

|  |  |
| --- | --- |
| Document name | Flyway Guide |
| Version no. | 1.0 |
| Release date |  |



|  |
| --- |
| Guidelines of Lunch Committee  (HR-GLC) |

|  |
| --- |
| Guidelines of Lunch Committee  (HR-GLC) |

FLYWAY

##### Document History

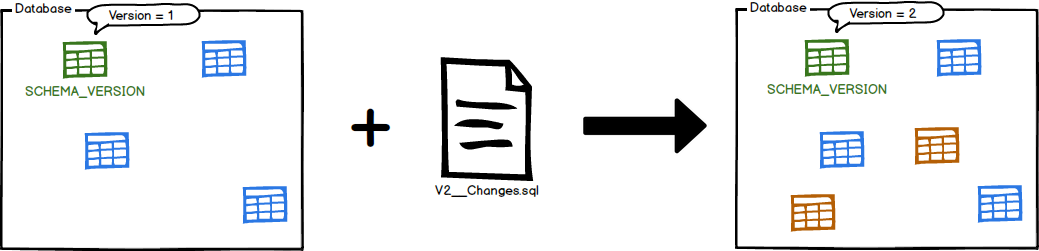
| Version No. | Authored / Modified by | Reviewed by, Date | Approved by, Date | Remark / Change History |
| --- | --- | --- | --- | --- |
| 1.0 | Sakshi Kathuria  23-June-15 |  |  | Initial Document |

FLYWAY

* Flyway is an open-source database migration tool. It strongly favors simplicity and convention over configuration.
* It is based around just 6 basic commands: [Migrate](http://flywaydb.org/documentation/command/migrate.html), [Clean](http://flywaydb.org/documentation/command/clean.html), [Info](http://flywaydb.org/documentation/command/info.html), [Validate](http://flywaydb.org/documentation/command/validate.html), [Baseline](http://flywaydb.org/documentation/command/baseline.html) and [Repair](http://flywaydb.org/documentation/command/repair.html).
* Migrations can be written in [SQL](http://flywaydb.org/documentation/migration/sql.html) (database-specific syntax (such as PL/SQL, T-SQL) is supported) or [Java](http://flywaydb.org/documentation/migration/java.html).
* It has a [Command-line client](http://flywaydb.org/documentation/commandline). If you are on the JVM, we recommend using the [Java API](http://flywaydb.org/documentation/api) (also works on Android) for migrating the database on application startup. Alternatively, you can also use the [Maven plugin](http://flywaydb.org/documentation/maven), [Gradle plugin](http://flywaydb.org/documentation/gradle), [SBT plugin](http://flywaydb.org/documentation/sbt) or the [Ant tasks](http://flywaydb.org/documentation/ant).
* And if that not enough, there are [plugins](http://flywaydb.org/documentation/plugins.html) available for Spring Boot, Dropwizard, Grails, Play, Griffon, Grunt, Ninja and more!
* Supported databases are [Oracle](http://flywaydb.org/documentation/database/oracle.html), [SQL Server](http://flywaydb.org/documentation/database/sqlserver.html), [SQL Azure](http://flywaydb.org/documentation/database/sqlazure.html), [DB2](http://flywaydb.org/documentation/database/db2.html), [DB2 z/OS](http://flywaydb.org/documentation/database/db2zos.html), [MySQL](http://flywaydb.org/documentation/database/mysql.html), (including Amazon RDS),[MariaDB](http://flywaydb.org/documentation/database/mariadb.html), [Google Cloud SQL](http://flywaydb.org/documentation/database/googleCloudSql.html), [PostgreSQL](http://flywaydb.org/documentation/database/postgresql.html) (including Amazon RDS and Heroku), [Redshift](http://flywaydb.org/documentation/database/redshift.html), [Vertica](http://flywaydb.org/documentation/database/vertica.html), [H2](http://flywaydb.org/documentation/database/h2.html), [Hsql](http://flywaydb.org/documentation/database/hsql.html),[Derby](http://flywaydb.org/documentation/database/derby.html), [SQLite](http://flywaydb.org/documentation/database/sqlite.html) and [solidDB](http://flywaydb.org/documentation/database/solid.html).

1. **MIGRATE:**

* Migrates the schema to the latest version. Flyway will create the metadata table automatically if it doesn't exist.



Migrate is the centerpiece of the Flyway workflow. It will scan the filesystem or your classpath for available migrations. It will compare them to the migrations that have been applied to the database. If any difference is found, it will migrate the database to close the gap.

Migrate should preferably be executed on application startup to avoid any incompatibilities between the database and the expectations of the code.

### Example 1: We have migrations available up to version 9, and the database is at version 5.

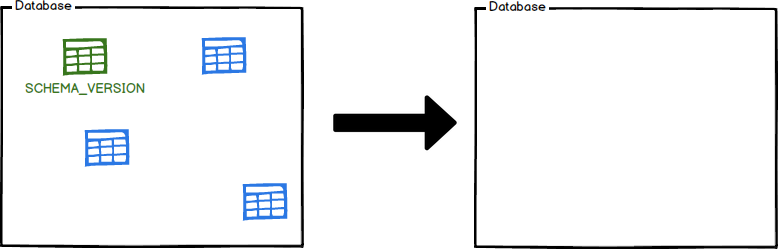
Migrate will apply the migrations 6, 7, 8 and 9 in order.

### Example 2: We have migrations available up to version 9, and the database is at version 9.

Migrate does nothing.

