**Flyway – Database migrations made easy**

By [Technical Services](http://blog.novatec-gmbh.de/author/tecadmin/) on September 29, 2014 · Category: [Java](http://blog.novatec-gmbh.de/category/development-programming/programming/java/), [Programming](http://blog.novatec-gmbh.de/category/development-programming/programming/), [Spring](http://blog.novatec-gmbh.de/category/spring/) · [Add Comment](http://blog.novatec-gmbh.de/flyway-database-migrations-made-easy/#wp-comments)

This post is a ‘step-by-step’ tutorial how Flyway could be integrated into an existing application. For this purpose I’ve created a sample application which uses EclipseLink as a JPA-Provider and the Spring-Framework for Dependency injection (DI). Furthermore, the sample application uses Maven for dependency management and for building the application. This tutorial examines how a database migration can be executed at build time with Maven (e.g. as a part of a CI-build) and also at run time.

**Step 1: Integrate Flyway dependencies**

With Maven it’s very easy to manage third-party libraries. If a database migration with Java classes is required, then the ‘flyway-core’ dependency with the latest version must be included to the projects ‘pom’-file. Following snippet shows how I’ve added this dependency into the applications ‘pom’-file.



|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>  <groupId>org.flywaydb</groupId>  <artifactId>flyway-core</artifactId>  <version>3.0</version>  </dependency> |

To execute a migration at building process it’s also important to add the ‘flyway-maven-plugin’ to the ‘pom’-file. To establish a connection to the database, it’s required to configure the plugin. In section ‘configuration’ we define a host, a user and a password, which are the credentials to get access to the database. The property ‘location’ is required if the path to the repository with the migration files doesn’t match the default path ‘db/migrations’. The following snippet shows how I’ve added this Maven-Plugin into the ‘pom’-file and defined the credentials to get access to a DB2 database instance.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <plugin>  <groupId>org.flywaydb</groupId>  <artifactId>flyway-maven-plugin</artifactId>  <version>3.0</version>    <dependencies>  <dependency>  <groupId>com.ibm</groupId>  <artifactId>db2jcc</artifactId>  <version>4.0</version>  </dependency>  </dependencies>    <configuration>  <user>db2admin</user>  <password>db2admin</password>  <url>jdbc:db2://localhost:50000/FLYWEXAM</url>  <locations>  <location>filesystem:src/main/resources/migrations</location>  </locations>  <skip>false</skip>  </configuration>  </plugin> |

**Step 2: Base configuration**

Flyway updates the database from one version to the next version using migrations. Each migration must have a unique version identifier and a description. The migrations are then sorted based on their version number and applied in order if it’s necessary. The information’s about the applied migrations are stored in a ‘SCHEMA\_VERSION’ table by default. This table is used to track the state of the database. As each migration gets applied, the metadata table is updated accordingly.

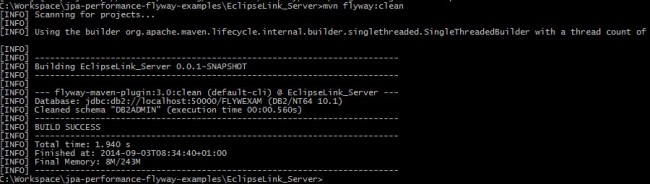
At this point of the tutorial, the ‘SCHEMA\_VERSION’ table doesn’t exist yet and it should be created first. If you start Flyway with an empty database then don’t worry. Flyway will try to locate the metadata table. If the database is empty, Flyway won’t find it and will create it instead. If you have an existing database that has not been filled by Flyway, this is the way you go.

First, create and save a snapshot or a migration script of your existing database that will recreate your current state of your database. Then, open up a shell and navigate to the sample project and execute following command:



|  |  |
| --- | --- |
| 1 | mvn flyway:clean |

This command deletes all tables and data in the database. This step guarantees that further migrations are running correctly. The following picture shows the results of the cleaning process.

