D3.2/D3.3/D4.5

KiWi Core System Documentation

Project title:	Knowledge in a Wiki
Project acronym:	KIWI
Project number:	ICT-2007.4.2-211932
Project instrument:	EU FP7 Small and Medium-Scale Focused Research Project (STREP)
Project thematic priority:	Information and Communication Technologies (ICT)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	ICT211932/SRFG/D3.2/D3.3/D4.5/R/P/PU
Responsible editors:	Sebastian Schaffert
Contributing authors:	Rolf Sint, Stephanie Stroka, Mihai Radulescu, Szaby Grünwald, Thomas Kurz, Marek Schmidt, Klara Weiand, Jakub Kotowski
Reviewers:	
Contributing participants:	SRFG, BUT
Contributing workpackages:	WP3
Contractual delivery:	31.08.10
Actual delivery:	--

Abstract:

This document provides the basic ideas behind the selected features and the overall design of the knowledge model for software project management.

Keyword List:

Project Management, Knowledge Management,

Project funded by the European Commission within the Seventh Framework Programme ©

Table of Contents

Foreword: Purpose of this Report

This report is the accompanying documentation to the KiWi system. It describes the overall architecture and the major components of the core system as well as the different KiWi extensions that are already available.

1. Introduction

2. KiWi Feature Overview

The KiWi system consists of a core system (backend) and a number of applications building on top of this core system (see also Section 4). In the following, we give an overview over the features of the core system as well as the most important applications. A more detailed description of the system components and architecture is given in subsequent sections.

- **Content and Metadata Indexing.** The KiWi System allows indexing of the unstructured content as well as the associated metadata of textual and multimedia content for the purpose of searching. The content indexed by KiWi can be stored either in the KiWi system itself or come from external sources using Web Service calls.
- **RDF and Ontology Support.** All metadata in KiWi is represented using the standardised RDF data model and format and can be queried and reused in other applications supporting RDF. KiWi comes with a number of core ontologies that are directly supported and used by the system, e.g. Dublin Core, SIOC, FOAF, HGTAGS, SCOT, and Geo.
- **Semantic Search**. The KiWi core system offers a powerful semantic search engine that allows to search over content and metadata alike. The search engine offers a traditional, keyword based search as well as "facetted browsing", guiding the user through a refinement of the search results. The search engine can be used either directly or as a Web Service.
- **Transactions and Versioning**. All updates (content as well as metadata) to the KiWi Core System are wrapped in transactions and can be undone at any later time. A KiWi transaction covers all work performed within a "unit of work", e.g. changing the textual description of a content item, filling a form, or updating the position of a point on a map.
- **Activity Logging**. Every activity performed by a user in relation to content stored in KiWi is logged. The logging data is used on the one hand for calculating community equity values of information, and on the other hand for building user models for personalisation. Also, activity information can be displayed in a user's Dashboard to identify the "things that are happening" in a user's universe.
- **Community Equity**. KiWi is fully integrated with Sun's Community Equity system that allows to determine the social value of content indexed by the KiWi system. Content that has a high level of social interactivity (e.g. read very often, updated very frequently, commented a lot, rated very high) has a high community equity value. Community Equity metadata is represented in RDF using KiWi's
- **Tagging**. The KiWi system offers a tagging service for all content indexed by the system. The tagging functionality can be used either directly in KiWi or integrated in the systems where the content originates using web services or widgets. KiWi uses semantic tagging instead of purely keyword-based tagging, i.e. tags can be disambiguated and connected e.g. in a SKOS thesaurus. Tagging metadata is represented using the HGTags and SCOT ontologies.
- **Commenting**. Similar to tagging, KiWi offers a commenting functionality for all content indexed by the system that can be used either directly or integrated into other applications using widgets or web services. Comment metadata is represented using the SIOC ontology.
- **Content Annotation (RDFa)**. KiWi allows to annotate unstructured content using the RDFa format. KiWi supports RDFa for displaying content in the browser, and offers a special RDFa editor that allows to mark up and annotate the content in the KiWi system.

- **Semantic Forms (RDFa Forms)**. Advanced users can develop custom tailored forms for editing content and metadata in the KiWi system. These "Semantic Forms" also use the RDFa format, but extend it to be used in forms.
- **Vocabulary Management**. For updating SKOS thesauruses, the KiWi system offers a vocabulary management tool that allows to associate concepts with each other and create new subconcepts. The vocabulary management tool also recognises "free" tags and allows to turn them into "semantic" tags by turning them into thesaurus concepts.
- **Metadata Editing**. In addition to the "user-friendly" ways of adding metadata (tagging, annotation, information extraction, vocabulary manager), the KiWi system also offers the possibility to directly edit the metadata associated with content in the system so that advanced users can change the metadata as desired.
- **Rule-based Reasoning (Enabling Technology)**. In addition to the ordinary RDF support, KiWi contains a rule-based reasoner that can reason over the RDF metadata contained in the index. KiWi already contains a set of simple rules for RDFS reasoning, which can be extended by developers by arbitrary functionalities, e.g. for implementing specific support for a certain schema.
- **Information Extraction (Enabling Technology)**. The KiWi core system offers several methods for advanced information extraction from the textual content indexed by the KiWi system. Information extraction is on the one hand used as a base for tag recommendations and for recommendations of related content, and on the other hand for semi-automatic annotation of the content in the Annotation tool. Recommendations are also available as Web Services.
- **Recommendations and Personalisation (Enabling Technology)**. In addition to the recommendations based on information extraction, the KiWi system also offers personalised recommendations and personalised search results based on the user model of the logged in user. Recommendations are then calculated based on the tagging activity performed by the user.
- **Advanced Structured Querying (Enabling Technology)**. Beyond the simple semantic search described above, the KiWi system also offers advanced structured querying using a variety of formalisms, e.g. SPARQL. A particularly salient feature of KiWi is the query language KWQL that combines the simplicity of search with the power of structured querying. KWQL queries can be used as alternative to the normal search and can also be composed using a visual editor.

