

FlywayDB Extension

Table of Contents

What happens during bootstrapping	2
How to Configure	3
Classpath	3
Configuration Properties	3
How to Use	5
Naming convention for scripts	5
Disabling for development	5
Idempotent constraints	6
Managing views etc	6
(Manual) baselining an existing database	7
More advanced use cases	8
(Automatic) baselining an existing database	8
Handling validation errors	8
Out-of-order migrations	10
Java-based migrations	10
Database vendor-specific scripts	11
A process for creating migration scripts	12
Using Liquibase to diff databases	13
Known issues	14
Dependencies	15

This module (`isis-module-flywaydb`) provides an implementation of the `DataNucleus PersistenceManagerFactory` class. This uses `Flyway` API to automatically apply any database migration scripts prior to the bootstrapping of the rest of the Apache Isis runtime.



The integration provided by this module runs a single instance of Flyway for the entire database, even if that database has multiple schemas (as is common in Apache Isis applications). In other words (although Flyway as a tool supports it), this integration does *not* allow for the tables in different schemas to have their own separate migration history.

What happens during bootstrapping

When the application bootstraps, it uses the run two DB migration files that reside in `db.migration` package of the webapp module:

- `V1__schema-simple.sql`

creates the `simple` schema

- `V2__table-simple_SimpleObject.sql`

creates the `simple.SimpleObject` table to hold the `SimpleObject` entity.

As a side-effect of running the migration, the flyway DB schema management tables are created.



While (for simplicity) the demo application uses files named `V1__`, `V2__` and so on, we recommend you use a more sophisticated naming convention based on dates. See [below](#) for further discussion.

All of this can be confirmed by invoking `Prototyping > HSQL DB Manager` to spawn off a Swing UI client that can view the (in-memory) database:

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
10		<< Flyway Schema Creation >>	SCHEMA	"flyway"		SA	2016-12-06 20:44:08.161	0	<input checked="" type="checkbox"/>
21	1	schema simple	SQL	V1__schema_simple.sql	1221918245	SA	2016-12-06 20:44:08.173	1	<input checked="" type="checkbox"/>
32	2	table simple-SimpleObject	SQL	V2__table_simple-SimpleObject.sql	-2131873747	SA	2016-12-06 20:44:08.178	1	<input checked="" type="checkbox"/>

This shows the migration scripts have been run, and the Flyway schema tables created and populated.

How to Configure

Classpath

Update your classpath by adding this dependency in your project's **webapp** module's **pom.xml**:

```
<dependency>
  <groupId>org.isisaddons.module.flywaydb</groupId>
  <artifactId>isis-module-flywaydb-dom</artifactId>
  <version>1.15.0</version>
</dependency>
```

Check for later releases by searching [Maven Central Repo](<http://search.maven.org/#search|ga|1|isis-module-flywaydb-dom>).

For instructions on how to use the latest **-SNAPSHOT**, see the [contributors guide](#).

Configuration Properties

Set the following configuration properties:

```
isis.persistor.datanucleus.impl.javax.jdo.PersistenceManagerFactoryClass=\
org.isisaddons.module.flywaydb.dom.FlywayJdoPersistenceManagerFactory
isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll=false
isis.persistor.datanucleus.impl.datanucleus.schema.validateAll=true
isis.persistor.datanucleus.impl.flyway.schemas=flyway
```

Disabling DataNucleus' **autoCreateAll** property in turn enables Flyway migrations.



Using **autoCreateTables=false** (instead of **autoCreateAll**) will also enable Flyway migrations, but will continue to allow DataNucleus to (re)create table constraints; see the section [below](#) for further discussion.

Enabling DataNucleus' **validateAll** property means you can be assured that your migration scripts have brought the database schema into the same shape as is expected by DataNucleus; if there is a mismatch then the app will fail to bootstrap.



See the section [below](#) for workarounds when validation errors are thrown that should not be.

The **flyway.schemas** property is intended to list all of the schemas that Flyway manages as a single indivisible set of tables; the **schema_version** table will be created in the first schema listed. If these schemas do not exist already then Flyway will automatically create them in the current database; however if any of them *do* exist then Flyway will not create any of the schemas.

If any of the tables that need to be managed happen to reside in a default schema (such as "dbo" for SQL Server, or "PUBLIC" for HSQLDB), then either:

- do *not* list that default schema within `flyway.schemas`, and also *do not* provide migration scripts to create the schemas themselves (relying on Flyway to create the schemas instead), or
- *do* list that default schema within `flyway.schemas`, and then *do* provide migration scripts to create the schemas themselves, or
- just list "flyway" as the one and only schema in `flyway.schemas`, and then *do* provide a migration scripts to create the schemas.

