Getting Started with GuttenBase

GuttenBase is a database copying and migration framework. The framework provides a new approach to copy (migrate) data between different database systems: GuttenBase is used to *program* sophisticated data migrations instead of just configuring existing yet limited tools. This approach gives programmers full control over the migration process (with much more flexibility when performing data migrations).

This small tutorial explains how to use GuttenBase in order to migrate a PostgreSQL database to MySQL.

Currently supported Databases:

- Microsoft SQL Server
- MS-Access
- MySQL
- PostgreSQL
- HSQLDB
- H2 Derby
- Oracle
- DB2
- Sybase
- (Vanilla database)

Since GuttenBase is a framework, it may easily be extended for your needs.

Step-by-step setup guide

Step 1 - Add GuttenBase dependency in Maven

To start using framework GuttenBase in Maven project, just add the following dependency to your pom.xml:

```
<dependency>
    <groupId>de.akquinet.jbosscc.guttenbase</groupId>
    <artifactId>GuttenBase</artifactId>
    <version>2.0.0</version>
</dependency>
```

Additionally, you will have to add dependencies of the JDBC drivers you intend to use. For example, MySQL and PostgreSQL:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.37</version>
</dependency>

<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-902.jdbc4</version>
</dependency>
```

Step 2 - Create Connections to the source and target Database servers

You will have to create so called connection infos first that describe your source and target database, respectively. This are just simple

wrappers for the JDBC connection parameters.

```
public class MySQLConnectionsInfo extends URLConnectorInfoImpl {
 public MySQLConnectionsInfo() {
  super("jdbc:mysql://localhost:3306/testdb", "user", "password",
"com.mysql.jdbc.Driver", "aev", DatabaseType.MYSQL);
 }
}
```

Then you can start to configure the GuttenBase repository right away.

```
public static final String SOURCE = "source";
public static final String TARGET = "target";
...
final ConnectorRepository connectorRepository = new
ConnectorRepositoryImpl();
connectorRepository.addConnectionInfo(SOURCE, new
PostgreSQLConnectionInfo());
connectorRepository.addConnectionInfo(TARGET, new
MySQLConnectionsInfo());
```

Step 3 - Copy data from source database to target database

In our scenario we want to make a 1:1 copy of our database (to an empty target database). In unsophisticated cases this can be as simple as follows:

1. Copy the schema from target to source. The tool gives its best to respect all differences in data type representation between different database. Yet in practice you will often have to give some ***hints*** to accomplish its work. More about hints later.
2. Check the schemas for compatibilty. This will look for equal table names, columns, data types, etc. and may also produce some warnings.
3. Run the copy tool.
4. Check empirically if the data has been correctly transfered.

