FullRelation Nano.H5

Download on sourceforge
Start Nano.H5 through WebStart

Project Members:

Thomas Schneider (admin)

<meta-tag>
crud, grud, crud2gui, crud2html, bean2html, entity2html, bean2gui, entity2gui, jpa2gui, jpa2html, jpa persistence provider, openjpa, hibernate, datanucleus, eclipselink, toplink, batoo, ormlite, ebean, data-editor, data-sheet, entity browser, jpa2, full stack framework, orm, o/r mapper, projector
</meta-tag>

<description>
crud, grud, crud2gui, crud2html, bean2html, entity2html, bean2gui, entity2gui, jpa2gui, jpa2html, jpa persistence provider, openjpa, hibernate, datanucleus, eclipselink, toplink, batoo, ormlite, ebean, data-editor, data-sheet, entity browser, jpa2, full stack framework, orm, o/r mapper, projector
</description>

Introduction

NanoH5 (or FullRelation) is an UI independent gui implementation framework providing a model driven design (MDA) and following the projector pattern. It is bound to the app framework **tsl2.nano.commons** and the jpa-service framework**tsl2.nano.serviceaccess**. It is possible to build a complete html5 application through a given class- or database-model. An Html5 presentation layer is provided as default.

- Everything will be filled for you by defaults - presenting a full application getting a database connection through any persistence provider (jpa 2.x)
- define it or implement it - all object-types have their representation as readable xml-file.
- ...or just use it as intelligent bean-browser or entity-browser

To do a *quick-start*, go to chapter *Starting / Test*.

A cool way to see the power of this tool can be seen in chapter *Online Quick-Start with a great selection of Database-Models on PonyORM*.

What does it for you?

- it starts a generic application with default mechanisms and configurations to present data (->jpa) in html5
- it provides all needed libraries for you - independent of the database and jpa-persistence tool you use
- it creates all jpa-entities for you (of course, you can use your own!)
- all (jpa) entities are configured and presented in a most reasonable way - but easily to be enhanced by you
- data is searchable and editable in a comfortable way - can be prepared to be printed and exported
- it provides several working modes: single standalone app, network/multi-user and/or access to an application-server

Technical Goals

- pure model implementation + platform independence (works on android, too).
- small, having as less as possible static dependencies to other libraries
- everything has a default - all defaults are configurable (Convention over Configuration)
- application, session and entity behaviors are configurable
- implementation through simple java beans + optional bean presenters
- you develop ui independent, but are able to use ui dependent extensions.

- no new language to learn. knowing html5 allows to improve layout and styling.
- navigates through given beans, showing bean-lists and bean-detail dialogs.
- resolves all bean/entity relations to be browsed.
- navigation can be a configurable workflow - or simply an entity browser
- pre-defines formatting, validation and presentation of relations
- pure html-5 (no javascript, only html5-scripting-api for websockts)
- using websockets to show status messages, input-assist and dependency field or timer refreshing
- pure jpa - jpa-annotations are read to resolve attribute-presentation
- independent of a special o/r mapper. all o/r mappers supporting javax.persistence with an EntityManager are usable.
- simple database replication on user-loaded data - offline working possible
- full key-navigation (shortcuts)
- framework designs interfaces and provides extendable implementations
- useable as standalone or web-service (with offline access), can connect to application-server or works standalone.
- many features through nano.common, and nano.incubation like a rule, sql, action engine, network executor etc.
- resolves all external (jdbc-drivers, etc.) jar-dependencies on runtime on an available network connection
- handling blobs of data (byte[]) to be presented in html as picture, audio, video or object.
- providing attributes as virtuals (not bound to a real bean-attribute, rule-expressions, sql-query-expressions and RESTful-service-call-expressions
- automatic translations through network connection
- planned interfaces:
  - xsd-->bean
  - java-interface-->java-bean (mock through internal proxy)

Using the *NanoHTTPD* Server as base, this client application creates html surfaces, sending them through the integrated server to an html browser. Entry point is the file *application.html* defining the browser request *http://localhost:8067* (configurable ;-)).

It is not a real web-application platform but a simple way to use html5 as graphical user interface - in a standard client application.

The base framework is tsl2.nano.common. It is a full stack framework as simple and generic as possible - without dependencies to other libraries (except: simple-xml).

The data access is done by:
tsl2.nano.serviceaccess
tsl2.nano.directaccess

It is possible to use an ejb container in an application server, but the default is set to use jpa (through any persistence provider) directly on the client (using *tsl2nano.directaccess*).

Why?

The base frameworks are grown through input of two industrial projects. the first project had to build a software fully configurable through a database. The second was a financial project.

tested project-environments:
windows-xp, windows-7, windows-terminal-server, ubuntu 12
oracle 11, hsqldb
glassfish 2.1, toplink
jboss-eap 6.1, hibernate

What is it for?

this software should provide a fast way to create a standard application through a database (or class-) model. through a complete set of configuration possibilities, a user may fit this application for his requirements. respecting a small set of rules, a software-developer is able to extend that application to do more specifics.

It is a full stack framework, but you can use it for other simple use cases:

- as an entity/database browser or editor
- as an JPQL or SQL statement tester with the integrated *ScriptTool*
- as a test base to check your model on different jpa persistence providers like *hibernate*, *elcipselink*, *openjpa*, *datanucleus* or *batoo*. you can switch between them with three mouseclicks.
- as a quick solution to generate your entity beans through the given database connection and to test that model.
- as a test-system, using an origin beans-jar-file to rebuild database on your local system on any database - transferring needed data through replication. all done in background for you.

...all available without any application server!

What this framework is not intended to be

- the web-application mode is not designed or tested for big data transfers or high network traffics.
- at the moment it is on construction - no guarantee can be given for any function or feature.
- no particular value is done for graphical design.

Usable modes

- Client/Server application
- Web application for small user-groups
- Standalone or through connection to an application server
- Application, started through a rest service in a web container (e.g. in jboss:http://localhost:8080/tsl2.nano.h5.0.8.0/web/start/user.home/free.port)

- with or without local replication database
- Usable as Entity-Browser configuring your data
- Usable as full-configurable application
- Usable as full-stack framework to develop model-driven applications
- Provides a JNLP file for Java Web Start

The *environment.xml* in your current environment directory defines access-modes. F.e. *http.connection* will define, if it is remote-accessible or not. For more informations, see chapter *The Environment*.

Provided Start Packages

This text is included in the README.txt of the nano.h5 download directory

package description:
- *tsl2.nano.h5.<version-number>.jar*: base "nano.h5" application. needs internet access on first use to download jdbc-drivers, ant and a jpa implementation.
- *tsl2.nano.h5.<version-number>-standalone.jar*: includes base "nano.h5" application with ant, jdbc-drivers for hsqldb and mysql libraries to work without internet access.
- *tsl2.nano.h5.<version-number>-signed.jar*: same as *tsl2.nano.h5.<version-number>-standalone.jar* but with signed content to be accessable through webstart (*nano.h5.jnlp*)
- *nano.h5.jnlp*: java webstart descriptor to start *tsl2.nano.h5.<version-number>-signed.jar*. will create its environment directory in its download directory.
- *sishell.<version-number>.jar*: "Structured Input Shell", a terminal application as toolbox for configurations, start scripts and administration. It is also integrated in *tsl2.nano.h5.<version-number>.jar*, but without starting support.

Commandline arguments

start parameters:

```
java [-Denv.user.home=true] -jar tsl2.nano.h5.<version>.jar [environment-path (default: .nanoh5.environment] [http-server-port (default: 8067)]
```

This call is implemented inside the *run.bat* script. Use that, if you are on windows. But normally, nano-h5 should be runnable through a double-click on the jar file *tsl2.nano.h5.<version-number>.jar*.

If you start it on *Windows*, a browser will be opened to show the initial screen. On other systems you should open an html-browser with file <environment-path>/temp/*application.html* or directly with: *http://<hostname>:<port>* (default: http://localhost:8067).

Start parameter can be given in several ways. The main arguments will be concatenated through the following rule:
1. META-INF/MANIFEST.MF:Main-Arguments
2. System.getProperties("apploader.args")
3. main(args) the real command line arguments

Third Party Libraries

Direct dependencies exist to the following libararies (contained inside the *tsl2.nano.h5.x.x.x.jar*):
SimpleXML xml serializer
Serviceaccess using: jpa-api, connector-api, jta-api, ejb-api, interceptor-api

The following library was a helper to generate an xsd-file for the beandef-xml files.
* Jing-Trang xml2xsd-Generator

For more indirect dependencies, have a look at chapter *Dependencies*.

JPA Persistence-Providers

At the moment, tested with JPA 2.0 providers, the following providers are known to implement*javax.persistence.spi.PersistenceProvider*:

- [Hibernate]
  : *org.hibernate.ejb.HibernatePersistence*
- [OpenJPA]
  : http://openjpa.apache.org: *org.apache.openjpa.persistence.PersistenceProviderImpl*. For further informations on reverse engineering see http://openjpa.apache.org/builds/2.3.0/apidocs/ or http://openjpa.apache.org/builds/2.2.1/apache-openjpa/docs/ref_guide_pc_reverse.html.
- [EclipseLink]: http://www.eclipse.org/eclipselink *org.eclipse.persistence.jpa.PersistenceProvider*
- [TopLink]
  : *oracle.toplink.essentials.PersistenceProvider*
- [Acme Pers.]: *com.acme.persistence*
- [DataNucleus]: http://www.datanucleus.org/ *org.datanucleus.api.jpa.PersistenceProviderImpl*
- [Batoo-JPA]
  : http://batoo.org *org.batoo.jpa.core.BatooPersistenceProvider*

Other smaller frameworks provide only access to JPA-Annotations without providing an EntityManager and JP-QL:

- [ORMLite]: http://ormlite.com *de.tsl2.nano.ormliteprovider.EntityManager*
  - only field annotations, no OneToMany

- [EBean]
  : http://www.avaje.org *de.tsl2.nano.ebeanprovider.EntityManager*

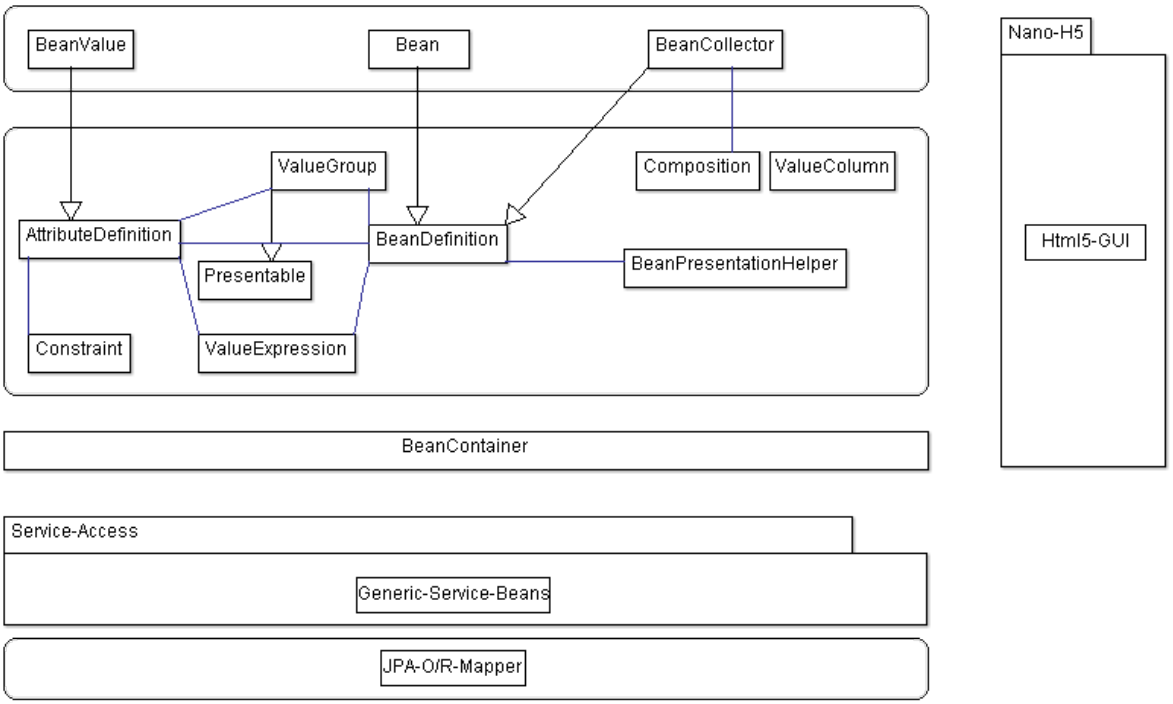
this ORM tools don't implement a jpa-persistenc-provider, but the libraries *tsl2.nano.ormliteprovider* and *de.tsl2.nano.ebeanprovider*extend the puristic base implementation of *NanoEntityManagerFactory*. Only native-SQL (no JP-QL) is available! As *tsl2.nano* works on JP-QL, some simple transformations to

native SQL are done by that EntityManager. These ORM-Tools are provided to have lightweight alternatives for mobile systems like Android.

A summary of lightweight ORM tools can be read here: ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/FACH-0147/FACH-0147.pdf

Architecture

The architecture is defined by the application framework tsl2.nano.common. The data and service layer is defined bytsl2.nano.serviceaccess.



The base for the model is any JPA-provider, the presentation of entity-beans will be configured through xml (using simple-xml as xml-persister and prefilling the beans with usable defaults). The presentation follows the projector pattern.

Model Driven Architecture (MDA))

Nano.h5 provides mechanisms to create a full configurable application from a given database definition file (ddl).

Creating an UML-Diagram with perhaps *ArgoUML*, or creating an ER-Diagram with f.e. *architect* you may generate a ddl (database-definition) script. Nano.H5 provides an ant-script (*mda.xml*) to generate an hsqldb-database generating entity beans through hibernate-tools or openjpa (script: *reverse-eng.xml*) for the given jdbc-connection. The ant-scripts use hibernate-tools or openjpa since there are no similar possibilities on other frameworks at the moment. We prefer the reverse engineering on *openjpa* because there are lots of options to be set.

The other option would be to have an existing *beans.jar*. This could be selected on the entry- or login-page of nano-h5. If you enter an absolute path for the attribute *JarFile* inside the *Persistence* page, this bean-file will be used to load the entity-classes. If you enter a relative path and the given file doesn't exist, it will be generated through hibernate-tools (if hibernate is in your environment!).

Process Description

1. perhaps create a new datamodel (use a tool like *ArgoUML* or *architect* to create an UML-Diagram) and export the resulting ddl-file to your new environment directory of nano.h5. the filename must end with .sql
2. start the nano.h5 application
3. input your database connection url and user/password (if you don't change anything, the **anyway** sample databae will be created and used
4. if you want a specific jpa-provider or generation tool, change that in the bottom panel
5. click the *OK* Button to start the first process

Now, the following will be done:

- nano.h5 downloads maven (~6MB) to load the following libraries:
- jdbc-driver (~1MB)
- ant (~2MB)
- the generator tool *hibernate-tools* (12MB) or *openjpa* (6MB)
- if the database-url points to a local database and it is an hsqldb or h2, the database will be downloaded, started and created throuth the given ddl-scripts (*.sql)
- if the given beans jar-file wasn't found, it will be created through the selected generator.
- now you see a list of available entities to work on

Framework mechanisms - how to extend the framework

What is a Bean?

In java a bean is a type/class defined by attributes (=properties). These attributes are readable through getter-methods ('get' + attribute-name) and changeable through setter methods ('set' + attribute-name).

Nano.H5 tries to change coppling between beans and attributes. A bean, defined by it's compiled class can have additional attributes to be presented. JPA annotations are interpreted to constrain the java-bean attributes and an additional presentation interface defines the layout of beans and attributes.

Where to get java entity beans from

Having a database connection and/or a ddl

If you don't give a beans-jar-file, nano.h5 tries to create the beans for you - using hibernate-tools or openjpa.

