Springfuse

Reference Documentation

Jaxio

Springfuse: Reference Documentation

Jaxio

Version 3.0.49-SNAPSHOT

Abstract

Springfuse Version 3.0.49-SNAPSHOT Reference Documentation

Legal notice

Table of Contents

1. Requirements 1
    - JDK 1.6 1
    - Maven 1
2. Conventions and integration 2
    - Camel case conventions 2
        - Underscore '_' Enables Java Camel Case Syntax 2
        - Native camel case support 2
    - Primary key conventions 2
        - Numerical Primary Keys 2
        - Primary Keys with 32 characters 3
        - Other Primary Keys 3
    - The 'Account' table 3
    - The 'Role' table 3
    - Other optional account's columns 4
        - The Email column 4
        - The Enabled column 4
    - Special columns for file handling support 5
    - ACCOUNT_ID column & Hibernate Filter 5
    - Version column and Optimistic Locking 6
    - Many to many and inverse attribute 6
3. Configuration 7
    - Id 7
        - Use a SEQUENCE per TABLE 7
        - Use a custom Id generator 7
    - Entity and property names 8
        - Force an entity name 8
        - Force a property name 8
        - Advanced property name calculation 8
    - Type Mapping 9
        - Force a type mapping locally 9
        - Number mapping customization 9
        - Date mapping customization 10
    - Associations 10
        - @ManyToOne 10
        - @OneToMany 11
        - @OneToOne 12
        - Inverse @OneToOne 12
        - @ManyToMany 12

List of Tables

2.1. Account table conditions 3
2.2. Role table conditions 4
2.3. Account's table email column conditions 4
2.4. Account's table enabled column conditions 5

List of Examples

2.1. Basic conversion 2
2.2. Example 2
2.3. Example of binary 5

Chapter 1. Requirements

The Springfuse requirements are really straightforward, all you need is a Java JDK 1.6 and maven 2.1+.

JDK 1.6

You can download the latest jdk at http://www.oracle.com/technetwork/java/

Maven

You can download the latest maven release at http://maven.apache.org/download.html

Chapter 2. Conventions and integration

Celerio has some built-in conventions. When these conventions are followed, Celerio generates cleaner Java code and some specific features. For example, by simply following some columns naming convention, you can rely on Celerio to generate all the infrastructure code and configuration that will allow you to handle file upload and download in your web application, in an optimal way.

Camel case conventions

Underscore '_' Enables Java Camel Case Syntax

'Camel Case' syntax is standard Java code convention. When Celerio encounters the character underscore '_' in a table's name or a column's name, it skips it and converts to upper case the next character when generating classes, variables or methods related to this table, or column.

Example 2.1. Basic conversion

For example, if your table name is `BOOK_COMMENT` , the generated entity class will be named `BookComment` ; a variable holding `BookComment` instance will be named `bookComment` and a setter will be named `setBookComment` , etc.

Native camel case support

If your table's and/or column's name use a camelCase syntax AND if the JDBC driver preserves this syntax, then Celerio takes it into account when generating classes, variables or methods related to this table, or column.

Example 2.2. Example

For example, if your table name is `bankAccount` , the generated entity class will be named `BankAccount` ; a variable holding `BankAccount` instance will be named `bankAccount` and a setter will be named `setBankAccount` , etc.

Choosing explicit names for your tables and columns is thus very important as it improves your source code readability without the burden of relying on configuration.

Primary key conventions

Numerical Primary Keys

Each numerical primary key column is mapped with `@GeneratedValue` and `@Id` annotations.

Important

If your database does not support identity columns, you should create the sequence 'hibernate_sequence'. Please refer to Hibernate reference documentation for more advanced alternatives.

Primary Keys with 32 characters

By convention, for all primary keys that are char(32), Celerio maps the column with the following annotations

```
@GeneratedValue(generator = "strategy-uuid")
@GenericGenerator(name = "strategy-uuid", strategy = "uuid")
@Id
```

annotations. so it uses Hibernate's UUIDHexGenerator. Therefore no sequence is needed for these primary keys.

Other Primary Keys

For primary key that are char(x) where x is different from 32, Celerio map the column with an "assigned" generator, which means you must provide manually the primary key value.

