First Look: XEP Object Persistence with InterSystems IRIS

Version 2018.1
2018-01-31

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

First Look: XEP Object Persistence with InterSystems IRIS
InterSystems Version 2018.1 2018-01-31

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

First Look: XEP Object Persistence with InterSystems IRIS

This quick start guide introduces you to InterSystems IRIS support for Java object persistence with the XEP Java object persistence API. It covers the following topics:

This First Look guide presents an introduction to XEP Java object persistence and walks through a simple scenario that demonstrates the main features of the API.

These activities are designed to use only the default settings and features, so that you can acquaint yourself with the fundamentals of XEP without having to deal with details that are beyond the scope of this overview. For the full documentation on XEP, see *Persisting Java Objects with InterSystems IRIS XEP*.

1 Rapid Java Object Storage and Retrieval

Java is an object oriented language, so it is natural for Java applications to model data as objects. However, this can cause problems when the application needs to store that data in a database. If you use JDBC to store and retrieve objects, you are faced with the problem of transforming your object data into a set of relational tables, and then transforming query resultsets back into objects. The usual solution to this problem is to use an object-relational mapping (ORM) framework such as Hibernate to automate the process. InterSystems ISIS does offer a standard Hibernate interface, and we recommend it for large, complex object hierarchies.

On the other hand, for applications that perform tasks such as real-time data acquisition, the main problem is speed rather than data complexity. Although Hibernate is by no means slow (if properly optimized), a Hibernate application may still need to use straight JDBC for time-critical tasks such as batch data import.

For applications that require extremely fast storage and retrieval of moderately complex data, XEP is by far the best alternative. It can be used as either a stand-alone ORM or as an alternative to JDBC in a Hibernate application, and will in either case be many times faster than either Hibernate or JDBC.

2 How Does XEP Work?

XEP (express event persistence) is a lightweight Java API that projects Java object data as *persistent events*. A persistent event is an instance of an InterSystems IRIS class (normally a subclass of %Persistent) containing a copy of the data fields in a Java object. Like any such instance, it can be retrieved by object access, SQL query, or direct global access.

Before a persistent event can be created and stored, XEP must analyze the corresponding Java class and *import a schema* into the database. A schema defines the structure of the persistent event class that will be used to store the Java objects. Importing the schema automatically creates a database extent for the persistent event class if one does not already exist.

The information from the analysis may be all that XEP needs to import a simple schema. For more complex structures, you can supply additional information that allows XEP to generate indexes and override the default rules for importing fields.

XEP contains three main classes that provide most of the important functionality:

- The EventPersister class is the main entry point for XEP. It provides methods to establish and control a TCP/IP database connection, import a schema and control schema generation, and create instances of the Event class.
- Instances of Event encapsulate a schema. The schema information is used by methods that store, update, or delete events, control index updating, and create instances of the EventQuery<targetclass> class (where targetclass is the Java class used to generate the current schema).
- Instances of EventQuery<targetclass> are used to query the extent of the persistent class corresponding to targetclass. Methods of the instance are used to define and execute the SQL query. The instance also acts as a container for the query resultset, and provides an iterator for it.

The following section provides a concrete example of how these classes work together.

3 Try It! Hands-on with XEP

We've developed a demo that shows you how to work with XEP and InterSystems IRIS.

You can download and inspect the code and run it for yourself.

For XEP requirements and setup, see "Installation and Configuration" in *Persisting Java Objects with InterSystems IRIS XEP*.

3.1 Overview of the XepSimple Program

The XepSimple demo program runs three simple methods, each of which is designed to demonstrate one of the three main XEP classes, EventPersister, Event, and EventQuery. These methods perform the following tasks:

- **runPersisterTasks()** creates an EventPersister object, uses it to establish a TCP/IP connection to the database, and imports a schema from Java sample class xep.samples.SingleStringSample. The EventPersister object is returned so it can be used by the next method.
- **runEventTasks()** uses the EventPersister object to create an Event object, and calls a class method of SingleStringSample to generate some instances of the class. Each instance is modified, and a method of the Event object is used to persist it in the database. When the samples have been processed, the Event object is returned so it can be used by the next method.
- **runEventQueryTasks()** closes the Event object after using it to create an EventQuery object. EventQuery methods are used to prepare and execute an SQL query, and to create an iterator for the returned resultset. The iterator allows each sample object in the resultset to be retrieved, updated, and saved.

