DB migration with Flyway

⇒ These notes are also demonstrated in a video (21:57).

We'll use Flyway to manage our database tables. To get started, use **File » Project Structure » Libraries** and

```
org.flywaydb:flyway-core:3.1
```

There are newer versions `3.2` and `3.2.1`, but I had trouble with them so I recommend sticking to `3.1`.

The paste this code at the top of your `main` function in your `SparkDemo` class, *before* you do anything related to the driver manager or template engine.

```
    Flyway flyway = new Flyway();
    flyway.setDataSource(Config.JDBC_URL, "", "");
    flyway.migrate();
```

You'll have to confirm the import, which should be just:

```
import org.flywaydb.core.Flyway;
```

The `setDataSource` method takes the JDBC connection string (which I defined as

a constant in a `Config` class – on 25 Feb), and then the database username and password, which I'm not using yet so I can leave them blank.

Next, create a subdirectory of your resource folder, called `db`. Then create a subdirectory of that called `migration`. It will show up in IntelliJ as `db.migration`, but in on the disk they are nested directories.

The format for SQL file names in the `db/migration` resource folder is:

```
V2__create_something.sql
```

That is, the filename must:

- start with capital `V`.
- then a version number – there are a variety of permissible formats but the simplest is just serial integers like `1`, `2`, `3`, `4`.
- then two underscores: `__`
- then some descriptive text about what this SQL script does – don't use spaces or other special characters. Underscore is okay.

- end with the extension .sql.

In the video, I started with V1__create_user_table.sql:

```
create table user
( id int primary key auto_increment
, email varchar(255) not null unique
, password varchar(255) not null default ''
)
```

After creating that file in the right place, running SparkDemo should give some log messages from Flyway, including "Migrating schema ... to version 1."

Next I created V2__sample_users.sql:

```
-- some sample users
insert into user (email) values ('league@acm.org');
insert into user (email) values ('alice@google.co');
```

Then after restarting the server, it migrates to version 2. We can investigate in the H2 Console using select * from users and indeed, it shows both of our users in the table.

Now let's say we want to add first and last names to the user table. It could be a problem to go back and change the V1 script that created the table initially. If our site is already live and has real users accessing it, we don't have the liberty to just recreate the tables from scratch. Migration is designed to help with this scenario, along with the SQL ALTER command.

So instead of editing the V1 script, we create a new version, V3__add_names_to_user.sql:

```
-- When altering, make sure not to invalidate
-- existing data!

alter table user add column
  ( first_name varchar(80));

alter table user add column
  ( last_name varchar(80) not null default '');
```

You just have to be careful that constraints like not null don't invalidate existing data. The V3 script above shows two possible choices: first_name is allowed to be null (so that will happen for existing users), and last_name cannot be null but we provide a default value (that will also be applied to existing users).

Now rerunning SparkDemo migrates to version 3, and we have those fields added. My existing users from script V2 were updated accordingly:

```
  ID   EMAIL              PASSWORD   FIRST_NAME   LAST_NAME
   1   league@acm.org                null
   2   alice@google.co               null
```

After the software is deployed, it would be a mistake to go back and modify any of the earlier SQL scripts. For example, what if I wanted to **change** V1 to add a creation timestamp?

```
create table user
( id int primary key auto_increment
, email varchar(255) not null unique
, password varchar(255) not null default ''
, created timestamp not null default now()
)
```

As soon as I run the migration, Flyway complains at us for making changes to a script that it had already applied:

```
Validate failed: checksum mismatch for version 1
```

Now, for flexibility and ease during development – before any version of our software has been deployed – we might want to try something else. As we churn through our application and figure out what our model needs to be, it's much more convenient if we can just update the SQL directly instead of having a hundred scripts applying tiny ALTER commands until we get it right.

To do that, we can ask Flyway to just delete all data and drop all tables before it runs the migration. **YOU WOULD NEVER USE THIS ON A PRODUCTION DATABASE,** but it's incredibly convenient in the early stages of development.

```
    Flyway flyway = new Flyway();
    flyway.setDataSource(Config.JDBC_URL, "", "");
    flyway.clean();  // WARNING: deletes all tables and data
    flyway.migrate();
```