```
new CopySchemaTool(connectorRepository).copySchema(SOURCE, TARGET);
final SchemaCompatibilityIssues schemaCompatibilityIssues = new
SchemaComparatorTool(connectorRepository).check(SOURCE, TARGET);
if (schemaCompatibilityIssues.isSevere()) {
    throw new SQLException(schemaCompatibilityIssues.toString());
}
new DefaultTableCopyTool(connectorRepository).copyTables(SOURCE,
TARGET);
new CheckEqualTableDataTool(connectorRepository).checkTableData(SOURCE,
TARGET);
```

In many cases, that's it!

Hints

Often migrating a database requires a custom solution, for example you may need to:

- rename tables/columns in the target database
- change column types
- split a column into multiple columns
- filter tables and columns...

In this case you may add configuration "***hints***" that influence the tools used in the code example above. Technically, there exists a default implementation of every hint that is added automatically when you add a connection info.

This default version may be overridden by your implementation subsequently.

Hints overview

Name	Description
`ColumnDataMapperProviderHint`	Used to find mappings for column data.
`ColumnMapperHint`	Used to rename the column name in target database. You may also select multiple target columns for a given source column.
`ColumnOrderHint`	Used to determine the order of columns in a SELECT statement. This will of course influence the ordering of the resulting INSERT statement, too.
`ColumnTypeMapperHint`	Used to map data types between source and target databases.
`ColumnTypeResolverListHint`	Used to determine mapping strategies between different column types.
`DatabaseColumnFilterHint`	Used to filter columns when initially querying the database.
`DatabaseTableFilterHint`	Used to filter tables when initially querying the database.
`EntityTableCheckerHint`	Used to look for entity classes, i.e. classes that have primary key
`ExportDumpExtraInformationHint`	Used to add extra information to the dumped data when exporting data to a JAR/ZIP file.
`ExporterFactoryHint`	Used to export schema information and table data to some custom format.
`ImportDumpExtraInformationHint`	Used to retrieve extra information from the dumped data when exporting data to JAR/ZIP file.
`ImporterFactoryHint`	Used to import schema information and table data from some custom format.
`MaxNumberOfDataItemsHint`	Used to determine maximum number of data items in an INSERT statement. (Technically: How many place holders ('?') does the database support in a JDBC statement)

NumberOfCheckedTableDataHint	Used to determine how many rows of the copied tables shall be regarded when checking that data has been transferred correctly. By default set to the value 100.
NumberOfRowsPerBatchHint	Used to determine the number of VALUES clauses in INSERT statement or statements in batch update.
RefreshTargetConnectionHint	Used to refresh the database connection periodically to avoid OutOfMemory errors.
RepositoryColumnFilterHint	Used to filter columns when querying connector repository.
RepositoryTableFilterHint	Used to filter tables when querying connector repository.
ResultSetParametersHint	Used to set fetch size, result set type and concurrency for JDBC result sets.
ScriptExecutorProgressIndicatorHint	Used to switch the logging implementation. Implementation of progress indicator may be a simple logger or fancy UI.
SelectWhereClauseHint	Used to configure the SELECT statement created to read data from source tables with a WHERE clause.
SplitColumnHint	Used to configure the column to be used for splitting data into chunks.
TableCopyProgressIndicatorHint	Used to log implementation. Implementation of progress indicator may be a simple logger or fancy UI.
TableMapperHint	Used to rename the table name in target database.
TableOrderHint	Used to determine order of tables during copying/comparison.
ZipExporterClassResourcesHint	Used to add custom classes to the generated JAR and configure the META-INF/MANIFEST.MF Main-Class entry.

There is always a default implementation added to a connector by the repository which may be overridden subsequently.

Hint examples

Below you find the practical examples of how to implement the hints. You should also take a look at the JUnit tests in the source distribution. They will provide a lot of information how to work with GuttenBase.

ColumnDataMapperProviderHint

Used for automatic mapping between database specific types. Each database system has its own list of custom data types Different database systems support different data types, that's why you often need to use the data type mapping between source and target database in migration process. The following examples show how to use the hint. First, you need to specify the default/additional mapping in the

method addMapping(). Then, you have to register the data mapper. The rest will be done automatically.

```
_connectorRepository.addConnectorHint(TARGET, new
DefaultColumnDataMapperProviderHint() {
 @Override
 protected void addMappings(final DefaultColumnDataMapperProvider
columnDataMapperFactory) {
  super.addMappings(columnDataMapperFactory);
  columnDataMapperFactory.addMapping(ColumnType.CLASS_LONG,
ColumnType.CLASS_STRING, ...);
  columnDataMapperFactory.addMapping(ColumnType.CLASS_BIGDECIMAL,
ColumnType.CLASS_STRING, ...);
 }
});
```

ColumnMapperHint

Used to override the column name in target database. During the migration process, you may want to rename the columns. In this case you should use the ColumnMapperHint. The following example shows how to use the hint and how to rename the columns. First, you need to register the column mapper.

```
connectorRepository.addConnectorHint(TARGET, new ColumnMapperHint() {
 @Override
 public ColumnMapper getValue() {
        return new CustomColumnRenameName()
                .addReplacement("name", "id_name")
                .addReplacement("username", "id_username");
    }
}
```

ColumnOrderHint

Determine the order of colums. Before you start the migration process, you can customize the order or sort columns in database. The following (unrealistic) example show how to sort the columns by the column name's has code.