The 'Account' table

The Account table is a special table that contains the user's login and password columns and eventually the email and enabled columns. It has an important role during the login phase. It is also used by the `AccountContext` generated class which store the current `account` information in the current thread.

Celerio detects automatically your 'Account' table. An account table candidate is expected to have at least the following columns:

Table 2.1. Account table conditions

Column's name	Mapped Java Type	Description
"username" OR "login" OR "user_name" OR "identifiant"	String	Login used by the end user to authenticate to this web application
"password" OR "pwd" OR "passwd" OR "mot_de_passe" OR "motdepasse"	String	Password (in clear) used by the end user to authenticate to this web application

If no Account table candidate is found, Celerio will do as if it had found one and will generate a mock Account DAO implementation that returns 2 dummy users (user/user and admin/admin) instead of generating a real JPA DAO implementation. It is up to you to replace this DAO implementation with your own implementation.

Note

You may also elect the account table by configuration.

The 'Role' table

The Role table is a special table that contains the account's roles. To be detected by Celerio, it must have a many-to-many or a many-to-one relationship with the found 'Account' table and contain the following mandatory column:

Table 2.2. Role table conditions

Column's name	Mapped Java Type	Description
"authority" OR "role_name" OR "role" OR "name_locale"	String	The generated code relies on the following authority's values: `ROLE_USER`, `ROLE_ADMIN`

Here is a sample SQL script (H2 Database) that complies to Celerio conventions

```
CREATE TABLE ACCOUNT (
    account_id char(32) not null, login
    varchar(255) not null,
    password varchar(255) not null,
    email varchar(255) not null,

    constraint account_unique_1 unique (login),
    constraint account_unique_2 unique (email),
    primary key (account_id)
);
CREATE TABLE ROLE (
    role_id smallint generated by default as identity,
    name_locale varchar(255) not null,

    constraint role_unique_1 unique (name_locale),
    primary key (role_id)
);
CREATE TABLE ACCOUNT_ROLE (
    account_id char(32) not null,
    role_id smallint not null,

    constraint account_role_fk_1 foreign key (account_id) references ACCOUNT,
    constraint account_role_fk_2 foreign key (role_id) references ROLE,
    primary key (account_id, role_id)
);
```

Other optional account's columns

The Email column

If the detected Account table has an email column, it is used by the generated code in few places, mostly as an illustration of the EmailService usage.

Table 2.3. Account's table email column conditions

Column's name	Mapped Java Type	Description
"email", "email_address", "email-Address", "mail"	String	The user's email.

The Enabled column

If the detected Account table has an enabled column, it is used by the generated code related to Spring

Security integration.

Table 2.4. Account's table enabled column conditions

Column's name	Mapped Java Type	Description
"enabled" OR "is_enabled" OR "isenabled"	Boolean	Only enabled users (enabled == true) can login.

Special columns for file handling support

When the following columns are present simultaneously in a table, Celerio generates various helper methods to ease file manipulation from the web tier to the persistence layer.

- 'prefix'_FILE_NAME (String)
- 'prefix'_CONTENT_TYPE (String)
- 'prefix_SIZE or 'prefix'_LENGTH (int)
- 'prefix'_BINARY (blob)

Example 2.3. Example of binary

```
mydoc_content_type       varchar(255)    not null,
mydoc_size               integer         not null,
mydoc_file_name          varchar(255)    not null,
mydoc_binary             bytea,
```

This convention will allow you to upload a file transparently, save it to the corresponding table, then download it using a simple URL.

ACCOUNT_ID column & Hibernate Filter

When a table contains a foreign key pointing to the Account table, Celerio assumes that the content of this table belongs to the user represented by the account_id foreign key.

An hibernate filter is configured to make sure that this table is loaded only by the current user.

The filter is enabled by the `HibernateFilterInterceptor`. Of course this default behavior may not always suit your needs. There are two ways of disabling it:

1. Remove the `@Filter` annotation from the Entity. This imply taking control over the entity.
2. Disable the filter programmatically using the HibernateFilterContext generated helper.
3. Disable globally this convention in Celerio's configuration file.