1. **CLEAN**

* Drops all objects in the configured schemas

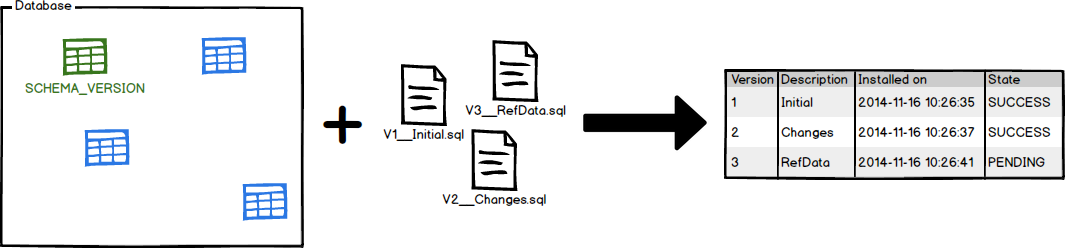


Clean is a great help in development and test. It will effectively give you a fresh start, by wiping your configured schemas completely clean. All objects (tables, views, procedures, ...) will be dropped.

Needless to say: do not use against your production DB!

1. **INFO:**

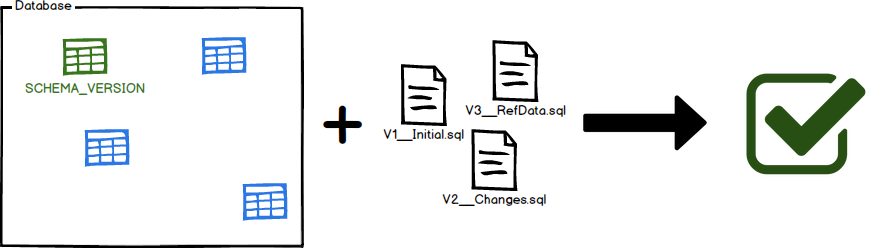
* Prints the details and status information about all the migrations



Info lets you know where you stand. At a glance you will see which migrations have already been applied, which other ones are still pending, when they were executed and whether they were successful or not.

1. **VALIDATE:**

* Validates the applied migrations against the available ones.

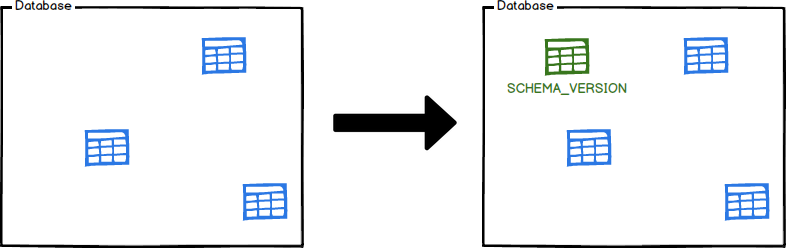


Validate helps you verify that the migrations applied to the database match the ones available locally.

This is very useful to detect accidental changes that may prevent you from reliably recreating the schema.

1. **BASELINE:**

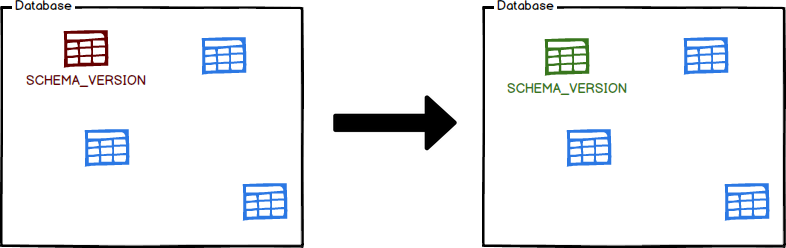
* Baselines an existing database, excluding all migrations up to and including baseline Version.



Baseline is for introducing Flyway to [existing databases](http://flywaydb.org/documentation/existing.html) by baselining them at a specific version. The will cause [Migrate](http://flywaydb.org/documentation/command/migrate.html) to ignore all migrations up to and including the baseline version. Newer migrations will then be applied as usual.

1. **REPAIR:**

Repairs the metadata table



Repair is your tool to fix issues with the metadata table. It has two main uses:

* Remove failed migration entries (only for databases that do NOT support DDL transactions)
* Realign the checksums of the applied migrations to the ones of the available migrations

**INTEGRATING FLYWAY WITH MAVEN CONFIGURATION:**

You need to add a plugin in pom:

**<plugin>**

**<groupId>org.flywaydb</groupId>**

**<artifactId>flyway-maven-plugin</artifactId>**

**<version>3.2.1</version>**

**<configuration>**

**<driver>com.mysql.jdbc.Driver</driver>**

**<url>jdbc:mysql://localhost:3306/mydb</url>**

**<user>root</user>**

**<password>root</password>**

**</configuration>**

**</plugin>**

Add Dependency in POM:

<dependency>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-maven-plugin</artifactId>

<version>3.2.1</version>

</dependency>

**Writing Migration Scripts:**

Create the migration scripts which contains the necessary schema updates that needs to be applied to the database during a migration, As specified earlier, Flyway supports migration scripts written in plain old SQL or in Java.

Whichever method you use, all you have to do is to have the migration files named appropriately, following the convention required by Flyway and have them in the location specified in the configuration.

Why a naming convention? The naming convention is required, as this is the default way Flyway keeps track of its versioning (I say default because when using Java, there is the possibility to override this naming mechanism. This is explained below). It uses it to determine the order in which migration scripts needs to be applied and to keep track of the scripts applied and ones pending.

The naming convention is as follows:

**V<VERSION>\_\_<DESCRIPTION>.<FORMAT>**

**Creating the first migration:**

Create the database manually from the sql command prompt.

You need to create a migration directory i.e. /src/main/resources/db/migration.

Inside the migration folder you can create sql scripts in which you write your sql queries.

Eg. V1\_Create\_Person.sql.

create table PERSON (

ID int not null,

NAME varchar(100) not null

);

Open the command prompt and go to the location of your project source folder and run the command:

* mvn compile flyway:migrate

If all went well, you should see the following output:

Flyway 3.2.1 by Boxfuse

Database: jdbc:mysql://localhost:3306/mydb (MySQL 5.5)

Validated 1 migrations (execution time 00:00.015s)

Creating Metadata table: `mydb`.`schema\_version`

Current version of schema `mydb`: << Empty Schema >>

Migrating schema `mydb` to version 1 - Create person table

Successfully applied 1 migration to schema `mydb` (execution time 00:00.563s).