3. Content Versatility: Same Content, Different Views

As we have already outlined in the introduction, what we call the ``Wiki Principles'' is actually applicable to many Social Software applications. It is therefore close at hand to build generic platforms for developing different kinds of Social Software. And indeed, there are several products already available that aim to deliver a platform that allows to combine wikis, weblogs, social networking, etc. Such platforms are for example Clearspace Community[1] from Jive Software, Community Server[2] from telligent, and Liferay Social Office[3] from Liferay.

However, all these systems only provide integration on the user interface level and still see wiki articles, blog posts, etc. as separate kinds of content that is visualised in a specific way. This has a number of serious disadvantages. For instance, something that started as a wiki article can never become a blog post, it will never be possible to attach comments to a wiki article if not foreseen in the data model, and new kinds of content (like our TagIT locations, cf. Section \ref{sec:tagit}) cannot be added easily without also doing fundamental changes to the system.

KiWi follows a completely different approach which we call “Content Versatility”. The underlying principle is that every piece of information is a combination of human-readable content and associated metadata, and that the same piece of information can be presented to the user in many different forms: as a wiki page, as a blog post, as a comment to a blog, as a photo, or even in a bubble in a map-based application. The decision how the information is displayed is taken based on the metadata of the content, and the context of the content and the user (e.g. metadata, user preferences, different applications). Metadata is represented using RDF and thus does not require a-priori schema definitions, so the data model of the system can be extended even at runtime.

In KiWi, we call the smallest piece of information a “content item”. A content item is identified by a URI and consists of human-readable content in the form of XHTML and associated metadata in the form of RDF. The KiWi core system provides means to store, update, search, and query content items, and offers automatic versioning of all updates (content and metadata). Whereas core properties of a content item (like the content, the author, and the creation date) are represented in XML and persisted in a relational database, all other properties can be flexibly defined using RDF properties or relations. The URI of a content item is generated in such a way that it is possible to make a KiWi system part of the Linked Open Data cloud. This allows to easily integrate KiWi content with other services on the Semantic Web.

[1] http://www.jivesoftware.com/products/clearspace-community

[2] http://communityserver.com/

[3] http://www.liferay.com/web/guest/products/social_office

4. KiWi Core Applications

Content Versatility makes it possible to offer completely different views on the content inside the KiWi system without any modifications to the core system or data model. We call such views “KiWi Applications”, and they are one kind of extension offered by the KiWi system (others are currently: KiWi Services, KiWi Widgets, KiWi Actions, and Exporters/Importers). In the following, we describe the set of applications that are part of the KiWi System to illustrate how the Wiki Principles and Content Versatility are realised in KiWi. The applications have been selected based on the requirements identified in the two KiWi use cases and accompanying SNML projects; additional applications are conceivable and reasonable. Additional applications will be offered as separate packages in later stages of the project.

KiWi applications share the same context, i.e. when the user browses a content item in the Wiki or Dashboard, she can switch to the TagIT application and view the same content item on a map or to the Inspector and get debugging information on the content item.

4.1. Extension Mechanism

KiWiPages, KiWiResourceResolver, ...

4.2. Wiki

The primary and most generic interface to the KiWi system is a Semantic Wiki. The layout and functionality (Figure above) of the KiWi Wiki is inspired by its predecessor IkeWiki\cite{ikewiki-eswc06}: the left column offers navigation functionality, the centre column contains the main (human-readable) wiki content, and the right column contains dynamic widgets that display additional information about the content item based on its metadata (e.g. a map or incoming and outgoing links).

The centre column by default displays the content of the content item. The section below the page title contains maintenance information about the item as well as the list of tags that have been associated with it. By clicking on the “Action” menu in the top right corner, the user can select to edit the content, edit the metadata (i.e., OWL Datatype Properties), edit the relations (i.e., OWL Object Properties) to other content items, edit the list of tags, and access the history of the item's content and metadata as provided by the versioning subsystem. In principle, the KiWi Wiki interface can thus be used in the same way as IkeWiki. Additional actions can be registered using KiWi's extension mechanism (e.g. domain specific actions like “Geolocate” or “Share”).

The widgets in the right column are visually part of the content box to emphasise that they contain additional information about the currently displayed item. Currently, the KiWi system offers to display the list of references (based on RDF relations with other items) and a “Stream of Activities” listing the recent activities associated with the content item (e.g. modification, tagging, annotation).