Having data stored in xml files

If you have xml files but no xsd schema definitions, create them through trang.jar. Download trang.jar, open a command line and go into the directory holding the data xml files:

```
java -jar trang.jar .* <myxsdfilename>
```

With an xsd file you are able to generate java classes through jaxb, which is integrated in oracles java implementation. the command:

```
xjc -d <src-base-directory> <xsd-schema-file>
```

will generate the desired java classes into *src-base_directory*.

Which bean attributes should be presented on default

If you connect/login to a persistence-layer/database for the first time, the framework tries to evaluate all bean-properties and attribute-properties. the order of attributes of each bean will be calculated through the implementation of*BeanPresentationHelper.getBestAttributeOrder()*. The best attribute to present the bean is done through*BeanPresentationHelper.getBestAttributeName()*.
This methods are called only if *bean.use.beanpresentationhelper.filter* is true - and use the following properties of *environment.xml*:

- bean.best.attribute.type (default: String.class)
- bean.best.attribute.regexp (default:.*(name|bezeichnung|description|id).*)
- bean.best.attribute.minlength (default: 2)
- bean.best.attribute.maxlength (default: 99)

All formats, constraints and basic presentation-properties will be evaluated through the type and jpa-annotations of an attribute.

Rules on evaluating the best order
1. attr.id() || attr.unique()
2. !attr.nullable()
3. !attr.isRelation()
4. bean.best.attribute.type (default: String.class)
5. bean.best.attribute.regexp (default:.*(name|bezeichnung|description|id).*)
6.a. bean.best.attribute.minlength (default: 2)
6.b. bean.best.attribute.maxlength (default: 99)
7. attr.getConstraint().getPrecision() > 0
8. attr.getType().isInterface()
|| (!BeanUtil.isStandardType(attr.getType()) && !BeanClass.hasDefaultConstructor(attr.getType()))
|| (!attr.isVirtual() && isGeneratedValue(bean.getDeclaringClass(), names[i]))
9. attr.isMultiValue() //always the multivalues at the end!

The found best attribute will be checked against an available database to be unique inside the current table-data.

Configuration through Serialization

The environment and the bean-definitions are the main constructs serializing it's properties to the file-system.
...
The *BeanDefinition* expects only *AttributeDefinition_s from deserializings - to be more readable. This means, that extensions of AttributeDefinition are not handled!*
...
On the data side, all collections are wrapped into an object of type _BeanCollector, single instances that are prepared to be presented are wrapped into an object of type *Bean*. All attributes of a *Bean* are wrapped into *BeanValue_s*. *The _BeanCollector* and the *Bean* are extensions of *BeanDefinition*, handling the attributes. While the used xml-serializer *Simple-XML* is not able to create the desired root instance through reading an xml (we have to define the instance type on calling simple-xml), extensions of a*BeanDefinition* are handled through a special mechanism, implemented in the class *Extension*.

Tip: If you delete your environment.xml file, it will be re-created on next restart - all beandefinitions will be re-created, too.

Registered services and definitions through the environment

The following services are registered through nano.h5 by default and should be acessed through the *Environment* class, if you are developing own classes or extensions:

- *IAuthorization*: provides access to check user privileges (e.g.: hasAccess(userName, actions))
- *XmlUtil*: provides de-/serialization (default: simple-xml), xpath, velocity-generation (if found in environment!)
- *Messages*: Resource-bundles for translations
- *IGenericService*: access to entity-beans
- *CompatibilityLayer*: indirect access to extern libararies (using reflection),

- *ClassLoader*: nano.h5 current classloader, containing all jars in your environment, including nested jars
- *UncaughtExceptionHandler*: handling/provding exceptions and messages
- *IPageBuilder*: implementation of the html5-building (here: *Html5Presentation* as extension of *BeanPresentationHelper*)
- *Profiler*: simple profiler to log simple performance aspects

The following service may be set to override the default mechanisms:
IConnector: defines the login/authentication. On default, the NanoH5 main instance is providing that through an Persistence instance.
Workflow: defines the navigation stack. On default a simple entity browser is used.
IGenericService: generic service to modify/query beans from
EntityManager: will be invoked into the GenericeService implementation. On default, the *javax.persistence* will be used by calling*Persistence.createEntityManagerFactory(persistenceUnitName).createEntityManager()*. If the desired orm framework doesn't provide a javax.persistence implementation, a simple *EntityManager* implementation can be set (see project *tsl2.nano.directaccess*class tsl2.nano.persistence.provider.NanoEntityManagerFactory).

Starting / Test

Quick-Start

- install a java jdk at least version 1.7. If you only install a java jre-version, the bean-jar file cannot be generated - then you have to select an existing one.
- get the newest version of tsl2.nano.h5
- start this jar file with *java -jar tsl2.nano.h5.[version-number].jar*. this will create the environments *nanoh5.environment* and a*run.bat*.
- go into the extracted sub-directory *sample* and start the *Windows*-batch file *runServer.bat*. This will start the sample database.
- start the *run.bat*
- if you are not on a *Windows* system, start your internet browser with adress *localhost:8067*, or just do a double-click on the file*application.html*
- In the center of the browser-page do a click on the centered link, starting with text 'Start...'
- Do a login filling the user-name 'time' and password 'time' and clicking the Ok-Button.
- After a while you will see all availabe entity-types to be browsable...

Online Quick-Start with a great selection of Database-Models on PonyORM

There are some online database-model creation/using provider like PonyORM (ponyorm.com) and ERDPlus (erdplus.com). This may be a fast possibility to see different database models working in tsl2.nano.h5.

Since most UML-model designers will generate a DDL (database definition language) script which may use a specific database dialect, we try to use HSqldb as local database server with compatibility modes switched on. See *Database Providers and Dialects*for more informations.

Here is a short description how to use a model from PonyORM and starting a full database-application with web-start on that model:

- be sure to use NOT java 8, as there the jnlp couldn't be started without a real certification!
- open the link https://editor.ponyorm.com/user/pony/OnlineStore in your browser
- select a datbase dialect on the top tab panel and click on it (prefered: oracle)
- click the button *select all*
- copy the selection to the clipboard using *Ctrl+C* or simply with the context menu of a right-mouse-click on the selection
- go to the tsl2.nano.h5 documentation page https://sourceforge.net/p/tsl2nano/wiki/Home/ and click on the link *Start Nano.H5 through WebStart* http://sourceforge.net/projects/tsl2nano/files/0.8.0-beta/nano.h5.jnlp
- after a while, Nano.H5 will open a page in your browser. click on the centered link of that page
- expand the detail panel
- click into the field *Database*
- paste the text from clipboard with Ctrl+V or simply using the context menu of a right-mouse click on the field
- click on the *OK* Button at the bottom
- after some minutes, Nano.H5 should show a list of available beans/tables.
- open a type, create a new bean and click 'configure' to configure the presentation of this bean type.

Known Problems on HsqlDB compatibility mode or DDL scripts provided by PonyORM

- mysql : AUTO_INCREMENT must be before PRIMARY KEY
- postgres: BYTEA <-- unknown
- oracle : Triggers with :NEW

Overview

The tsl2.nano.h5 framework can be started through it's jar *tsl2.nano.h5-xxxxx.jar*. A start script *run.bat* (will be generated on first start) is available to do this in *Windows*. Starting it, a given directory is used as a kind of workspace where you put all configuration and jar files into - to be used. This jars may be ant, an o/r-mapper like hibernate with all it's dependencies. The configuration files are*the environment.xml* and all xml files describing the presentation of each entity bean. Icons for all buttons and backgrounds are in the*icons* folder. The main jar file can contain all dependent jar files (as described in the manifest file) or outside in the same directory as the main jar.

On first start, the most important files will be created!

Normally, you will start with an example: download the file *tsl2.nano.h5.x.y.z-hibernate.zip* to have an example using lots of features of this project - you have a simple test database and are able to generate your bean jar, see a trivial application-extension, workflow, autorization and rule.

Feel free to test, whether a database-connection of a project you know is working with nano.h5....

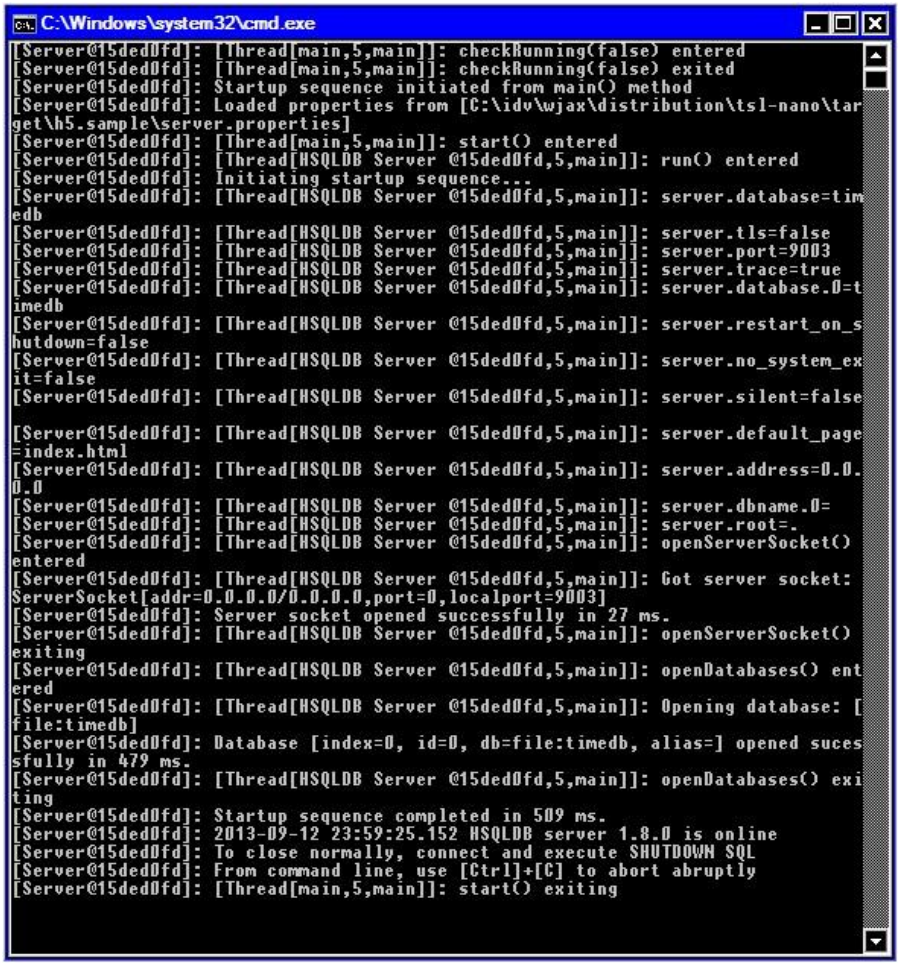
Using the sample-content

A sample environment is contained in the *tsl2.nano.h5.x.y.z.jar* file, containing all icons, jars and configurations for a project. It may be used for other nano.h5 projects. It uses:

- ant libraries to generate entitiy beans through hibernate-tools and the sample database
- hibernate with all dependencies as o/r mapper
- hibernate-tools
- hsqldb.jar as jdbc driver for a hsqldb database
- sample databae *timedb*
- sample icons for all buttons

Before you start *nano.h5*, you should start the sample hsqldb database:

config/runServer.bat

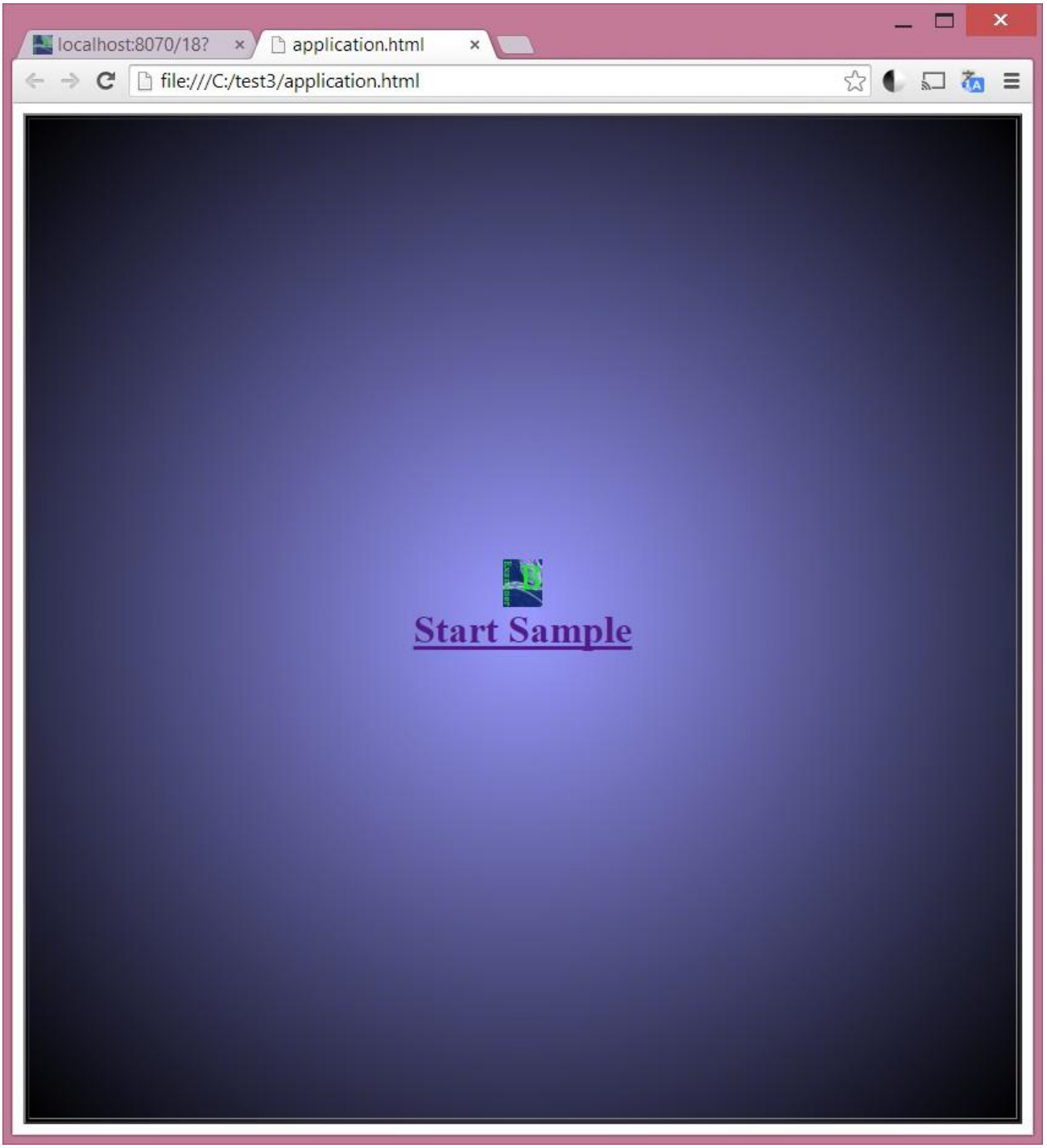


To start nano.h5 you have to call it with following syntax:

```
java -jar tsl2.nano.h5.x.y.z.jar [environment-path (default: .nanoh5.environment | env.user.home] [http-server-port (default: 8067)]
```

