Jarman Manual v0.1

Serhii Riznychuk

Contents

1 About

This book contain whole knowlage about what is Jarman by the client, user, buisness side. Explaining what purpose and how to develop this product.

2 DevOps

2.1 Making installers

2.1.1 TODO Linux

Currently Trashpanda Team not support any insallation distribution on linux machines.

2.1.2 Windows

Executable installator in windows are maked by two different program with dedicated configurations, in three steps

1. First of all you create jarman-client executable by running `lein uberjar` with default profile

2. The seconds you ivoke `launch4j` which wrapp, created in directory `target/uberjar/`, jar into exe, and add some jre configurations, parameters, guards(for exampel you get info dialog when jdk not been found on your computer) etc.

3. And the last step finished by `Inno Setup` which defined in confugred in `installer` folder, finally build `jarman-setup.exe`.

1. Launch4j Launch4j is tool for wrapping `.jar` into windows `.exe` executable file. Whole configuraion, you can find in path `<project-root>\jarman\installer\launch4j.x` Remember, program support only *absolute paths*.

   You can build executable by runnig dedicated CLI toolkit

   ```
   & launch4jc.exe <project-root>\jarman\installer\launch4j.xml
   ```

   or altirnatively go into `jarman/jarman` project path, and in powershell run

   ```
   lein launch4j
   ```

   plugin automaticaly build executable and put it into `installer` directory.

2. Inno Setup Program help automatically create setup processing gui wizard. Mainly all needed configuration are declared in `installer/setup-client.iss`. That's file define what, where will be installed. To understading more, just open the file. After program finished to work, file `jarman-setup.exe` will be created in `inno-target` folder.

   ```
   & ISCC.exe <project-root>\jarman\installer\setup-client.iss
   ```

3. Post install file structure

   Jarman files dividing logicaly as "something that is just program" and "all user data, configuration, plugins".

   - `C:\\P..  F..\\Jarman\\` - is main folder where `Jarman.exe`, and related binary or another application must be located.
   - `${USERHOME}\.jarman.d` - directory must contain all needed plugin, configurations, backups and any file dumps.
   - `${USERHOME}\.jarman` - central configuration file, which help to set some behavior pattern for jarman
   - `${USERHOME}\.jarman.data.clj` - in this file we define all configuraiton to database that up whole jarman buisness logic infrastructure.

3 Metadata

3.1 Cause

While you try to say about "*something* or *stuff* must be dynamically generated" the main keyword is "dynamically". When you try to do some automatically build GUI or metadata logistic, some algorythm require a data, which in any scope discribe some "logic" for render or logic for composition something. Indeed, jarman don't understand how and what they must build, even that things discribe in #CREATEREF view.clj. Plugins, core jarman app need information about system tables, any other aspects that discribe one or anoter. For example name of table $user_{accountsystem}$ in database also must have some not ugly name for user, which will work with that table by the client side. Also we need to *inform* user about permission to editing this table, or specifying for system if is table is something internal or it's buisness logic. Jarman system require avoid those information and allow to use it in any of cases, to get information about environtment, used tables, fields, refernces etc. Take and look closelly by the example how to declare one.

3.2 Structure

Please open #CREATEREF .jarman.data.clj, which contain declaration of metadata for all/partial mentioned tables in database. Metadata not require to cover whole db, is just list of tables which you prefare to managing from the program client side. In this file you can find variable `metadata-list` with declared metadata to db tables in one long list. Please notice that metadata cannot be created for tables like "metadata" or another bad names, that not allowed, or currently use for system. #LANG warning abount bad naming you can see in runtime. All declarated and validated #CREATEREF metadata pushed into `metadata` table name in database.

One metatable entity describes on top level table name and serialized properties, for example `{:id 412 :table "cache" :props "{}"}`, where `:table` describe 1:1 table name, and `:props` properties used for building UI and db-logic to program.

Properties 'prop' data

```
{:id 1
 :table "cache_register"
 :prop {:table {:field :cache_register
:representatoin "user"
:description nil
:is-system? false
:is-linker? false
:allow-modifing? true
:allow-deleting? true
:allow-linking?  true}
:columns [{:field :id_point_of_sale
   :field-qualified :cache_register.id_point_of_sale
   :representation "Cache register"
   :description "This table used for some bla-bla"
   :component-type ["l"]
```

```
   :default-value nil
   :column-type [:bigint-20-unsigned]
   :private? false
   :editable? false}
  {:field "name"
   :field-qualified ...
   :representation "name" ...}...]
:columns-composite [{:field :url-site
     ...
     :constructor map->Url
     :columns
     [{:field :site-url
       :constructor-var :label}
      {:field :site-label
       :constructor-var :url}]}]}}
```

Deserialized ‘prop‘(look above) contain specially meta for whole table behavior and some selected column(not for all, in this version, only column ’id’ hasn’t self meta info).