4.3. TagIT

TagIT is an application originally developed in a separate project with the goal to create the "Youth Atlas of Salzburg"[4], which has been running as a prototype successfully for over a year and has now been ported to the KiWi platform (Figure above). In TagIT, users browse through a map (based on Google Maps) where they can "tag" locations with descriptions and provide interesting information about them, e.g. cafés, bars, sports parks, hiking tours, etc. Such tags can be associated with categories from a SKOS thesaurus and with additional multimedia data like photos or videos. Other users can then browse or search for interesting locations using the same interface. In addition to the web-based platform, TagIT also offers a mobile client that can run on a GPS-enabled mobile phone. When visiting an interesting location, users can then start the TagIT application, take a picture of the location, add a short description and directly upload the "tag" using UMTS. The tag is automatically geolocated and immediately available for other users.

Although quite different on the user interface and in the way it is used, TagIT actually closely follows the wiki principles: everyone can add and edit tags, the system is easy to use, tags can be linked, tags are versioned, and different kinds of content are supported. On top of KiWi, tags are realised as content items and can thus be displayed in both the TagIT user interface and the previously described KiWi Wiki (in which case a small map widget is displayed in the right column showing the position).

4.4. Search

In addition to these different ways of presenting content, the KiWi core system also provides a generic search functionality accessible from within all KiWi applications. When a user switches to search and selects a content item, he is redirected back to the previous application afterwards. KiWi currently allows a combination of full-text search, metadata search (tags, types, persons), and database search (date, title). A more sophisticated search language is currently under development as Deliverable D2.2.

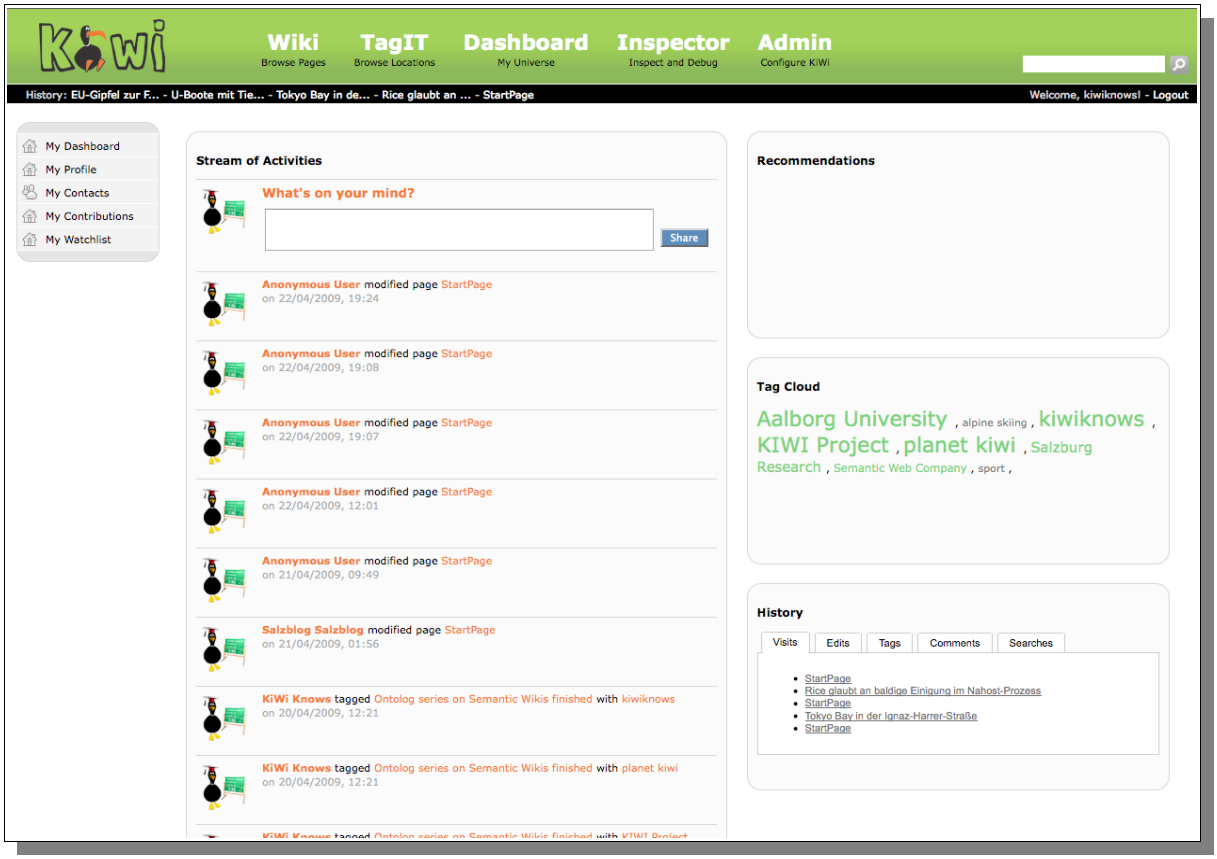
The KiWi search interface implements a so-called "facetted search" based on Apache SOLR (see Figure above): the user starts with a keyword search, resulting in a list of content items ordered by relevance or time. In case the user is not satisfied with the results, he then has the option to refine his search using one or more of the facets offered in the right column of the search result box. Currently, the KiWi system offers the facets "tags", "types", and "persons", which we have identified as the core facets needed in any system. For each facet, only the criteria occurring in the currently displayed search results are listed, together with a count of the content items matching the criterion. Selecting one of the criteria narrows down the search.

All search facets are included in the full-text search box; this decision has been made to provide all search functionality in one place without confusing the user and to allow advanced users to directly search using the text field. Also, it makes it much simpler to bookmark searches or include them in a user's personal stream of activities on the Dashboard.

