries.FIND_WHITELIST_ENTRY,
        query = "SELECT c FROM WHITELIST c " +
                "WHERE c.calledHandle= :calledNumber and c.callingHandle= :callingNumber")
public final class WhiteListEntry implements Serializable {…}
```