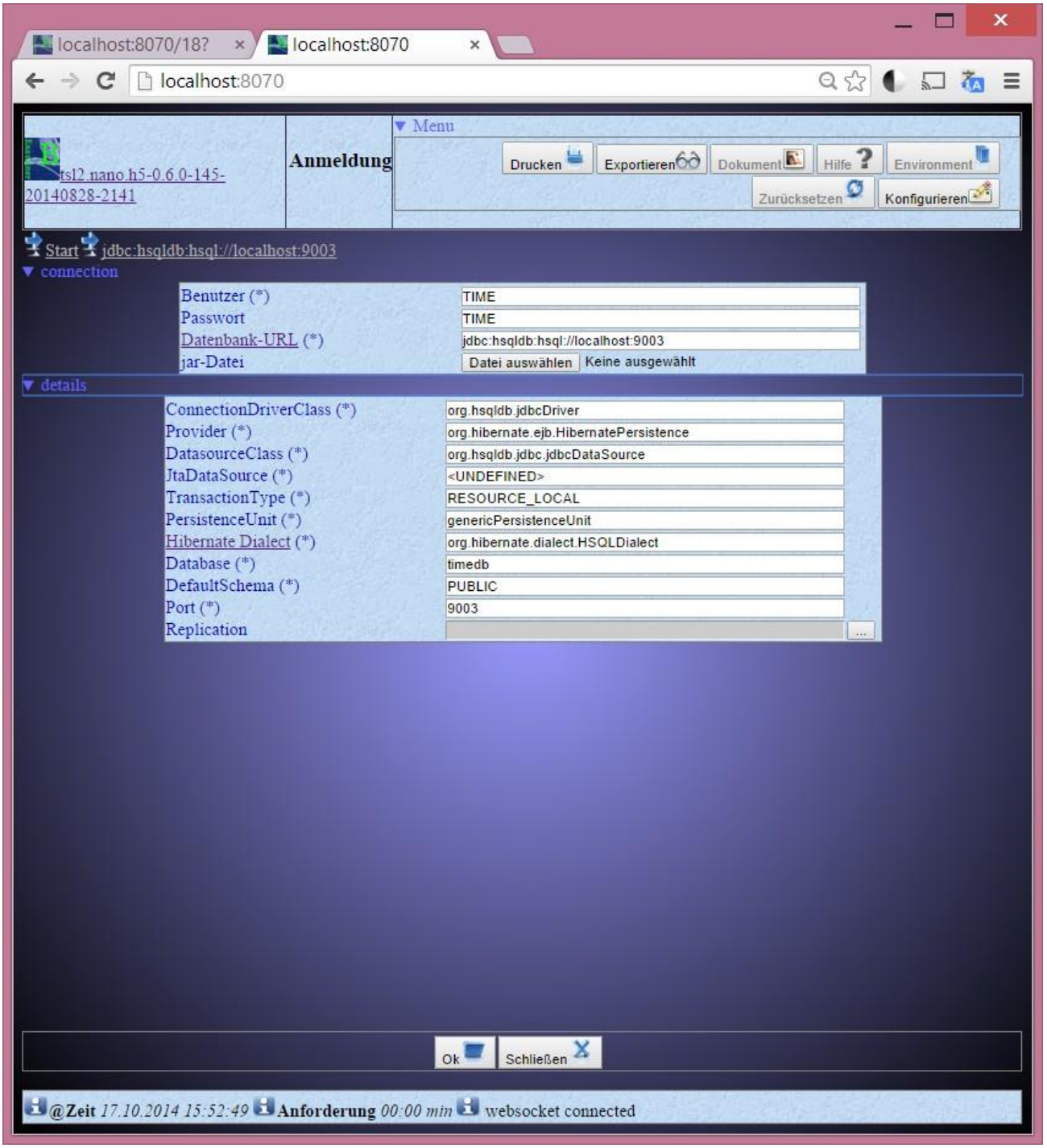
This call is implemented inside the *run.bat* script. Use that, if you are on windows. But normally, nano-h5 should be runnable through a double-click on the jar file *tsl2.nano.h5.x.y.z.jar*.

If you start it on *Windows*, a browser will be opened to show the initial screen:

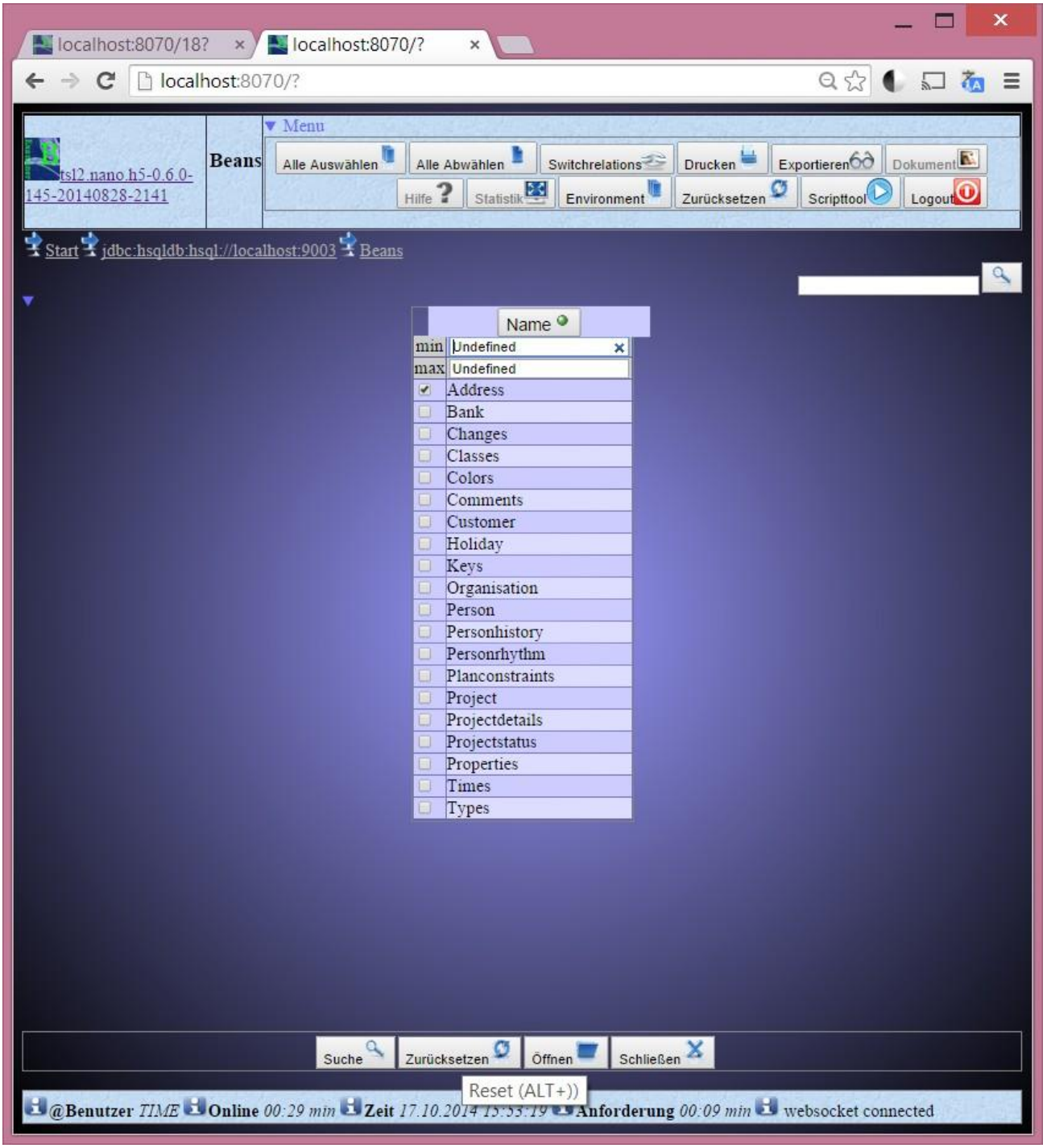


If you are not on *Windows*, you should open an html-browser with file *application.html*.

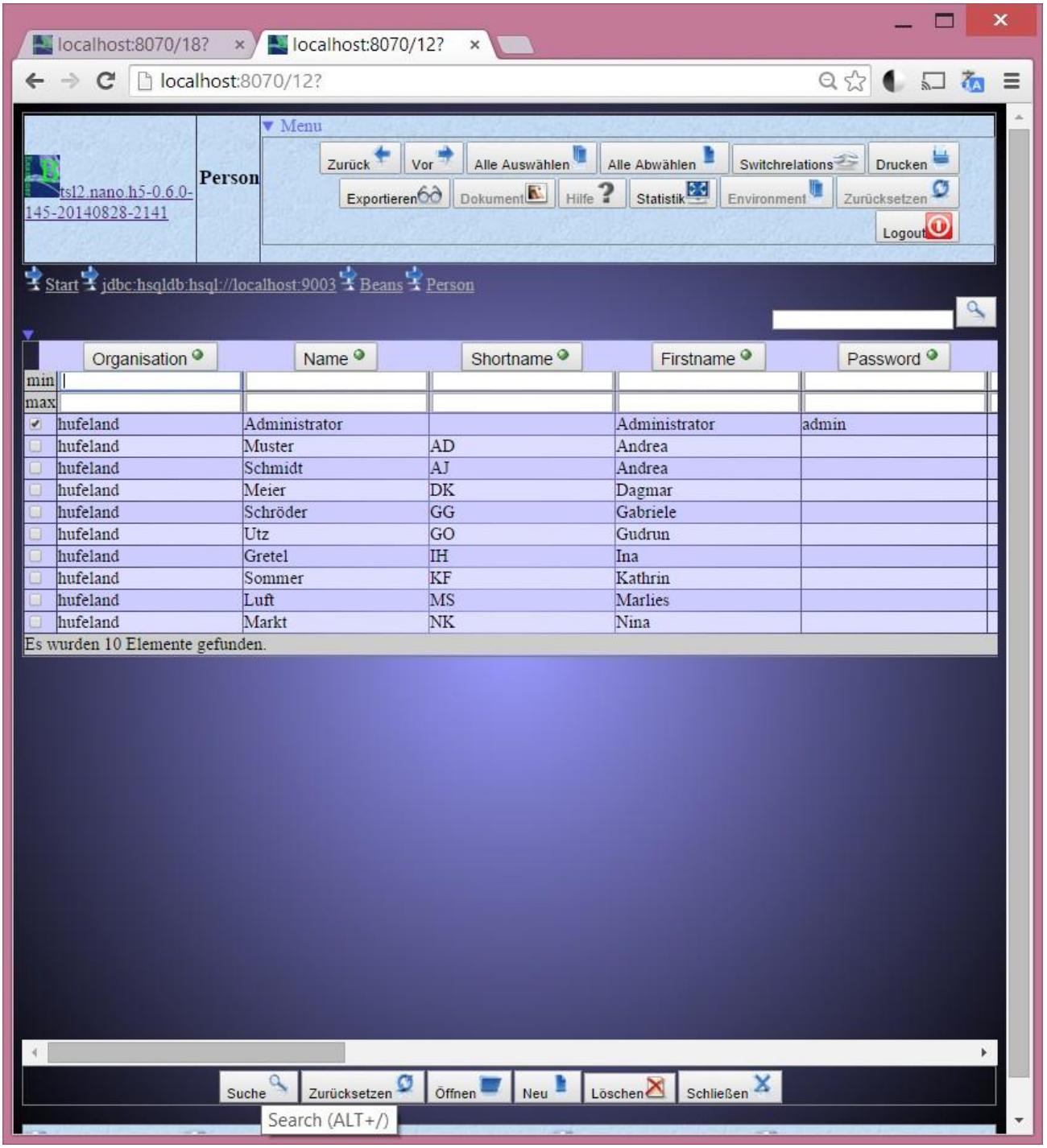
Now you can login to the sample database. It is fully configurable, which o/r mapper and database should be used. After pressing Ok, a *persistence.xml* will be generated to be found by the *javax.persistence* implementation.



All entities of the jar-file, containing the entities, will be listed. You can filter the list and select one or more to edit them.



Then you will get a search page with a search filter and an empty list. Pushing the search-button will create the result list.



If you click a column header (here f.e. *comments*), the list will be sorted by this column - clicking on that column a second time, the sorting will be done in the other direction.

The possible actions will be described in the chapter *Application and Page Actions*.

Extending the Sample

The file *environment.xml* defines the application behaviour. For further informations read chapter *The Environment*.
It is possible to change the presentation of each bean. Inside the environments directory *beandef* all beans have configuration files to change their presentation and behaviour. If the file *environment.xml* doesn't exist, the bean definitions will be created on next application start.

It is possible to create an own java project to define own application and bean behaviour. This is described in chapter *Creating an own project*.

The Environment

Loading a *Nano.h5* application will create and use an environment directory as workspace for it. Resources like icons, configuration xml-file and libraries will be put there - being on top of classpath.

Everything of your application will be accessible through this environment. It provides all system-/ application properties and all application services.

A description of all configuration attributes can be found here

Internationalization: Languages and automatic Translations

The *messages.properties* is the language file to translate every application specific text. You can overwrite it through putting your own file into the environment-directory. All bean or bean-attribute-names will be translated - if no presentation xml file are present. To define for example a german translation, you would create the file *messages_de.properties* to the environment-dirctory.
All framework specific and generic texts will be translated through the internal *de.tsl2.nano.messages.properties*.

The environments file *messages.properties* generated on new beans provides default entries for all available fields and actions. Change it to define other translations (will be done since you change the bean-definition names in the xml files directly. It is possible to add html tags like:

```
myfieldid=<a href=another-url.html>my-translation</a>
```

Tip: to have a good internationalization result on generating a new environment, put your *messages.properties* file into a new self created environment directory. This *messages.properties* may hold some glossaries or a full specific translation.

The framework provides automatic translation through a network connection. So, if no *messages*[locale].properties_ was found in the environment directory, the framework will try to create a translation for the *messages.properties* from english to the current locale of the java vm - only, if *tsl2nano.offline* is not true. This machine translation is only a first try - you should inspect this file on your own.

To create different translations you can start the virtual machine with different locales. You do that f.e. with vm-arg *set LANG=-Duser.country=FR -Duser.language=fr*.

Application and Page Actions

On top of each html page you will see on the left side an application icon (clicking on it, it tries to load the help file from web). On top right, all page-specific buttons are shown. The following list tries to describe them. They depend on the current type of bean. A bean list will have other buttons than a beans detail page.

Bean's search page top buttons

- *select all*: will select all listed items to be accessed through 'open' or 'delete' buttons
- *de-select all*: will de-select all listed items
- *next page*: if a bean search filter results in more matches than shown in this result page, the next result items will be shown
- *previous page*: if you walk through the result pages with *next-page*, you can go back until the first page.
- *switch relations*: if your current bean-list or bean-detail contains one-to-many relations (f.e. persons have collection of adresses), they can be shown now. this may result in longer page-build times! clicking the button on the same page another time will turn off the one-to-many relations.

All other page buttons (on the top)

- *print*: shows a page with non-interactive presentation of the current page. use the browsers back-button to return to the application
- *export*: shows a page with a pure text presentation like a csv file - to be copy/pasted into another file. use the browsers back-button to return to the application.
- *document*: if configured in *environment.xml*, a text-file (a rtf-file is possible, too) can be search and replaced with key/values of the current page.
- *configure*: opens a configuration page for the current bean. you are able to change the presentation text of this bean, all attributes to be presented - presentation properties like layout and layout constraints.
- *help*: if a help html-file for the current page/bean can be found, it will be shown. use the browsers back-button to return to the application.
- *refresh*: all configurations will be reloaded
- *exit*: stops the current user-session.

Bean search and manipulation buttons

- *search*: searches for all beans of the current type and filter.
- *reset*: resets all search-filter fields and the result list
- *open*: opens all selected beans. only active, if on page-creation at least one selected items is available.
- *new*: creates a new item of the given type. default values found by configuration will set - navigation history values will be set, too.
- *delete*: deletes all selected items.

Bean detail buttons

- *save*: saves/persists the current bean.
- *close*: closes the current page, returning to the last one without saving.

Buttons defined through actions inside the beans/entities itself

Each entity can define actions to be presented as buttons itself. See chapter *Bean Actions*.

Dependencies

While the *tsl2.nano* framework has only a static dependency to simple-xml, using datbase-connections, bean or ddl generation will result in additional dependencies to other libraries. It's easy to add these libraries putting them into the environment path - they will be load on runtime, too. If you have a network connection, the *tsl2.nano* framework tries to resolve the dependencies for you.

The Jar-Resolver

The *JarResolver* is able to resolve dependencies on runtime through a network connection using maven. It knows some important classes (driver, persistence-provider etc.) and their depending jar-files.

