

Ramecera™

Screwdriver application

UserGuide-3.5.1

October, 2020

Index

Screwdriver application.....	1
UserGuide-3.5.1.....	1
1 Preface.....	1
1.1 Graphic conventions.....	1
1.2 Tags used in this document.....	1
2 Getting Started.....	2
2.1 Install process.....	2
2.2 The ELE file.....	2
2.3 Let's go.....	3
2.3.1 An ELE file.....	4
2.3.2 A moment of discussion.....	4
2.4 Plugins (accessories) supported by Screwdriver.....	4
2.5 Java code conventions.....	5
2.6 Classes and paths having pre-established names.....	6
2.7 ".screwMaven" directory.....	7
2.8 "box" and "logs" directory.....	7
2.9 Screwdiver properties file.....	7
2.10 Processing the snippets.....	8
3 Processing the JSPs.....	8
3.1 Introduction.....	8
3.2 The markup dress+, method+ and dao+.....	9
3.3 Validation of data.....	10
3.3.1 Validation according to Hibernate.....	10
3.4 The crud+ markup.....	10
3.5 The formn+ markup.....	10
4 The DB process by Hibernate.....	10
4.1 A moment of discussion.....	11
5 Security.....	12
5.1 The Authentication Provider.....	12
5.2 The Authentication Manager.....	12
5.3 The group authorities: GBAC system.....	13
6 Perfective maintenance.....	13
6.1 The knife system.....	14
7 Java Source code style.....	15
7.1 Screwdriver java style.....	16

Your use of this text is governed by the terms of the end user license agreement EPL.
Author: Massimo Maglioni – October, 2020

See files TermsOfUse.txt and Eclipse Public License - Version 1.0.html attached to every distribution of Screwdriver application and to its Drill plug-in.

1 Preface

Screwdriver is now ready to work: it is a stand alone java-maven application, it includes all source codes and much more.

What's its purpose?

It is a Java source code generator, capable of building a Java web application characterized by the synergy of the Maven, Hibernate and Spring MVC frameworks. It is made up, at release 3.0.0, of approximately 23,000 lines. Its reduces development time, therefore increases developer productivity. Obviously he cannot create a complete application, he does not have a magic wand. But it does many things: for example, it sets all the @Controller classes and all the @RequestMapping methods: the developer has only to implement them. This is a dream.

How is it possible?

The developer must describe the application by means of a new language, the ELE language (short for "elementary", remembering sir Arthur Conan Doyle: "Elementary, my dear Watson"). Really it is an elementary language. Someone will say: but then you don't solve the problem, you move it. No, I move it but I reduce the difficulties by an order of magnitude. It is a step forward in reaching the goal that for many years has been considered only an utopia, the transition from the procedural paradigm to the declarative paradigm.

The generated application is not strange, it does not include proprietary software or particular structures or frameworks: it is a classic, good Spring / Hibernate / Maven application, so it can later be maintained without knowing absolutely anything about *Screwdriver*, which generated it.

Screwdriver still does not produce Spring Boot applications, it will be in future releases. But the "old" Spring 4 is still a giant in information technology, it is certainly not at sunset.

Screwdriver includes Spring Security. If I don't need a DB, logically Hibernate is absent.

1.1 Graphic conventions

blabel+ : the name of a markup, always followed by sign +

ELE line : listings of ELE files

Screwdriver: an assigned name

constant width : used for program listings as well as within paragraphs to refer to program elements such as variable or method names

classification: used for listings of object types

1.2 Tags used in this document

<nationalCode> example: <https://www.berlin.de/> national code is "de"

<myFirm> example: <https://www.ibm.com/us-en/> my firm is "ibm"

<myApp> example: <https://www.microsoft.com/de-de/windows/> my app is "windows"

<rail> the name of every rail is simply formed by the 4-character string "rail" followed by a number (start value=0) example: **rail0**, **rail1**, ecc.

2 Getting Started

2.1 Install process

Use the plug-in *Screwdriver* (internal name `it.ramecera.drill`), present in Eclipse Marketplace, to install *Screwdriver* project. You have to use Eclipse for Enterprise Java Developers, version not older than Mars.

By the install process, you obtain also an "Hello world"... anything but simple. It consists of

1. the *Screwdriver application*, an Eclipse project named `screwdriver-n.n.n`;
2. a JPA project named `waxds-copper`;
3. inside *Screwdriver*, the file `silver.ele`, in order to generate the web application *silver*; *silver* in turn uses the JPA project *waxds-copper*;
4. inside *waxds-copper*, the Data Definition Language files to obtain the DB tables used by *silver*: `createSchema.sql`, `dropCreateAll.sql`, `function_upd_dt.sql` (reference DBMS: Postgres 11).

Screwdriver uses a language composed of particular markups, all composed of alphanumeric characters, followed by the + sign and any data. Example:

datan+1953-09-12

(when writing a date, we will always use the XML standard: yyyy-MM-dd).