In later stages of the project, it is planned to make the set of widgets customisable to adapt it to different application domains. For example, in a biology scenario, an interesting facet could be different protein structures, in a music scenario it could be different instruments, and in a history scenario it could be different countries.

4.5. Dashboard

[4]http://tagit.salzburgresearch.at



The Dashboard is a user's personal(ized) start page in the KiWi system. The Figure above shows an early stage of its user interface, which is currently still under development. The Dashboard follows the same general layout as the Wiki, i.e. the left column provides generic navigation while the centre and right columns contain the actual content. While the look of the Dashboard is freely customisable by the user, the KiWi core system by default provides the following information:

•the *Stream of Activities* is the most important part of the Dashboard: it contains an aggregated list of activities that happened in the user's "universe", i.e. updates to content items that are either explicitly watched by the user or implicitly added to the user's universe e.g. because they have been edited by the user, because they have been edited or rated "good" by one of the "friends" of the user, or because they have been recommended to the user based on previous activites by the personalisation component of KiWi

•the *Recommendations* widget provides a list of additional content items that might be relevant to the user; different recommendation algorithms are investigated as part of the Deliverable D2.8

•the *History* widget lists the content items the user visited or worked on recently to give quick access to the issues the user is currently concerned with

•the *Tags* widget lists the tags used by the user and is a quick and flexible means of structuring and accessing the content items that are relevant to the user; clicking on a tag redirects to the search interface described below

Besides the main view, the Dashboard is also the place where the user manages his own profile. The most important part of the user's profile is the list of friends, which is the primary way to use the social networking functionality of KiWi. The requirements analysis carried out as part of the KiWi use cases showed that social networking is a crucial aspect in a modern knowledge management system like KiWi as it helps define the context the user works in.

4.6. Admin

4.7. Inspector

The KiWi Inspector is an application developed for advanced users and developers. It provides a more technical insight into the current context. It currently provides the following functionalities:

•*Content Item Inspector*: displays the RDF data associated with the current content item as RDF/XML; the shown RDF data is the same as would be displayed to a Linked Open Data client when accessing the KiWi system

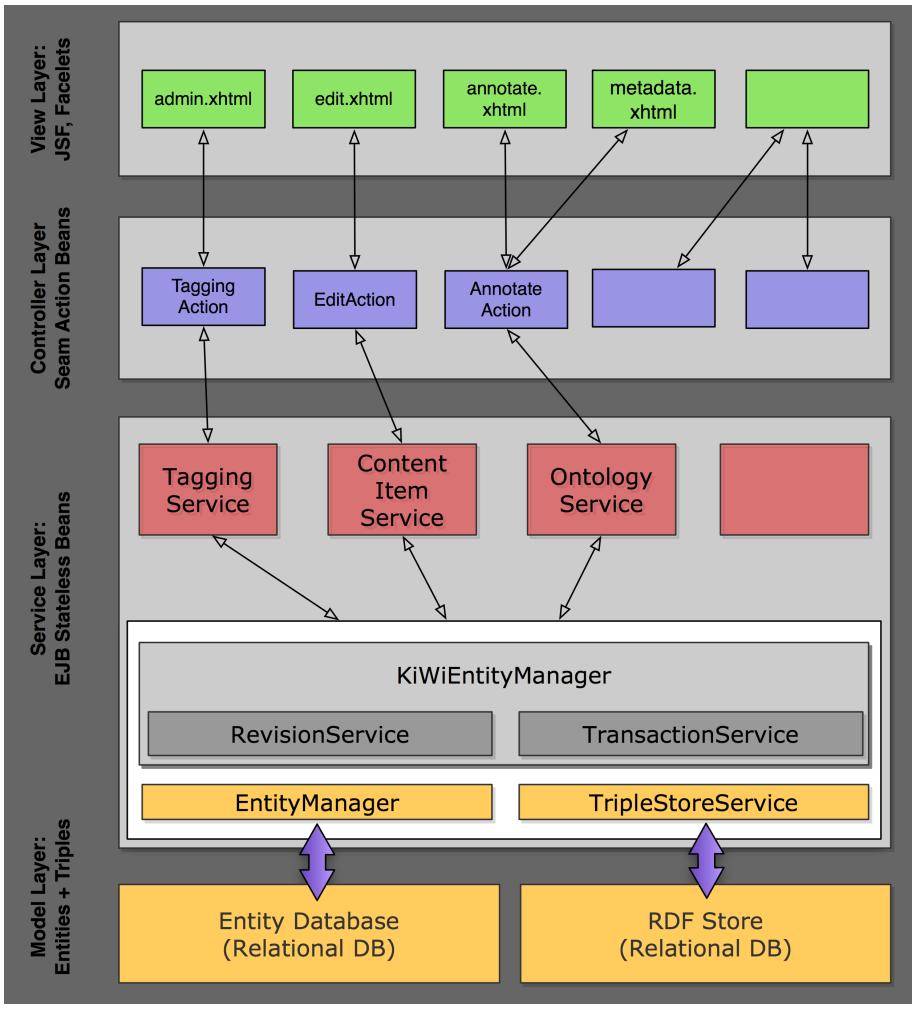
•*Tag Inspector*: displays a list of all tagging actions of the current content item and the RDF data generated by them as RDF/XML

•*User Inspector*: displays the RDF data associated with the currently logged in user as RDF/XML

In addition, the KiWi Inspector will later also provide more details about the revisions (particularly metadata revisions) of the current content item and additional debugging information as needed.

