



Loading initial data with Spring

August 20th, 2019, 7 min read

Java JPA Spring Spring boot Spring Data

I've been answering Spring related questions on Stack Overflow for the past three years now, and one of my most popular answers is about [how to load initial data](#). While my solution over there works really fine, there are multiple solutions to this problem, and in this tutorial I'll demonstrate which ones you have. This also allows me to fulfil my promise I made in [my earlier tutorial about Spring Data JPA](#), where I said I would create a tutorial like this one.

Using Spring's JDBC initializer

One of the nice features of Spring boot is that it will automatically pick up any **schema.sql** and **data.sql** files on the classpath if you're using an embedded datasource.

For example, let's say we have the following entity:

```
@Data  
@Builder  
@NoArgConstructor
```

```
@AllArgConstructor
@Entity
@Table(name = "marvel_character")
public class MarvelCharacter {
    @Id
    @Column(name = "hero_name")
    private String heroName;
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "last_name")
    private String lastName;
}
```


In this case, we can set up our schema by creating a **schema.sql** file like this:

```
create table marvel_character (
    name          varchar(100)    not null,
    first_name    varchar(100)    not null,
    last_name     varchar(100)    not null,
    primary key (name)
);
```

Additionally, we can create a **data.sql** file like this:

```
insert into marvel_character (hero_name, first_name, last_name) values ('Iron man', 'I', 'M')
insert into marvel_character (hero_name, first_name, last_name) values ('Thor', 'Thor', 'O')
insert into marvel_character (hero_name, first_name, last_name) values ('Black widow', 'B', 'W')
insert into marvel_character (hero_name, first_name, last_name) values ('Hawkeye', 'C', 'H')
```

```
insert into marvel_character (hero_name, first_name, last_name) values ('Spider-man',  
insert into marvel_character (hero_name, first_name, last_name) values ('Captain Ameri  
insert into marvel_character (hero_name, first_name, last_name) values ('Hulk', 'Bruce  
insert into marvel_character (hero_name, first_name, last_name) values ('Ant-man', 'Sc
```



There are a few caveats though. First of all, if you use JPA and an embedded datasource, your **schema.sql** won't be taken into effect, because Hibernate's DDL generation has priority over it.

A solution to that problem is to set the following property:

```
spring.jpa.hibernate.ddl-auto=none
```

Additionally, since **Spring boot 2**, the schema is only initialized by default for embedded datasources. To allow loading data for all types of datasources, you have to set the following property:

```
spring.datasource.initialization-mode=always
```

If you want to use multiple datasources, like an in-memory H2 database for development, and a MySQL database for production, you can name your files like **schema-h2.sql** and **data-h2.sql**.

However, this will only work if you set the following property:

```
spring.datasource.platform=h2
```

Last but not least, if you want to change the name of the files in a different way, or you want to execute multiple SQL scripts, you can use the `spring.datasource.data` property:

```
spring.datasource.data=classpath:script1.sql, classpath:script2.sql
```

🔗 Using Hibernate

I've already mentioned it before, but with Hibernate you can also automatically generate your DDL based on your entities by setting:

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Additionally, when you do this, Hibernate will pick up a file called **import.sql** on the classpath, in which you can add your insert-statements, like we did before:

```
insert into marvel_character (hero_name, first_name, last_name) values ('Iron man', 'I  
insert into marvel_character (hero_name, first_name, last_name) values ('Thor', 'Thor'  
insert into marvel_character (hero_name, first_name, last_name) values ('Black widow',  
insert into marvel_character (hero_name, first_name, last_name) values ('Hawkeye', 'Cl  
insert into marvel_character (hero_name, first_name, last_name) values ('Spider-man',  
insert into marvel_character (hero_name, first_name, last_name) values ('Captain Ameri  
insert into marvel_character (hero_name, first_name, last_name) values ('Hulk', 'Bruce  
insert into marvel_character (hero_name, first_name, last_name) values ('Ant-man', 'Sc
```



Just like with the Spring JDBC initializer, you can also configure which SQL scripts you want to load, by using the `spring.jpa.properties.hibernate.hbm2ddl.import_files` property:

```
spring.jpa.properties.hibernate.hbm2ddl.import_files=script1.sql, script2.sql
```

Using a repository

While the previous solutions used SQL statements to insert data into the database, you can also use your Spring data repository if you created one.