```
connectorRepository.addConnectorHint(SOURCE, new ColumnOrderHint() {
 @Override
 public ColumnOrderComparatorFactory getValue() {
   return () -> Comparator.comparingInt(Object::hashCode);
   }
 }
```

ColumnTypeMapperHint

Different databases use different data types. That's why the column data types are not compatible sometimes. In this case, use ColumnTypeMapperHint.

```
_connectorRepository.addConnectorHint(TARGET, new ColumnTypeMapperHint()
{
  @Override
  public ColumnTypeMapper getValue() {
    return new DefaultColumnTypeMapper()
      .addMapping(DatabaseType.GENERIC, DatabaseType.GENERIC, "BIGINT",
"INTEGER");
  }
});
```

ColumnTypeResolverListHint

Determine the mapping strategies. To specify multiple mapping strategies for different column types, use ColumnTypeResolverListHint. This hint allows you to perform some heuristic checks on given column type. The example will check column type name and determine what Java type is appropriate in this case. For example: if the declared type of the column contains any of the strings "CHAR" or "TEXT" then the column type is identified as a string.

```
connectorRepository.addConnectorHint(SOURCE, new
ColumnTypeResolverListHint() {
  @Override
 public ColumnTypeResolverList getValue() {
    return () -> Arrays.asList(new HeuristicColumnTypeResolver(), new
ClassNameColumnTypeResolver());
  }
}
```

DatabaseColumnFilterHint

You can filter the columns in your database and define which columns should be migrated/copied to new database. The following example shows how to use the hint and how to filter columns: the hint selects all the columns from the table "foo_user" except the one called "password". Only that data will be copied to target database.

```
connectorRepository.addConnectorHint(SOURCE, new
DatabaseColumnFilterHint() {
 @Override
 public DatabaseColumnFilter getValue() {
    return columnMetaData ->
!columnMetaData.getTableMetaData().getTableName().equals("FOO_USER")
  || !columnMetaData.getColumnName().equals("PASSWORD");
  }
}
```

DatabaseTableFilterHint

You can filter the tables in your database and define which tables would be migrated/copied to new database. In this example, the hint selects all tables with a name containing String "USER". Only these tables will be copied to target database.

```
connectorRepository.addConnectorHint(SOURCE, new
DatabaseTableFilterHint() {
 @Override
 public DatabaseTableFilter getValue(){
    return new DefaultDatabaseTableFilter() {
      @Override
   public boolean accept(final TableMetaData table) throws SQLException
{
        return table.getTableName().toUpperCase().contains("USER");
      }
    };
  }
}
```

EntityTableCheckerHint

By default this hint checks if the given table has a primary key column named "ID". This information may be usefuly during migration.

```
connectorRepository.addConnectorHint(SOURCE, new
EntityTableCheckerHint() {
  @Override
 public EntityTableChecker getValue() {
    return tableMetaData -> {
      for (final ColumnMetaData columnMetaData :
tableMetaData.getColumnMetaData()) {
        final String columnName =
columnMetaData.getColumnName().toUpperCase();
        if (columnMetaData.isPrimaryKey() && (columnName.equals("ID") ||
columnName.equals("IDENT"))) {
          return true;
        }
      }
      return false;
    };
  }
}
```

ExportDumpExtraInformationHint

As a special feature GuttenBase allows to export data into a ZIP/JAR file. The ExportDumpExtraInformationHint gives you the possibility to add additional custom information to the dumped data. In this example, you can add the information (as key value pair objects) to the dumped data in HashMap form.

```
public static final String KEY = "SEQUENCE_NUMBER";
public static final long VALUE = 4711L;
...
connectorRepository.addConnectorHint(EXPORT, new
ExportDumpExtraInformationHint(){
 @Override
 public ExportDumpExtraInformation getValue() {
    return (connectorRepository, connectorId, exportDumpConnectionInfo)
-> {
      final Map<String, Serializable> result = new HashMap<>();
      result.put(KEY, VALUE);
      return result;
    };
  }
```

ExporterFactoryHint

You can export a database, namely schema information and table data to some custom format. In this example, the schema information and data are being exported to gzipped file with serialized data. You can customize the hint to your preferences.