Version column and Optimistic Locking

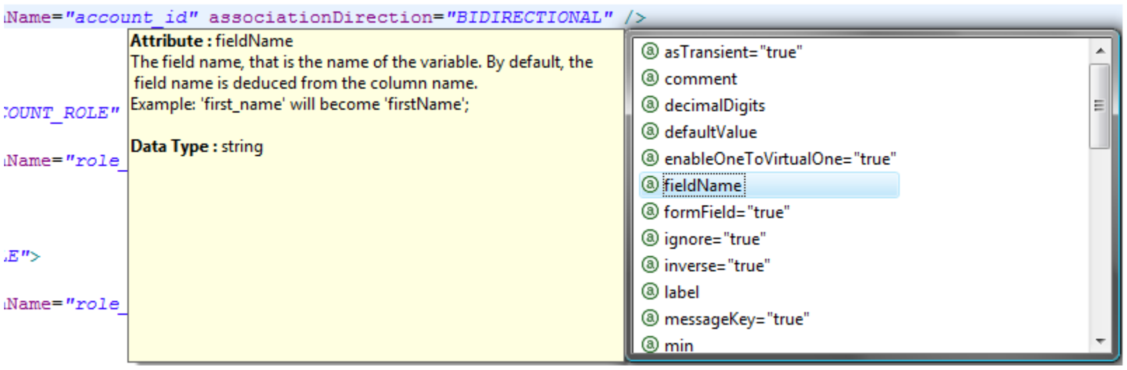
If your table has a column named `VERSION` whose type is an int, Celerio assumes by convention that you want to enable an optimistic locking strategy. As a result, the property is annotated with `@Version` .

Many to many and inverse attribute

Which side of the relation is marked as inverse="true" ? By convention, the side whose corresponding column's order is the highest on the "Middle table".

Chapter 3. Configuration

Before editing your configuration file, make sure that Eclipse displays the documentation present in the `celerio.xsd` file and that it suggests the available tags. From Eclipse, you cannot work efficiently without the help of the XSD documentation.



Attribute : fieldName
The field name, that is the name of the variable. By default, the field name is deduced from the column name.
Example: 'first_name' will become 'firstName';

Data Type : string

Tag completion and documentation under Eclipse

Id

If you rely on conventions, you do not need to configure anything regarding Ids. These examples are for advanced usage.

Use a SEQUENCE per TABLE

In case you use a sequence per table to generate your primary key values, you must configure Celerio in order to take it into account. Here is an example:

```
<entityConfigs>
  <entityConfig tableName="ADDRESS" sequenceName="ADDRESS_SEQ"/>
</entityConfigs>
```

assuming the PK of the ADDRESS table is mapped to an Integer, here is how would look the generated code:

```
@Column(name = "ADDRESS_ID", nullable = false, unique = true, precision = 5)
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "ADDRESS_SEQ")
@Id
@SequenceGenerator(name = "ADDRESS_SEQ", sequenceName = "ADDRESS_SEQ")
public Integer getAddressId() {
  return addressId;
}
```

Use a custom Id generator

In certain cases, generally when you work with legacy databases, you may need to use a custom Id generator in order to be consistent with the legacy system. Here is an example:

```
<entityConfig tableName="ADDRESS">
  <columnConfigs>
```

```
    <columnConfig columnName="ADDRESS_ID">
      <generatedValue generator="myCustomerGenerator" />
      <genericGenerator name="myCustomerGenerator" strategy="com.yourcompany.appli
        <parameters>
          <parameter name="sequence" value="YOUR_EVNTUAL_SEQ" />
        </parameters>
      </genericGenerator>
    </columnConfig>
  </columnConfigs>
</entityConfig>
```

leads to:

```


@Column(name = "ADDRESS_ID", nullable = false, unique = true, precision = 5)
@GeneratedValue(generator = "myCustomerGenerator")
@GenericGenerator(name = "myCustomerGenerator",
 strategy = "com.yourcompany.appli.customgen.CustomerGenerator",
 parameters = @Parameter(name = "sequence", value = "YOUR_EVNTUAL_SEQ"))
@Id
public Integer getAddressId() {
  return addressId;
}
```

