The database management application **JEntigrator**.

White paper.
Alexander Imas(alexander.imas@gmail.com).
2017, Darmstadt.

1.Definition.

The **JEntigrator** is a file-based scalable database management application implemented as a **java** program. It is intended to manage semi-structured evolving data objects and to serve as a basis to build applications using such data objects. It is published as an open source project under the GNU GPL license on the github platform. Binaries can be found on the maven platform.

2. The concept.

There are hundreds of database management systems on the market. The reason to develop another one is the fact that existing systems don't efficiently support management of evolving data objects.

According to the object-oriented paradigm the real world is considered as a set of objects. An object has a state and behavior. A real object can be represented as a program object having a set of named fields reflecting object's state and a set of methods reflecting its behavior. We distinguish evolving and none evolving objects. None evolving objects change only values of state variables during their lifetime. Number and names of state variables and set of methods do not change. Evolving objects can change a set of state variables and methods during their lifetime.

A relational database keeps data in tables. The structure of data objects defines the scheme of the database – a set of related tables. The database designer creates the scheme and the application designer creates a front-end application that provides interface to the end user and communication with the database. For not evolving data objects all this work will be done only once in the developing time. The end user takes over management of state variables in the working time through the application interface.

A much more complicated challenge is management of evolving data. Both the scheme and the application code must be changed when data objects change their structure. That dramatically increases costs and time of the application evolution(migration).

NoSQL databases have no scheme. The application developer doesn't need the assistance of the database designer and can complete the evolution himself . The participation of the application developer in migration is inevitable.

A **facet-oriented data model** provides the possibility to design/evolve data objects in the working time without any coding. This work can be delegated to the end user. The facet-oriented data model is based on the assumption that a real object can be represented as a composition of more elementary components called **facet**s.

A **facet** is defined as a portion of data and associated with it behavior. For example the **phone** facet includes the phone number (data) and the call (behavior). Another example is the **image** facet including the image file (data) and viewing of the image(behavior).   A **facet** can be implemented as a program object where the data component is represented as a set of named fields and the behavior is implemented as a facet's handler – the code processing these fields.

The most fundamental fact is that the set of facets is essentially smaller and stabler than the set of data objects. The second important fact is that the data object structure can be built or modified in the same fashion as the content – by assigning data to the object without changing of the application code. The data object can be considered as a facet container and the data object design means to add/remove facets to/from it . The application developer must provide the permanent part of the application: a pool of facets and the interface to manage universal data objects – facet data containers. The end user takes over the design and maintenance of data objects, again, without any coding. Both end users and developers benefit from this architecture. An end user can evolve data objects without assistance of developers. A facet developer must provide only a facet code without intervention into the application code.

The initial application **JEntigrator** contains only a small pool of common facets and interface to manage universal facet containers called **entities.** To adapt the application to user’s business area the **JEntigrator** uses the extension mechanism. An extension is an entity containing a library with facet handlers relevant to certain business area. It is enough to insert an extension entity into the database to enhance the facets pool. The application code stay intact.

3. Demo.

Lets take a closer look on demo databases . The web viewer provides read access to databases through a web browser. The viewer responses on user’s actions and shows the relevant context page. The demo home page contains four demo databases: Blank , Community   Movie and Northwind. Each database illustrates different aspects of the application. The Blank is an empty database containing minimum of entities and facets. The   Community database demonstrates some common facets. The Movie database implements a graph database using the data from the well known sample **neo4j**’s movie graph database. The Northwind database simulates a relational database using the data from the of well known sample **Microsoft**’s Northwind relational database. The click on the database item shows the base navigator context – the origin point for navigation through the database.   Note how simple and intuitive it is to navigate through the database and find entities. The data inside of a database is fully transparent. You can investigate any entity in the entity editor context , browse files in the entity’s folder through the folder context , integrate data from multiple entities into the view context , etc.   Chapters below tell more how to use different contexts.

3.1. Select entities.

There are three ways to find searched entities: select by properties, search by label and list by category.