```
connectorRepository.addConnectorHint(EXPORT, new ExporterFactoryHint() {
 @Override
 public ExporterFactory getValue() {
  return PlainGzipExporter::new;
 }
}
```

ImportDumpExtraInformationHint

Of course you may also import the data from a ZIP file into some real data base.The ImportDumpExtraInformationHint gives you a possibility to retrieve additional information from the dumped data. In this example, we store the extra information from the dumped data to a field.

```
private Map<String, Serializable> _extraInformation;
connectorRepository.addConnectorHint(IMPORT, new
ImportDumpExtraInformationHint() {
 @Override
 public ImportDumpExtraInformation getValue() {
    return extraInformation -> _extraInformation = extraInformation;
  }
}
```

ImporterFactoryHint

You can import a database from a file, particularly schema information and table data from some custom format.

```
connectorRepository.addConnectorHint(IMPORT, new ImporterFactoryHint() {
 @Override
 public ImporterFactory getValue() {
  return PlainGzipImporter::new;
 }
}
```

MaxNumberOfDataItemsHint

During the migration process you need to specify the maximum number of data items that can be inserted via a single INSERT statement.

```
connectorRepository.addConnectorHint(SOURCE, new
MaxNumberOfDataItemsHint() {
  @Override
 public MaxNumberOfDataItems getValue() {
    return targetTableMetaData -> 30000;
  }
}
```

NumberOfCheckedTableDataHint

This hint is used by the CheckEqualTableDataTool to specify how many rows of the copied tables shall be regarded to ensure that the data has been transferred correctly.

```
connectorRepository.addConnectorHint(SOURCE, new
NumberOfCheckedTableDataHint() {
  @Override
 public NumberOfCheckedTableData getValue() {
    return () -> 100;
  }
}
```

NumberOfRowsPerBatchHint

You may specify how many rows will be inserted in single transaction. This is an important performance issue. The value also must not be too high so data buffers are not exceeded. In this example, the value of rows per batch is set to 1000. Using multiple VALUES clauses in a single statement is much faster than using separate single-row insert statements, but not all databases support this.

```
connectorRepository.addConnectorHint(TARGET, new
NumberOfRowsPerBatchHint() {
 @Override
 public NumberOfRowsPerBatch getValue() {
   return new NumberOfRowsPerBatch() {
     @Override
     public int getNumberOfRowsPerBatch(TableMetaData
targetTableMetaData) {
        return 1000;
    }
     @Override
   public boolean useMultipleValuesClauses(TableMetaData
targetTableMetaData) {
        return false;
     }
   };
  }
}
```

RefreshTargetConnectionHint

Some JDBC drivers seem to accumulate data over time, even after a connection is commited() and all statements, result sets, etc. are closed. This will cause an OutOfMemoryError eventually. To avoid this the connection can be closed and re-established periodically using RefreshTargetConnectionHint.

```
connectorRepository.addConnectorHint(CONNECTOR_TARGET, new
RefreshTargetConnectionHint() {
 @Override
 public RefreshTargetConnection getValue() {
    return (noCopiedTables, sourceTableMetaData) -> noCopiedTables % 2
== 0;
  }
}
```

RepositoryColumnFilterHint

You can filter the columns in your database, when inquiring connector repository. In this example, the hint selects all of the columns, except one column with a name "password".

```
connectorRepository.addConnectorHint(SOURCE, new
RepositoryColumnFilterHint() {
 @Override
 public RepositoryColumnFilter getValue(){
    return column ->
!column.getColumnName().equalsIgnoreCase("password");
  }
}
```

RepositoryTableFilterHint

You can filter the tables in your database, when inquiring connector repository. In this example, the hint selects a table which has a name containing String "USER". Only these tables will be copied to target database.

```
connectorRepository.addConnectorHint(SOURCE, new
RepositoryTableFilterHint() {
 @Override
 public RepositoryTableFilter getValue() {
  return table -> table.getTableName().toUpperCase().contains("USER");
 }
}
```

ResultSetParametersHint

Using this hint you may set some parameter, namely the fetch size, result set type and concurrency tye for result set. By default the fetch size is set to the value 2000, the result set type is ResultSet.TYPE_FORWARD_ONLY and concurrency type is ResultSet.CONCUR_READ_ONLY.