Features

- installs maven by itself (through a network connection)
- transforms known class names (with package) to known jar-dependencies
- creates dynamically a pom.xml holding all dependencies
- loads all given dependencies through maven to the current path

You can switch off using jar-resolving throuth the NetworkClassloader by setting the java-argument *tsl2nano.offline=true*. Or you set the ENV property *classloader.usenetwork.loader = false*.

Framework dependencies

Static Dependencies (direct referenced by tsl2.nano):

- simple-xml-2.7.jar

Dynamic Dependencies (used by tsl2.nano, but through compatibility-layer)

- XmlUtil: velocity-1.6-dep.jar
- CommonTest: junit-4.8.2.jar
- AntUtil: ant-launcher.jar, ant.jar, ant-nodeps.jar
- BeanEnhancer: javassist-3.12.0.GA.jar

No Dependencies, but useful to do the work

- jdbc database driver (like hsqldb.jar)
- jpa o/r mapper (like hibernate, toplink, eclipselink, openjpa, batoo-jpa, ormlite or ebean)
- generator tool to create entity beans (like hibernate-tools or openjpa)

Hibernate 4 for example would have the following dependencies:

- commons-collections-3.2.1.jar
- commons-logging-1.1.1.jar
- commons-beanutils-1.8.0.jar
- commons-io-1.3.2.jar
- commons-lang-2.4.jar
- commons-codec-1.6.jar
- dom4j-1.6.1.jar
- javassist-3.12.0.GA.jar
- antlr.jar

Runtime Configuration

Extensions of a BeanDefinition / Presentation

The *Extension* is a workaround for de-/serializings through *simple-xml*. While you must know the type of your deserializing root element, it is not possible, to load any class-extensions.

This class can be used as member of your base-class. after deserializing you can create the desired extension instance through informations of this extension-instance.

```
USE:

- mark all extending members with annotation {@link Transient} (not with javas keyword 'transient')
- on serialization of your extension class, call {@link #Extension(Object)} in your method annotated with {@link Commit}.
    {AT}Persist
    protected void initSerialization() {
        extension = new Extension(this);
        if (extension.isEmpty())
            extension = null;
    }
- on de-serialization, call {@link #to(Object)} getting the desired instance.
```

Plugins inside a BeanDefinition or AttributeDefinition

Plugins can be added to the beandefinition or an attribute-definition. A plugin must implement the interface *IConnector* with its method *connect(connectionEnd)*. This method will be called after deserialization - so you can do anything with all properties of a bean instance.

Your plugin implementation must provide the following:
implementation of IConnector
serializable
* must have a default constructor

Example with plugin implementation 'RuleCover':

```
<attributeDefinition ...>
...
   <plugin class="de.tsl2.nano.h5.RuleCover">
        <rule for-property="constraint.nullable" name="specification-rule-name" />
   </plugin>
</attributeDefinition>
```

Example for programmatically adding plugins:

```
    myattribute.addPlugin(myPlugin);
```

Specifying Rules, Queries and Actions to be used on Beans and Attributes

To do a structured work on a usable specification, rules, queries and actions can be defined before implementing or configuring the presentation. This is done by creating these items as xml-files inside the environments *specifiation* directory. To see how it works, hit the applications menu button 'sample-codes' and have a look into the specification directory.

All items in the specification directory will be tested against their own *specification* entries. These are assertions for simple test or boundary conditions. The tests/checks are done on creating the instances of the rules. To switch these tests off, set the environment variable *"rule.check.specifications"* to false.

Example for a query:

```
<query name="personen" nativeQuery="false">
   <query><![CDATA[select p from Person p]]></query>
</query>
```

Example for an action, starting an ant-script:

```
<action name="ant" declaringClass="de.tsl2.nano.execution.ScriptUtil">
   <parameter name="arg1">
      <type javaType="java.lang.String"/>
   </parameter>
   <parameter name="arg2">
      <type javaType="java.lang.String"/>
   </parameter>
   <parameter name="arg3">
      <type javaType="java.util.Properties"/>
   </parameter>
   <constraint name="arg2">
      <definition type="java.lang.String" nullable="true" length="-1">
         <defaultValue class="java.lang.String">help</defaultValue>
         <scale>-1</scale>
         <precision>-1</precision>
      </definition>
   </constraint>
   <constraint name="arg1">
      <definition type="java.lang.String" nullable="true" length="-1">
         <defaultValue class="java.lang.String">C:\eigen\tsl2\tsl2-workspace\tsl2-nano\target\test.h5.sample\h5.sample/antscript
         <scale>-1</scale>
         <precision>-1</precision>
      </definition>
   </constraint>
   <operation>ant</operation>
</action>
```

This action can be referenced in your presentation beandef:

```
<beanDefinition ...>
...
      <action class="de.tsl2.nano.h5.SpecifiedAction" name="ant" />
</beanDefinition>
```

This action will be shown as button in your beansdefs detail view.

The rule cover

The rule cover is a plugin-mechanism to override fix values of beandefinition properties through dynamic evaluation of referenced specification rules.

Example, referencing a rule for a property of your beandefinition:

```
<beanDefinition ...>
...
   <plugin class="de.tsl2.nano.h5.RuleCover">
        <rule for-property="constraint.nullable" name="specification-rule-name" />
   </plugin>
</beanDefinition>
```

Three rule types are known:
- *Rule*: a boolean/numeric operation
- *RuleScript*: a java script expression
- *RuleDecisionTable*: a decision table where the first column defines the parameter names and all following column are decision values. the last line defines the result vector.

Defining a rule through javascript

Use a *RuleScript* to define a rule with a javascript expression.

Example returning a map with an html style to be used as layoutconstraint on an attribute:

```
String redColorStyle = "color: red;";
String greenColorStyle = "color: green;";

//define the rule
RuleScript<String> presValueColor =
    new RuleScript<String>(
        "presValueColor", "var map = new java.util.HashMap(); map.put('style', value > 10 ? '"
        + redColorStyle
        + "' : '" + greenColorStyle + "'); map;", null);

//this will persist the rule to '<ENV>/specification/rule/presValueColor.xml'
ENV.get(RulePool.class).add(presValueColor);
```

...the xml equivalent would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<ruleScript name="presValueColor">
   <operation>var map = new java.util.HashMap(); map.put(&apos;style&apos;, value &gt; 10 ? &apos;color: red;&apos; : &apos;colo
</ruleScript>
```

...the rule-cover could be done with:

```
RuleCover.cover(Charge.class, ATTR_VALUE, "presentable.layoutConstraints", "%" + presValueColor.getName());
RuleCover.cover(Charge.class, ATTR_VALUE, "columnDefinition.presentable.layoutConstraints", "%" + presValueColor.getName());

charge.saveDefinition();
```

Defining a rule through a decision table

Use a *RuleDecisionTable* to define a rule through an excel sheet exporting its data through an *csv* file.

Example creating a csv and referencing this csv through a RuleDecisionTable:

```
String redColorStyle = "color: red;";
String greenColorStyle = "color: green;";

//create a csv file holding a decision-table
TableList tl = new TableList<>(2);
tl.add("matrix", "<1>", "<2>", "<3>", "<4>", "<5>", "<6>", "<7>");
tl.add("weekday", "Mo", "Di", "Mi", "Do", "Fr","Sa", "So");
tl.add("result", greenColorStyle, greenColorStyle, greenColorStyle, greenColorStyle, greenColorStyle, redColorStyle, redColorSty
String ruleDir = ENV.get(RulePool.class).getDirectory();
FileUtil.save(ruleDir + "weekcolor.csv", tl.dump());

//now, we create the rule referencing the weekcolor.csv decision-table:
RuleDecisionTable dtRule = RuleDecisionTable.fromCSV(ruleDir + "weekcolor.csv");
ENV.get(RulePool.class).add(dtRule);
```

...the xml equivalent would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<ruleDecisionTable name="weekcolor">
   <operation>[BASEDIR]/.nanoh5.timesheet/specification/rule/weekcolor.csv</operation>
```

```
</ruleDecisionTable>
```

...the csv would be:

```
matrix  <1> <2> <3> <4> <5> <6> <7>
weekday Mo  Di  Mi  Do  Fr  Sa  So
result  color: green;   color: green;   color: green;   color: green;   color: green;   color: red; color: red;
```

...the rule-cover could be done with:

```
RuleCover.cover(Charge.class, ATTR_FROMDATE, "presentable.layoutConstraints", "&" + dtRule.getName());
RuleCover.cover(Charge.class, ATTR_FROMDATE, "columnDefinition.presentable.layoutConstraints", "&" + dtRule.getName());

charge.saveDefinition();
```

Defining an action through javascript

Use a *ActionScript* to define an action with a javascript expression.

Example:

```
TODO
```

Defining a rule through standard java operations

Use a *Rule* to define an action with an expression containing standard java operations.

Example:

```
TODO
```

The *ScriptTool*

After login, you are able to execute queries and scripts like ant-scripts. You have to enable it in your *environment.xml* to see the*ScriptTool* in the list of *BeanCollectors*.

With *ScriptTool* you can define and save queries which will be loaded on next start time as virtual beandefinition - means, it is inside the bean-type list.

Layout and Styling

Using CSS

Html styles can be added through import of css files. On building each html page, the file *meta-frame.html* will be searched inside the environments *css* directory: *css/meta-frame.html*.
The *meta-frame.html* defines the html header and an empty body. Inside the header you should define the css style files you want to be used. The style files must define a class *menu* through the tags *ul* and *li*. Then, each page will have a menu at the top - instead of a button panel.

An example is provided with *tsl2.nano.h5.x.y.z.jar*. After first start, a css directory will be created. To turn on the given css-styling, rename the file *meta-framex.html* to *meta-frame.html* This is only an example - not yet working perfectly!

Configuring Presentation, Layout and Constraints

Layouts and LayoutConstraints must be of type *Map* or *String*. If a string is given, this string will be set as element style.

Overview styles for Application, BeanCollectors and Beans

The environment properties provide a fast way to define the applications page style. Change the entry *application.page.style* to use a custom background - perhaps a picture or a color. This must be an html5 css style expression.

Example setting a radial gradient background color:

```
<property name="application.page.style">
   <object class="java.lang.String">background: radial-gradient(#9999FF, #000000);</object>
</property>
```

Example setting a simple background image:

```
<property name="application.page.style">
   <object class="java.lang.String">background-image: url(icons/spe.jpg); background: transparent;</object>
</property>
```

The same can be done for a bean-collector, defining the property *beancollector.grid.style* and for bean-details with *bean.grid.style*.

Tip: Use CSS3Generator to prepare your css styles.

Collectors, Beans and Attributes

Each bean and each bean-attribute can define a layout and layout-constraints through it's presentable object (see class*IPresentable*). It is easy to do layout definitions programmatically through the *BeanPresentationHelper*, but in this chapter we describe only the xml configuration. Help on programmatical configurations, see chapter *Programmatical configuration*. Inside the application (on runtime), there is a *BeanConfigurator* providing to edit your _BeanDefinition_s as administrator.

Setting the layout for beans or attributes may often be done through setting their styles. Be carefull with this, while this overwrite framework-styles. The best way is, to have a look at the html source to see, which style was set before.

TODO describe

Bean-Collector Presentation

A bean-collector shows a set of bean-instances. The collector is defined by a bean-type or directly by a collection of items. The bit-field *mode* defines, which actions can be done on a bean-collector.

Mode:
Searchable: a search and reset action can be done on that bean-collector Filter: a table filter will be provided
Editable: an open action is available Creatable: new and delete actions are available
* *Assignable*: an assign action can assign the current selection to the previous edited bean.

If an item was selected and opened, this item (a *Bean*) will be presented as detail in a new page.

letting the beancollector start the search action on activation

Normally a bean collector provides a search mask to filled and started through user input. If only a small number of items are available in the database, it will be loaded directyl, depending on the environments entry *beancollector.do.search.on.count.lowerthan*with a default of 20.

If you want to have a pre-filled collection, you have to define at least one *minsearch* or _maxsearch value inside the bean definitions column definitions.

Example:

```
 <columnDefinition name="value" columnIndex="9" sortIndex="-1" isSortUpDirection="true" width="-1">
    <minsearch class="java.lang.Integer">2</minsearch>
    <maxsearch class="java.lang.Integer">8</maxsearch>
...
 </columnDefinition>
```

Of course you have to use values that are compatible with the attributes type.

You are able to predefine/memorize beans to be used on new beans or to constrain the search attributes. Go into the detail page of a bean and click 'memorize' at the top menu to store that bean inside your current session as default bean of that type.

creating a detailed summary for a bean collector

Example using the standard summary function of the bean collector on numbers:

```
<columnDefinition standardSummary="true">
...
</columnDefinition>
```

Example using the query *sumvalue* defined in specification:

```
<columnDefinition>
...
    <summary class="de.tsl2.nano.h5.expression.SQLExpression">
        <expression><![CDATA[§sumvalue]]></expression>
    </summary>
...
</columnDefinition>
```

where the *sumvalue* expression is defined inside the specifications with:

```
<query name="sumvalue" nativeQuery="false">
   <query><![CDATA[select sum(value) as Gesamt from Charge where party.name = '${party.name}' ]]></query>
</query>
```

the variable *party.name* must be defined by the user before while selecting (see action *memorize*) a party to use.

direct queries through *QueryResult*

To define direct sql or jpa-ql queries beside the standard mechanism though compiled beans you have the possibility to show a list of values direct through *QueryResult* which is an extension of *BeanCollector*.
QueryResult_s can be defined through the _ScriptTool. There you can test your queries and save it as *QueryResult*. The*QueryResult* will be loaded into the list of available bean-collector on next application start.

Example:

```
<queryResult clazz="java.lang.Object" name="virtual.times-overview" isNested="false" isdefault="true" xmlns="beandef.xsd">
   <extension declaringClass="de.tsl2.nano.h5.QueryResult">
      <member name="queryName">
         <object class="java.lang.String">times-overview</object>
      </member>
   </extension>
</queryResult>
```

The examples queryName *times-overview* has to exist in directory *specification/query*.

direct controlling a set of beans through the *Controller*

The *Controller* is an extension of *BeanCollector* to present a collection of beans through their actions. These actions should be simple change-actions for attributes - like de- or increasing their values. This feature should be used for touch-screen applications.

To provide such a controller you can create an xml-file inside your environments *beandef/virtual* directory. All beandefs inside the*virtual* directory will be added to the bean-type list which is presented after login.

Example:

```
<controller clazz="org.anonymous.project.Times" name="virtual.time-actions-controller" isNested="false" isdefault="true" xmlns="
   <extension declaringClass="de.tsl2.nano.h5.Controller">
      <member name="beanName">
         <object class="java.lang.String">time-actions</object>
      </member>
   </extension>
</controller>
```

Bean Presentation

A bean - as container of it's attributes - will always be presented as detail page.

[TODO screenshot]

A detail page presents the attributes of a bean as it is defined in it's bean-definition xml file. for more informations see chapter*Configuration through Serialization*.

Value Expressions

Each bean and attribute can be presented by and created by a text expression. On beans, this expression (called *ValueExpression*) should contain at least one bean-attribute (perhaps the id, but if another attribute is more readable but unique, you would prefer this). These *ValueExpression* will be used to present a bean in a list, or if it is used as relation in another bean. The creation through a string works only, if the bean provides a default-constructor and the attributes in the expression have setter-methods.

Two kinds of expressions are possible:

Standard *java.text.MessageFormat* expression

Example:

```
{forename}, {name}, {birthday}/{birthcity}
```

C-like *printf* expression

TODO: explain

Example:

```
%birthday$TD
```

Value Groups and Sub-Panels

Optional a bean-definition can define value groups - a list of attributes - to be presented in its own panel.

Multiple Value Relations

Whether multiple value relations (one-to-many) are shown is defined inside the environments filter*default.present.attribute.multivalue*. The bean name must match the regex-filter.

Attribute Presentation

The attributes are the most important and complex definitions. They contain sub-definitions described in the next chapter. Each attribute will be presented by a label, defined by the *Presentable.getLabel()* the value itself, mostly shown as input field and optional, if the value has a relation to other beans or is a collection of values - a selection button. The label will be translated through a resource bundle - the message.properties. It is possible to embed html-tags

inside the translation.