As mentioned above , each data object in the **JEntigrator** is represented by the universal facet container called entity. An entity has two mandatory fields – identification key and label , unique within the database. A key is a final string assigned to the entity by creation that cannot be changed later. A label is a human friendly name that can be changed during entities lifetime. A third mandatory field is a type string that define the primary facet of the entity – the category. Some fields belong to the **property** group and are included into the global database index. The **label** and **entity**(type) fields are properties.

The Design item in the base navigator shows the property selector. To find entities having certain value of the property select the property and its value. The entity selector shows all corresponding entities. For example , in the **Community** database selection of property **entity** with **Contact** value shows all contacts the Entity selector. The click on the **Entity** button opens the Entity Facets context. The click on **Value** button shows the list of all selected entities. The click on the **Property** button shows the list of all entities, having the selected property independent of its values.

The Search item in the base navigator shows the search context panel to select entities by labels. Type a part of label into the input text box and press the **Enter** button. The list of labels appears in the entities selector. Select an entity to open a list of its facets.

The All categories item in the base navigator shows a list of all facets. The click on the facet item shows the list of all entities having this facet assigned. That’s all about standard selection tools. Special selection procedures may be created by programming additional query entities like the template Query all entities**.** The first important note to make is that :

- *a user does not need any pre-information about the structure of the database before browsing the database.*

3.2. Work with entities.

3.2.1. The structure of an entity.

Each entity is represented by a separate XML file in the **JEntigrator.** Logically it contains a set of sections called elements having a unique name within an entity. There are two mandatory elements – **attributes** and **property**. **Attributes** contains system information and **property -** fields included into the global database index. Number and names other elements in the entity depend on assigned facets.

An element in turn can contain any number of cores. A core is a set of three named strings called **type, name** and **value.** A core must have unique name within the element. Let us inspect entities Icons and ACTED_IN in the **Movie** database. We can see that both entities besides mandatory **attributes** and **property** contain two other elements – **fhandler** and **jfacet.** These elements keep information about programs that can be used to process this entity. The next important note to make is that :

- *each entity contains within itself the information about programs that can process it .*

In other words the **JEntigrator** is a data-driven application.

The **folder** facet assigns a separate directory to the entity where files can be stored in the native format. For example the Icons contains all icons.

3.2.2. The functionality of an entity.

The **JEntigrator** provides access to functionality of an entity through the entity facets list that displays a list of facets assigned to an entity. A user opens the desired facet's context to interacts with the entity. Again:

- *entities of the same type may have different sets of facets.*

3.2.3. Complex entities.

In the real world many objects are compounds of other objects. The **JEntigrator** provides the possibility to set component-container relation between entities. A component may have many different containers (n to m relation type). A complex entity is a container having some components. Components can be containers too. Note that this construct makes the database relational in the sense that:

- *by changes only the targeted component must be updated.*

The standard way to access the certain facet of the certain component may be a long navigation through components and facets. The digest context shows an overview of a complex entity as a tree sorted by facets and components. A user call this context from the "Context" menu in the "Entity" facet. The "Components" menu item shows the structure of the complex entity.

3.2.4. The graph.

A graph is one of very common types of relations. A graph data structure consists of nodes connected by edges. The clone of Neo4j demo database **Movie** includes the extension graph adding the graph functionality to the **JEntigrator.** We can see facets Nodes and Edges in the list of categories. Persons and movies belong to nodes, relations ACTED_IN, DIRECTED, etc.. belong to edges. The facet node displays a list of connections to other nodes in the graph. The click on the Graph menu item shows the node and its relations to other nodes as a graph. The click on a node or an edge displays the pop up menu. The nodes menu consists of 4 items: Entity, Relations, Expand and Network. The **Entity** menu item opens the facet list of the node. The **Relations** clears display and shows the relations of the node. The **Expand** adds relations of the node to the graph. The **Network** shows all nodes and relationship connected to a node. The demo is a connected graph so **Network** shows all nodes and all edges in the database. It takes some time to build large networks. The graph context displays controls to search nodes and filter edges.