As already mentioned, *Screwdriver* generates web application supported by Spring, Hibernate and Maven.

The possibility of building a web application without Spring or without Hibernate is there, but for now it is only a case study for *Screwdriver*.

Using Hibernate, you must first create a separate JPA project, and have all the tables converted into entities automatically by Eclipse. I recommend using Data Source Explorer view, although some bugs are still present. I also recommend not building the JPA project separately and then copying the java package directly into the Eclipse Spring project, in the usual `model` directory; although this may seem convenient (at the time of the creation of the war there will be one less jar to insert), separating JPA projects we get a more modular software, and this is very important; make a little effort and learn how Maven handles deployments of linked modules.

2.2 The ELE file

The input code is a file with extension `.ele` (elementary¹), simply we call it **ELE file**, and contains plain text; each line, conventionally called *train*, is composed of an initial markup, called *locomotor*, optionally followed by one or more secondary markups, called *wagons*. Example:

```
requemap+series/dt+goal+reports/dt.jsp
```

1 Remembering Sherlock Holmes by sir Arthur Conan Doyle: "Elementary, my dear Watson!"

`requmap+` is the locomotor, `goal+` the wagon; note that in this case the + character is present 3 times, because the + sign is also the conjunction between the locomotor and the wagon.

Consequently, in a line (a train) the quantity of signs + is always odd.

The character # in position 1 indicates a comment line. Empty lines are allowed, they are used to make certain parts of the code stand out.

2.3 Let's go

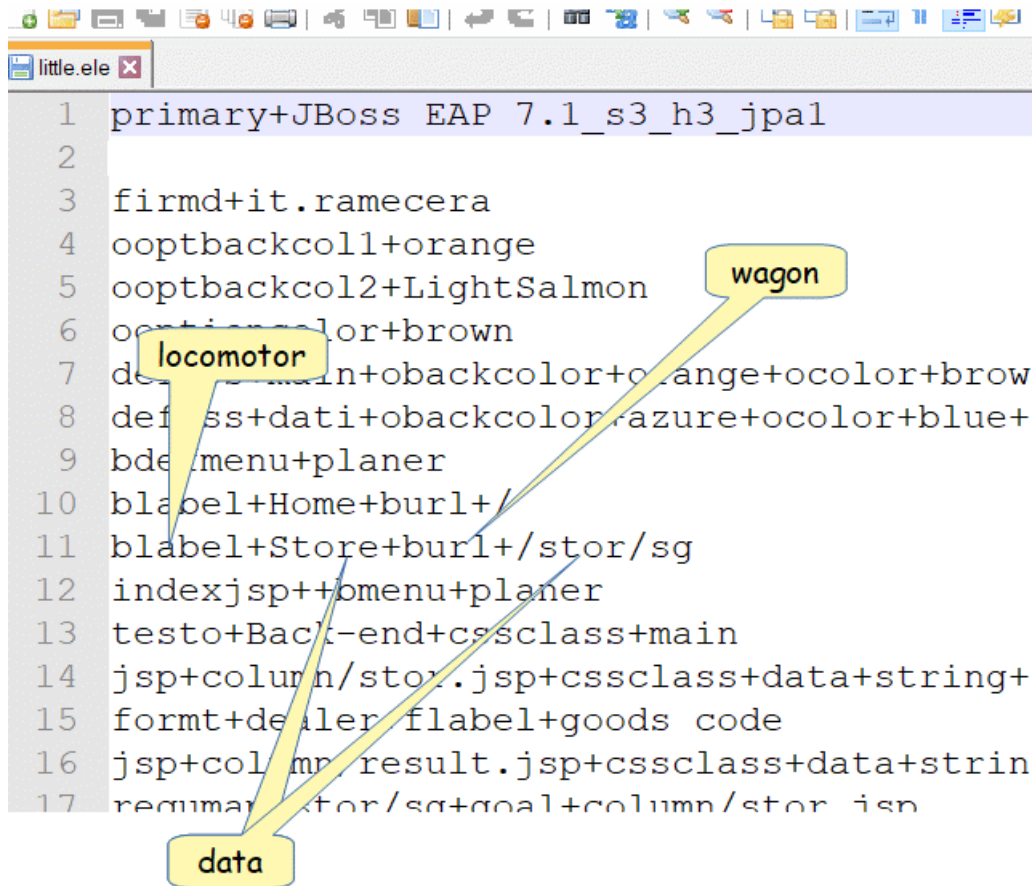
The procedure for creating the new project is as follows:

- 1) create the JPA project(s) using the Data Source Explorer view; it is necessary a distinct JPA project for every involved DB schema, not only for every DB; initially I advise not to use Maven, later yes.
- 2) prepare your file ELE: for instance, if your application is named *vanadium*, this file is `vanadium.ele` (you must use only lower case characters; actually the first proposed sample application is *silver*);
- 3) create an application property file: its name is `vanadium.properties`;
- 4) run *Screwdriver* to process your file ELE;
- 5) after passing the various tests, and the application ended successfully, hit the menu command **Screwdriver>Bring out the last generated application** (in subsequent modifications this command is no longer necessary because the project is already present in the Package Explorer; a refresh will suffice).