5. KiWi Architecture

The applications described in the previous section are built on top of the KiWi Platform, which provides all the core functionalities needed by most Semantic Social Software applications. In the following, we briefly describe architecture, core data model, and core services offered by the KiWi platform.



The KiWi system is implemented on top of JBoss Seam[5] and Java Enterprise Edition (Java EE 5), and thus follows a component- and service-oriented architecture. The Figure above depicts the overall structure of the KiWi system:

The *model layer* comprises the KiWi data model (see below) and is represented in a relational database, a triple store, and a full-text index. Entities are persisted using the Hibernate framework[6], which maps Java objects to relational tables. The KiWi triple store is a custom implementation also based on the relational database, because existing triple store implementations provide insufficient support for features like versioning and additional metadata about triples that are needed by KiWi. The full-text index is implemented using Hibernate Search and currently allows to search over title, textual content, tags, authors, and RDF literals.

The *service layer* provides services to other components in the KiWi system. Of central importance is the KiWi Entity Manager, which provides unified access to content items and RDF metadata (see

[5]http://www.seamframework.org

[6]http://www.hibernate.org

below). Further core services are the revision service -- taking care of versioning, and the transaction service, allowing to manage all updates to KiWi content in reliable transactions. Both services are heavily used internally by the KiWi Entity Manager and usually not used by further components. Besides these core services, the service layer may contain additional services that offer certain common functionalities. For example, the KiWi system currently offers an “ontology service” that provides convenient access to the triple store using higher-level concepts like “classes” and “properties”, and a “content item service” that allows to easily access all functionalities associated with content items (creating, loading, updating).

The *controller layer* consists of action components that implement a specific functionality in the KiWi system. For example, the Semantic Wiki application contains a “view action”, a “edit action”, a “annotation action”, and a “history action”, and the TagIT application contains a “explorer action” and a “tagging action”. Action components are usually very close to some functionality offered in the user interface, and they make use of service components to access the content in the KiWi system.

The *view layer* is implemented using Java Server Faces (JSF), which are used to generate the HTML presentation of the KiWi user interface and the user interaction with the system. JSF pages are linked with action components in the controller layer. Also part of the view layer are web services offered by KiWi. Currently, there are web services for accessing the triple store and SKOS thesauruses, and there is a “linked open data” service offering the content of the KiWi system to linked open data clients.

6. KiWi Data Model

KiWi's core data model makes use of few concepts, namely Content Items, Tags, and Triples. Additional functionality is added by "KiWi Façades".

6.1. Content Items

6.1.1. Description

The *content item* is the core concept of the KiWi system. It represents a "unit of information" in KiWi, e.g. a page about a certain topic, a user profile, etc. When a user accesses the KiWi system, he is always interacting with exactly one (primary) content item, the context content item. The context content item can be viewed, modified, and annotated by the user. Though changes might also affect other content items, the context content item is always the primary content item.

Each content item has both, a machine readable symbolic representation and a human readable textual or multimedia representation.

- content items are all different kinds of content and data items that are stored in the KiWi system, i.e. (wiki) pages, multimedia, users, roles, rule definitions, layout definitions, widgets, and possibly more. The KiWi system is not restricted a priori to specific content formats.
- URIs or blank nodes serve as machine-readable symbolic representations of resources, to be used in extended triples in the triple store. The URI is used to embed a resource in its context and provide machine-readable meaning, e.g. by annotation with formal annotations, reasoning, etc.
- the textual / media content related to a resource is meant for human consumption. The content resembles a wiki page, is easy to edit, supports linking, and is versioned. In the current design each wiki article in a specific language is represented as an own content item describing a single concept. The assumption for this design is that the content about a topic and its authors may differ from language to language. The definition of connections between content items with equivalent content but different languages can be accomplished with metadata relations.

These assumptions distinguish the KiWi data model from other wikis and content management systems. Most important, it treats all kinds of resources equally, leading to a very clean and simple model where every resource has both, a machine-readable and a human-readable description.

A consequence of the direct relationship between content items and RDF resources is that every RDF resource, even those representing widgets, layouts, users, or even rules, can also be described in human readable form.

Textual content is represented internally as (structured) XML documents that can be queried and transformed to other representations like HTML or XSL-FO (for PDF and other print formats) using standard XML query languages (XQuery, XSLT) or using the rule-based reasoning language developed in KiWi. The XML format used for page representations resembles a subset of HTML, taking into account only core structuring elements.