Example:

```
persistence.connectionUrl=<a href="http://www.databasedrivers.com/jdbc/">Database-URL</a>
```

There is a standard filter to present only non-technical single value fields. So, the following types of attributes will not be shown on default:
- multiple values (like collections)
- id fields (mostly filled by generic sequences)
- timestamps (filled automatically by system; use DATE and TIME for user inputs)

The *BeanPresentationHelper.isDefaultAttribute(IAttribute attribute)* implements that filter and can be switched off by setting the environment variable *bean.use.beanpresentationhelper.filter* to *false*. Or you can set switches for the three types directly:
- default.present.attribute.id: regular expression - e.g.: .
- *default.present.attribute.multivalue: regular expression - e.g.: .*
- default.present.attribute.timestamp: regular expression - e.g.: .*

Be careful, jpa generators like hibernate may change their implementation on generated temporal types from DATE to TIMESTAMP.

Dynamic Attribute Presentation

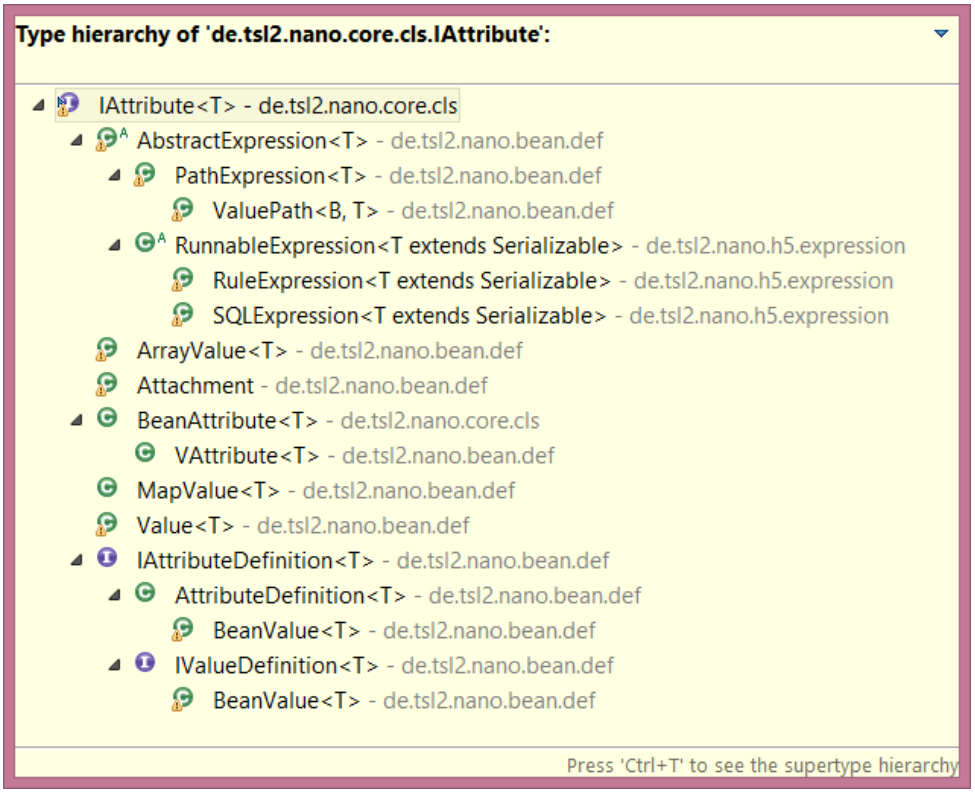
The presentation of an attribute can be defined through an xml file. To use dynamic presentation values, you can use the plugin*RuleCover* which will invoke the given rule for a defined property.

Attribute Declaration

This is the most important attribute property. Normally it is a standard bean-attribute, defined by the declaring class and the attribute name. The following declaration types are available:

- bean-attribute: (default, *de.tsl2.nano.core.cls.BeanAttribute*) the attribute value is defined by the declaring class and the attribute name (accessed through getter and setter methods)
- virtual-attribute: the attribute value is any fixed value
- value-expression: the attribute value is defined by an expression
  - path-expression: the expression is a path from a bean-attribute to another one - through one or more relations. expression example: *person.organization.name*.
  - rule-expression: the expression is an existing rule name.
  - sql-expression: the expression is an existing query name.
  - restful-expression: the expression is the url of a restful-service

All available attribute instances:



Example for an attribute through a relation path:

```
<attribute name="path-test">
   <attributeDefinition id="false" unique="false" composition="false" cascading="false" generatedValue="false">
      <declaring class="de.tsl2.nano.bean.def.PathExpression" declaringClass="org.anonymous.project.Times" type="java.lang.Ob
         <expression><![CDATA[id.dbBegin]]></expression>
      </declaring>
```

```
         <constraint type="java.lang.Object" nullable="true" length="-1">
            <scale>-1</scale>
            <precision>-1</precision>
         </constraint>
         <description>path-test</description>
         <presentable class="de.tsl2.nano.h5.Html5Presentable" type="2" style="4" visible="true" searchable="true">
            <label>DbBegin</label>
            <description>DbBegin</description>
            <enabler class="de.tsl2.nano.action.IActivable$2" active="false"/>
         </presentable>
         <doValidation>true</doValidation>
      </attributeDefinition>
   </attribute>
```

if a relation is an iterable or map, it can be specified through a parameter.

Examples for one-to-many relation paths:

```
1. customer.address[first].city
2. customer.address[0].city
3. customer.address.city     (the first address, too)
4. customer.address[last].street
5. customer.address[-1].code (the last address)
6. customer.address[street=berlinerstrasse].city
```

Example for a virtual attribute:

```
   <attribute name="space-1421869962419">
      <attributeDefinition class="de.tsl2.nano.bean.def.BeanValue" id="false" unique="false" composition="false" cascading="false
         <declaring class="de.tsl2.nano.bean.def.VAttribute" declaringClass="de.tsl2.nano.bean.IValueAccess" name="value" virtual
         <constraint type="java.lang.Object" nullable="true" length="-1">
            <defaultValue class="java.lang.String">[space]</defaultValue>
            <scale>-1</scale>
            <precision>-1</precision>
         </constraint>
         <description>space-1421869962419</description>
         <presentable class="de.tsl2.nano.h5.Html5Presentable" type="2" style="4" visible="false" searchable="true">
            <label>Space-1421869962419</label>
            <description>Space-1421869962419</description>
            <enabler class="de.tsl2.nano.action.IActivable$1" active="true"/>
         </presentable>
         <doValidation>true</doValidation>
      </attributeDefinition>
   </attribute>
```

Constraints

The attribute constraint defines the

- *type*: attributes java type
- *nullable*: if the attributes value may be null (empty)
- *format*: an attributes value has to match the formats expression to be valid
- *length*: maximum length of input characters to be valid
- *scale, precision*: scale and precision if attribute is of type number
- *min, max, allowedValues*: the attributes value range. you may define a min and max range - or only a list of allowd values.

Presentable

These are generalized properties to define the presentation in any gui (graphical user interface, surface):

- *type, style*: bit-field to define a gui specific user-input
- *visible*: whether to show the attribute
- *label*: a label for the attributes input field
- *description*: an attribute description to be used f.e. as tooltip or title
- *layout*: in nano.h5: a map of html5 attributes to be set for child elements, if attribute is a container
- *layoutconstraints*: in nano.h5: a map of html5 attributes to be set for the attributes html element
- *icon*: optional icon to be shown with that attribute

Presentable types and styles:

```
UNDEFINED           = -1
UNSET               = 0
TYPE_INPUT          = 1
```

```
TYPE_SELECTION      = 2
TYPE_OPTION         = 4
TYPE_DATE           = 8
TYPE_TIME           = 16
TYPE_LABEL          = 32
TYPE_TABLE          = 64
TYPE_TREE           = 128
TYPE_ATTACHMENT     = 256
TYPE_GROUP          = 512
TYPE_FORM           = 1024
TYPE_PAINT          = 2048
TYPE_INPUT_MULTILINE= 4096
TYPE_DATA           = 8192
TYPE_INPUT_NUMBER   = 32768
TYPE_INPUT_TEL      = 65536
TYPE_INPUT_EMAIL    = 131072
TYPE_INPUT_URL      = 262144
TYPE_INPUT_PASSWORD = 524288
TYPE_INPUT_SEARCH   = 1048576
TYPE_OPTION_RADIO   = 2097152
STYLE_SINGLE        = 1
STYLE_MULTI         = 2
STYLE_ALIGN_LEFT    = 4
STYLE_ALIGN_CENTER  = 8
STYLE_ALIGN_RIGHT   = 16
STYLE_ALIGN_TOP     = 32
STYLE_ALIGN_BOTTOM  = 64
STYLE_DATA_IMG      = 128
STYLE_DATA_EMBED    = 256
STYLE_DATA_OBJECT   = 512
STYLE_DATA_CANVAS   = 1024
STYLE_DATA_AUDIO    = 2048
STYLE_DATA_VIDEO    = 4096
STYLE_DATA_DEVICE   = 8192
STYLE_DATA_FRAME    = 16384
```

Example for some layoutconstraints:

```
<attribute name="comment">
   <attributeDefinition id="false" unique="false" composition="false" cascading="false" generatedValue="false">
      <declaring class="de.tsl2.nano.core.cls.BeanAttribute" declaringClass="org.anonymous.project.Times" name="comment"/>
      <constraint type="java.lang.String" nullable="true" length="128">
         <scale>0</scale>
         <precision>0</precision>
      </constraint>
      <presentable class="de.tsl2.nano.h5.Html5Presentable" type="4097" style="4" visible="true" searchable="true">
         <label>Comment</label>
         <description>Comment</description>
         <layoutconstraint name="colspan">11</layoutconstraint>
         <layoutconstraint name="border">1</layoutconstraint>
         <layoutconstraint name="size">150</layoutconstraint>
         <enabler class="de.tsl2.nano.action.IActivable$1" active="true"/>
      </presentable>
      <doValidation>true</doValidation>
   </attributeDefinition>
</attribute>
```

Data and attachments

The following types declare data types that should be editable through the attachment type:

```
TYPE_ATTACHMENT     = 256
TYPE_DATA           = 8192
```

To have a file selector box as user input, you should use TYPE_ATTACHMENT.
To define the detailed style of your data, you can set one of the following styles:

```
STYLE_DATA_IMG      = 128
STYLE_DATA_EMBED    = 256
STYLE_DATA_OBJECT   = 512
STYLE_DATA_CANVAS   = 1024
STYLE_DATA_AUDIO    = 2048
STYLE_DATA_VIDEO    = 4096
STYLE_DATA_DEVICE   = 8192
STYLE_DATA_FRAME    = 16384
```

These styles correspond to the equal named html5 tag names. The default style is *img*, where no additional informations are required. If you use the style *object*, you have to define the *mimetype* for the object. You do this through a layout property (e.g. type="image/svg+xml").

Using the type *TYPE_ATTACHMENT* and style *STYLE_DATA_OBJECT* would result in the html-tag

```
<object data="myfile.svg" type="image/svg+xml">
</object>
```

Example, showing an svg as attachment:

```
   <attribute name="icon">
      <attributeDefinition id="false" unique="false" composition="false" cascading="false" generatedValue="false">
         <declaring class="de.tsl2.nano.core.cls.BeanAttribute" declaringClass="org.anonymous.project.Item" name="icon"/>
         <constraint type="[B" nullable="true" length="255">
         </constraint>
         <presentable class="de.tsl2.nano.h5.Html5Presentable" type="256" style="1028" visible="true" searchable="true">
            <label>Icon</label>
            <description>item.icon</description>
            <enabler class="de.tsl2.nano.action.IActivable$1" active="true"/>
         </presentable>
         <columnDefinition name="icon" columnIndex="10" sortIndex="-1" isSortUpDirection="true" width="-1" standardSummary="fals
            <presentable class="de.tsl2.nano.h5.Html5Presentable" type="256" style="1028" visible="true" searchable="true">
               <label>Icon</label>
               <description>item.icon</description>
            <layout name="type">image/svg+xml</layout>
               <enabler class="de.tsl2.nano.action.IActivable$1" active="true"/>
            </presentable>
         </columnDefinition>
         <doValidation>true</doValidation>
      </attributeDefinition>
   </attribute>
```

Available mime types (e.g. for the tag *object*):

```
application/pdf
application/postscript
application/postscript
application/x-pcl
application/vnd.hp-PCL
application/x-afp
application/vnd.ibm.modcap
image/x-afp+fs10
image/x-afp+fs11
image/x-afp+fs45
image/x-afp+goca
text/plain
application/rtf
text/richtext
text/rtf
application/mif
image/svg+xml
image/gif
image/png
image/jpeg
image/tiff
text/xsl
image/x-emf
```

Using the style *embed* would embed the whole attachment content into the html response file - only, if no *pluginspage* attribute was defined. all other styles only reference the given source file with an *src* attribute.

To show for example a pdf file in your html page, you can add a plugin attribute. Add the following layout tag to the attributes presentable tag.

```
    <layout name="pluginspage">http://www.adobe.com/products/acrobat/readstep2.html</layout>
```

Please have a look at the html5-tags and their corresponding attributes, that could be provided through the attributes layout definition map.

Column-Definitions

The column-definitions describe the beans presentation in bean-table with columns - as the bean-collector do.

- *width*: column width
- *columnIndex*: at which position this column is shown (number between 0 and bean-attribute-count
- *sortIndex*: sort position, if multiple sort conditions are supported

- *isSortUpDirection*: if true, the sorting direction is *up*, otherwise *down*
- *presentable*: presentable properties like them in the attribute-constraints

Rule-Attributes

Rule attributes are virtual attributes showing the result of a specified rule.

Showing calculated values through integrated rule engine

The integrated rule engine provides calculations of rules - optional calling sub-rules - having one or more input parameter and a result value after execution. Input parameter can be constraint through java-types and value-ranges.

A rule calculates a combination of mathematical and conditioning expressions.

Example:

```
A1 & A2 ? ((x1 + x2) * $myrule2) : 0
```

Where A1, A2 are boolean values and x1, x2 numbers and myrule2 another referenced (using '§') rule.

Example xml-file for a rule:

```
<rule name="test">
   <parameter name="A">
      <type>BOOLEAN</type>
   </parameter>
   <parameter name="x1">
      <type>NUMBER</type>
   </parameter>
   <parameter name="x2">
      <type>NUMBER</type>
   </parameter>
   <constraint name="result">
      <definition type="java.math.BigDecimal" nullable="true" length="-1">
         <min class="java.math.BigDecimal">0</min>
         <max class="java.math.BigDecimal">10</max>
         <scale>-1</scale>
         <precision>-1</precision>
      </definition>
   </constraint>
   <constraint name="x1">
      <definition type="java.math.BigDecimal" nullable="true" length="-1">
         <min class="java.math.BigDecimal">0</min>
         <max class="java.math.BigDecimal">1</max>
         <scale>-1</scale>
         <precision>-1</precision>
      </definition>
   </constraint>
   <operation>A ? (x1 + 1) : (x2 * 2)</operation>
</rule>
```

Example for a rule using another sub-rule:

```
<rule name="test-import">
   <parameter name="A">
      <type>BOOLEAN</type>
   </parameter>
   <parameter name="x1">
      <type>NUMBER</type>
   </parameter>
   <parameter name="x2">
      <type>NUMBER</type>
   </parameter>
   <operation>A ? 1 + (A ? (x1 + 1) : (x2 * 2)) : §test</operation>
```

Presenting data or media like images, audio, video

If an attribute holds data of type *byte[]*, it may be presented as image or any other non-textual type. The default-mechanism will declare the attributes type as *TYPE_DATA* which will save the attributes value to a temporary file to be sent through http to the client and to be shown as image - or if a special style was defined, as *embed*, *object*, *audio*, *video*, *canvas* or *device* - as known from*HTML5* embedded content tags.