That's all! But you need do some other simple operation:

- 6) If you have set up a local AS, in **Properties>Java Build Path>Libraries** indicate it; proceed to the test phase later.
- 7) After achieving some stability, if a DB is accessed, the JPA projects can be mavenized, so this reference can be deleted, it becomes a POM dependency; anyway it is necessary to keep the JPA project in the workspace, in closed state, because Screwdriver, every time it starts, analyzes the JPA source code.
- 8) If you don't want mavenize JPA, no problem: the JARs corresponding to these JPA projects must be copied to `main/webapp/WEB-INF/lib`.
- 9) It is possible to build this JAR (double) clicking on `makeJar.jar`. In general, it is possible to create a JPA jar file through the command `File>Export>JAR file`, remembering to check the options "Export generated class files and resources" and "Export java source files and resources".
- 10) In the JPA project, the application server must be declared too; if the JPA project is already mavenized, as an alternative you can introduce a dependency `<groupId>javax.persistence</groupId>
<artifactId>javax.persistence-api</artifactId> <version>2.2</version>` in `pom.xml`; anyway follow the instructions in chap. 4.

2.3.1 An ELE file



The image shows a screenshot of a text editor window titled 'little.ele'. The window contains a list of 17 lines of text, each representing a configuration entry. Three yellow callout boxes with black text and arrows point to specific lines: 'locomotor' points to line 7, 'wagon' points to line 11, and 'data' points to line 17. The text in the editor is as follows:

```
1 primary+JBoss EAP 7.1_s3_h3_jpal
2
3 firmd+it.ramecera
4 ooptbackcoll+orange
5 ooptbackcol2+LightSalmon
6 ooptbackcolor+brown
7 de+main+obackcolor+orange+ocolor+brow
8 def+ss+dati+obackcolor+azure+ocolor+blue+
9 bde+menu+planer
10 blabel+Home+burl+/
11 blabel+Store+burl+/stor/sg
12 indexjsp++omenu+planer
13 testo+Back-end+cssclass+main
14 jsp+column/stor.jsp+cssclass+data+string+
15 formt+dealer flabel+goods code
16 jsp+column result.jsp+cssclass+data+strin
17 regumar+stor/sg+goal+column/stor isn
```

2.3.2 A moment of discussion

"Well, but who will help me write this ELE file?"

"You will do as all developers do: they take the source of an old application that looks like, and modify it. This method also applies to the ELE file".

"So where's the advantage?"

"The advantage is that the ELE file is much more synthetic: one modification of a single parameter causes a cascade of modifications in the whole application."

2.4 Plugins (accessories) supported by Screwdriver

In addition to Spring and Hibernate, the following accessories or frameworks are available:

- Jackson
- Apache Log4j
- Tiles
- Webflow
- Captcha
- i18n
- Antisamy
- LDAP
- Quartz Scheduler

Some of these software are, in a large part of the ICT world, considered obsolete. Also in some cases *Screwdriver* implements only older versions. It is not important, the core business is the Hibernate-Spring-Maven-Log4j load-bearing structure.

2.5 Java code conventions

According to *Screwdriver*'s philosophy, it is an unnecessary complication to give the developer the possibility to choose, always, any name of directories or files, according to his personal taste. *Screwdriver* establishes the names, according to customs now widespread. Many developers care a lot about this freedom, but the advantage given by *Screwdriver* is enormous, compared to this "humiliation".

This principle is one of the necessary conditions to fit the transition from the procedural paradigm to the declarative paradigm.

Being known by the world community of java developers, I do not mention here all the basic files and directories, however mandatory, such as `webapp/WEB-INF`, `webapp/META-INF`, `pom.xml` or `web.xml`.

So the following files have been made mandatory (of course, only if you decide to use that particular framework or functionality):

1. `src/main/webapp/WEB-INF/spring/DispatcherServlet.xml`
`src/main/webapp/WEB-INF/spring/<myApp>-security.xml`
2. `§§§/web/annotation/LoggedUser.java`
3. `§§§/security/Belonger.java` (the bean that represents the session user, in the context of security)
4. `src/main/webapp/WEB-INF/tiles/tiles.xml`
5. `§§§§/dao`, contains an interface for each table of the schema DB
6. `§§§§/dao/impl`, contains the implementation of the interface in the previous point
7. `§§§§/service`, contains an interface for each table of the schema DB
8. `§§§§/service/impl`, contains the implementation of the interface in the previous point

Where `§§§` indicates the basic package, consisting of 3 segments `<nationalCode>/<myFirm>/<myApp>`: example: `it.ramecera.vanadium`.