6.1.2. API

Package: `kiwi.model.content`

The representation of ContentItems in the source code is divided into several classes, of which we

describe ContentItem itself, TextContent, and MediaContent here:

ContentItem. Is the main entity for representing content items in KiWi. It has the following fields (some annotations left out for brevity):

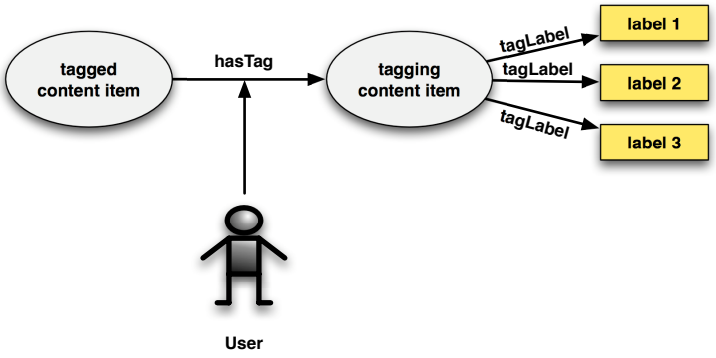
```
@Entity
public class ContentItem implements KiWiEntity, ... {
    @Id
    @RDF(Constants.NS_KIWI_CORE+"id")
    private Long id;

    @Temporal(value = TemporalType.TIMESTAMP)
    @RDF(Constants.NS_KIWI_CORE+"createdOn")
    private Date created;

    @Temporal(value = TemporalType.TIMESTAMP)
    @RDF(Constants.NS_KIWI_CORE+"modifiedOn")
    private Date modified;

    @OneToOne(fetch=FetchType.EAGER)
    @RDF(Constants.NS_KIWI_CORE+"hasTextContent")
    private TextContent textContent;

    @OneToOne(mappedBy="contentItem", fetch=FetchType.LAZY)
    @RDF(Constants.NS_KIWI_CORE+"hasMediaContent")
    private MediaContent mediaContent;

    @OneToOne( mappedBy="contentItem", fetch=FetchType.LAZY)
    @RDF(Constants.NS_KIWI_CORE+"hasExternContent")
    private ExternContent externContent;

    @OneToOne(mappedBy="contentItem", ...)
    @Immutable
    private KiWiResource resource;

    @OneToMany(fetch = FetchType.LAZY, mappedBy="taggedResource")
    private Set<Tag> tags;

    @RDF(Constants.NS_KIWI_CORE+"title")
    private String title;

    @RDF(Constants.NS_KIWI_CORE+"language")
    private Locale language;

    @ManyToOne(fetch=FetchType.LAZY)
    @RDF(Constants.NS_KIWI_CORE+"author")
    private User author;

    @CollectionOfElements(fetch=FetchType.LAZY)
    @RDF(Constants.NS_SKOS+"altLabel")
    private Set<String> tagLabels;

    private boolean deleted;

    @OneToMany(fetch=FetchType.LAZY)
    @RDF(Constants.NS_KIWI_CORE+"hasComment")
    private List<ContentItem> comments;

    private Long contentItemScore;

    @OneToMany(mappedBy="revisedContentItem", fetch=FetchType.EAGER)
    private List<Revision> revisions;
    …
}
```

INSERT HERE A PRETTY-PRINTED SOURCE FRAGMENT WITH SOME EXPLANATION: ContentItem, TextContent

6.2. Tags

6.2.1. Description

Tagging is one of two ways of annotating content items in the KiWi system. In KiWi, tags serve many different purposes, for example associating content items with certain topics or grouping content items in "knowledge spaces". There are two kinds of tags: explicit tags are explicitly added to a content item by a user; implicit tags are created by the system, e.g. based on automatic mechanisms like information extraction from text or reasoning on existing tags.



Conceptually, an explicit tag is a 3-ary relation between two content items (the tagged content item and the tagging content item) and a user (the tagging user or tagger). An implicit tag is a binary relation between two content items, a tagged and a tagging one. Tagging content items are identified using one or more labels that are available for annotating content items. In case of ambiguous tag labels (i.e. the same tag label for different content items), the KiWi system asks the user to choose the appropriate content item. If the user enters a new label that is not yet used elsewhere, it is displayed like a wiki-link to a non-existing page; when the user clicks on it, he is given the choice to either associate the label with an existing content item or to create a new content item explaining this tag label. Internally, a tag is furthermore given maintenance information like creation time and date and a URI for uniquely identifying a tag.

For example, the content item that describes "Mickey Mouse" could be tagged with the label "Mouse", thereby associating it with the content item describing "Mouse" (the animal). The tagged content item would be "Mickey Mouse", the tagging content item would be "Mouse", and the tag label used for tagging would be "Mouse", which is a tag label of the content item "Mouse".

Inside the system, a tag is mapped to an RDF structure that can be used for deriving additional RDF metadata by means of reasoning. Also, tags can be "lifted" to taxonomy or ontology concepts by advanced users, e.g. by using the "meaning of a tag" (MOAT)[7] or "social semantic cloud of tags" (SCOT)[8] ontologies. In this case, more information about the meaning and context of a tag becomes available, e.g. for reasoning or querying.

Tags can be used by the KiWi system for many different purposes. For example, tags can help with searching by offering a facetted search interface or by offering tag clouds. Furthermore, it is possible to derive user preferences from the tags she has used or to identify users with similar

[7]http://moat-project.org/

[8]http://scot-project.org/scot/

interests via clustering. Similarly, tags can also be used for grouping related content items, e.g. for defining group work spaces or for clustering thematically related items. Beyond that, the way how tags are used is left to the application developers and users that implement a specific instance of the KiWi system.