This will have the result of creating a new `flyway.schema_version` table.

Of these options, the demo application for this module goes with the last approach.



Flyway's `migrate` command supports a large number of other [configuration options](#). All of these can be enabled/set using a property name consisting of `isis.persistor.datanucleus.impl.` prefix along with the `flyway.xxx` property as a suffix.

How to Use

The sections below describe some of the common use cases when using Flyway to manage database migrations.

Naming convention for scripts

When Flyway runs it searches for files named `Vnnnnn__`, where `nnnn` is some number. In the Flyway documentation (and, for that matter, in the demo app for this module), those numbers start at 1. However, if your team uses [feature branches](#) then this is likely to cause conflicts when those feature branches are merged back into master/trunk.

Luckily, Flyway does not require the [version numbers](#) to be contiguous; moreover the number can contain '_' or '.' as a separator. Therefore, (along [with others](#)) we recommend that the number used is the current date/time, for example:

```
VyyyyMMdd.hhmm__description-of-the-change.sql
```

When the feature branches are merged, you (the developer) should check that any new migrations have a later timestamp than the version of the current production database; chances are they will be. But if necessary, the filename/timestamp can be updated, eg to be the current date/time that the merged is performed.

Alternatively, an "out-of-order" migration (as discussed [below](#)) could be used.

Disabling for development

When developing Apache Isis applications, it's common practice to prototype and run integration tests in memory, by default using HSQLDB. In production, however, some other database will most likely be used (PostgreSQL, MySQL, MS SQL Server etc).

Now Flyway - by design - does not attempt to abstract over different database vendors. In other words, the SQL migration scripts that are designed for production will quite possibly not work for development environment. The long and short of this is that you will most likely want to simply disable the flyway migration when running in-memory against HSQLDB.

This can be done simply by setting either the `autoCreateAll` or the `autoCreateTables` property back to `true`, either in the `persistor_datanucleus.properties` file, eg:

```
isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll=true
```

Or, (if using `mvn jetty:run` or `org.apache.isis.WebServer`) with a system property, eg:

```
mvn -pl webapp jetty:run  
-Disis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll=true
```



Flyway migrations are **not** disabled if only `autoCreateConstraints` property is enabled. This enables the use case for dropping all constraints prior to migration, and then having DataNucleus recreate them after. See [below](#) for further discussion.

Idempotent constraints

As part of the release process, some DBAs prefer to drop all database constraints, then recreate them at the end of the release. This helps ensure that what is deployed to the database is only what should be there.



Obviously this isn't feasible for very large databases; in such cases dropping only foreign key/alternate indices (but not primary keys) may be more appropriate.

To effect this:

- dropping constraints can be done using the "[before migrate](#)" [callback](#) from Flyway.

Create a script `beforeMigrate.sql` which will drop all these objects, place in `db.migration` package alongside any other migration scripts.

- recreating constraints can be done using DataNucleus' `autoCreateConstraints` property.

In `webapp` module's `persistor_datanucleus.properties` file, also set:

```
isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateConstraints=true
```

When the application is bootstrapped, Flyway will run the `beforeMigrate.sql` script to drop all constraints and other objects, and then DataNucleus will reinstate those constraints as it initializes.

Managing views etc

In addition to the tables that support an Apache Isis application, there may be additional artifacts such as views and stored procedures that also need to be deployed. For example, such views might be to support third-party reports or other similar tools.

Since such views may need to be updated whenever the underlying tables change, it makes sense to manage them as part of the codebase of the Apache Isis application.

On the other hand, since these are not part of the application, DataNucleus cannot be used to automatically create these artifacts. Instead, Flyway's [repeatable migrations](#) can be used to run the scripts.

A repeatable migration is simply a file with the prefix `R__`, residing in the `db.migration` package as usual; for example `R__reporting-views.sql`. Typically these scripts should drop all views and then recreate them; ie they should be idempotent.

Flyway will run these scripts whenever they file is changed (it maintains a checksum of the file).

(Manual) baselining an existing database

If you want to start using Flyway for with an existing database that is already in production, then it must be [baselined](#).

This involves Flyway creating its `schema_version` table, and inserting a row to represent the "current" version of that database. Thereafter only scripts with a number higher than that version will be applied.