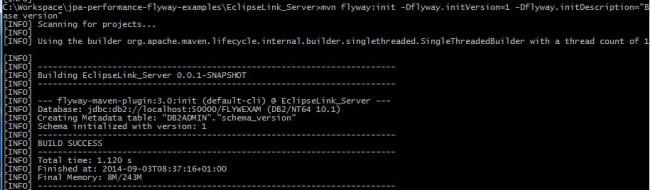
[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/2.jpg)

After the cleaning process, we can generate the ‘SCHEMA\_VERSION’ table. Use the following command to create the table:

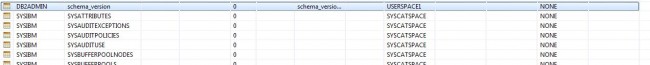


|  |  |
| --- | --- |
| 1 | mvn flyway:init -Dflyway.initVersion=<version> –Dflyway.initDescription=<description> |

The following screenshot shows this process.

[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/2_1.jpg)

As you can see in the screenshot below, the ‘SCHEMA\_VERSION’ table has been created. For this purpose, we initialized the table with version 1.

[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/3.jpg)

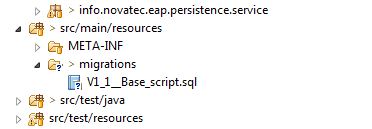
[SCHEMA_VERSION table after init](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/3_1.jpg)

After you created the ‘SCHEMA\_VERSION’ table, it’s now time to import the snapshot which was created before. For this step, the snapshot or migration script must be moved to the folder, where Flyway can find it. In this example the manually created directory ‘migrations’ is acting as a repository for Flyway. After you move the file to the repository, it’s important to customize the name of the migration file. Flyway only locates files which are valid with the Flyway-typical naming conventions. This requires a prefix (by default: V), a version number which must consists of one or more numeric parts and are separated with a dot (.) or an underscore (\_), two underscores (\_\_) as a delimiter and a description~~.~~The complete filename must look like:



|  |  |
| --- | --- |
| 1 | [prefix][version]\_\_[description].sql |

The following screenshot shows, how it’s customized for the sample application.

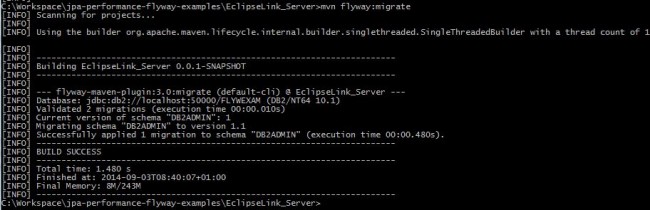
[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/5.jpg)

For the data recovery, you execute the following command:



|  |  |
| --- | --- |
| 1 | mvn flyway:migrate |

The next screenshot shows that the database has been restored. The version of the database has been changed from 1.0 to 1.1.

[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/6.jpg)

In addition, a new entry to the ‘SCHEMA\_VERSION’ table has been inserted. This is the evidence that the snapshot has been migrated successfully.

[SCHEMA_VERSION after migration](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/7.jpg)

Thus, the production data has been restored. Furthermore our first point has been established and works like a charm. All further migrations are executed with the same steps. Every time when you have to update the database in the future, simply create a new migration with a version number higher than the current one. If Flyway starts it will find it and upgrade the database accordingly. However, it’s important to make sure that the version number is correct.

**Step 3: Configuration for Java migrations**

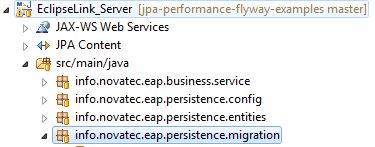
As I described before, the sample application uses the Spring Framework for DI. The required dependencies are defined in the projects ‘pom.xml’ file. To check the state of the database while bootstrapping, it’s important to squeeze in the Flyway dependencies to the normal startup process. This requires a reconfiguration of the applications ‘ApplicationContext’. A Java migration is necessary, if BLOBs or CLOBs should be inserted. It’s also useful to change a massive amount of data or for migrations which requires the usage of regular expression.