6.2.2. API

Package: `kiwi.model.tagging`

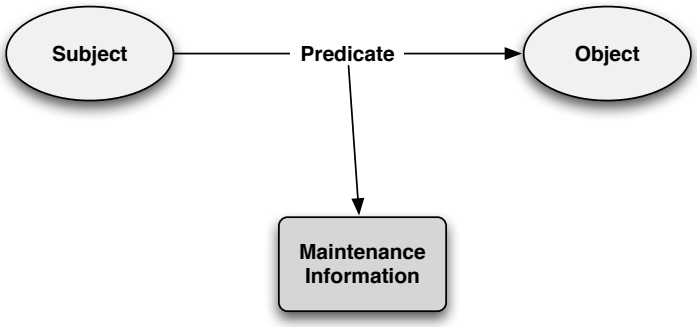
```
@Entity
public class Tag implements KiWiEntity, ... {

    @Id
    private Long id;

    @ManyToOne(cascade = {CascadeType.REFRESH}, fetch=FetchType.LAZY)
    @RDF(Constants.NS_HGTAGS+"taggedBy")
    private User taggedBy;

    @Temporal(TemporalType.TIMESTAMP)
    @RDF(Constants.NS_HGTAGS+"taggedOn")
    private Date creationTime;

    @ManyToOne(cascade = {CascadeType.REFRESH}, fetch=FetchType.LAZY)
    @RDF(Constants.NS_KIWI_CORE+"meaning")
    private ContentItem taggingResource;

    @ManyToOne(cascade = {CascadeType.REFRESH}, fetch=FetchType.LAZY)
    @RDF(Constants.NS_HGTAGS+"taggedResource")
    private ContentItem taggedResource;

    @NotNull
    @OneToOne(fetch=FetchType.EAGER)
    private KiWiResource resource;

    @NotNull
    private boolean deleted;
    …
}
```

Fig. 1: Class Tag: representing a tagging action in the database and KiWi system

6.3. RDF Metadata: Extended Triples

6.3.1. Description

Machine-readable metadata is represented using what we call extended triples. Extended triples contain additional maintenance information that is used internally by the KiWi system for various tasks like versioning, transactions, associating a triple with a certain workspace, user, or group, or for reason maintenance (i.e. storing why a certain triple has been asserted). In principle, an extended triple can thus be seen as a “triple with attributes”.



Note that these attributes could also be represented as a RDF subgraph using triples and reification. However, such a representation has several disadvantages compared to the extended triples proposed for KiWi:

•it requires reification, meaning that the original triple, which assumingly provides the most interesting information, is broken up into parts that have to be reassembled, and

•it mixes up several levels of abstraction, which is inconvenient not only for machines and reasoning, but also for the user

•it makes it difficult to filter out the information that is used for internal purposes and this not supposed to be exchanged with external systems

The implementation of extended triples is straightforward and fits easily with already existing tools and standards without disguising the original meaning. Triple attributes containing maintenance information are only represented programmatically inside the system, avoiding problematic situations. To the outside world, extended triples look like ordinary triples and can be exported into the usual Semantic Web formats like RDF.

In the current implementation, extended triples are represented in special tables in the relational database, and queries to the triple store are executed in Hibernate's object oriented query language HQL. We chose not to use one of the existing triple store implementations because they are restricted to simple triples without the possibility to add additional metadata, they have only basic transaction support \cite{stroka09transactions}, and they offer poor scalability if one wants to use reasoning. Building KiWi on top of these systems has proven to be extremely difficult and has been abandoned in favour of the more flexible database solution. Mapping SPARQL queries to Hibernate is currently under development.

6.3.2. API

Package: kiwi.model.kbase

```
@Entity
@Immutable
public abstract class KiWiNode implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private Long id;

     …
}
```

Fig. 2: Class KiWiNode: base class for all RDF nodes in KiWi

```
@Entity
public abstract class KiWiResource extends KiWiNode ... {

    @OneToOne(fetch=FetchType.EAGER)
    protected ContentItem contentItem;

    @Transient
    private HashMap<String,KiWiLiteral> properties;

    @Transient
    private HashMap<String,Boolean> undef_properties;

    @Transient
    private boolean prefetched = false;

    public String getProperty(String propLabel, Locale loc) …

    public Collection<KiWiTriple> listOutgoing(String propLabel, int limit) …

    public Collection<KiWiTriple> listIncoming(String propLabel, int limit) …

    public String getLabel(Locale loc) …

    public Collection<KiWiResource> getTypes() …

    public boolean hasType(KiWiUriResource type) …

    …
}
```

Fig. 3: Class KiWiResource: offers functionalities common to all RDF resource nodes (URI and blank) in KiWi

```
@Entity
public class KiWiUriResource extends KiWiResource ... {

    @Index(name="uri_index")
    @NotNull
    private String uri;
    …
}
```

Fig. 4: Class KiWiUriResource: resource with a string URI

```
@Entity
public class KiWiAnonResource extends KiWiResource ... {

    @Column(unique=true)
    @Index(name="anonid_index")
    private String anonId;
    …
}
```

Fig. 5: Class KiWiAnonResource: resource representing a blank node; anonId is an internal identifier

```
@Entity
public class KiWiTriple  implements Serializable {

    @Id
    @NotNull
    private Long id;

    @NotNull
    @Immutable
    @ManyToOne(fetch = FetchType.EAGER, optional=false)
    @Index(name="idx_triple_subject")
    private KiWiResource    subject;

    @NotNull
    @Immutable
    @ManyToOne(fetch = FetchType.EAGER, optional=false)
    @Index(name="idx_triple_property")
    private KiWiUriResource property;

    @NotNull
    @Immutable
    @ManyToOne(fetch = FetchType.EAGER, optional=false)
    @Index(name="idx_triple_object")
    private KiWiNode        object;

    @Immutable
    @ManyToOne(fetch = FetchType.LAZY, optional=true)
    @Index(name="idx_triple_context")
    private KiWiUriResource context;

    @Immutable
    @ManyToOne(fetch = FetchType.LAZY, optional=true)
    private User  author;

    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    private Boolean deleted;

    …
}
```