Entity and property names

Force an entity name

By default, an entity name is deduced from the Table name. To force the entity name to a different value, use the `entityName` attribute of the `entityConfig` element, for example.

```
 <entityConfigs>
  <entityConfig tableName="ACCOUNT" entityName="UserAccount"/>
 </entityConfigs>
```

Force a property name

By default, a property name is deduced from the column name. To force the property name to a different value, use the `fieldName` attribute of the `columnConfig` element, for example.

```
<entityConfigs>
  <entityConfig tableName="ACCOUNT">
    <columnConfigs>
      <columnConfig columnName="user_dob" fieldName="birthDate"/>
    </columnConfigs>
  </entityConfig>
</entityConfigs>
```

leads to:

```
Date birthDate;
```

Advanced property name calculation

By default Celerio calculates Java field name based on the underlying column name. The `fieldNaming` element allows you to change the column names passed to Celerio to calculate the default field names. The example below removes well known prefix pattern from column names:

```
<configuration>
  <conventions>
    <fieldNaming regexp="^.{3}_{1}" replace=""/>
  </conventions>
</configuration>
```

In that case, column names such as `XYZ_SOMETHING_MEANINGFUL` now lead to Java field name `sometingMeaningful` instead of `xyzSometingMeaningful`.

Type Mapping

Celerio has some conventions regarding type mapping. You can change them either either locally or globally using rules

Force a type mapping locally

You can force the mapped type using the `mappedType` attribute of the columnConfig element. For example to force a mapping to an Integer you would do:

```
<entityConfigs>
  <entityConfig tableName="ACCOUNT">
    <columnConfigs>
      <columnConfig columnName="year" mappedType="M_INTEGER"/>
    </columnConfigs>
  </entityConfig>
 </entityConfigs>
```

Number mapping customization

You can configure number mapping rules globally. For example, to map all columns whose size and decimal digits are > 1 to BigDecimal, proceed as follow:

```
<configuration>
  <numberMappings>
    <numberMapping mappedType="M_BIGDECIMAL" columnSizeMin="1" columnDecimalDigits
  </numberMappings>
</configuration>
```

First rule that matches is used. For example to map to either Boolean, Double or BigDecimal you can do:

```
<configuration>
  <numberMappings>
    <numberMapping mappedType="M_BOOLEAN" columnSizeMin="1" columnSizeMax="2" colu
    <numberMapping mappedType="M_DOUBLE" columnSizeMin="1" columnSizeMax="11" colu
    <numberMapping mappedType="M_BIGDECIMAL" columnSizeMin="11" columnDecimalDigit
  </numberMappings>
</configuration>
```

Note that the `columnSizeMin` is inclusive and `columnSizeMax` is exclusive.

Date mapping customization

You can configure date mapping rules globally. For example, to map all date/time/timestamp column to Joda Time LocalDateTime, proceed as follow:

```
<configuration>
  <dateMappings>
    <dateMapping mappedType="M_JODA_LOCALDATETIME" />
  </dateMappings>
</configuration>
```

And for example to map differently the columns whose name is VERSION you can add the following mapping rule:

```
<configuration>
  <dateMappings>
    <dateMapping mappedType="M_UTILDATE" columnNameRegExp="VERSION"/>
    <dateMapping mappedType="M_JODA_LOCALDATETIME" />
  </dateMappings>
</configuration>
```

Associations

@ManyToOne

By default, Celerio generates the code for a `@ManyToOne` association when it encounters a column having a `foreign key` constraint and no `unique` constraint.

The variable name of the many to one association is deduced by default from the `fieldName` of the column. For example if the `fieldName` is `addressId` , the many to one variable name will be `address` . In case where the `fieldName` already matches the name of the target entity, Celerio adds the "Ref" suffix to the variable name. Here are few simplified examples:

```
// column name is 'addr_id'
@ManyToOne Address addr;
```

```
// column name is 'address'
@ManyToOne Address addressRef;
```

```
// column name is 'anything_else'
@ManyToOne Address address;
```

In any case, use the `manyToOneConfig` element to force a different variable name. For example:

```
<columnConfig columnName="addr_id">
    <manyToOneConfig var="myAddress"/>
</columnConfig>
```

will lead to

```
@ManyToOne Address myAddress;
```

The `manyToOneConfig` element also allows you to tune the JPA fetch type and the JPA cascade types. Please refer to the XSD for more information.