Furthermore `§§§§` indicates the basic package, plus a fourth element, the **rail**; the term **rail** is linked to the name of the JNDI-DBschema pair to use; example:

`it.ramecera.vanadium.rail2`: the application is called `vanadium`, it uses a DB, `rail` is fixed and mandatory, and 2 is the number of the JPA project I link to; every JNDI-DBschema couple must have its own JPA project.

Therefore the names of every rail is simply constituted by the 4-character string "rail" followed by a number (start value=0), and in this document they are indicated through `<rail>`: summing up: `§§§§ = <nationalCode>/<myFirm>/<myApp>/<rail>`.

The names of the interfaces corresponding to the individual tables respect the

`I<rail><table-name>Dao` (example: `IRail2CarsDao.java`)

and

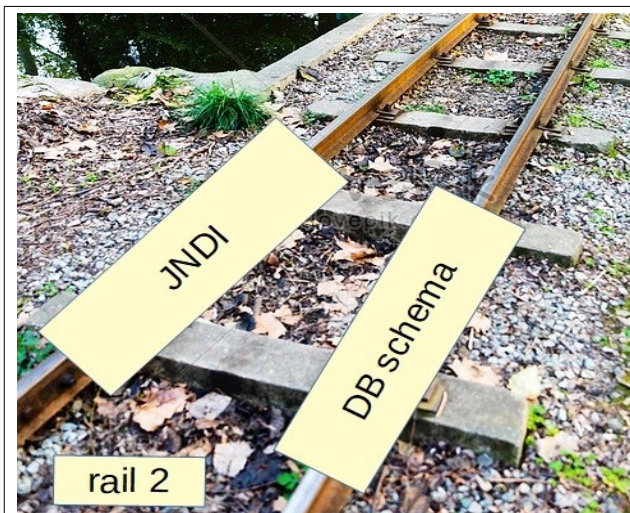
`I<rail><table-name>Service`

schemes, while their implementations respect the

`<rail><table-name>DaoImpl`

scheme and

`<rail><table-name>ServiceImpl`.



ELE line example:

```
hdefrail+2+hjndi+waxds+hdbschema+co
pper+hdbms+postgres
```

```
locomotor: hdefrail
```

```
wagons: hjndi, hdbschema, hdbms
```

```
rail: 2
```

Everything is automatically created by *Screwdriver* according to what is declared in ELE file.

Spring's @Controller type classes go inside `$$$/web/controller`.

2.6 Classes and paths having pre-established names

- ◆ **DoorControllerBelong**: located in `<nationalCode>/<myFirm>/<myapp>/web/controller`, it controls the `index.jsp` home page, it is generated by the presence of the `indexjsp+` markup.
- ◆ **FormsControllerBelong**: it is located in `<nationalCode>/<myFirm>/<myapp>/web/controller`, it is unique, and controls all JSPs made up of a form, to be filled in by the user.
- ◆ **Belonger**: found in `<nationalCode>/<myFirm>/<myapp>/security`, it is generated by the presence of the `uv+` markup.
- ◆ **<myJsp>Form**: it is located in `<nationalCode>/<myFirm>/<myapp>/web/model`, and `<myJsp>` indicates the name of each of the JSPs consisting of a form.

For web apps, directories have been made mandatory (logically, only if you decide to use that particular framework or functionality):

- ◆ `src/main/webapp/resources`
- ◆ `src/main/resources`
- ◆ `src/main/webapp/WEB-INF/views`
- ◆ `src/main/webapp/WEB-INF/views/answers`
- ◆ `src/main/webapp/WEB-INF/views/home`
- ◆ `src/main/webapp/WEB-INF/views/includes`
- ◆ `src/main/webapp/WEB-INF/spring`
- ◆ `src/main/webapp/WEB-INF/flows`
- ◆ `src/main/webapp/WEB-INF/tiles`

Spring's default resolver: **InternalResourceViewResolver** implies that all views extensions are `.jsp`; if not, it can still be adapted.

The data to be passed to each JSP not declared as a **form** (see below the classification of JSPs in **plain** and **form**) are always represented by the variable

`#{belong}`, and are passed through the `ModelAndView` constructor. It is mandatory to have no more than one transit variable, all data indeed can be packaged in a single bean for each JSP.

The name of the application is however converted to lowercase, and goes to dial:

- the 3rd qualifier of the package
- the id attribute of the `<web-app>` tag in `web.xml`
- the Artifact Id of Maven
- other less important parameters.

Unless you use Tiles, an include with `header.jsp` and `footer.jsp` is introduced in JSPs; these 2 JSPs are found in `src/main/webapp/WEB-INF/views/include`.

When the class implementing an interface is one, the interface name is always the name of the class preceded by the character I.

2.7 “.screwMaven” directory

The `.screwMaven` directory contains the file `.project` and `.classpath` to insert in each web application generated by Screwdriver.