**Second Migration:**

If we add a second migration i.e. src/main/resources/db/migration/ V2\_Add\_People.sql:

insert into PERSON (ID, NAME) values (1, 'Sakshi');

insert into PERSON (ID, NAME) values (2, ‘Mr. Foo’);

insert into PERSON (ID, NAME) values (3, ‘Ms. Bar');

and execute it by issuing:

> mvn compile flyway:migrate

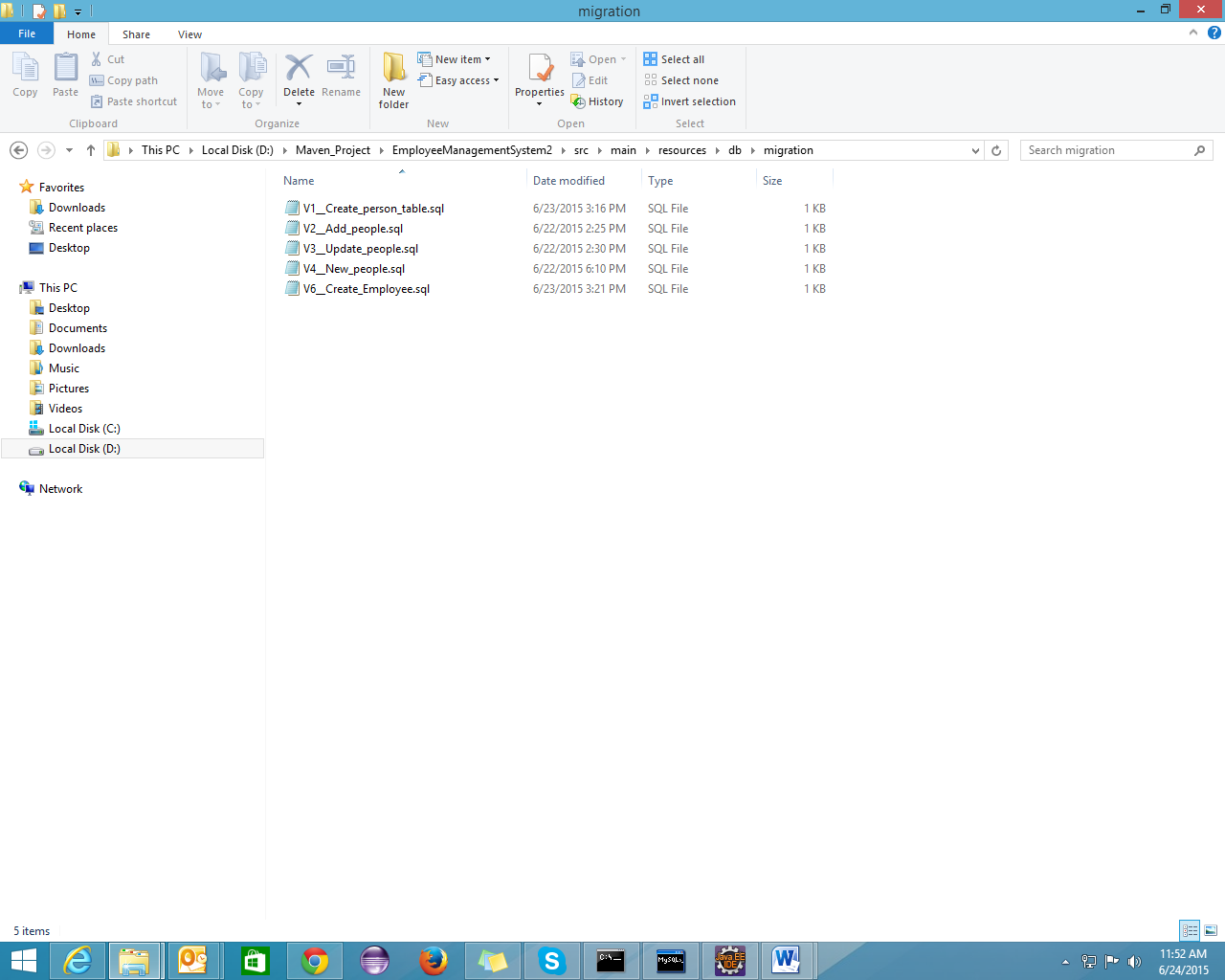
You will get the following output:

[INFO] Current version of schema "PUBLIC": 1

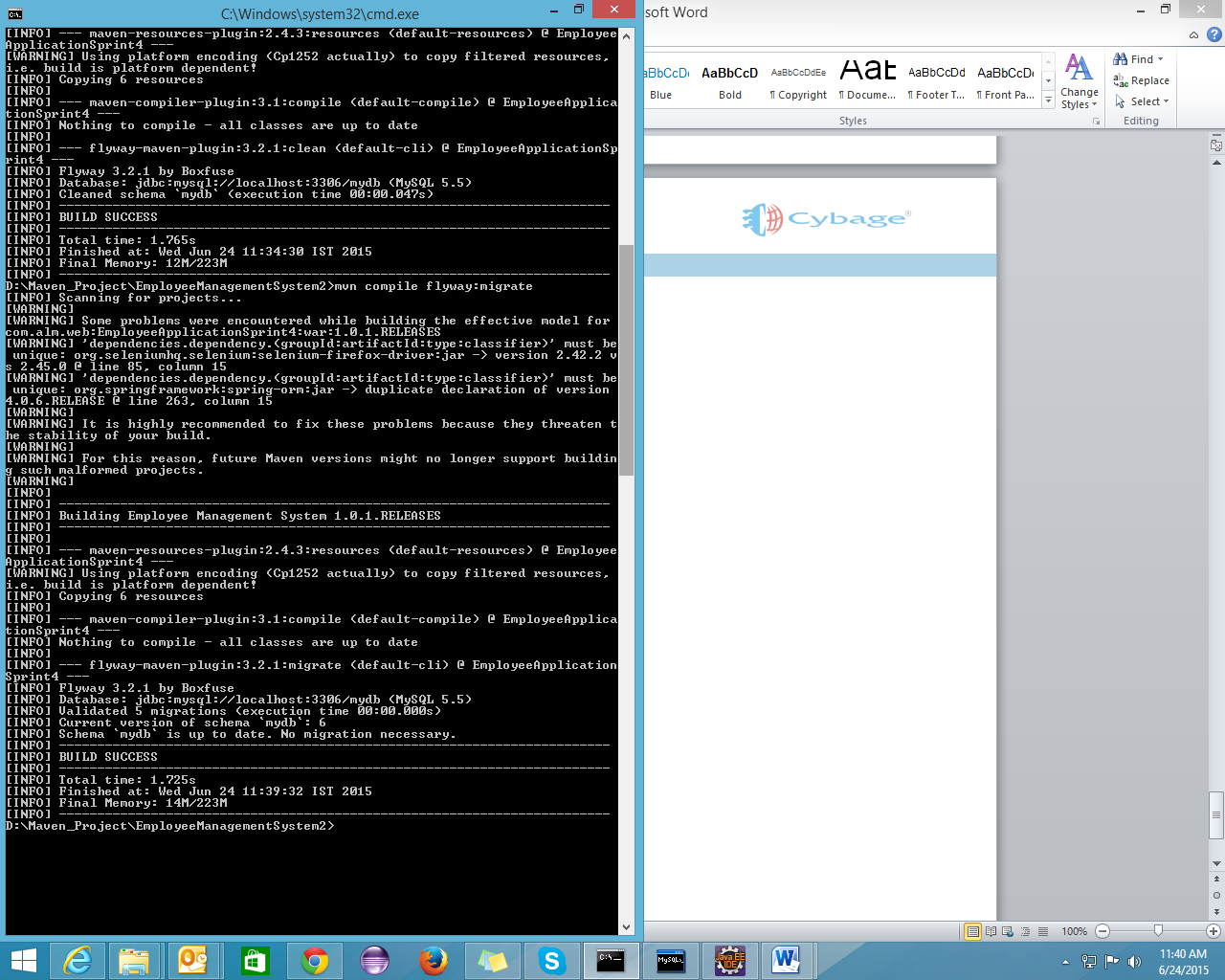
[INFO] Migrating schema "PUBLIC" to version 2

[INFO] Successfully applied 1 migration to schema "PUBLIC" (execution time 00:00.090s).

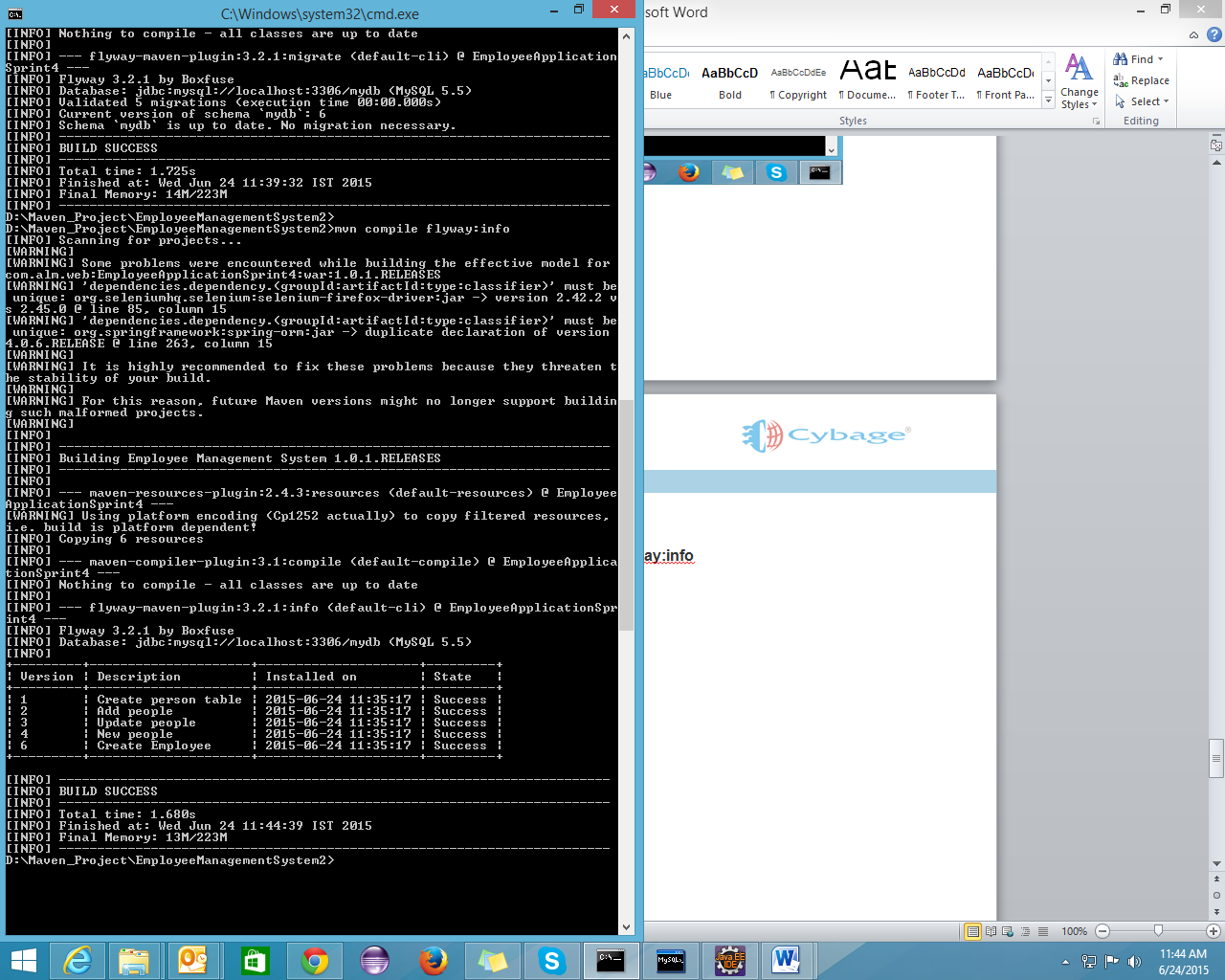
Similarly, you can create other sql scripts according to your requirement.