Interactive actions on images

Using pixel or vector pictures as attribute content and defining a dependency listener of type *WebSocketDependencyListener* will give you the possibility to let the user interact on data content. your dependency listener will have access to the clients mouse click position to refresh other attributes depending on that click into the picture or vector graphic.

Attribute Encryption

Set the *secure* property on your attribute - if it is of type String - to have an attribute encryption. The framework provides two implementations:

1. hashes: de.tsl2.nano.core.util.Hash
2. crypt : de.tsl2.nano.core.util.Crypt

If you use hash, the presentation will automatically be a password field. Setting a new value will directly hash the value.
If you use crypt, the en- decryption will be done on saving or loading the bean and its attributes.

Bean Actions

Each entity can define methods that will automically be presented as buttons on a detail panel. This methods must follow the following constraints:

1. the method must be public in any class inside the current class hierarchy.
2. the method name must start with prefix 'action' or must have the annotation @Action.

With Annotation *@Action* you can define parameter names, if you have method parameters. Additionally you can define more parameter constraints through Annotation *@Column*.

Example without Annotation:

```
public void actionRemoveRuleCover(String child, String rule) {
    RuleCover.removeCover(attr.getDeclaringClass(), attr.getName(), child);
}
```

Example with Annotation:

```
@de.tsl2.nano.bean.annotation.Action(name = "addListener", argNames = { "Observer Attribute",
    "Observable Attribute", "Rule-Name" })
public void actionAddListener(
        @Constraint(pattern = "(\\w+") String observer,
        @Constraint(pattern = "(\\w+") String observable,
        @Constraint(pattern = "[%§!]\\w+") String rule) {
    BeanDefinition def = ENV.get(BeanConfigurator.class).def;
    Html5Presentation helper = (Html5Presentation) def.getPresentationHelper();
    helper.addRuleListener(observer, rule, observable);
}
```

It is possible to add any action on runtime to your bean.

- Simply go to configuration page of your bean (use page button 'configuration' if you are inside your beans detail page).
- use *Create specification Action*, if you didn't yet define the desired specification action and define a name and an existing class+method (mypackage.MyClass.myMethod).
- use *Add Action* and write the specified action name in the next page

Example creating a new bean action on runtime:

```
New specified Actionname    : testaction
Action-Expression           : de.tsl2.nano.core.util.NetUtil.getRestfulJSON
```

...this will result in file _.../specification/action/testaction.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<action name="testaction" declaringClass="de.tsl2.nano.core.util.NetUtil">
   <parameter name="arg1">
      <type javaType="java.lang.String" />
   </parameter>
   <parameter name="arg2">
      <type javaType="[Ljava.lang.Object;"/>
   </parameter>
   <operation>getRestfulJSON</operation>
</action>
```

...now you can *Add Action testaction* to your bean.

After saving the bean configuration, the new action should be available. test it, activating the new Button *Testaction* and fill the first argument with test REST-Service-URL: *http://headers.jsontest.com/*. After activating the button *TestAction* again, the REST service will be called and the returned JSON structure will be shown an a new simple html table.

Changing Layouts through the *BeanConfigurator*

The chapter before described the structure of all definitions. *Nano.H5* provides an administration tool inside the application to do some configurations on runtime. If you are inside the detail-page of a bean, there is an action 'Configuration' on your headers button-group. If you activate it, you are inside a configuration mode, letting you change layout and styling of the current bean.

[TODO: png]

The value-expression defines the presentation of the current bean-definition. Inside the brackets there are any characters + several attribute-names to be present a bean-value.
F.e., if you have a bean *Person* with attributes *id* and *name*, but want to show only the name of a person in a list, you define: *{name}*.

Presentable

Inside the *Presentable* configuration, you are able to change f.e. the layout of a bean. F.e. you can add a border through it's layout-constraints:

[TODO png]

which result in

[TODO png]

Adding Change Listeners and Rule Covers

The BeanConfigurator provides the ability to add change listeners and rule covers on bean attributes.

Change Listener

A change listener will be defined through an observer attribute, an observable attribute and a rule, calculating a reaction for the observer attribute if the observable changes.

You start the configuration after entering the configuration panel selecting an attribute and clicking the button 'Add Rule Listener'. Then you define three names (case-sensitive!):

1. observer attribute name
2. observable attribute name
3. existing rule name (defined in specification directory)

Then you hit 'Add Rule Listener' to finish the action. Close the configuration panels until you reach the panel having a 'save' button. Hit the save button to save the new configuration. The changes will be loaded after restart or reset.

Rule Covers

A rule cover covers any property to evaluate the properties value on runtime. A rule cover will be defined through a member of an attribute and a rule name. direct members of an attribute are:

- constraint (having properties to constrain the user input)
- presentable (having properties to define the gui presentation)
- columnDefinition (having properties to define the presentation inside a table)

You start the configuration after entering the configuration panel selecting an attribute and clicking the button 'Add Rule Cover'. Then you define two names (case-sensitive!):

- property of attribute (any property in the attributes member hierarchy. e.g.: presentable.layoutConstraints)
- existing rule name (defined in specification directory)

Then you hit 'Add Rule Cover' to finish the action. Close the configuration panels until you reach the panel having a 'save' button. Hit the save button to save the new configuration. The changes will be loaded after restart or reset.

Resolving *many-to-many* constellations

Mostly many-to-many constellations will be solved through an additional table in the middle.

Bean1*()-->(1)Middle(1)-->()*Bean2

Nano.H5 recognizes this constellations and solves the bindings through an internal mechanism called *composition*. For further informations see implementation of class *Composition* and *BeanCollector*.

Resolving Compositions

Nano.h5 recognizes a *composition* through the following jpa annotations of a bean attribute:

```
@OneToMany(...)
@JoinColumn(...nullable=false)
```

If the relation is uni-directional, the child bean is not persistable by itself. *Nano.h5* will present no *save* button but an *assign* button. So, changes will be persisted only on the parent bean.

Working on a page using the Keyboard

The gui provides shortcuts/hotkeys for each button to activate. As html provides basic field tabbing (of course, that depends on your browser), it should be possible to work fast using your keyboard only. Each button/action shows its hotkey/shortcut through its tooltip. The tooltip is shown positioning your mouse (;-)) cursor on the element.

On entering a search-page, the *search* button will have always the default-focus. If a search was done, the *open* button will have the default-focus.

On entering a detail-page, the *save* button, if existing, will have the default-focus.

Actions and Buttons

Each action is able to define a shortcut (keystroke). if not set, the default will be evaluated through the actions name. If that name contains a '&', the following character will be used as shortcut - if no '&' is contained and no keystroke was defined, the first character of the name will be used. E.g., if the name/label is "Clo&se", you have to hit *ALT+s* to activate this action.

Tableheader Buttons

Each standard table of beans will provide a table-header with Buttons to do up/down sorting (on first activation, up-sorting will be done, on second activation, down-sorting). On these Buttons we can't use the algorithm above to avoid shortcut collisions. These buttons will have shortcuts depending on their indexes. So, the first table-header column will be activated through a hit to *ALT+1*.

Networking Modes

The *Nano.H5* application can be started in different modes.

- *Standard Single Access Mode:*
  - environment property *http.ip* is *localhost*. no other network node is able to connect to the application. useable as a simple client/server application.

- *Network Single Access Mode:*
  - environment property *http.ip* is a network ip of your system. all network nodes are able to connect to the application of your system, using one environment and session. useable in an intranet to do some teamwork.
  - *Network Multiple Access Mode:*
  - environment property *http.ip* is a network ip of your system. all network nodes are able to connect to the application of your system, using their own environment and session. may be used in an intranet.
  - *Network Multiple Security Mode:*
  - environment property *http.ip* is a network ip of your system. all network nodes are able to connect to the application of your system, using their own environment and session working with ssl (under construction!). may be used in the internet.

Authorization, Roles and Permissions

Permissions are set after connecting to the datasource through the persistence-unit. The permissions define activation of buttons and visiblity of fields. if all fields are invisible, no data and actions are available!

The application class *NanoH5* has a method *createAuthorization()* that defines a subject with a user-principal and it's roles, defined by permissions. If a file *[username]-permissions.xml* is found, it will be used to fill the subject - if not, an admin-role with a wildcard-permission will permit anything!

Permissions contain a name - perhaps ended by a wildcard - and comma separated actions (a wildcard is possible, too. The permissions work on actions and data. The *BeanContainer* provides to ask for permissions: call*BeanContainer.instance().hasPermission(name, actions) to check access to a call or to any data. The _Environment* provides access to the implementation of *IAuthorization*. Call _Environment.get(IAuthorization.class).hasAccess(name, actions) to check for permissions.

To define special permissions for a user, change the content of the file *[username]-permissions.xml* and use the translation resourcebundle *messages.properties* to find the field and action names to fill in.
To use your own Authorization, set your implemenation of IAuthorization as service in the environment.

The framework does all checks for you. But if you need extended access to authority informations, read the following details.

Permissions on actions

To check whether a user can access a button (action) you call the *hasAccess* of IAuthorization or *hasPermission* of BeanContainer with the id of the action. The second method parameter may be a wildcard (*) or *executable*.

Permissions on data

To check whether an entity should be accessed by the current user, you call the *hasAccess* of IAuthorization or *hasPermission* of BeanContainer with the class-name + toString() representation of the current object. The second method parameter tells whether to read or write the object.

Navigation and Workflows

A navigator will guide the user through his application session. Before/after each page, the navigator will evaluate the next bean to present. This may be list of entities or simply the details of an entity.

The application can work on a simple navigation stack - a simple implementation is provided by the *EntityBrowser*. If a configuration file *workflow.xml* is found inside your environment-path, this workflow will be used, to navigate the user through his application session.

The EntityBrowser

The EntityBrowser works on a Navigation-Stack, putting all available entity-types to the first bean-collector to be able to browse through all beans and data.

A configured Workflow

If a configured workflow is available, this workflow will be used as navigator.
A workflow holds one or more activities defined by an enabling *condition* and an execution *expression*. After finishing a page, all activities will be checked for their entry/enabling condition. Only one *activity* should have a positive condition. This activity will be executed. The execution will be calling an EJB-QL given by the expression and returning the result of that query as next navigation bean.

Example:

```
<workflow name="test.workflow">
   <activity name="timesByProject">
      <condition>project&amp;true</condition>
      <expression>select t from Times t where t.project.id = :prjname</expression>
      <query-parameter length="1">
         <parameter>prjname</parameter>
      </query-parameter>
   </activity>
   <activity name="organisation">
      <condition>!organisation.activated</condition>
      <expression>select p from Organisation p where ? is null</expression>
      <query-parameter length="1">
         <parameter>prjname</parameter>
      </query-parameter>
   </activity>
   <activity name="person">
      <condition>organisation.activated &amp; (!person.activated)</condition>
      <expression>select p from Person p where p.organisation = ?</expression>
      <query-parameter length="1">
         <parameter>organisation</parameter>
      </query-parameter>
   </activity>
</workflow>
```

All parameters are stored and given by the workflow. The following parameters will be stored automatically:
response = *{last user action name}*
{activity-name}.size = count of entities found by your activity expression
{activity-name} = if your activity expression found exactly one entity, this entitiy will be referenced here.
{activity-name}.activated = true, if this activity was already activated

Use that parameters for your activity condition.

TODO: use bean-path to inspect condition expressions like 'times.type = F'.

Database Replication - Working Offline

It is possible to replicate the data loaded from a remote database. The environments property *use.database.replication* must be true (default is false!). The replication will be done through a second persistence-unit 'replication' (see *persistence.xml*). For each user an own local replication database will be created.

The replication is done in it's own thread to avoid conflictions with the application. The O/R mapper will create all tables through bean informations. A special bean holds the information (time, id, change-type) about the changes done by the user. All data, loaded and edited by the current user will be replicated to a local database. Of course the dependent data will be replicated, too, to have a valid datastore.

The replication connection is configurable like the persistence-unit is. But at the moment, only the combination hsqldb and hibernate are tested - and it's only usable in standalone-mode - without application-server!

The replications hsqldb database will be started internally (configuration :see META-INF/persistence.tml).
- database: replication-<user-name>
- port: 9898

If the model cycles entity relations, replication may fail. Sometimes, it helps to increase the heap memory (specially for the permormance) - if not, you should select other bean instance (base instances) before to replicate.

Working on multiple Databases

In standalone mode, database connections are defined inside the *persistence.xml*. Define multiple _persistence_untit_s if you need access to multiple databases. Of course, they need their mapping entity beans. So define the beans jar file in the persistence.xml and store it in the environments directory.

Working inside an application server, the server will provide a database pool. connections are defined for example in jboss in a deployed xml file (or the jboss configuration file like standalone.xml) having datasource entries.

Using an Applicationserver

If you don't want to have a standalone appliation, you are able to use an application server like jboss. To do this you must have:

- the *application servers client libraries* must be in your classpath. so copy them to your environment directory. with jboss eap 6.1 it would be *jboss-client.jar*.
- any entity beans and *remote interfaces* of your ejb's should be in your classpath. so copy the jar holding them to your environment directory.

- of course, the *entity beans must be deployed* to that server perhaps in an ear or war-file.
- be sure to have *tsl2.nano.serviceaccess* packed into that ear/war-file. the service *IGenericService is essential.
- add environement property 'applicationserver.authentication.service' (default: **org.nano.[environment-name].service.remote.IUserService**) with service class name.
- add environment property 'applicationserver.authentication.method' (default: **login**) with service method to call. this method has to have two string-parameters for user and password. the method has to call *ServiceFactory.instance().createSession(userObject, mandatorObject, subject, userRoles, features, featureInterfacePrefix)*
- on the login page input user and password
- create a *jndi.properties* file in your environment. with jboss this would look like the following:

Example jndi for jboss eap 6.1

```
    java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
    java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
    java.naming.provider.url=remote://localhost:4447
    java.naming.security.principal=<jndi-prefix>
    java.naming.security.credentials=<perhaps the application name>
    jboss.naming.client.ejb.context=true
```

Example authentication service method

```
UserService.authenticate(String user, String password) {
...
    //fill the server side ServiceFactory
    if (!ServiceFactory.isInitialized()) {
        ServiceFactory.createInstance(this.getClass().getClassLoader());
        //registriere shared und error messages
        ResourceBundle bundle = ResourceBundle.getBundle("org.mycompany.myproject.shared_messages",
            Locale.getDefault(),
            this.getClass().getClassLoader());
        Messages.registerBundle(bundle, false);
        bundle = ResourceBundle.getBundle("org.mycompany.myproduct.error_messages",
            Locale.getDefault(),
            this.getClass().getClassLoader());
        Messages.registerBundle(bundle, false);
    }

    /*
     * a user session will be created on server side.
     */
    final Collection<String> userRoles = new LinkedList<String>();
    final Collection<String> mandatorFeatures = getMandatorFeatures();
    ServiceFactory.instance().createSession(null, getMandant(), null, userRoles, mandatorFeatures, null);
...
}
```

Generating your Entities

It is possible to generate your entity beans through a tool like hibernate-tools or openjpa. This is prepared in *Nano.H5*, for further informations see chapter *Model Driven Architecture (MDA))*.

If hibernate-tools creates your beans, it may create additional beans to use composed primary keys. To use them with *nano.h5* you should extend these beans, to fill the id-bean from dependent entity fields. So, enhance the setter methods of this unique fields to fill the id-beans corresponding field, too.