2.8 “box” and “logs” directory

The `box` directory has the following subdirectories: `snippets`, `include`, `recycleBin`.

- **snippets**: code fragments, in different languages: *css, jsp, java, properties, xml*;
- **includes**: it collects all ELI file; an ELI is a variant of ELE file, it contains a group of predefined markups;
- **recycleBin**: files that are deleted by screwdriver in the evolutionary maintenance processes; preceding version of files, when they are updated by screwdriver.

The `logs` directory contains log files generated during *Screwdriver* processes; it also contains the `statistics.log` file, generated after each run and containing the statistics of the ELE markups used.

2.9 Screwdriver properties file

Screwdriver starts reading a properties file, whose name will be the name of the application to be built, followed by its natural extension: `<myapp>.properties`.

This file must contain the following labels:

- **input directory and output directory**: they are self-explanatory;
- **inside**: true or false, says if I decided to insert java code already inside the `.ele` file, immediately after the declaration of a method by markup `method+`; obviously it is recommended to do it only for very small methods, because doing so I mix java code and ELE code, not always appropriate.
- **freeze**: the list of files that, in a subsequent processing of the same application, must absolutely not be modified, obviously because you have added custom code you don't want to lose. While it is true that every time, in a maintenance, *Screwdriver* modifies an existing class, it asks the user for permission first; the `freeze` functionality is an additional guarantee.

Example:

```
input_directory= C:\notebook\metalEW\screwdriver
output_directory= C:\notebook\metalEW
inside=false
freeze=WoodTaskImpl.java, MatterControllerGroup.java, single.jsp
```

2.10 Processing the snippets

Screwdriver's snippets can be of type **css**, **java**, **JSP**, **properties**, **XML**, or simple **txt**, that is fragments that can be inserted in the most disparate places. In the **java** type, it represents an entire class, which is read by *Screwdriver*, and then modified in different ways: 1) in a complex way, an object of the abstract class FileComponent is created (example: **DoorControllerBelong.java**); 2) in a simple way, the snippet is an entire java class, first modified by the static method **FileComponent.addClass**, then inserted in the application during construction (example: **SimpleUsernamePasswordAuthenticationToken.java**).

3 Processing the JSPs

3.1 Introduction

Each **jsp+** markup defines a JSP by giving it a path and a name, relative to the mandatory position **src/main/webapp/WEB-INF/views**. The JSP name must begin through a lowercase letter, if uppercase it is automatically converted; even if they are in different directories, for an easier search in the file system, the overall set of JSPs must not have duplicates.

A JSP can be in 4 types: **form** type, **plain** type, **model** and **message** type (codes assigned respectively: 0, 1, 2, 9). The JSP **form** is mainly input, and is called through the **@ModelAttribute** annotation, with a method that returns a simple String; example:

```
@RequestMapping ("/create")
public String belong_createForm_2 (
    @ModelAttribute CreateForm createForm) {
    return "events/create";
}
```

the **plain** JSP is mainly output, and is called by the controller method through the **ModelAndView** class; example:

```
@RequestMapping ("/my")
public ModelAndView events_1 () {
    MyPlain myPlain = new MyPlain ();
    return new ModelAndView ("events/my", "belong", myPlain);
}
```

while the JSP model is reserved for viewing the profile of the user who is currently authenticated.

The `formt+`, `form+`, `formn+` and `formc+` markups following a JSP markup warn through their presence that it is a **form** JSP (i.e. screwdriver attributes the **form** type when there is at least 1 of the 4 possible markups), and declare the names of the controls that the user must fill in (`formt` = text-box, `forma` = text-area, `formn` = list-box, `formc` = check-box).

By declaring a subsequent JSP on the `.ele` file, I assure that the list of controls has ended.

JSPs are called by URL, and each URL is composed of a root, unique for the whole application (for example `www.hammer.ramecera.it`) followed by a variable suffix (for example `www.hammer.ramecera.it/events/create`); often this suffix coincides with the path/name of the JSP to be called (for example: `/events/create`, extension is not declared), but not necessarily. The URL-JSP association is obtained through the `goal+` wagon of the `requmap+` locomotor. Example:

```
requmap+matter/share+goal+events/create.jsp
```

If the JSP must have a navigation bar, this is called by the `bmenu+` wagon; this markup declares the name of the particular menu you want to insert into the JSP; the declaration of all the possible menus, obviously before their use, is made by the markup `bdefmenu+`.

For each JSP a bean is created, as an abstract class, in position

```
src/main/java/$$$/web/model/xxxxxxx/
```

where `$$$` has already been described and `xxxxxxx` is the name of the directory immediately below the `view` directory, which contains a group of JSPs. The properties that make up this bean are declared through `integer+`, `boolean+`, `string+` or `arraystring+` markups, and correspond to the data that will be shown on the web page.