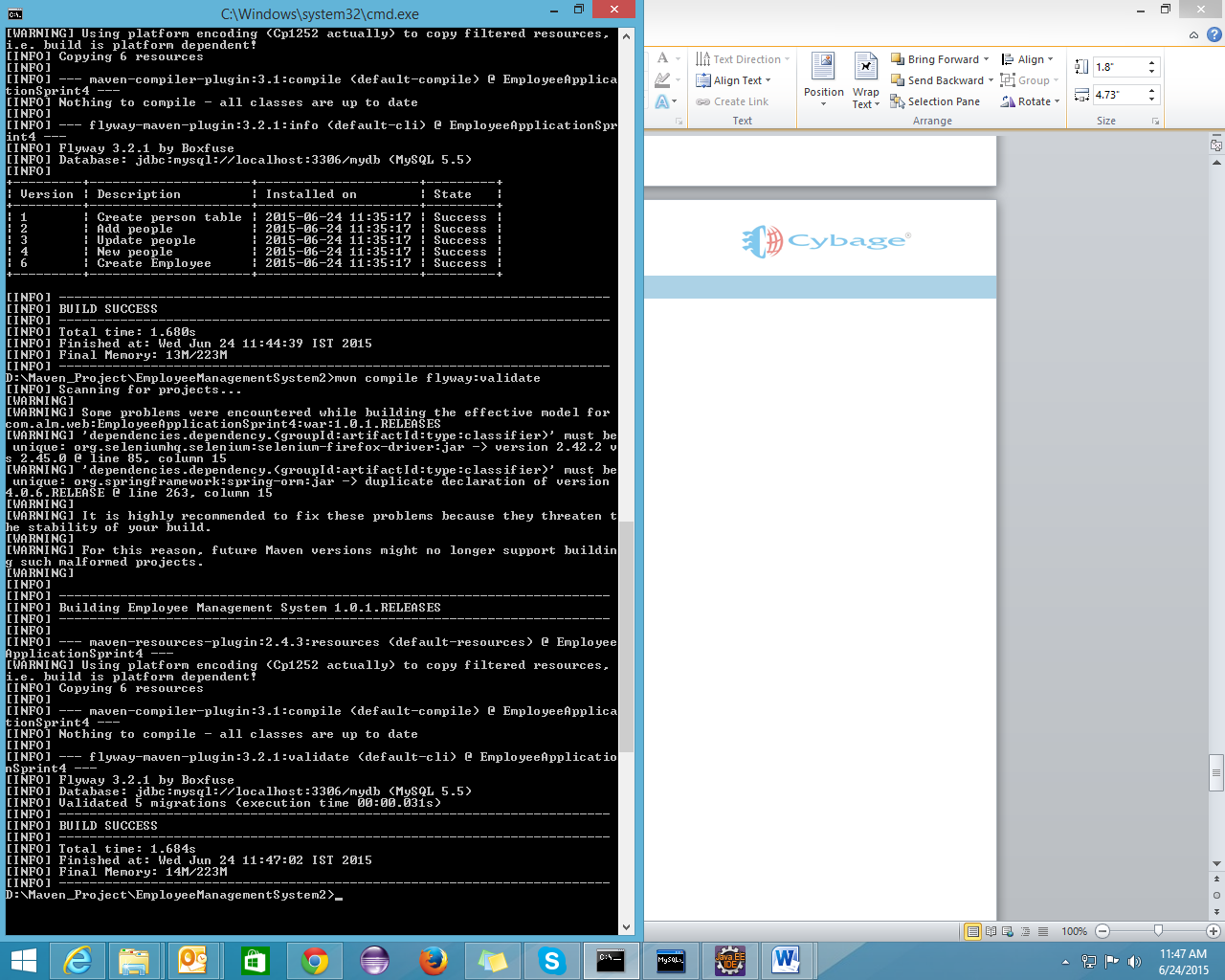
**mvn compile flyway:migrate**



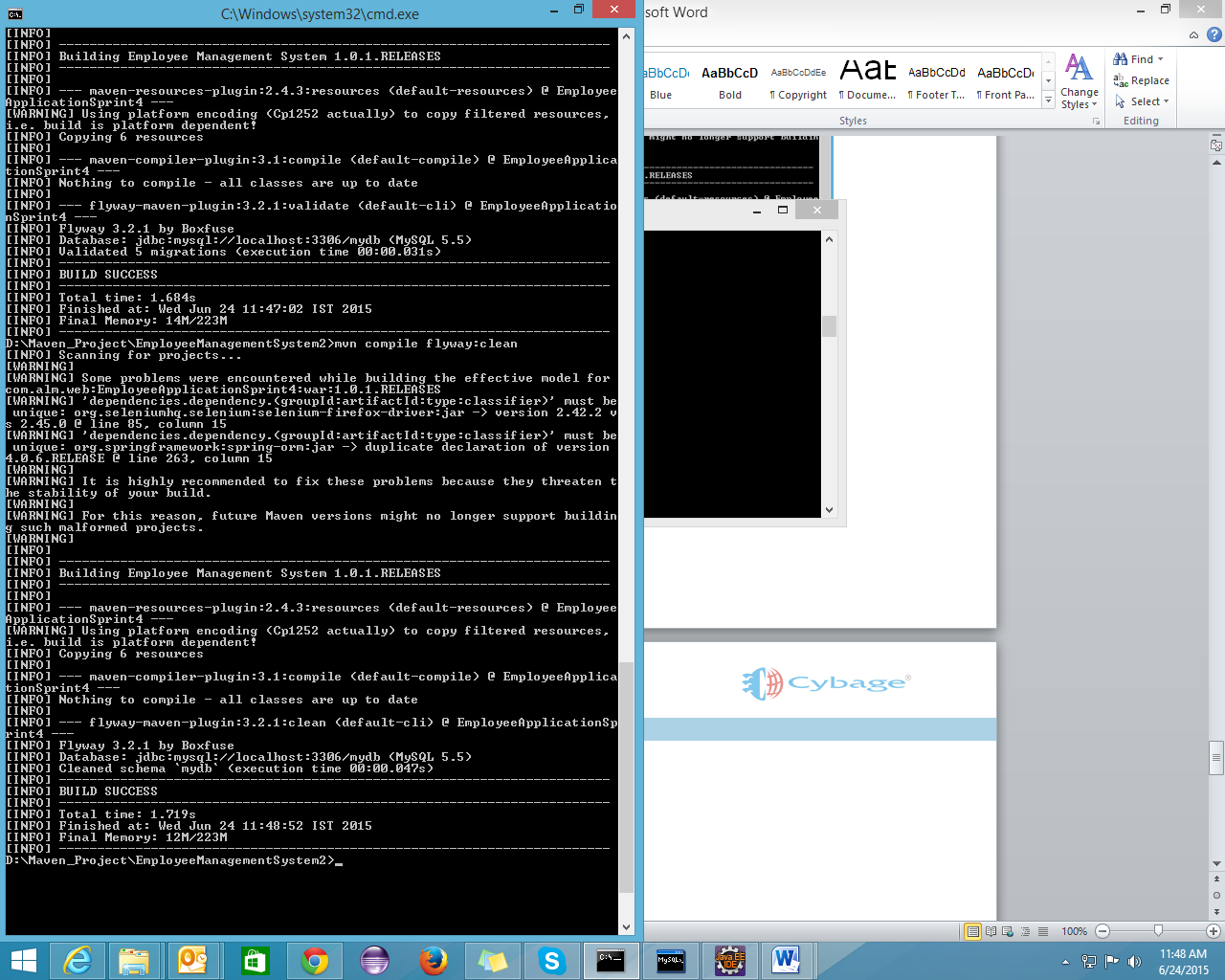
**mvn compile flyway:info**

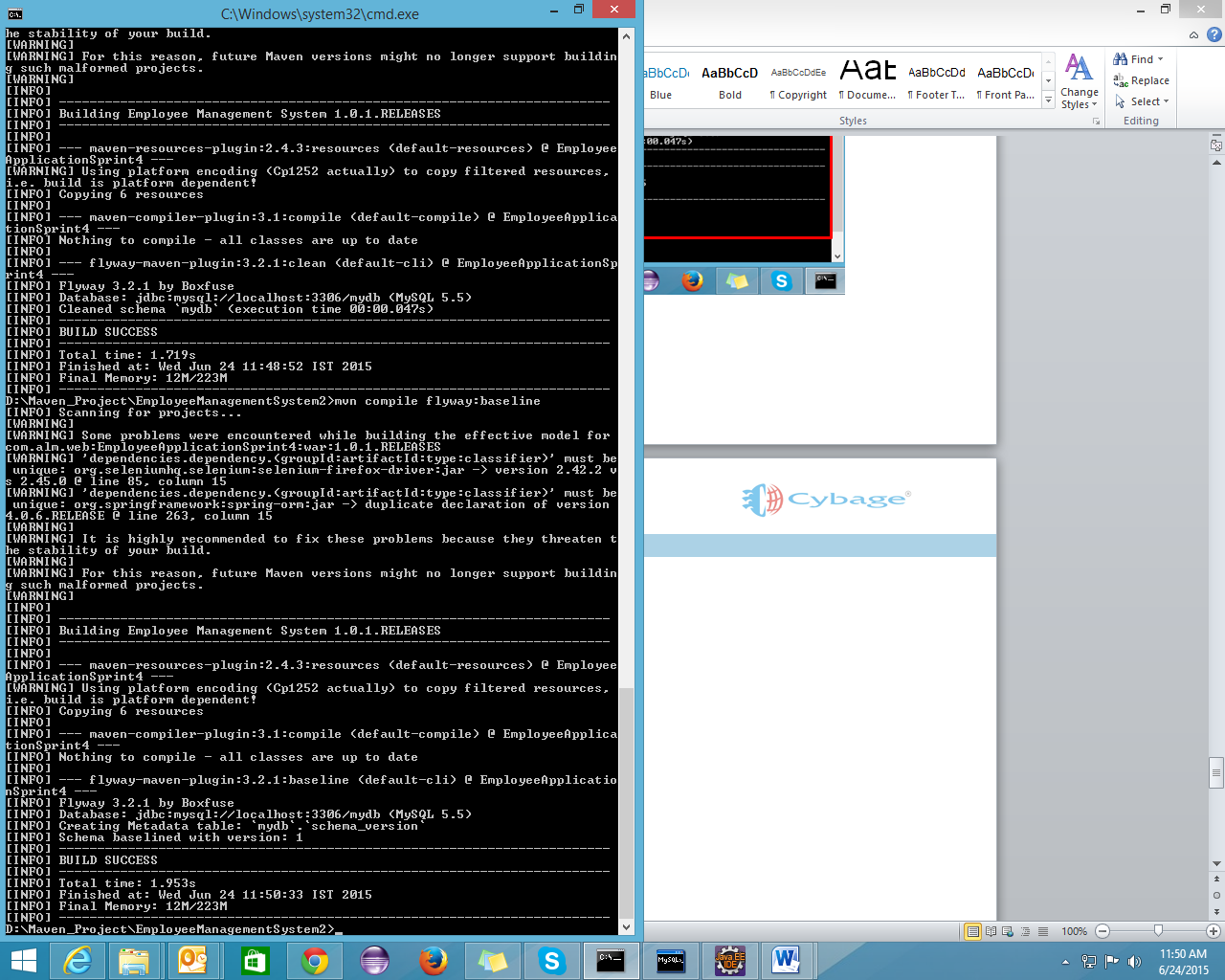


**mvn compile flyway:validate**



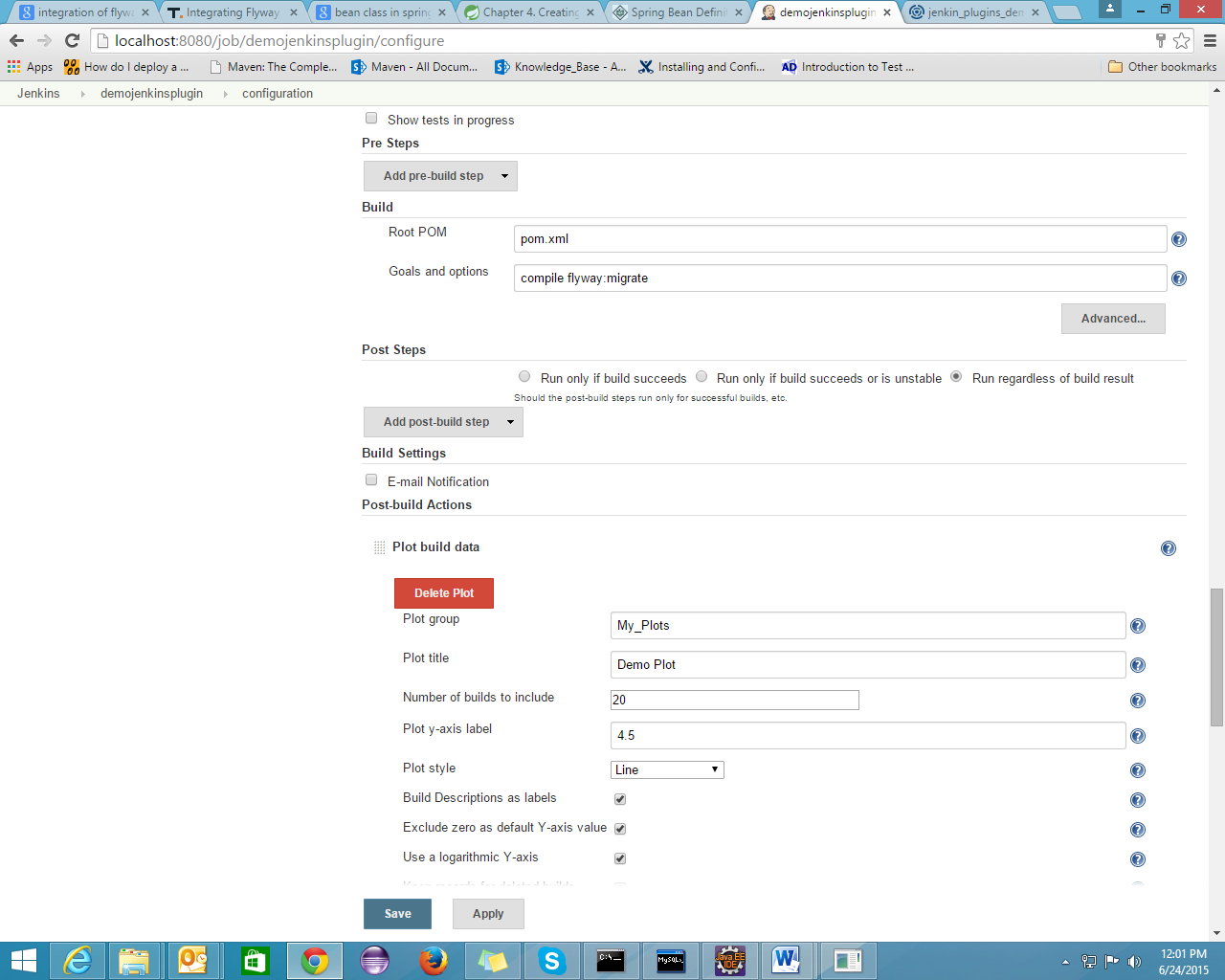
**mvn compile flyway:clean**



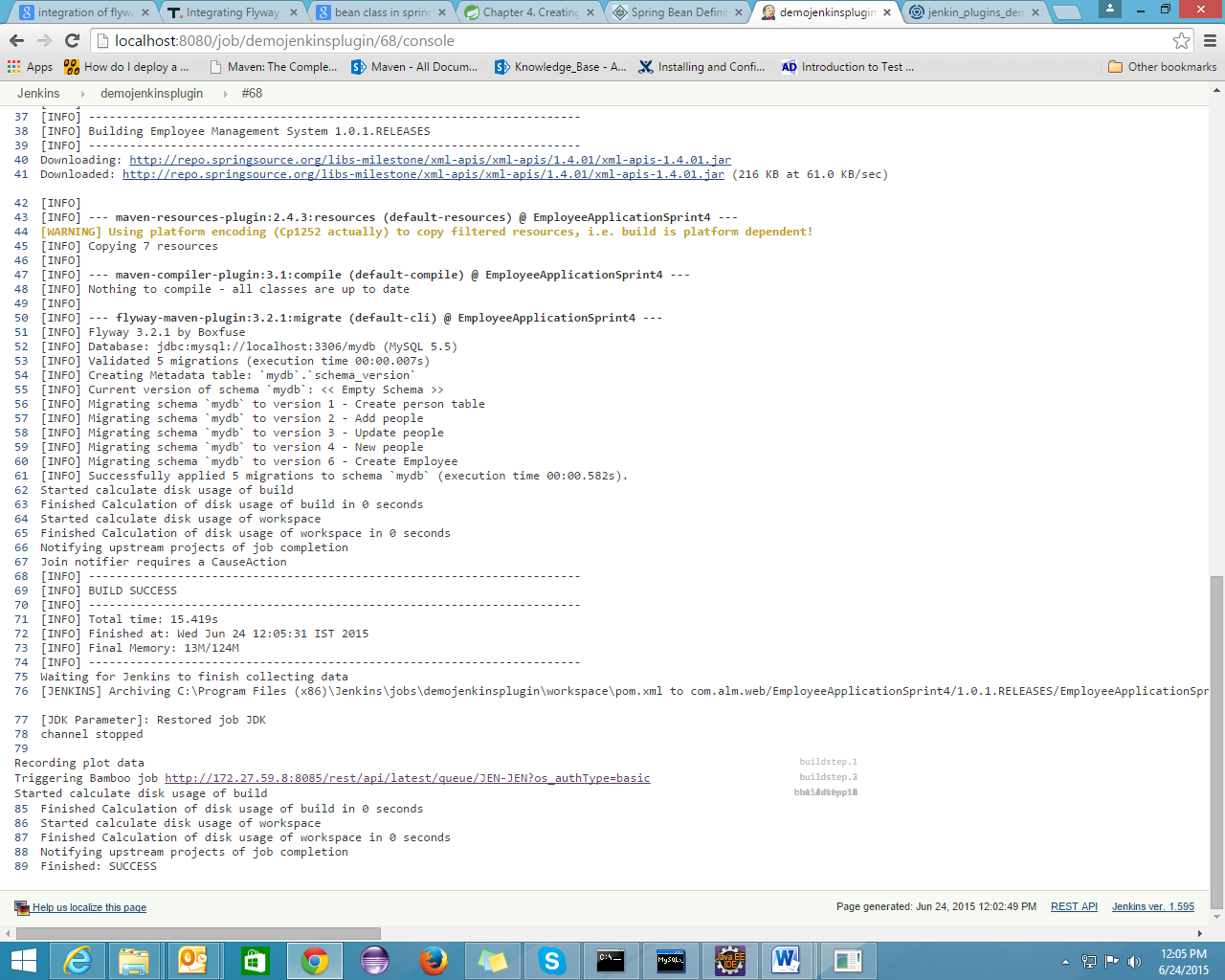
**mvn compile flyway:baseline**

**Integration of Flyway with Jenkins**

If you want to migrate your database through Jenkins, then you can simply write the commands in goals and option column of build.



**Compile flyway:info**





**Integration with Spring Framework**

When integrating Flyway into a Spring Application what you want to do is to have Flyway kick in, find the migration files (either in SQL or JAVA) and apply the necessary modifications to the database at application start-up. The following steps show how to achieve this:

#### 1. Project’s Dependencies Set-up

<dependency>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-core</artifactId>