Short meta description for table:

- `:representation` - is name of table which was viewed by user. By default it equal to table name.
- `:is-linker?` - specifing table which created to bind other table with has N to N relations to other.
- `:is-system?` - mark this table as system table.
- `:allow-modifing?` - if it false, program not allowe to extending or reducing column count. Only for UI.
- `:allow-modifing?` - if true, permit user to modify of column specyfication(adding, removing, changing type)
- `:allow-linking?` - if true, than GUI must give user posible way to adding relation this data table to other.

Short meta description for columns

- `:field` - database column name.
- `:field-qualified` - table-dot-field notation.
- `:representation` - name for end-user. By default equal to `:field`.
- `:description` - some description information, used for UI.
- `:column-type` - database type of column.
- `:private?` - true if column must be hided for user UI.
- `:editable?` - true if column editable
- `:component-type` - influed by column-type key, contain list of keys, which describe some hint to representation information by UI. All of this types place in variable ‘*meta-column-type-list*‘

3.3 Metadata

Module generate metainformation for database tables(but exclude metainformation for table defined in '*meta-rules*' variable. All metadata must be saving in 'METATABLE' database table.

One metatable entity describes on top level table name and serialized properties, for example `{:id 412 :table "cache" :props "{}"}`, where `:table` describe 1:1 table name, and `:props` properties used for building UI and db-logic to program.

Properties 'prop' data

```
{:id 1
 :table "cache_register"
 :prop {:table {:field :cache_register
:representatoin "user"
:description nil
:is-system? false
:is-linker? false
:allow-modifing? true
:allow-deleting? true
:allow-linking?  true}
:columns [{:field :id_point_of_sale
   :field-qualified :cache_register.id_point_of_sale
   :representation "Cache register"
   :description "This table used for some bla-bla"
   :component-type ["l"]
   :default-value nil
   :column-type [:bigint-20-unsigned]
   :private? false
   :editable? false}
  {:field "name"
   :field-qualified ...
   :representation "name" ...}...]
:columns-composite [{:field :url-site
     ...
     :constructor map->Url
     :columns
     [{:field :site-url
       :constructor-var :label}
      {:field :site-label
       :constructor-var :url}]}]}}
```

Deserialized 'prop'(look above) contain specially meta for whole table behavior and some selected column(not for all, in this version, only column 'id' hasn't self meta info).

Short meta description for table:

- `:representation` - is name of table which was viewed by user. By default it equal to table name.
- `:is-linker?` - specifing table which created to bind other table with has N to N relations to other.

- `:is-system?` - mark this table as system table.
- `:allow-modifing?` - if it false, program not allowe to extending or reducing column count. Only for UI.
- `:allow-modifing?` - if true, permit user to modify of column specyfication(adding, removing, changing type)
- `:allow-linking?` - if true, than GUI must give user posible way to adding relation this data table to other.

Short meta description for columns

- `:field` - database column name.
- `:field-qualified` - table-dot-field notation.
- `:representation` - name for end-user. By default equal to `:field`.
- `:description` - some description information, used for UI.
- `:column-type` - database type of column.
- `:private?` - true if column must be hided for user UI.
- `:editable?` - true if column editable
- `:component-type` - influed by column-type key, contain list of keys, which describe some hint to representation information by UI. All of this types place in variable ‘*meta-column-type-list*‘

3.4 TODO FAQ

I want change column-type (not component-type)?

- Then user must delete column and create new to replace it

I want change component-type for gui must be realized "type-converter" field rule, for example you can make string from data, but not in reverse direction.

- This library no detected column-type changes.

3.5 TODO Composite fields concept

3.5.1 *Problem*

Mainly database tables has flatt column structure, except the NoSQL tables, where one column may represent whole datastrcutre. Jarman realize metadata mechanism which help resolve problem with undestanding types of each columns for internal frontend toolkit, but also allow grouping component in aggregation entityes. That mechanism called `Composite columns`. That type of columns allow creating some "groups" of fields, which finally would pack in some Components.

How does it work? Take a look on simple(not technical) case(realized in `table.clj`):

1. When you invoking SQL (select! :table$_{\text{name}}$:user ...)

2. You get data vector like that

```
[{:user.login "user"
  :user.password "1234"
  :user.aaaa "1"
  :user.bbbb "2"
  :user.ccccc "1"}...]
```

3. After, you use meta for building Editable View (right side of =table.clj). You get columns from metadata like belove

```
[{{:field :user.login :column-type }
  {:field :user.password :column-type ...}
  {:field :user.aaaa... }
  {:field :user.bbbb... }
  {:field :user.cccc... }}]
```

4. But now i little bit change structure of metadata, and add new type of columns is `Composite colums`, this will has some grouped columns

```
(defrecord SomeRecord [a b c])

[{{:field :user.login }
  {:field :user.password }
  {:field :UNION
   :constructor SomeRecord
   :columns
   [{:field :user.aaaa... }
    {:field :user.bbbb... }
    {:field :user.cccc... }]}}]
```

   In this example we see, that all repeat char-name columns now in section `:UNION`. Those section just logically group 1+ columns in big columns category.