3.2 The markup `dress+`, `method+` and `dao+`

Among the `jsp+` wagons we have `dress+`, which indicates the name of a method associated with the JSP bean (we mentioned this bean in a previous paragraph); the method is defined by the markup `method+`, which indicates the method name, followed by the `dao+` wagon, which indicates the list of DAO interfaces called in the method; it can also be followed by the `others+` wagon, which indicates the will to add other arguments to the signature of the method: it will simply be a list of classes, while the name of the object is conventionally **other** followed by a progressive. These methods are always abstracts, and just they require that the class also be defined as abstract. The implementation takes place in a subsequent class, in the same package, which takes the name of the previous class followed by the suffix **Imp1**. This method is designed to be called within a controller class (i.e. defined by a `@Controller` annotation), and this will provide the reference to those interfaces that are present in the method signature. These interfaces are all obtained through the `@Autowired` annotation, inside the controller class.

Nothing new: these are widespread practices in the Java-Spring developer community, but the burden is all on Screwdriver.

3.3 Validation of data

A distinction must be made between the validation according to Hibernate, activated by the `hvalid+` markup, and the one according to Spring, activated by `svalidator+`. Markups that implement validation start through the prefix `val`.

3.3.1 Validation according to Hibernate

This validation is actually not related to the presence of a database, so the hibernate name is purely conventional.

It is sufficient to declare, immediately after each `formt+` markup, one or more `valcons+` (validation constraints) markup. Among the `valcons+` wagons we have `valtag+`, which indicates the label corresponding to the message you want to get out. Elsewhere in the `.ele` file these labels must be defined by `vallabel+`, which in turn carries `vallang+` and `valmess+` along with it.

3.4 The crud+ markup

Among wagons of the locomotor `jsp+` we also have `crud+`. When the JSP is of the **form** type, it is optional to declare this wagon; it has the effect of creating 4 buttons corresponding to the 4 functions C, R, U, D, and setting up their controllers; the developer will then add the source code of the 4 corresponding methods, and each will return a ModelAndView object. The `crud+` wagon has no value, its presence means **true** and its absence **false**.

3.5 The formn+ markup

When the JSP is of the **form** type, a particular treatment is reserved for list-boxes (I could also call them combo-boxes, because in an HTML page usually the concepts of combo-box and list-box coincide).

The mandatory wagons are:

`flabel+`, specifies the explanatory text near the list-box;

The optional wagons are:

`fcartel+`, specifies the string that appears on the list-box when it is closed;

`hdefrail+`, a number indicating the DB-schema coupled to the JNDI;

`hsql+`, the query that fills the list-box;

`mybutton+`, the name of the button, placed next to the list-box, dedicated to sending the chosen DB table row online (since this button is missing, sending is operated by the overall button of the form).

4 The DB process by Hibernate

For each `hdbschema+` markup the developer has to:

a) create a JPA project in the same workspace, which will have as logical name the pair: JNDI-DBschema; the beans, or entities, corresponding to the DB tables, will be placed in a package named according to the following structure:

```
<domain>.screwdriver.<jndi>.<DBschema>.
```

For example, if through the `hjndi+` markup I defined a logical name `silverDS`, and the DB schema is `heritages`, the project name will be (it is recommended to use lower case) `silverds-heritages`, and the package

```
it.ramecera.screwdriver.silverds.heritages.
```

The presence of the `persistence.xml` file within the JPA project is very important, because the *Screwdriver* application searches for it and reads it to know the list of DB tables.

4.1 A moment of discussion

"Why all these rules? Wasn't it better when Spring allowed me to choose, and then to match everything up?"

"No, it wasn't better. Let us remember the humiliation speech that I have talked about: following transparent and equal rules for everyone, after all, is a great advantage, especially for colleagues who, after you, will have to maintain the code."

b) before creating the JPA project, make sure that the connection to the DB exists through the Eclipse Data Source Explorer view. Then create the JPA project using the **File>New>JPA Project** command; the recommended choices are (they vary with versions of Eclipse)

- Target Runtime: jre 1.8 or newer
- JPA version: 2.1 until Eclipse 2019-12; 2.2 from Eclipse 2020-03 on
- Basic JPA Configuration: Default configuration for..

then **Next>Next**: (we are arriving to "JPA facet" panel)

- Platform: Generic 2.1 until Eclipse 2019-12; Generic 2.2 from Eclipse 2020-03 on
- Disable Library Configuration
- Connection... the connection must recall a DB already configured in the Eclipse Data Source Explorer view, possibly through the use of filters to simplify the search for the various DB schemas and tables.
- Override default catalog for connection (only if the schema DB is not the default one)
- Override default schema for connection (only if the DB schema is not the default one)
- Discover annotated class automatically.