If you have some inheritance involved on the 'one' side of the many to one association, the table referenced by the foreign key is not enough to identify the target entity. In that case, set the `targetEntityName` attribute of the `columnConfig` element. For example:

```
<columnConfig columnName="address_id" targetEntityName="HomeAddress"/>
```

On legacy schema, the foreign key constraint may not be present and Celerio will not generate the many to one association you would expect. Hopefully you can configure Celerio to do as if a foreign key constraint was present by setting the `targetTableName` attribute of the `columnConfig` element. For example:

```
<columnConfig columnName="address_id" targetTableName="ADDRESS"/>
```

@OneToMany

One to many association is configured on the 'many' side of the association, more precisely on the same `columnConfig` as the one used for the associated many to one association. This may be a bit confusing at first but it has the advantage to group together, both associations on the side that really owns the association.

Celerio generates the code for one to many association when a many to one association is present and when the `associationDirection` attribute of the `columnConfig` element is `BIDIRECTIONAL` . For example:

```
<entityConfig tableName="Account">
    <columnConfig columnName="address_id"
                  associationDirection="BIDIRECTIONAL"/>
</entityConfig>
```

will lead (assuming address_id refers to Address) to something like:

```
// in Account.java
@ManyToOne Address address;
```

```
// In Address.java
@OneToMany List<Account> accounts;
```

In the example above `accounts` is simply the plural of the Account entity that Celerio guessed. We were of course lucky on this one.

Use the `oneToManyConfig` element of the `columnAttribute` to set the name of the one to many association to a different value. As you will see, you can also set the name of an element of the collection to control the name of the associated helper methods that Celerio generates (adder, remover, etc.). Here is an example:

```
<entityConfig tableName="Account">
    <columnConfig columnName="address_id"
                  associationDirection="BIDIRECTIONAL">
```

```
        <oneToManyConfig var="people" elementVar="resident"/>
    </columnConfig>
</entityConfig>
```

will lead to

```
// In Address.java
@OneToMany List<Account> people;

public void addResident(Account resident) {
// skip...
}
```

The `oneToManyConfig` element also allows you to tune the JPA fetch type and the JPA cascade types. Please refer to the XSD for more information.

@OneToOne

By default, Celerio generates the code for a `@OneToOne` association when it encounters a column having a `foreign key` constraint AND a `unique` constraint.

One to one associations are very similar to many to one associations. To change the variable name, the JPA fetch type or the cascade types of the one to one association, use the `oneToOne` element of the `columnConfig` element.

Inverse @OneToOne

Inverse one to one association is for one to one association what one to many association is for many to one association.

Celerio generates the code for inverse one to one association when a one to one association is present and when the `associationDirection` attribute of the `columnConfig` element is `BIDIRECTIONAL`.

Inverse one to one association is configured on the owning side of association, that is on the `columnConfig` that has the foreign key and unique constraints. As for one to many association, this may be a bit confusing at first but it has the advantage to group together, both associations on the side that really owns the association.

@ManyToMany

Many to many association necessarily involves a join table. When Celerio detects a join table, it generates the code for the many to many relation. Celerio assumes that a table is a join table when it has 2 foreign keys and no other columns, except eventually a primary key column and a column used for optimistic locking.

To fine tune the many to many association, you must declare an entityConfig for the join table. You may use the `manyToManyConfig` element. to set the related variables and adder/remover/etc. method names. You can use the `inverse` attribute to force the inverse side of the association. For example:

```
<entityConfig tableName="account_role" associationDirection="BIDIRECTIONAL">
 <columnConfigs>
  <columnConfig columnName="account_id">
   <manyToManyConfig var="theAccounts" elementVar="anAccount"/>
  </columnConfig>
  <columnConfig columnName="role_id" inverse="true">
   <manyToManyConfig var="theRoles" elementVar="aRole"/>
  </columnConfig>
 </columnConfigs>
</entityConfig>
```

Note

In case Celerio does not detect the join table, for example if an extra column is present, you can force it by setting to `true` the `middleTable` attribute of the `entityConfig` element.