To be able to insert data into the database when the application is started, all you have to do is create a bean of type `ApplicationRunner` or `CommandLineRunner`, for example:

```
@Bean
public ApplicationRunner initializer(MarvelCharacterRepository repository) {
    return args -> repository.saveAll(Arrays.asList(
        MarvelCharacter.builder().heroName("Iron man").firstName("Tony").lastName("Stark").build(),
        MarvelCharacter.builder().heroName("Thor").firstName("Thor").lastName("Odinson").build(),
        MarvelCharacter.builder().heroName("Black widow").firstName("Natasha").lastName("Romanov").build(),
        MarvelCharacter.builder().heroName("Hawkeye").firstName("Clint").lastName("Barton").build(),
        MarvelCharacter.builder().heroName("Spider-man").firstName("Peter").lastName("Parker").build(),
        MarvelCharacter.builder().heroName("Captain America").firstName("Steve").lastName("Rogers").build(),
        MarvelCharacter.builder().heroName("Hulk").firstName("Bruce").lastName("Banner").build(),
        MarvelCharacter.builder().heroName("Ant-man").firstName("Scott").lastName("Lang").build()
    ));
}
```

The benefit of this approach is that you can do this programmatically. The downside however is that if you make a mistake in your entity mapping, you won't notice it immediately, because your entities might still work out fine.

Using a custom DataSource

Rather than relying on the `DataSource` that's being provided by the Spring boot's autoconfiguration, you can also create your own `DataSource`. Using the `EmbeddedDataSourceBuilder`, you can easily add scripts you want to execute, for example:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setName("marvelDB")
        .setType(EmbeddedDatabaseType.H2)
        .addScript("classpath:script1.sql")
        .addScript("classpath:script2.sql")
        // ...
        .build();
}
```

Be aware, this approach only works for embedded datasources, the regular `DataSourceBuilder` doesn't have a fluent way for adding SQL scripts to load on startup.

Using database migrations

When you're handling more complex DDLs, or you need more versioned control over your DDLs, you can use a database migration framework like [Flyway](#) or [Liquibase](#).

The nice part is that both integrate nicely with Spring boot. For example, to use Flyway, all you have to do is to add the following dependency:

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

Additionally, you have to make sure that `spring.flyway.enabled` is `true`. Considering that this is the default value, there shouldn't be any issue.

Now you can create a folder called **db/migrations** in your **src/main/resources** folder, so that it's available on the classpath. Within this folder, you can create versioned SQL migration files, using [the naming convention provided in the documentation](#).

For example, we can create a **V1__create_table_marvel_hero.sql**, containing the `CREATE TABLE` statement like before:

```
create table marvel_character (
  hero_name  varchar(100)  not null,
  first_name varchar(100)  not null,
  last_name  varchar(100)  not null,
  primary key (hero_name)
);
```

Additionally, you can create a **V1.1__insert_marvel_hero.sql** file, for example:

```
insert into marvel_character (hero_name, first_name, last_name) values ('Iron man', 'I  
insert into marvel_character (hero_name, first_name, last_name) values ('Thor', 'Thor'  
insert into marvel_character (hero_name, first_name, last_name) values ('Black widow',  
insert into marvel_character (hero_name, first_name, last_name) values ('Hawkeye', 'C]
```

Did you forget a column, or did you forget to add some records? No worries, crate a **V1.2__insert_more_marvel_hero.sql** file, and add the records you need:

```
insert into marvel_character (hero_name, first_name, last_name) values ('Spider-man',  
insert into marvel_character (hero_name, first_name, last_name) values ('Captain Ameri  
insert into marvel_character (hero_name, first_name, last_name) values ('Hulk', 'Bruce  
insert into marvel_character (hero_name, first_name, last_name) values ('Ant-man', 'Sc
```

Conclusion

With that, we've seen plenty of ways to set up your initial data. For every specific requirement, there is a solution. Whether or not you go for a fully fledged database migration framework, or for a simple SQL file, the choice is up to you.

[Back to tutorials](#) • [Report an issue on GitHub](#) • [Discuss on Twitter](#)



 Hey there, I'm Dimitri

I'm a full-stack developer who likes [testing out](#) new and interesting frameworks and [blogging](#) about them. I prefer working with **Java** and **JavaScript**.

[Privacy policy](#) [Post an idea](#) [Contact](#) [RSS](#)