5. It's simple, just like you have `[1 1 2 2 3 3 3]` vector, and you want group it by logical value, and you get `[[1 1] [2 2] [3 3 3]]`. This field also contain Constructor, - and that certain kill-feature, which allow group(or better say wrapp) in some defrecord, and remapp one fealds to others. For example you can group data, to mapp all your need into some Aggregative component, which is much more better to wrapping, and passing istead of some map with fealds. For Example you have columsn `{:ftp_login "1" :ftp_password "2"}`, but more comformtable way to managment is converting to some rerecord `(FTPRecord "1" "2")`. New metadata allow make grouping and ungrouping from flatt columsn to componet and from componetns to columns.

```
Record field names
:user.cccc --------+
```

```
:user.bbbb -----+  |
:user.aaaa --+  |  |  ;; take raw data and create componet from it
     |  |  |
SomeRecord.  a  b  c  ; <= send those type to GUI componetn => GUI component
     |  |  |
:user.cccc --+  |  |  ;; converting back to the raw params
:user.aaaa -----+  |
:user.bbbb --------+
:user.login "user"     ;; also adding rest k-v
:user.password "1234"  ;; also adding rest k-v
```

3.5.2 How to use it?

First of all you need some agregation component

```
(defrecord Url [label url])
```

Now define metadata for user, where user have extra url field's.

```
{:id nil, :table_name "user",
 :prop
 {:table (table :field :user :representation "User"),
  :columns
  [(field :field :login :field-qualified :user.login :component-type [:text])
   (field :field :password :field-qualified :user.password :component-type [:text])
   (field :field :first_name :field-qualified :user.first_name :component-type [:text])
   (field :field :last_name :field-qualified :user.last_name :component-type [:text])
   (field-link :field-qualified :user.id_permission :component-type [:link]
       :foreign-keys [{:id_permission :permission} {:delete :cascade, :update :cascade}] :
  :columns-composite
  [{:field :user-site-url
    :field-qualified :user.user-site-url
    :component-type [:url]
    :constructor map->Url
    :columns [{:field :profile-label,
       :field-qualified :user.profile-label,
       :constructor-var :label
       :component-type [:text],
       :default-value "Domain"}
      {:field :profile-url,
       :field-qualified :user.profile-url,
       :constructor-var :url
       :component-type [:text],
       :default-value "https://localhost/temporary"}]}]
```

Please take a look on `:columns-composite` key section. Those section discribe *Composite columns*.

```
{:field :user-site-url
 :field-qualified :user.user-site-url
 :component-type [:url]
 :constructor map->Url
```

```
 :columns [{:field :profile-label,
    :field-qualified :user.profile-label,
    :constructor-var :label
    :component-type [:text],
    :default-value "Domain"}
   {:field :profile-url,
    :field-qualified :user.profile-url,
    :constructor-var :url
    :component-type [:text],
    :default-value "https://localhost/temporary"}]}
```

Composite columns has your own keyword syntax, as in simple fields, but also addtional keywords

- `:constructor` - in this key you specify constructor which create some Object instanse from mapped colums discribed in `:columns` section.
- `:columns` - is simple standart field, which have additional `:constructor-var` key.

For example *Url* have two fields *url* and *label* and you must specify which columns are mapped into the specifyc column in *record* field `:user.profile-label` put into `label` in defrecord `URL`

```
      (URL. label url) ----> {:user.user-site-url #URL{"Domain", "https://.."}}
      |    |
:user.profile-label ----------+    |
:user.profile-url ----------------+
```

Builded component are menaged by the key `field.quialified`, specified in declaration of composite column.

4 Development

4.1 Codebase

Jarman code base is single repository, with all libraries, toolkits. Any cli toolkits also any managment code partially embed into jarman, this information are important by the servicing side, when we have some cli tool which can fix some problem to some version of client app was installed. Also that mean that new version jarman not would contain some legasy or any backward compatibility. If client app has 1.2.3 version, then toolkit you will fix some issues must have also 1.2.3 release version.

4.1.1 Git Flow

Code, documentation and any things which has relation to jarman or main environment, also an project files as a documentation must be on one git repository, without any division on submodules.

Version explaining for central GUI client.

```
1.2.3
^ ^ ^-- patch
| +---- minor release
+------ major release
```

Every major release pushed on **dedicated branch**, with name "release-<number>". Major release mean "new architecture" or "new logical" concept, which require some buisness environment restriction. Minor release mainly touch some additional feature, and good practise for this feature is making branch, but is not required, and relate do programmer comfort. Patch updates it's some fix for platform or any visual changes, that must be updated by user. You can create your branch per feature. Jarman haven't any ticketing system or etc. Every development changes pushed directly to the **master** branch. All changes must be noted in file `changelog.org`

What important! You cannot increasing version jarman cli or jarman-box. All changes must be declared as changes for jarman central gui app, and pushed into the main *changelog.org* file. Also not allowed making small patches only for cli applications. Argued this bahavior the same code base, which is not divided on libraries.