The following listing shows the typical configuration of the ‘dataSource’ for the ‘EntityManagerFactoryBean’ and the ‘PlatformTransactionManager’ with Spring.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84 | package info.novatec.eap.persistence.config;    import java.util.HashMap;  import java.util.Map;    import javax.sql.DataSource;    import org.flywaydb.core.Flyway;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.ComponentScan;  import org.springframework.context.annotation.Configuration;  import org.springframework.context.annotation.DependsOn;  import org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor;  import org.springframework.orm.jpa.JpaTransactionManager;  import org.springframework.orm.jpa.JpaVendorAdapter;  import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;  import org.springframework.orm.jpa.vendor.Database;  import org.springframework.orm.jpa.vendor.EclipseLinkJpaVendorAdapter;  import org.springframework.transaction.PlatformTransactionManager;  import org.springframework.transaction.annotation.EnableTransactionManagement;    import com.zaxxer.hikari.HikariConfig;  import com.zaxxer.hikari.HikariDataSource;    *@Configuration*  *@ComponentScan*("info.novatec.eap.persistence")  *@EnableTransactionManagement*  public class JpaConfig {    *@Bean*  public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean() {  final LocalContainerEntityManagerFactoryBean localContainerEntityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();  localContainerEntityManagerFactoryBean.setDataSource(this.dataSource());  localContainerEntityManagerFactoryBean.setPackagesToScan("info.novatec.eap.persistence");  localContainerEntityManagerFactoryBean.setJpaVendorAdapter(this.jpaVendorAdapter());  localContainerEntityManagerFactoryBean.setJpaPropertyMap(this.jpaProperties());    return localContainerEntityManagerFactoryBean;  }    *@Bean*(destroyMethod = "close")  public DataSource dataSource() {  final HikariConfig config = new HikariConfig();  config.setMaximumPoolSize(10);  config.setDataSourceClassName("com.ibm.db2.jcc.DB2SimpleDataSource");  config.setPoolName("testPool");  config.addDataSourceProperty("serverName", "localhost");  config.addDataSourceProperty("databaseName", "FLYWEXAM");  config.addDataSourceProperty("user", "db2admin");  config.addDataSourceProperty("portNumber", "50000");  config.addDataSourceProperty("password", "db2admin");    return new HikariDataSource(config);  }    private Map<String, Object> jpaProperties() {  final Map<String, Object> map = new HashMap<String, Object>();  map.put("eclipselink.weaving", "false");    return map;  }    private JpaVendorAdapter jpaVendorAdapter() {  final JpaVendorAdapter jpaVendorAdapter = new EclipseLinkJpaVendorAdapter() {  {  setShowSql(true);  setDatabase(Database.DB2);  setGenerateDdl(false);  }  };    return jpaVendorAdapter;  }    *@Bean*  public PlatformTransactionManager transactionManager() {  return new JpaTransactionManager(this.entityManagerFactoryBean().getObject());  }    *@Bean*  public PersistenceExceptionTranslationPostProcessor exceptionTranslator() {  return new PersistenceExceptionTranslationPostProcessor();  }  } |

To get access to java migration classes, a package must be created, which serves as a repository for Flyway. For this example I’ve created a package ’info.novatec.eap.persistence.migration’.

[](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/8.jpg)

Next, we create a new bean for Flyway:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | *@Bean*(initMethod = "migrate")  public Flyway flyway() {  final Flyway flyway = new Flyway();  flyway.setDataSource(this.dataSource());  flyway.setLocations("info.novatec.eap.persistence.migration", "filesystem:src/main/resources/migrations");    return flyway;  } |

In the context of this bean we instantiate the class ‘Flyway’ and set the ‘data Source’ with a setter.  With ‘setLocations’ we define the package path which contains the migration java files. While migrating, Flyway checks if all migrated files are available. For the implicit migration process, it’s important to define ‘initMethod=migrate’ as bean property. This setting guarantees that Flyway checks the version of the database and migrate it to a new state while bootstrapping the application.