```
class MyBean {
    @Id MyBeanId id;
    @Column MyType myUniqueField;
}

class MyBeanId {
    String field1;
    String myUniqueFieldId;
}
```

There are other tools to generate your entities:

- Netbeans provide a generating plugin
- "Eclipse Dali"http://www.eclipse.org/webtools/dali/ on WTP package
- "Telosys-Tools":http://tools.telosys.org/ (includes an eclipse plugin)
- "MinuteProject":http://minuteproject.wikispaces.com/JPA2
- Kodo (probably now integrated into openjpa)

But these tools are not supported and discussed in *Nano.H5*

Reverse engineering with OpenJPA

As on hibernate-tools it is difficult to set properties, openjpa provides lots of properties to assign the generation process for your project. The ant script *reverse-eng.xml* shows some of these properties.

Some hints:
don't define more than one persistence-unit in your persistence.xml
constrain the *schema* otherwise all schemats will be respected
the provider should be set to org.apache.openjpa.persistence.PersistenceProviderImpl openjpa-2.3.0 is not able to find/generate primary keys --> no class can be generated! Please at least 2.4.0
edit the persistence.xml *and set the property* openjpa.jdbc.DBDictionary* to value="hsql(SupportsSchemaForGetTables=false)"

Interactive attribute content

There are several possibilities to add interactive content to your page.

- Presenting an Attribute as one of:
  - Canvas (canvas filled through drawing informations of the attributes value)
  - SVG (svg can contain interactive content and can embed or reference bitmaps, "see tutorial":http://www.petercollingridge.co.uk/interactive-svg-components
  - Javascript (attributes value contains javascript)
  - WebGL (attribute value as web-gl)

- Rich Client GUI interactions through WebSockets (see next chapter)

These attributes may be BeanAttributes, getting their content f.e. from a database - or virtual attributes getting their content from somewhere else.

Rich Client GUI interactions through WebSockets

The use of websockets and their internal javascripts is optional. Nano.h5 provides four mechanisms to use websockets:

1. as application message service: providing status messages from main application
2. as input assist: supporting the user with available values
3. as dependency listener: refreshing other dependent values after change a fields value or doing a mouse click on data like pictures
4. as dependency listener: refreshing other dependent values through calls of RESTful services
5. as file attachment transferrer: providing to transfer local content to the server
6. as restful service request: the attribute has to present an _iframe. the current user interaction event will be sent to the service - the result will be embedded into the iframe.
   The main exception handler is connected to a WebSocketServer. All messages, sent through **Message.send(msg)** are transferred to all connected child sessions. After getting the 'submit' event, a progress bar is shown.

The applications message service

After user request - for example a forms submit - the application is doing some work with interaction to other servers like an application-server or a database. This may take long times, so the websocket connection is a comfortable way to show working status messages - with text and progressbar.

Attributes input assist to show available values

Each attribute having an input-assist instance (on persistable attributes it is a DefaultInputAssist instance) tries to provide an input-assistance through the websocket connection. This is done through the NanoWebSocketServer combined with the WebSocketExceptionHandler.

Dependency listeners to re-calculate their values after changing a source value

To refresh dependent values after changing a special value, do the following:
create a new class, implementing WebSocketDependencyListener.evaluate() add an instance of this dependency listener to the special values change-handler.

available implementations:
WebSocketDependencyListener
WebSocketRuleDependencyListener

Example: creating a dependency listener

```
BeanDefinition b = BeanDefinition.getBeanDefinition(org.anonymous.project.Person);
((AttributeDefinition)b.getAttribute("organisation")).changeHandler().addListener(new WebSocketDependencyListener(((AttributeDef

    @Override
    protected Object evaluate(Object value) {
        //any new value...
        return "my-refreshed-value:" + Util.asString(value);
    }
});
```

more complex Example: creating a dependency listener

```
final BeanDefinition b = BeanDefinition.getBeanDefinition(org.anonymous.project.Person);
((AttributeDefinition)b.getAttribute("organisation")).changeHandler().addListener(new WebSocketDependencyListener(((AttributeDef

    @Override
    protected Object evaluate(Object value) {
        //new value of attribute 'organisation'
        Object value = evt.newValue;
        //here we set dynamically which attribute depends on changes
        setAttribute(b.getAttribute("shortname"));
        //the evt.source holds the changed bean value
        IValueDefinition srcValue = (IValueDefinition)evt.getSource();
        //here we get the old value of the dependent attribute 'shortname'
        //through srcValue.getInstance() you could get all other values with Bean.getBean(srcValue)
        Object lastAttributeValue = getAttribute().getValue(srcValue.getInstance());
        //return the refreshed value for attribute 'shortname'
        return value + "/" + lastAttributeValue;
    }
});
```

If you use the *WebSocketRuleDependencyListener* you set a rule name to call a specification rule for evaluate the new attribute value. This dependency listener can be embedded into the owning attributedefinition of the presentation xml file.

```
    BeanDefinition b =
        BeanDefinition.getBeanDefinition(BeanClass.createBeanClass("org.anonymous.project.Person").getClazz());
    ((AttributeDefinition) b.getAttribute("name")).changeHandler().addListener(
        new WebSocketRuleDependencyListener<T>(b.getAttribute("shortname"), "shortname", "rule-shortname"));
```

Monitoring and refreshing a value through REST services and a Timer

If you use the *WebSocketServiceListener* you can define the following properties:
restfulUrl: url to a RESTful service
parameter: name of the first methods parameter - assigned to the bean attributes value
* timer: through *createTimer(periodInSeconds)* (xml-tag: *timer*) to do a scheduled client refresh.

Transferring local file attachments

Simply set your beans attribute type to be an attachment. The client will use a file-selector to select your file. After selecting the file, this file will be sent to the application and the bean can access this file, stored inside the environments temp path.

Example:

```
BeanDefinition b = BeanDefinition.getBeanDefinition(MyEntity.class);
((AttributeDefinition)b.getAttribute("my-attribute-name")).getPresentation().setType(IPresentable.TYPE_ATTACHMENT);
```

Calling a restful service to embed the result into an iframe

TODO: develop and describe...

Another feature is to define a timer through *createTimer(periodInSeconds)* (xml-tag: *timer*) to do a scheduled client refresh.

Technical details: How it works

there is a template file *websocket.client.js.template* that will be filled and embedded into the response html. some of the declared functions are called by event handlers of input fields of this html. So, if someone changes the method names of that template, he has to change the dependent environment properties, defining this function names.

Example environment-property:

```
websocket.inputassist.function=inputassist(event)
```

so this function has to be implemented inside the template script:

```
...
function inputassist(e) {
    var code;
    if (e.keyCode)
        code = e.keyCode;
    else if (e.which)
        code = e.which;
    var text = e.srcElement.value + String.fromCharCode(code);
```

```
    var request = '@' + e.srcElement.id + ':' + text;
    console.log('sending input-assist request: \'' + request + '\'');
    socket.send(request);
}
...
```

As you can see, all messages will change attribute properties.

Developing, Deploying and Debugging

- tsl2.nano code server

If you change sources of a plugin, you should start ant script *2nano.xml*. This will generate all jars to a target directory and creates the product-version *tsl2.nano.h5.x.y.z.jar* into the targets test directory *test.sample.h5*.

The version is read from *build.properties* of nano.h5.

Debugging

To debug your app or just the framework, you should enhance the logging output. Edit the file *logfactory.xml* and add the text *debug*to the attribute *standard* of tag *logfactory*. This will enhance not only the logging output but the details of the tooltips of each html-field.

It is possible to enhance the logging level as start parameter, too. add the parameter *-Dtsl2.nano.log.level=debug*.

Testing

To start the application in test mode, add the parameter *-Dtsl2.nano.test=true*. This will change the behaviour of creating new items. on test mode, all duty fields will be filled with default values to enable test engines save new entities without user input.

Testing JPA-Providers and the persistence.xml

If you want to test some properties in the persistence.xml without letting *nano.h5* generate it from template persistence.tml, change the environment property *login.save.persistence* to *false*.

Help on Html5

Normally, you don't have to create html-pages by yourself, but if you are interested in html5, have a look at the following tutorials/references:

- W3C-Html5-Reference
- Html-5.com
- Webkompetenz
- Html5 Poster

To analyse the html-page in your browser, for example in your Chrome-Browser, you can analyse components by mouse-right-click on *analyse element* to debug and change the current bean presentation.

jar-library dependencies

- velocity-1.6-dep.jar: XmlUtil.transform (using CompatibilityLayer)
- TODO: describe all jars!

Creating a new *NanoH5* version

Change the following files, if you change a version number of any nano project:
tsl2.nano.incubation/projects/.properties
tsl2.nano.h5/src/resources/build.properties
tsl2.nano.h5/src/resources/run.bat

Run the script tsl2.nano.incubation/2nano.xml. un-comment the first entry of *refactore.clean* before.

Creating an own project

Nano.H5 is based on the framework *tsl2.common* and it's bean package. The bean package provides a generic and comfortable way to describe your user interface. If the standards of *Nano.H5* don't fulfil your needs, you can develop own beans on top of *Nano.H5* - without creating special gui-elements or interaction, this will be done by the framework - generating html-pages through the*BeanPresentation* implementation. Of course, this implementation is extendable, too. Have a look at chapter *Dependencies* to know, which jar-files you should copy to the environment directory (f.e. *h5.sample*).

If you download and unpack *test.h5.sample*, you yield an eclipse project referencing the *tsl2.nano* jar-files.

The implementation *Loader.java* and *MyApp.java* provide an own entry for the sample application. The *Loader* only tells java to load*MyApp*. *MyApp* overwrites three methods. Only *createBeanCollectors* defines own beans and an own navigation stack.

Here is the implementation:

Loader

```
public class Loader extends AppLoader {
    public static void main(String[] args) {
        new Loader().start("my.app.MyApp", args);
    }
}
```

MyApp

```
public class MyApp extends NanoH5 {
    private static final Log LOG = LogFactory.getLog(MyApp.class);

    /**
     * @throws IOException
     */
    public MyApp() throws IOException {
    }

    /**
     * @param ipport ip + port
     * @param builder
     * @param navigation
     * @throws IOException
     */
    public MyApp(String ipport, IPageBuilder<?, String> builder) throws IOException {
        super(ipport, builder);
    }

    @Override
    @SuppressWarnings("rawtypes")
    protected BeanDefinition<?> createBeanCollectors(List<Class> beanClasses) {

        /*
         * create your own bean collectors as described in the next chapter
         */

        //...

        /*
         * define your own navigation stack
         */
        return beanCollector;
    }

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        startApplication(MyApp.class, MapUtil.asMap(0, "http.connection"), args);
    }
}
```

Programmatical configuration

The next code pieces generate examples for all *nano.h5* aspects. The generated xml-files are used in the specialized chapters as examples. You can put them all together in *MyApp.createBeanCollectors(..)*.

Creating a workflow

```
/*
 * Sample Workflow with three activities
 */
    LinkedList<BeanAct> acts = new LinkedList<BeanAct>();
    Parameter p = new Parameter();
    p.put("project", true);
    p.put("prjname", "test");
    acts.add(new BeanAct("timesByProject",
        "project&true",
        "select t from Times t where t.project.id = :prjname",
        p,
        "prjname"));
    p = new Parameter();
    p.put("prjname", "test");
    acts.add(new BeanAct("organisation",
        "!organisation.activated",
        "select p from Organisation p where ? is null",
```

```
        p,
        "prjname"));
    p = new Parameter();
    p.put("organisation", "test");
    acts.add(new BeanAct("person",
        "organisation.activated & (!person.activated)",
        "select p from Person p where p.organisation = ?",
        p,
        "organisation"));
    Workflow workflow = new Workflow("test.workflow", acts);
    Environment.persist(workflow);
```

Creating a rule with sub-rule

```
/*
 * use a rule with sub-rule
 */
LinkedHashMap<String, ParType> par = new LinkedHashMap<String, ParType>();
par.put("A", ParType.BOOLEAN);
par.put("x1", ParType.NUMBER);
par.put("x2", ParType.NUMBER);
Rule<BigDecimal> testRule = new Rule<BigDecimal>("test", "A ? (x1 + 1) : (x2 * 2)", par);
testRule.addConstraint("x1", new Constraint<BigDecimal>(BigDecimal.class, BigDecimal.ZERO, BigDecimal.ONE));
testRule.addConstraint(Rule.KEY_RESULT, new Constraint<BigDecimal>(BigDecimal.class, BigDecimal.ZERO,
    BigDecimal.TEN));
Environment.get(RulePool.class).add("test", testRule);

//another rule to test sub-rule-imports
Environment.get(RulePool.class).add("test-import",
    new Rule<BigDecimal>("test-import", "A ? 1 + §test : (x2 * 3)", par));

BigDecimal result =
    (BigDecimal) Environment.get(RulePool.class).get("test-import")
        .run(MapUtil.asMap("A", true, "x1", new BigDecimal(1), "x2", new BigDecimal(2)));

LOG.info("my test-import rule result:" + result);
```

Defining a query

```
/*
 * define a query
 */
String qstr = "select db_begin from times t where t.db_end = :dbEnd";

HashMap<String, Serializable> par1 = new HashMap<>();
Query<Object> query = new Query<>("times.begin", qstr, true, par1);
QueryPool queryPool = Environment.get(QueryPool.class);
queryPool.add(query.getName(), query);
```

Defining an action

```
/*
 * define an action
 */
Method antCaller = null;
try {
    antCaller = ScriptUtil.class.getMethod("ant", new Class[] { String.class, String.class, Properties.class });
} catch (Exception e) {
    ManagedException.forward(e);
}
Action<Object> a = new Action<>(antCaller);
a.addConstraint("arg1", new Constraint<String>(Environment.getConfigPath() + "antscripts.xml"));
a.addConstraint("arg2", new Constraint<String>("help"));
Environment.get(ActionPool.class).add("ant", a);
```

Defining a Controller as Collector of Actions of a Bean

```
/*
 * define a Controller as Collector of Actions of a Bean
 */
final BeanDefinition timeActionBean = new BeanDefinition(Times.class);
timeActionBean.setName("time-actions");
BeanDefinition.define(timeActionBean);
```

```
final Controller controller = new Controller(timeActionBean, IBeanCollector.MODE_SEARCHABLE);
timeActionBean.getActions().clear();
timeActionBean.addAction(new SecureAction("times.actions.one.hour.add", "+1") {
    @Override
    public Object action() throws Exception {
        //this.shortDescription =
        return controller;
    }
});
timeActionBean.addAction(new SecureAction("times.actions.one.hour.subtract", "-1") {
    @Override
    public Object action() throws Exception {
        return controller;
    }
});
timeActionBean.saveDefinition();
controller.saveVirtualDefinition(timeActionBean.getName()+ "-controller");
```

Defining a specific bean-collector presenting a query (SQL or JPA-QL)

```
/*
 * define a specific bean-collector presenting a query (SQL or JPA-QL)
 */
qstr = "\nselect t.day as Day, p.name as Project t.dbbegin as Begin, t.dbend as End, t.pause as Pause\n"
    + "from times t join project p on p.id = times.projid\n"
    + "where 1 = 1\n";

query = new Query<>("times-overview", qstr, true, null);
queryPool.add(query.getName(), query);

QueryResult qr = new QueryResult<>(query.getName());
qr.saveVirtualDefinition(query.getName());
```

Defining own beans to present your entities another way

```
/*
 * define own beans to present your entities another way
 */
Collection<Times> times = Environment.get(IBeanContainer.class).getBeans(Times.class, UNDEFINED, UNDEFINED);

BeanCollector<Collection<Times>, Times> beanCollector =
    new BeanCollector<Collection<Times>, Times>(times, BeanCollector.MODE_ALL);

AttributeDefinition space1 = beanCollector.getPresentationHelper().addSpaceValue();
beanCollector.addAttribute("path-test", new PathExpression<>(Times.class, "id.dbBegin"), null, null);
beanCollector.addAttribute("rule-test", new RuleExpression<>(Times.class, "§test-import"), null, null);
beanCollector
    .addAttribute(
        "sql-test",
        new SQLExpression<>(
            Times.class,
            "?" + query.getName(), Object[].class),
        null, null);
beanCollector.addAttribute("virtual-test", "I'm virtual", null, null, null);
beanCollector.addAttribute("picture", new Attachment("picture", Environment.getConfigPath()
    + "/icons/attach.png"), null, null);
beanCollector.setAttributeFilter("path-test", "creation", "dbEnd", "pause", space1.getName(), "project",
    "comment");
//more fields on one line (one field has grid-width 3)
beanCollector.getPresentable().setLayout((Serializable) MapUtil.asMap(L_GRIDWIDTH, 12));
//let the field 'comment' grow to full width
beanCollector.getAttribute("comment").getPresentation()
    .setLayoutConstraints((Serializable) MapUtil.asMap(ATTR_SPANCOL, 11, ATTR_BORDER, 1, ATTR_SIZE, 150));