The review of the graph extension let us make some important notes.

- *The base application **JEntigrator** does not change if we add an extension.*
- *All extension information is enclosed in the extension's entity within the database. There are no additional files outside of the database.*
- *The **JEntigrator** recognizes new facets automatically from the extension's library and includes them in the list of categories.*

- *The **JEntigrator** absorbs new facet contexts from the extension's library and displays them to a user through the facet's menu.*

3.2.5. Associations.

The **JEntigrator** provides two ways to group data objects together : bookmarks and indexes. "Data objects" here are entities, files or web links. An entity of the **bookmarks** type is an unordered collection of links to original objects. The entity Authors in the **Community** database is an example of bookmarks. An entity of the **index** type contains links to original data objects hierarchically ordered as a tree. The Favorites entity in the **Community** database is an example of index. Note that:

- *adding of bookmarks and indexes does not change original data objects and the global database index.*

3.2.6. Tables.

Entities having types 'view' or 'query' serve as table generators. They content entity selectors in form of class files stored within entity's folder. The query selector class returns only a list of selected entities. The structure of the table is defined by information within the query entity. The view selector returns the ready table model. The corresponding facet loads this class, makes selection and represents results in form of table. For example tables in the Northwind database are implemented as Queries and views as Views . Note that the **JEntigrator** builds tables for views and queries by collecting data from entity files located on the disk. This procedure is slower as getting tables in traditional DBMS, where tables are native data containers located in the memory.

4. Summary.

The application **JEntigrator** aims to manage evolving semi-structured data objects. The database is implemented as a file directory where each data object called **entity** is stored in a separate XML file. An entity is considered as an universal facet container. The evolution (migration) of an entity is carried by adding/removing facets to it through the GUI without any coding. Entities of the same type may have different sets of facets or properties. The database contains the global properties index including all properties of all entities. An entity contains three mandatory properties : an identification **key** (immutable and unique within the database), a **label** (editable string unique within the database) and an **entity** . The **key** is a string assigned to the entity by creation. The **label** is a human friendly name of the entity. The **entity** is a type (category) of the entity. A user manage entities through the application GUI.

Distinctive features.

- The **JEntigrator** integrates the application and the data management system in one pure **Java** program.
- The **JEntigrator** is a low resource consuming application. The data is resident on the disk. Only currently processing entities are loaded into memory.
- The **JEntigrator** is a scalable application. Additional features can be added to it by

inserting into the database extension entities, containing extension libraries.

- The **JEntigrator** encloses applications into entities within the database. The only external file is the **jentigrator.jar**.
- The **JEntigrator** provides the possibility to build complex entities through container-component relations between entities. By changes only the targeted component must be updated.
- The **JEntigrator** provides a unified comprehensive graphic user interface. A user can browse the database like a file system without any previous information about its structure. Each piece of data can be reached through a few clicks.

5. Third party software used.

Environment.

Oracle Java EE

commons-codec-1.10.jar

collections-generic-4.01.jar

colt-1.2.0.jar

concurrent-1.3.4.jar

j3d-core-utils-1.3.1.jar

j3d-core-1.3.1.jar

jung-3d-2.0.1.jar

jung-3d-demos-2.0.1.jar

jung-algorithms-2.0.1.jar

jung-api-2.0.1.jar

jung-graph-impl-2.0.1.jar

jung-io-2.0.1.jar

jung-jai-2.0.1.jar

jung-visualization-2.0.1.jar

jung-jai-samples-2.0.1.jar

jung-samples-2.0.1.jar

stax-api-1.0.1.jar

vecmath-1.3.1.jar

wstx-asl-3.2.6.jar

Development tools.

Eclipse IDE : Eclipse Java EE Neon

LibreOffice : Version: 5.1.4.2

Libraries included into databases.

jquery-3.1.1.js

jstree.js

vis.js

Numerous code snippets found on the Internet.