To ensure that the ‘EntityManagerFactoryBean’ starting up after the migration, we are using the Annotation ‘@DependsOn(“flyway”)’. The next listing shows the enhanced configuration class.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | *@Bean*  *@DependsOn*(value = { "flyway" })  public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean() {  final LocalContainerEntityManagerFactoryBean localContainerEntityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();  localContainerEntityManagerFactoryBean.setDataSource(this.dataSource());  localContainerEntityManagerFactoryBean.setPackagesToScan("info.novatec.eap.persistence");  localContainerEntityManagerFactoryBean.setJpaVendorAdapter(this.jpaVendorAdapter());  localContainerEntityManagerFactoryBean.setJpaPropertyMap(this.jpaProperties());    return localContainerEntityManagerFactoryBean;  } |

**Step 4: Create a Java type migration file**

Java migrations must follow the same naming conventions as migration scripts based on SQL. This means that a prefix, a version, a separator and a description must be specified as class name for Flyway. It should also be mentioned that a class and also a SQL file always migrated within a transaction. Furthermore, each Java migration class must implement the interface ‘JdbcMigration’. The following listing shows a correct implemented migration class.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package info.novatec.eap.persistence.migration;    import java.sql.Connection;  import java.sql.PreparedStatement;    import org.flywaydb.core.api.migration.jdbc.JdbcMigration;    public class V2\_0\_\_enhancementOnCustomerTable implements JdbcMigration {    @Override  public void migrate(Connection connection) throws Exception {  final PreparedStatement statement = connection.prepareStatement("INSERT INTO CUSTOMER (VERSION,TIMESTAMP,FIRSTNAME,LASTNAME,EMAIL) "  + "VALUES (1,'2014-09-01 09:49:00.0','Max','Mustermann','max.mustermann@blub.de')");  try {  statement.execute();  } finally {  statement.close();  }  }  } |

The application uses the Spring Framework. So it makes sense to use ‘SpringJdbcMigration’ instead of ‘JdbcMigration’. The execution processes of both implementation classes are identical. The main difference between these is that ‘SpringJdbcMigration’ enables to use the ‘SpringJDBC’-Template class to define SQL statements. The following class implements the interface ‘SpringJdbcMigration’.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package info.novatec.eap.persistence.migration;    import org.flywaydb.core.api.migration.spring.SpringJdbcMigration;  import org.springframework.jdbc.core.JdbcTemplate;    public class V2\_1\_\_addDataToCustomer implements SpringJdbcMigration {    *@Override*  public void migrate(JdbcTemplate jdbcTemplate) throws Exception {  jdbcTemplate.execute("INSERT INTO CUSTOMER (VERSION,TIMESTAMP,FIRSTNAME,LASTNAME,EMAIL) "  + "VALUES (1,'2014-09-01 09:49:00.0','Max','Mustermann','max.mustermann@blub.de')");  }  } |

**Step 5: Execution**

As described in step 3, a migration is executed if it’s necessary while bootstrapping the application. This applies to Java classes and also to SQL scripts. Thus, the second key point, as described above, has been implemented.

The question is if Maven also executes Java migration classes. The answer is ‘Yes’. For this purpose we need to adjust the ‘flyway-maven-plugin’. A second ‘location’ with the name of the package which I’ve created before must be added. The next listing shows how it’s defined.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <plugin>  <groupId>org.flywaydb</groupId>  <artifactId>flyway-maven-plugin</artifactId>  <version>3.0</version>    <dependencies>  <dependency>  <groupId>com.ibm</groupId>  <artifactId>db2jcc</artifactId>  <version>4.0</version>  </dependency>  </dependencies>    <configuration>  <user>db2admin</user>  <password>db2admin</password>  <url>jdbc:db2://localhost:50000/FLYWEXAM</url>  <locations>  <location>filesystem:src/main/resources/migrations</location>  <location>classpath:info.novatec.eap.persistence.migration</location>  </locations>  <skip>false</skip>  </configuration>  </plugin> |

This setting enables Flyway to access Java migration classes and considers them for migration. The next screenshots shows the ‘SCHEMA\_VERSION’ table again. The ‘script’ column shows that Java and also SQL migrations have been performed on the database.

[SCHEMA_VERSION after migrations](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/9.jpg)

For further information’s and tutorials, I recommend the documentation of this project, which can be found[here](http://flywaydb.org/documentation/). I’ve also attached the [flyway-sample](http://blog.novatec-gmbh.de/wp-content/uploads/2014/09/flyway-sample.zip) application.