After the last method returns, the connection to the database is terminated by closing the EventPersister object.

class XepSimple

```
package xepsimple;
import com.intersys.xep.*;
import xep.samples.SingleStringSample;
```

```
public class XepSimple {
  public static String targetName = "xep.samples.SingleStringSample";
  public static void main(String[] args) throws Exception {
    try {
      EventPersister persisterObj = runPersisterTasks();
      Event eventObj = runEventTasks(persisterObj);
      runEventQueryTasks(eventObj);
      persisterObj.close();
    } catch (XEPException e) {
      System.out.println("main() failed:\n" + e);
    }
  } // end main()

// See later sections of this tutorial for full listings of these methods:
//  public static EventPersister runPersisterTasks() { ... }
//  public static Event runEventTasks(EventPersister xepPersister) { ... }
//  public static void runEventQueryTasks(Event xepEvent) { ... }

} // end class XepSimple
```

3.2 runPersisterTasks(): Using the Persister Class

The **runPersisterTasks()** method creates an EventPersister object and establishes a TCP/IP connection to the database. It then deletes any old sample database entries and imports a schema for the xep.samples.SingleStringSample class (see "The SingleStringSample Class" at the end of this chapter for a listing of SingleStringSample).

Create an EventPersister, connect to the database, and import a schema

EventPersister extends CacheConnection (the InterSystems implementation of JDBC Connection), which allows it to use the same physical connection and transaction context as JDBC calls. The EventPersister.**connect()** method takes arguments for host, port, namespace, username, and password (just like a standard JDBC connection), and establishes a TCP/IP connection to the specified InterSystems IRIS namespace.

```
public static EventPersister runPersisterTasks() {
  EventPersister newPersister = null;
  try {
    newPersister = PersisterFactory.createPersister();
                   // (host,  port,  namespace,  user,  password)
    newPersister.connect("127.0.0.1",1972,"User","_SYSTEM","SYS");
  } catch (XEPException e) {
    System.out.println("connectPersister() failed:\n" + e);
  }
```

We call **deleteExtent()** to delete any sample objects from previous runs of XepSimple. XEP provides two different schema import models: flat schema and full schema. In this example, the **importSchema()** method creates a flat schema for the SingleStringSample class. Importing the schema automatically creates a database extent for the SingleStringSample persistent event class if one does not already exist.

```
  try {
                        // targetName = "xep.samples.SingleStringSample"
    newPersister.deleteExtent(targetName);  // remove old sample objects
    newPersister.importSchema(targetName);  // import flat schema
  } catch (XEPException e) {
    System.out.println("import failed:\n" + e);
  }

  return newPersister;
} // end runPersisterTasks()
```

The main difference between the flat schema and full schema models is the way in which Java objects are projected to persistent events. A flat schema is the optimal choice if performance is essential and the event schema is fairly simple. A full schema offers a richer set of features for more complex schemas, but may have an impact on performance.

After the schema has been imported, **runPersisterTasks()** returns the EventPersister object so that the next method, **runEventTasks()**, can use it to create an instance of Event.

3.3 runEventTasks(): Using the Event Class

The **runEventTasks()** method creates an instance of Event, calls a class method of SingleStringSample to generate an array of sample data objects, and then uses an Event method to persist each sample object to the database.

Create an instance of Event; generate, modify, and store sample data

getEvent() is the last EventPersister method that will be called, but the *xepPersister* object also contains our database connection, so we won't close it until we want to disconnect.

```
  public static Event runEventTasks(EventPersister xepPersister) {
//                      targetName = "xep.samples.SingleStringSample"
    Event sampleEvent = xepPersister.getEvent(targetName);
```

In this loop, we generate some instances of SingleStringSample, modify the name property of each sample object, and use the Event.**store()** method to persist each object to the database.

```
//  Call a class method to generate an array of 12 SingleStringSample objects
    SingleStringSample[] sampleArray = SingleStringSample.generateSampleData(12);
    int count = 0;
    try {
      for (int i=0; i<12; i++) {
        count++;
        SingleStringSample sample = sampleArray[i];
        sample.name = "SingleStringSample object " + count;
        sampleEvent.store(sample);
        System.out.println("initialized sample \"" + sample.name + "\"");
      }
    } catch (XEPException e) {
      System.out.println("runEventTasks() failed:\n" + e);
    }
    System.out.println("Stored " + count + " SingleStringSample objects.");

    return sampleEvent;
  } // end runEventTasks()
```

When the loop finishes, we return the Event object so that the next method, **runEventQueryTasks()**, can use it to create an instance of EventQuery. The output from **runEventTasks()** will look similar to this:

```
initialized sample "SingleStringSample object 1"
initialized sample "SingleStringSample object 2"
initialized sample "SingleStringSample object 3"
. . .
initialized sample "SingleStringSample object 10"
initialized sample "SingleStringSample object 11"
initialized sample "SingleStringSample object 12"
Stored 12 SingleStringSample objects.
```

3.4 runEventQueryTasks(): Using the EventQuery Class

The **runEventQueryTasks()** method creates an EventQuery object and uses it to prepare and execute an SQL query. It then creates a resultset iterator and uses it to retrieve, update, and save each sample object in the resultset.