Finally **Finish**.

c) after creating the JPA project it is necessary to declare in Java Build Path what the application server will be, and the right JRE; then you will access the DB tables, through the connection in Data Source Explorer, opening the connection; then you can give the command **File>New>other>JPA>JPA Entities from Tables**. Check the option **List the generated classes in persistence.xml**. If Eclipse reports an error, check that the **Properties>JPA>Discover annotated classes automatically** option is active. For each table it is necessary to check the characteristics read, in particular the key generator and the foreign keys.

d) now you can convert it to Maven using the **Configure>Convert to Maven Project** command.

The Maven project obtained does not have, and must not have, the 2 usual `main/java` directories in `src`. In the Maven project you get the jar by running **Run as>Maven build...** and then assigning the value "package" to the text-box goals, then **Run as>Maven install**.

e) in Java Build Path of the project you are building, it is necessary to declare the link to this JPA project, and to insert the jar in the web-app library (in `webapp/WEB-INF/lib`). Better still is to insert the dependency towards the JPA jar into the POM of your web app.

Remember that the name of a DB table must not end through the string PK, and no column, including the primary key, can be called `id`: use `identifier`, or perhaps you can use `key`, or `id` as suffix, as `meta1_id` or similar (the latter prohibition is due to a bug present in Eclipse, knowing that the usual key name in Spring is just `id`).

For each `hdefrail+` markup, a file with the name `Generic<rail>DaoImpl.java` is also created, in the package `<domain><myapp><rail>`. In the previous paragraph Java code conventions (see) this sequence is indicated through the string `§§§§`. By `<myapp>` we mean the application that calls and uses the JPA project.

For example, if the domain is `it.ramecera`, the application `silver`, the rail number is `0`, inside the application we are building the file will be called `GenericRail0DaoImpl.java` and will be placed in the `it.ramecera.silver.rail0` package. It is an abstract class that implements the interface `it.ramecera.silver.generic.GenericCrud` up to Hibernate 4, `it.ramecera.silver.jpa.dao.GenericDao` from Hibernate 5 onwards.

5 Security

The XML file that manages Security takes the name `<myapp>_security.xml`, and is located in `webapp/WEB-INF/spring`.

5.1 The Authentication Provider

The authentication provider is identified by the `xprovider+` markup, which can take the following values: ***user-service***, ***class***, ***jdbc***, ***ldap***.

5.2 The Authentication Manager

In the case of `xprovider+class`, user recognition is managed by a special java class, which is indicated by the authentication-provider tag; the name of this class should not be specified, because it is mandatory: `<myapp>userAuthenticationProvider`. The `userpass+` markup has 3 possible values: ***simple***, ***domain*** and ***trivial***; ***trivial*** when the userid and password are specified in clear text in the `<authentication-manager>` tag; eg .:

```
<s:authentication-manager>
  <s:authentication-provider>
    <s:user-service>
```

```

        <s:user name="sardinia" password ="thyrsus" authorities ="ROLE_USER" />
        <s:user name="admin" password ="flumendosa" authorities ="ROLE_ADMIN" />
    </s:user-service>
</s:authentication-provider>
</s:authentication-manager>

```

All this is discussed in Screwdriver in `it.ramecera.screwdriver.component.SpringSecurity_xml.java` class, `synthesis` method for creating the XML file, and in `it.ramecera.screwdriver.component.UserAuthenticationProvider_java.java` class, `synthesis` method for creating the java source.

5.3 The group authorities: GBAC system

In applications with Spring Security you will need a DB schema containing certain tables for user authentication. However, you can create the tables that concern user groups (GBAC system), which will remain unused if you do not wish to make use of group-authorities.

The name of the schema DB and the tables are free, but we recommend:

- SCREWXGROUP

for the DB schema; and

- MGROUP_AUTHORITIES
- MUSERS
- MGROUPS
- MGROUP_MEMBERS
- MUSER_AUTHORITIES

for tables; DDLs are found in the `gbacCreateTables.sql` and `usersCreateTables.sql` files within the Eclipse screwdriver project.

The JPA project must be called through the same name as the schema DB, for example SCREWXGROUP, and the sources must be in the package

`<domain><myapp>.jpa.<dbschema>`. Tag `<myapp>` refers to the application name, or a collective name of multiple applications, that make use of JPA.

So nothing changes substantially, compared to the other schema DBs. Once the tables are created, exactly follow the instructions for JPA projects described in the previous chapter.

6 Perfective maintenance

After adding Java code to the classes that were generated by Screwdriver, sooner or later I will have to change the structure of the application, for example because I added a new table to the database. I will modify the ELE file, but by Screwdriver's subsequent processing the updated classes will cover the old ones, and I will lose the code added by me. This problem is solved or mitigated by acting on several fronts:

- 1) the most important is the knife system;
- 2) *Screwdriver* never changes the classes found in the `web/model` directory – and subdirectories – whose name ends by the string "Impl" (we refer to the classes,

connected to each JSP, whose purpose is to prepare the data to display); no notice is produced to warn of this foresight;

3) all files that the developer wishes are not altered by *Screwdriver* can be declared in the external XML properties file, under the "freeze" label; they can be not only *.java but also *.properties, *.jsp, *.css and *.xml; no warning is produced to warn of this foresight;

4) whenever *Screwdriver* is about to replace a file, because the new code is different from the old one, the application asks the developer for permission, who can answer Yes or No. If he answers Yes, however the old file is not lost, because it ends in the Screwdriver recycle bin (**box/recycleBin**).