As a minimum, baselining involves simply running the `baseline` command:

```
flyway -driver=... \  
-url=... \  
-user=... \  
-password=... \  
-baselineVersion="yyyyMMdd.hhmm" \  
-baselineDescription="Initial take-on" \  
baseline
```

where `yyyyMMdd.hhmm` can be the current date/time.



It's also possible to specify command-line options using a `flyway.conf` [configuration file](#).

If you wish, you could also generate scripts to represent the current state of the database. These won't be used by Flyway in the baselined system, but could be used to start the app against a completely empty database (if Flyway isn't otherwise [disabled](#)).

For example, scripts can be geneated for MS SQL Server using the [Generate and Publish Scripts](#) wizard (Tasks > Generate Scripts); save these as `VyyyyMMdd.hhmm__initial-take-on.sql`. This follows the date/time naming convention discussed [above](#).

More advanced use cases

And here are some slightly more advanced use cases to consider.

(Automatic) baselining an existing database

Rather than [manually baselining an existing \(production\) database](#), Flyway also supports automatic baselining. With this option enabled, if Flyway is run against a database with no `schema_version` table, then it will automatically create that table and populate it with a single baseline row.

This can be configured by updating the `webapp` module's `persistor_datanucleus.properties` file:

```
isis.persistor.datanucleus.impl.flyway.baselineOnMigrate=true
isis.persistor.datanucleus.impl.flyway.baselineVersion=1
```

Change the `flyway.baselineVersion` if you want some other value to be used as the baseline version.

Handling validation errors

Sometimes `validateAll` can result in DataNucleus throwing an exception even if the actual database matches the schema. The underlying reason for this occurring will vary; one reason is a buggy JDBC driver misreporting database metadata. It is however possible to workaround this issue.

By way of example, when running against MS SQL Server you may find that BLOB/CLOB columns are reported as being invalid. One common example is the `CommandJdo` entity (in the [command spi](#) module), with its `exception` and a `memento` properties. This is defined as:

```
public class CommandJdo {
    ...
    @javax.jdo.annotations.Column(allowsNull="true", jdbcType="CLOB")
    private String exception;
    ...
    @javax.jdo.annotations.Column(allowsNull="true", jdbcType="CLOB")
    private String memento;
    ...
}
```

In MS SQL Server this is mapped to a table with a column of type `TEXT`. However, this results in DataNucleus throwing an exception, to the effect that the datastore defines a `LONGVARCHAR`, while the (class) metadata defines a `CLOB`.

The workaround is to redefine the JDO metadata using an `.orm` file. For example, `CommandJdo` can be made to work by adding `CommandJdo-sqlserver.orm`:

```

<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://xmlns.jcp.org/xml/ns/jdo/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/orm
                          http://xmlns.jcp.org/xml/ns/jdo/orm_3_0.xsd">

  <package name="org.isisaddons.module.command.dom">
    <class name="CommandJdo"
           schema="isiscommand"
           table="Command">
      <property name="exception">
        <column name="exception" jdbc-type="CLOB" sql-type="LONGVARCHAR"
allows-null="true"/>
      </property>
      <field name="memento">
        <column name="memento" jdbc-type="CLOB" sql-type="LONGVARCHAR" allows-
null="true"/>
      </field>
    </class>
  </package>

</orm>

```

This should reside in the appropriate package (`org.isisaddons.module.command.dom` in this case).

Another example is the `DocumentAbstract` entity (in the `document subdomain` module), with its `blob_byte` and a `memento` properties.

```

public class DocumentAbstract {
    ...
    @javax.jdo.annotations.Column(allowsNull = "true", name = "blob_bytes", jdbcType =
"BLOB", sqlType = "BLOB")
    private byte[] blobBytes;
    ...
    @javax.jdo.annotations.Column(allowsNull = "true", name = "clob_chars", jdbcType =
"CLOB", sqlType = "CLOB")
    private String clobChars;
    ...
}

```

The fix in this case is the following `DocumentAbstract-sqlserver.orm` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://xmlns.jcp.org/xml/ns/jdo/orm"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/orm
                          http://xmlns.jcp.org/xml/ns/jdo/orm_3_0.xsd">