Create an EventQuery instance, execute the query, and iterate through the resultset

The *sqlQuery* query string is passed to **createQuery()**, which creates our EventQuery object. We no longer need the Event object, so we close it. *sqlQuery* includes two variables (the question marks in `"BETWEEN ? AND ?"`), and the **setParameter()** method is used to assign values to them.

```
  public static void runEventQueryTasks(Event xepEvent) {
    EventQuery<SingleStringSample> xepQuery = null;
    try {
      String sqlQuery = "SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN ? AND
?";
      xepQuery = xepEvent.createQuery(sqlQuery);
      xepEvent.close();

      xepQuery.setParameter(1,3);  // assign value 3 to first SQL parameter
      xepQuery.setParameter(2,12);  // assign value 12 to second SQL parameter
      xepQuery.execute();   // get resultset for IDs between 3 and 12
    } catch (XEPException e) {
      System.out.println("query execution failed:\n" + e);
    }
```

The call to **execute()** returns the query resultset, which is stored in *xepQuery*, and **getIterator()** returns a resultset iterator. In the loop, we get each sample object from the resultset and print the current *name* property. We then change *name* before saving the sample object back to the database with **set()**.

```
    try {
      EventQueryIterator<SingleStringSample> iter = xepQuery.getIterator();
      while (iter.hasNext()) {
        SingleStringSample newSample = iter.next();
        System.out.println("Retrieved \"" + newSample.name + "\"");
        newSample.name = "updated " + newSample.name;
        System.out.println("   and changed name to \"" + newSample.name + "\"");
       iter.set(newSample);
      }
    } catch (XEPException e) {
      System.out.println("query execution failed:\n" + e);
    }

    System.out.println("Successfully updated objects in query resultset!\n");
  } // end runEventQueryTasks()
```

The output will look similar to this:

```
Successfully executed SQL query:
SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN ? AND ?
Retrieved "SingleStringSample object 3"
   and changed name to "updated SingleStringSample object 3"
Retrieved "SingleStringSample object 4"
   and changed name to "updated SingleStringSample object 4"
 . . .
Retrieved "SingleStringSample object 11"
   and changed name to "updated SingleStringSample object 11"
Retrieved "SingleStringSample object 12"
   and changed name to "updated SingleStringSample object 12"
```

3.5 The SingleStringSample Class

In case you're curious, here's a listing of our xep.samples.SingleStringSample sample class:

```
public class SingleStringSample {
  public  String name;
  public SingleStringSample() {}
  SingleStringSample(String str) {
      name = str;
  }

  public static SingleStringSample[] generateSampleData(int objectCount) {
      SingleStringSample[] data = new SingleStringSample[objectCount];
      for (int i=0;i<objectCount;i++) {
          data[i] = new SingleStringSample("single string test");
      }
      return data;
  }
}
```

3.6 Next Steps

We've kept this exploration simple to give you a taste of XEP without bogging you down in details. For example, we persist extremely simple Java objects that don't require annotation or other mechanisms to aid in schema generation and optimization.

So be sure not to confuse this exploration of XEP with the real thing! When you bring XEP to your production systems, you will need to understand the full range of tools offered by the API.

See *Persisting Java Objects with InterSystems IRIS XEP* for a comprehensive description of the tools provided by the XEP Java persistence API. It should give you give you a good idea of how to use XEP in production. The sources listed at the end of this document describe other facets of InterSystems IRIS Java support.

4 For More Information about XEP and Object Persistence

For more information on Java object persistence and other InterSystems Java interoperability technologies, see:

- The chapter on "Using the InterSystems IRIS Hibernate Dialect" in *Persisting Java Objects with InterSystems IRIS XEP* describes the InterSystems IRIS dialect of Hibernate. This dialect implements support for Java Persistence Architecture (JPA), which is the recommended persistence technology for complex object hierarchies in Java projects.
- *First Look: JDBC and InterSystems IRIS* — provides an introduction to connecting to InterSystems IRIS via JDBC: quick facts, a special feature, and a chance to try it for yourself. This is the simplest starting point for becoming familiar with InterSystems IRIS Java support.
- InterSystems IRIS provides Java APIs for easy database access via SQL tables and objects. See the following books for detailed information on each type of access:
  - See *Using JDBC with InterSystems IRIS* for SQL table access. The InterSystems JDBC driver allows InterSystems IRIS to establish JDBC connections to external applications, and provides access to external data sources via SQL.
  - See *Persisting Java Objects with InterSystems IRIS XEP* for object access. XEP is optimized for transaction processing applications that work with simple to medium complexity object hierarchies and require extremely high speed object data persistence and retrieval.
- *Using the InterSystems Spark Connector* describes how to use the InterSystems IRIS implementation of the Apache Spark Data Source API. The Spark Connector allows Apache Spark big data analytics applications to take full advantage of InterSystems IRIS clustered server technologies.