6.1 The knife system

type	generated code	definitive code
java	<code>/* screwdriver_knife */</code>	<code>/* screwdriver_upper */</code> <code>...</code> <code>/* screwdriver_lower */</code>
css	<code>/* screwdriver_knife */</code>	<code>/* screwdriver_upper */</code> <code>...</code> <code>/* screwdriver_lower */</code>
properties	<code># screwdriver_knife</code>	<code># screwdriver_upper</code> <code>...</code> <code># screwdriver_lower</code>
xml	<code><!-- screwdriver_knife --></code>	<code><!-- screwdriver_upper →</code> <code>...</code> <code><!-- screwdriver_lower --></code>
jsp	<code><!-- screwdriver_knife --></code>	<code><!-- screwdriver_upper →</code> <code>...</code> <code><!-- screwdriver_lower --></code>

The "knife system" is a particular type of merge. The files generated by *Screwdriver* can contain one or more knives from the list present in the "generated code" column; in a final transformation they become those present in the "definitive code" column; That 3 dots represent the code that the developer will insert later.

When I edit the ELE file, the produced file will still have the same number of knives; the new file will replace the old one, but everything that was written within the **upper_screwdriver / lower_screwdriver** pair will be reported in the new file absolutely safe. This is an example for the java code case:

internal generated code:

```
producedMaterial = Strings.replaceFirst(producedMaterial, "+importForm+",
impForm);
s += Candies.LF + Candies.TABS + "@RequestMapping(value = \"/screw" + rm.jsp
+ "\", method = RequestMethod.POST)" + Candies.LF;
```



```
/* knife_screwdriver */
s += Candies.TABS + "public String compile" + rm.jsp+ "(";
```

generated new code:

```
producedMaterial = Strings.replaceFirst(producedMaterial, "+importForm+",
impForm);
s += Candies.LF + Candies.TABS + "@RequestMapping(value = \"/screw" + rm.jsp
+ "\", method = RequestMethod.POST)" + Candies.LF;
/* upper_screwdriver */
/* lower_screwdriver */
s += Candies.TABS + "public String compile" + rm.jsp+ "(";
```

pre-existent, different old code:

```
producedMaterial = Strings.replaceFirst(producedMaterial, "+importForm+",
impForm);
compact = newLocation(a, b);
/* upper_screwdriver */
newIssue = "pdf " + document;
s += Candies.TABS + newIssue + Candies.LF;
/* lower_screwdriver */
s += "public String compile" + rm.jsp+ "(";
```

final result:

```
producedMaterial = Strings.replaceFirst(producedMaterial, "+importForm+",
impForm);
s += Candies.LF + Candies.TABS + "@RequestMapping(value = \"/screw" + rm.jsp
+ "\", method = RequestMethod.POST)" + Candies.LF;
/* upper_screwdriver */
newIssue = "pdf " + document;
s += Candies.TABS + newIssue + Candies.LF;
/* lower_screwdriver */
s += Candies.TABS + "public String compile" + rm.jsp+ "(";
```

as you can see, the 2 lines have been transferred to the new code.

In other words, the two lines `/* upper_screwdriver */` `/* lower_screwdriver */` limit a free zone where the developer can write knowing that his code will not be lost.

In this particular merge, special treatment is reserved for `import` declaration lines. For these lines a trivial merge occurs, in other words the imports add up. The end result is an excess of imports, which however do not cause particular problems and can be easily adjusted using the Eclipse editor features (for example **Organize imports**).

7 Java Source code style

Source code is formatted by file `Leoncavallo.xml`, according to Eclipse functionality
Window>Preferences>Java>Code style>Formatter
and

Window>Preferences>Java>Editor>Save Actions

The file `Leoncavallo.xml` is present in the Eclipse project *Screwdriver*.

7.1 Screwdriver java style

Every developer has the right to have his own style, but when working together you have to agree to use code written in a style that doesn't match yours.

Screwdriver code uses camelCase; right curly bracket are placed on a new line.

Not all beans present the set and get methods; they are present only if it was necessary to insert business logic into any of them; on the other hand, inserting them later through the Eclipse functionality is very easy.

Moreover, Screwdriver code uses some capital letters at the end of the word to indicate the role of a property or variable. They are:

Q: the quantity of elements, int

E: an array, map, list or collection

J: the index in a loop, may be int or long

M: the maximum value in an array

S: the result of a cast from int to String

W: the penultimate element in a sequence

example: charQ = quantity of a particular character in a text

Special care has been used in keeping the scope of each variable, method and class to a minimum.