  <package name="org.incode.module.document.dom.impl.docs">
    <class name="DocumentAbstract"
          schema="incodeDocuments">
      <field name="blobBytes">
        <column name="blob_bytes" jdbc-type="BLOB" sql-type="LONGVARBINARY"
allows-null="true"/>
      </field>
      <field name="clobChars">
        <column name="clob_chars" jdbc-type="CLOB" sql-type="LONGVARCHAR"
allows-null="true"/>
      </field>
    </class>
  </package>
</orm>
```

The last thing to do is to instruct DataNucleus to also read these additional `.orm` files. This can be done using:

```
isis.persistor.datanucleus.impl.datanucleus.Mapping=sqlserver
```

where `sqlserver` matches the filename (`DocumentAbstract-sqlserver.orm` and so on).

Out-of-order migrations

Sometimes it is necessary to run "outOfOrder" migrations; that is, to run a migration whose number is less than that of the current production database.

To enable this feature, add the following to the `webapp` module's `persistor_datanucleus.properties` file:

```
isis.persistor.datanucleus.impl.flyway.outOfOrder=true
```

Java-based migrations

In addition to SQL-based migrations, Flyway also supports [migrations implemented in Java](#). These follow the same naming convention as SQL-based migrations, and also reside in the `db.migration` package. Of course, they must be compiled and reside on the classpath, very similar to Apache Isis fixture scripts.

Database vendor-specific scripts

In addition to SQL-based migrations, Flyway also supports [migrations implemented in Java](#). These follow the same naming convention as SQL-based migrations, and also reside in the `db.migration` package. Of course, they must be compiled and reside on the classpath, very similar to Apache Isis fixture scripts.

A process for creating migration scripts

Suppose you've developed a new feature which will require a database schema change; how should the migration scripts be created and tested? Here's one approach:

- obtain a backup of the current production database (which is already under Flyway's control; [baseline](#) it if not)

In fact, all that is required is the schema of this database. So, as a minor refinement, you could set up a CI pipeline that hooks onto your nightly database backups; this would restore the production database to some scratch DB, then truncate all tables, then creates a backup of that truncated database.



See [truncate-all-tables.sql](#) for a script that does this for MS SQL Server.

- in your development environment, restore the current production database (or truncated version) twice:
 - restore once to **current** DB
 - restore another to **test** DB
- create a completely blank **dev** DB

You could either create an empty database, or zap an existing scratch DB.



See [drop-all-tables.sql](#) for a script that does this for MS SQL Server.

- run app against this empty **dev** database, with **autoCreateAll=true**

This disables Flyway, causing DataNucleus to create the schema based on its current metadata

- next, use a comparison tool to compare **current** against **dev**.



One option is to use the command line tools provided by [liquibase](#) (itself a DB migration framework that "competes" with Flyway; here we just leverage its diff utility). See [below](#) for details of how to use liquibase's commandline tool.

- Save SQL scripts capturing the difference
- Finally, run app against **test** DB, this time with Flyway re-enabled and with DataNucleus validation re-enabled also:
 - **autoCreateAll=false** (or **autoCreateTables=false**) re-enables Flyway, causing it apply the migration scripts
 - **validateAll=true** causes DataNucleus to check that the resultant DB schema matches that required by the entity metadata.

If there is an issue then the app will fails to start; use the errors in the console to diagnose the issue and then go round the loop.

Using Liquibase to diff databases

[Liquibase](#) is another Java-based migration tool that "competes" with Flyway; its scope is rather broader than Flyway which some prefer. Here we just leverage its diff utility in order to help generate migration scripts.

The [delta.sh](#) shows how this can be done for a SQL Server database. It is invoked as follows:

```
PROD_URL="jdbc:sqlserver://localhost;instance=.;databaseName=current"
DEV_URL="jdbc:sqlserver://localhost;instance=.;databaseName=dev"
USERNAME="sa"
PASSWORD="pass"

sh delta.sh $PROD_URL $DEV_URL $USERNAME $PASSWORD
```

(Referring back to the process described [above](#)) this compares the current production database to the development database.

The `delta.sh` script uses a [schema.txt](#) file which lists all of the database schemas to compare. This should be the same list of schemas as configured in `persistor_datanucleus.properties` (the `flyway.schemas` property), described [above](#). Adjust as necessary.

Obviously, the above script requires that `liquibase` shell script is on your `$PATH` (or `liquibase.bat` on your `%PATH%`).

Known issues

None known at this time.

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/ext/flywaydb/impl -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns these compile/runtime dependencies:

```
org.flywaydb:flyway-core:jar:4.0.3
```

For further details on 3rd-party dependencies, see:

- [Flyway DB](#)