<version>${Flyway.version}</version>

</dependency>

#### 2. Configuring Flyway to integrate into Spring’s Container

The next steps would be the configuration needed to have Spring pick up Flyway as a managed bean and have it play nicely with other beans it would need to work it…to be specific the entity manager factory.

If you are using XML to configure spring the necessary configuration would look thus:

<!-- Flyway configuration -->

<bean id="flyway" class="org.Flyway.core.Flyway" init-method="migrate">

<property name="baselineOnMigrate" value="true" />

<property name="locations" value="filesystem: /path/to/migrations/" />

<property name="dataSource" ref="dataSource" />

</bean>

Some additional things to take note of in the configurations:

***Flyway Configuration.***  
When configuring Flyway you need to specify the init method (which was done using the init-methodproperty in xml and initMethod property of the @Bean annotation in JavaConfig)

This is used to instruct Spring to call the migrate() method once all the properties of the Flyway bean has been initialized. The migrate() method is what is responsible for performing the migration logic: that is finding the migration scripts, applying them and keeping a tab of successful migrations etc. Which is why it is specified as the init method as you would want this to be executed as soon as the Flyway bean is initialized in Spring’s container.

The baseLineOnMigrate is also another interesting part of the configuration. It comes in handy when Flyway is initially used for the first time and no SCHEMA\_VERSION table exists. It instructs Flyway that before the migration scripts are applied, it should create a migration entry within the SCHEMA\_VERSION table which would serve as the baseline, and thus the available migration script would only be applied if their version is higher than the baseline version.

The dataSource is used to specify the dataSource.

The last item of interest in the configuration is the location through which you specify where Flyway should find the migration scripts. Flyway has the ability to scan either the filesystem or classpath for these migration scripts. In the above example the value given to the location is prefixed with “filesystem:” indicating to Flyway to use its file system scanner to locate the migration scripts. If no prefix is specified or “classpath:” is used as the prefix, then Flyway uses its classloader scanner to locate the migration scripts.

When you will run your application, database changes will get applied to your database.

**Advantages**

Developers no longer need to remove the entire test database in order to create a new test database from scratch (e.g. using schema creation scripts from DDL generation tools). Further, if generation of test data costs a lot of time, developers can avoid regenerating test data for small, non-destructive changes to the schema.

<http://flywaydb.org/>

<http://blog.trifork.com/2014/12/09/integrating-flywaydb-in-a-spring-framework-application/>