```
connectorRepository.addConnectorHint(SOURCE, new
ResultSetParametersHint() {
  @Override
 public ResultSetParameters getValue() {
    return new
DefaultResultSetParametersHint.DefaultResultSetParameters();
  }
}
```

ScriptExecutorProgressIndicatorHint

You can log the migration process, particularly start, end, execution of a SQL statement, etc. In this example, the simple implementation will just log to console.

```
connectorRepository.addConnectorHint(SOURCE, new
ScriptExecutorProgressIndicatorHint() {
  @Override
 public ScriptExecutorProgressIndicator getValue() {
    return new LoggingScriptExecutorProgressIndicator();
  }
}
```

SelectWhereClauseHint

Using this hint you may configure the WHERE clause of the SELECT statement used to read data from the source table, e.g. to apply custom filters.

```
connectorRepository.addConnectorHint(SOURCE, new SelectWhereClauseHint()
{
 @Override
 public SelectWhereClause getValue() {
    return tableMetaData -> {
      switch (tableMetaData.getTableName()) {
        case "FOO_USER":
          return "WHERE ID <= 3";
        case "FOO_USER_COMPANY":
        case "FOO_USER_ROLES":
          return "WHERE USER_ID <= 3";
        default:
          return "";
      }
    };
  }
}
```

SplitColumnHint

Sometimes the amount of data read by the SELECT statement exceeds any buffer. In these cases we need to split the data by some given numeric unique value, usually the primary key.
I.e., the data is read in chunks where these chunks are split using the ID column.

```
connectorRepository.addConnectorHint(SOURCE, new SplitColumnHint() {
 @Override
 public SplitColumn getValue() {
    return new DefaultSplitColumn();
  }
}
```

TableCopyProgressIndicatorHint

You can log the migration process, particularly copying of tables In this example, the simple implementation will just log to console.

```
connectorRepository.addConnectorHint(SOURCE, new
TableCopyProgressIndicatorHint() {
  @Override
 public TableCopyProgressIndicator getValue() {
    return new LoggingTableCopyProgressIndicator();
  }
}
```

TableMapperHint

Configure the table name in target database. During the migration process, you may want to rename the tables. The following example shows how to use the hint and how to rename the tables. First, you need to register the table mapper. Then you can rename the table by name: change "offices" to "tab_offices", "orders" to "tab_orders".

```
connectorRepository.addConnectorHint(TARGET, new TableMapperHint() {
 @Override
 public TableMapper getValue() {
        return new CustomTableRenameName()
                .addReplacement("offices", "tab_offices")
                .addReplacement("orders", "tab_orders");
    }
}
```

TableOrderHint

You can customize the order of tables during migration. By default the comparor would order a collection of tables by their natural ordering of table names.

```
connectorRepository.addConnectorHint(SOURCE, new TableOrderHint() {
  @Override
 public TableOrderComparatorFactory getValue() {
    return () -> Comparator.comparingInt(System::identityHashCode);
  }
}
```

ZipExporterClassResourcesHint

You can export a database to a special ZIP/JAR file. This hint allows you to create a self-contained executable JAR that will startup with a Main class customizable by the framework user. If necessary you can add custom classes and resources to the resulting JAR. You can also specify a startup class for the resulting JAR File. The following example shows how to register and use the hint.

```
connectorRepository.addConnectorHint(EXPORT, new
ZipExporterClassResourcesHint() {
 @Override
 public ZipExporterClassResources getValue() {
    return new DefaultZipExporterClassResources() {
      @Override
    public Class<?> getStartupClass() {
        return MyStartup.class;
      }

   @Override
    public Map<String, URL> getUrlResources() {
        final Map<String, URL> result = new HashMap<>();
        result.put("GIF",
this.getClass().getResource("/data/test.gif"));
        return result;
      }
    };
  }
}
```