/*
 * add a specified action
 */
beanCollector.addAction(new SpecifiedAction<>("ant", null));

/*
 * save it as beandef
 */
BeanDefinition.define(beanCollector);
beanCollector.saveDefinition();
```

Performance

Generic features like authorization for actions and data, filtering of columns and field, loading bitmaps etc. may slow down the application performance. The following tips may increase the performance:

set the following *environment.xml* properties:

- extract all sub-jars from main jar 'nano.h5.xxx.jar'. nested jar loading is really slow.
- if you don't need data-permissions, disable check of data-permission: *check.permission.data=false*
- set log-level to info: *default.log.level=8*
- disable multiple field/column filter: *collector.use.multiple.filter=false*
- get only 50 lines per search: *service.maxresult=50*
- don't use the attribute pre-filter: *bean.use.beanpresentationhelper.filter=false*
- turn off replication: *use.database.replication = false*

Android

The *nano.h5* framework works on android systems, too. But upto this time I coudn't find a proper O/R mapper supporting full JPA 2.0 and providing an EntityManager through an persistence.xml file.

The O/R mappers *EBean* and *ORMLite* are able to work on JPA-annotations, but no *javax.persistence.EntityManagerFactory* was implemented. The standard jpa persistence providers use libraries that don't work on android - or they have to many methods which causes the exception:

```
    Dex Loader] Unable to execute dex: method ID not in [0, 0xffff]: 65536
Conversion to Dalvik format failed: Unable to execute dex: method ID not in [0, 0xffff]: 65536
```

So, the nano.h5 framework is useless on android systems at this moment.

Other restrictions on android:
*all libraries in your environment directory must contain dex-classes. you can't simply add a standard library without transforming it to*dalvik classes.
ant wont work, so
* as hibernate doesn't work on android too, *hibernate-tools* as bean-jar generator is not usable. The same problem exists on openjpa!
on android an error occurs creating an xml-file for authorization. perhaps an empty authorization file will be created resulting in an error on next application start.
adding eclipse-link to the apk file results in error: *Unable to execute dex: method ID not in [0, 0xffff]: 65536*
* Your project is too large. You have to many methods. There can only be 65536 methods per application. see herehttps://code.google.com/p/android/issues/detail?id=7147#c6

Tutorials

Starting from beginning with a new project

Creating a new data model

Introduction

I've looked for free solutions to create a data model through a java application - or better through an online html5 application. UML-2 using stereotypes to define entity relationship informations would be my first selection. ArgoUML 0.34 is such a java application - but since its development was stopped, i preferred another tool, that is directly done to create ER models: The SQL Power Architect:http://www.sqlpower.ca/page/download?fileName=http://download.sqlpower.ca/architect/1.0.7/community.
All the found online tools have constraints, so they are not usable in a free way:

Here is a list of currently available browser solutions to create model graphs:

- https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools
  - http://www.jrromero.net/tools/jsUML2
  - Gliffy (https://www.gliffy.com/)
  - Creatly.com
  - Diagram.ly
  - LucidChart
  - MxGraph (jGraph)
  - Draw.io
  - PonyORM
  - Mogway ER Designer (Java Webstart, creates DDL) http://mogwai.sourceforge.net/erdesignerng/erdesignerng.jnlp

The Model

- Please downlad SQL Power Architect from http://www.sqlpower.ca/page/download?fileName=http://download.sqlpower.ca/architect/1.0.7/community.
- After extracting it and starting the architect.jar file, you can directly start modelling through a click on right mouse button and selecting 'new table'.
- Create your model (using 'T' for a new table and 'C' for a new column)
- Hit Menu 'Create SQL Script' using the defaults and saving the shown sql script into your tsl2.nano.h5 environment directory 'anyway' with name 'anyway.sql'.
- Go into this environment directory and start the script 'mda' with parameter 'anyway'.

Database Providers and Dialects

There are defined ANSI SQL Standards, but most datbase providers use additional specific function sets. So, JPA providers like hibernate use distinguished database dialects on different database systems.

For a small and fast java database, providing most features of SQL-92, HSQLDB is the first selection. HSQLDB provides additional features to switch on compatibility mode for some dialects:
- DB2 : SET DATABASE SQL SYNTAX DB2 TRUE;
- MySQL : SET DATABASE SQL SYNTAX MYS TRUE;
- MS SQL Server : SET DATABASE SQL SYNTAX MSS TRUE;
- Oracle : SET DATABASE SQL SYNTAX ORA TRUE;
- PostGres : SET DATABASE SQL SYNTAX PGS TRUE;

These dialects can be switched on through the connection-url, too:
- DB2 : ;sql.syntax_db2=true
- MySQL : ;sql.syntax_mys=true
- MS SQL Server : ;sql.syntax_mss=true
- Oracle : ;sql.syntax_ora=true
- PostGres : ;sql.syntax_pgs=true

This is useful if DDL generators create specific dialects - or simply for testing.

List of data-sources:

http://publib.boulder.ibm.com/infocenter/wsdoc400/v6r0/index.jsp?topic=/com.ibm.websphere.iseries.doc/info/ae/ae/rdat_scriptool.html

JDBC-Drivers

```
IBM DB2
jdbc:db2://<HOST>:<PORT>/<DB>
COM.ibm.db2.jdbc.app.DB2Driver

JDBC-ODBC Bridge
jdbc:odbc:<DB>
sun.jdbc.odbc.JdbcOdbcDriver

Microsoft SQL Server
jdbc:weblogic:mssqlserver4:<DB>@<HOST>:<PORT>
weblogic.jdbc.mssqlserver4.Driver

Oracle Thin
jdbc:oracle:thin:@<HOST>:<PORT>:<SID>
oracle.jdbc.driver.OracleDriver

PointBase Embedded Server
jdbc:pointbase://embedded[:<PORT>]/<DB>
com.pointbase.jdbc.jdbcUniversalDriver

Cloudscape
jdbc:cloudscape:<DB>
COM.cloudscape.core.JDBCDriver

Cloudscape RMI
jdbc:rmi://<HOST>:<PORT>/jdbc:cloudscape:<DB>
RmiJdbc.RJDriver

Firebird (JCA/JDBC Driver)
jdbc:firebirdsql:[//<HOST>[:<PORT>]/]<DB>
org.firebirdsql.jdbc.FBDriver

IDS Server
jdbc:ids://<HOST>:<PORT>/conn?dsn='<ODBC_DSN_NAME>'
ids.sql.IDSDriver

Informix Dynamic Server
jdbc:informix-sqli://<HOST>:<PORT>/<DB>:INFORMIXSERVER=<SERVER_NAME>
com.informix.jdbc.IfxDriver

InstantDB (v3.13 and earlier)
jdbc:idb:<DB>
jdbc.idbDriver

InstantDB (v3.14 and later)
```

```
jdbc:idb:<DB>
org.enhydra.instantdb.jdbc.idbDriver

Interbase (InterClient Driver)
jdbc:interbase://<HOST>/<DB>
interbase.interclient.Driver

Hypersonic SQL (v1.2 and earlier)
jdbc:HypersonicSQL:<DB>
hSql.hDriver

Hypersonic SQL (v1.3 and later)
jdbc:hsql:<URL>
org.hsql.jdbcDriver

H2 (old)
jdbc:h2:<URL>
com.h2database.Driver

H2
jdbc:h2:<URL>
org.h2.Driver

Derby/Cloudscape/JavaDB
jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*
org.apache.derby.jdbc.EmbeddedDriver
org.apache.derby.jdbc.ClientDriver

Microsoft SQL Server (JTurbo Driver)
jdbc:JTurbo://<HOST>:<PORT>/<DB>
com.ashna.jturbo.driver.Driver

Microsoft SQL Server (Sprinta Driver)
jdbc:inetdae:<HOST>:<PORT>?database=<DB>
com.inet.tds.TdsDriver

Microsoft SQL Server 2000 (Microsoft Driver)
jdbc:microsoft:sqlserver://<HOST>:<PORT>[;DatabaseName=<DB>]
com.microsoft.jdbc.sqlserver.SQLServerDriver

MySQL (MM.MySQL Driver)
jdbc:mysql://<HOST>:<PORT>/<DB>
org.gjt.mm.mysql.Driver

Oracle OCI 8i
jdbc:oracle:oci8:@<SID>
oracle.jdbc.driver.OracleDriver

Oracle OCI 9i
jdbc:oracle:oci:@<SID>
oracle.jdbc.driver.OracleDriver

PostgreSQL (v6.5 and earlier)
jdbc:postgresql://<HOST>:<PORT>/<DB>
postgresql.Driver

PostgreSQL (v7.0 and later)
jdbc:postgresql://<HOST>:<PORT>/<DB>
org.postgresql.Driver

Sybase (jConnect 4.2 and earlier)
jdbc:sybase:Tds:<HOST>:<PORT>
com.sybase.jdbc.SybDriver

Sybase (jConnect 5.2)
jdbc:sybase:Tds:<HOST>:<PORT>
com.sybase.jdbc2.jdbc.SybDriver
```

Actual list: http://infocenter.pentaho.com/help/index.jsp?topic=%2Fsupported_components%2Freference_jdbc_drivers.html

Known Problems and some Solutions

- Performance going down:
  - to low memory for jvm:
    - enhance the maximum vm memory on start: change run.bat and add option '-Xmx1024m' on line starting with 'java'

    - non-persistable entities (annotated as @Entity, but not bound to any database table) are loading slow
        - remove that beans from bean list view: create a regular expression hiding that entities for property 'bean.class.presentation.regexp' in environment.xml (Example: *^(?!(.MyBean1)|(.MyBean2)).**).
    - class-loading is slow through nested jar loading
        - extract all jar-files from tsl2.nano.h5.xxx.jar and copy them into the environments directory.
    - persistence provider has problems on specific beans:
        - use another persistence provider

- bean attributes have wrong default properties read from jpa annotations
    - if your beans don't decapitalize names the standard way, field names not be found. E.g. a name like MYName should decapitalized like this: MYName (not mYName!, see *Introspector.decapitalize(String name)*). If your beans can't be changed, you can switch the environment property *bean.attribute.decapitalize* to *false*.

- Beans showing currencys with EURO can't be saved
    - Set the character encoding of your browser's current page to *UTF-8*. E.g. in Chrome: Menu=>Tools=>Encoding

- no datasource for Sqlite available - apache BasicDatasource uses JavaBeans what is not usable on android
- ORMLite (4.48) not usable through persistence.xml because no EntityManagerFactory/EntityManager implementation available
- hsqldb 1.8 is not able to do *select max(count(color)) from Colors group by color*
- hibernate3 is - on special circumstances - not able to load fields with empty ('') strings. This results in an**java.lang.StringIndexOutOfBoundsException: String index out of range: 0**
- hibernate4 is not able to generate correct case-sensitive class-names if table-names contain *''. Example: SYS_TABLE will result in class _SysTable*, but variables try to declare *Systable*.
- hibernate-tools 4 needs slf4j-api-1.6.1.jar! other slf4j libraries will result in problems.
- If an Enum value overwrites a method (like *toString()*), the declaring class of this value is anonymous. serializing it through simple-xml will write the class with postfix perhaps *$1*. This can't be de-serialized through simple-xml.
- Some array types can't be de-serialized through simple-xml.
- a field holding a collection is not enabled
    - on oneToMany relations a cascading type must be given to be enabled

- jpa-providers:
    - eclipse-link: nullpointer on java.net.URL on loading persistence.xml
        - eclipseLink loads the METAINF/persistence.xml through another classloader. the bean-jar file will be searched from META-INF/ parent directory
        - use prefix '!' to define, that no environment-prefix will be added. E.g. '!mybeans.jar'. to enable text-input on jarfile property, change environments 'login.jarfile.fileselector' to false.
        - persistable fields of type Date have to annotate a temporal type
    - datanucleus: metamodel has to be statically generated and compiled before. this is done automatically on compiling the beans if the datanucleus libraries are present.
    - openjpa: all jar-file names are invalid:
        - openjpa needs the beans jar file packed into the main jar file!
        - dynamic class enhancement must be enabled (see https://openjpa.apache.org/builds/2.2.2/apache-openjpa/docs/ref_guide_pc_enhance.html)! start the jvm with option: *-javaagent:[path-to-extraced-openjpa-zip-file]/openjpa-2.x.x.jar*
        - openjpa properties: https://openjpa.apache.org/builds/2.2.2/apache-openjpa/docs/ref_guide_conf_openjpa.html
    - openjpa schema-tool does not create schema.xml for hsqldb:
        - in persistence.tml <property name="openjpa.jdbc.DBDictionary" value="hsql(SupportsSchemaForGetTables=false)"/> hinzufuegen
        - see http://openjpa.apache.org/builds/2.3.0/apache-openjpa/docs/ref_guide_dbsetup_dbsupport.html
    - ebean 3.3.2: "is not an enhanced entity bean. Subclassing is not longer supported in Ebean"
        - ebean versions after 2.8.1 don't support DynamicProxies any more. enhancing has to be done statically through maven-builds.
    - batoo-jpa: the jar-file has to have a protocol prefix like 'file:' (e.g.: file:./mybeans.jar). to enable text-input on jarfile property, change environments 'login.jarfile.fileselector' to false.
    - ebean 2.8.1: "@OneToMany MUST have Cascade.PERSIST or Cascade.ALL because this is a unidirectional relationship"
    - ormlite: "Generated-id field 'myID' in MyEntity can't be type STRING. Must be one of: INTEGER INTEGER_OBJ LONG LONG_OBJ UUID"
    - ormlite: "No fields have a DatabaseField annotation in class org.anonymous.project.Address"

- SimpleXml: TransformationException on Lists
    - use ListWrapper instead of type List

Changelog

Version	Date	Description
0.0.1	06.07.2013	First alpha Version
0.0.2	21.09.2013	beta Version (full basic feature implementation)
0.0.3	01.01.2014	beta Version (basic features + authorization + workflows + rules)
0.0.4	20.03.2014	full featured version (replication, attribute- and beandef-extensions QueryResult, RuleAttribute, many refactorings)

0.0.4b	11.05.2014	lots of corrections, partially working on android
0.0.4c	25.05.2014	classloader transformation changed to enable loading persistence.xml through EclipseLink and OpenJPA
0.0.5	01.06.2014	supporting non-jpa-persistence-providers like ebean, ormlite through NanoEntityManagerFactory
0.1.0	29.06.2014	supporting non-jpa-persistence-providers like ebean, ormlite through NanoEntityManagerFactory
0.6.0	13.07.2014	using websockets to support rich client gui interactions
0.7.0a	01.11.2014	new plugin interface on beandefintions (path changed to 'presentation'). specified rules and actions are pluggable into beandef or attributedef now.
0.7.0b	09.11.2014	gui fading, offline message, web-cache through manifest, testing batoo-jpa
0.7.0c	15.11.2014	reverse-engineering with openjpa support, jar resolving enhanced
0.7.0d	11.01.2015	auto creating new databases through an equal named sql file
0.7.0e	27.03.2015	attachments extended
0.7.0f	10.05.2015	new: RESTful service access, mouseclick access on dependency listeners
0.7.0g	17.05.2015	new: Secure Attributes: hashes or encrypts attribute values on runtime
0.7.0h	17.10.2015	webstart/jnlp, war, automatic translation
0.8.0a	15.11.2015	refactorings, replication enhanced, dependency listeners now persistable/configurable, changes on beandef xml and xsd
0.8.0b	07.01.2016	rule-listener, rule-cover enhanced, new app package: timesheet, bean-actions parametrized with annotations
0.8.0c	01.02.2016	many fixes, bean-configuration provides creation of actions (e.g. for REST service calls showing the JSON result)
0.8.0d	08.02.2016	statistics now showing bar charts with xgraph
0.8.0e	24.02.2016	bugfixes, environment-action --> administration-action