Fig. 6: Class KiWiTriple: entity representation for extended triples in KiWi

6.4. User

6.4.1. Description

6.4.2. API

Package: kiwi.model.user

```
@Entity
public class User ... implements KiWiEntity {

    @Column(unique=true, nullable=false)
    @RDF(Constants.NS_FOAF + "nick")
    private String login;

    @RDF(Constants.NS_FOAF + "firstName")
    private String firstName;

    @RDF(Constants.NS_FOAF + "surname")
    private String lastName;

    @Email
    @RDF(Constants.NS_FOAF + "mbox")
    private String email;

    @Temporal(value = TemporalType.TIMESTAMP)
    @Column(updatable = false, nullable=false)
    private Date created;

    @OneToMany(fetch = FetchType.LAZY, mappedBy="author")
    private Set<ContentItem> authoredContent;

    @OneToMany(fetch = FetchType.EAGER)
    private Set<ContentItem> watchedContent;

    /** the tags this user has created */
    @OneToMany(fetch = FetchType.LAZY, mappedBy="taggedBy")
    private Set<Tag> tags;

    @RDF(Constants.NS_FOAF + "knows")
    @ManyToMany(fetch = FetchType.LAZY)
    private Set<User> friends;

    @RDF(Constants.NS_KIWI_CORE + "hasProfilePhoto")
    @OneToOne
    private ContentItem profilePhoto;

    @NotNull
    @OneToOne(fetch=FetchType.EAGER)
    private KiWiResource resource;

    @NotNull
    private boolean deleted = false;

    …
}
```

could also be represented using façades!

6.5. Revisions and Updates

6.5.1. Description

Revisions and Updates are important to persist the history of entities to allow non-destructive updates on the content. In KiWi, updates are generated during every transaction and linked together when the transaction ends. Our requirements were to store transactional updates and to keep the relation to the ContentItem, which enables the tracing of historical modifications of contents. Furthermore, not only visible updates are persisted (text and media updates), but also modifications on meta-data (RDF tags, rules and triples). Thus, we are keeping the history of TextContentUpdates,

MediaContentUpdates, RuleUpdates, DeletionUpdates (removing and restoring a ContentItem), RenamingUpdates, MetadataUpdates (RDF) and TaggingUpdates.

Another requirement, based on the further improvement of KiWi, was to allow automatic updates of composed ContentItems. Composed ContentItems are contents containing other contents, hold by reference. This enables to have parent-child relationships and other forms of transclusion between contents. Thus, updates during one transaction may not only involve one ContentItems, but several ones. We, therefore, have decided to link updates of a ContentItem with a ContentItem version (CIVersion), and CIVersions with a Revision. Hence, we keep the relation between updates and ContentItems and between versions of several ContentItems.

6.5.2. API

Package: kiwi.model.revision

As mentioned before, the revision object holds a set of CIVersions (at least one). One important attribute of the Revision object is, that MetadataUpdates are directly linked to the Revision, as they cannot always be associated with a CIVersion. Besides that, the Revision keeps the creation date and the user, to identify who and when the modifications have been completed.

```
@Entity
@Immutable
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Revision {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private Long id;


    @OneToMany(mappedBy="revision")
    private Set<CIVersion> contentItemVersions;


    @OneToOne
    private MetadataUpdate metadataUpdate;


    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;


    @ManyToOne(fetch=FetchType.LAZY)
    private User user;


    public Revision() {
        this.creationDate = new Date();
    }

    ….
}
```

The CIVersion object references to the update objects, which can be either discrete, meaning that the updates represent the state, which was valid at the time of the update, or continuous, which means that just the changes to the previous version are stored. In other words, if one likes to restore a ContentItem version by continuous updates, all modifications have to be undone in reverse order.

Besides updates, the CIVersion contains a title, which is shown in the user interface and a color, which identifies the textual modifications in the version preview. It also links to the ContentItem and to the Revision, which it belongs to. The boolean flag *origin* identifies whether the user intended to modify this version (*origin*=true), or whether the version was build automatically by the system (*origin*=false).

```
public class CIVersion implements Serializable {

    private static final long serialVersionUID = -6172743983508044935L;

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private Long id;

    private String title;

    @OneToOne
    private TextContentUpdate textContentUpdate;

    @OneToOne
    private MediaContentUpdate mediaContentUpdate;

    @OneToOne
    private TaggingUpdate taggingUpdate;

    @OneToOne
    private RuleUpdate ruleUpdate;

    @OneToOne
    private DeletionUpdate deletionUpdate;

    @OneToOne
    private RenamingUpdate renamingUpdate;

    @ManyToOne(fetch=FetchType.LAZY)
    private ContentItem revisedContentItem;

    /**
     * the colour of this revision, shown in the preview view.
     * by default it's black
     */
    @NotNull
    @Column(length=7,nullable=false)
    private String colour = new String("#000000");

    @NotNull
    private Long versionId;

    @ManyToOne
    private Revision revision;

    private boolean origin;
}
```

6.6. KiWi Façades

A particularly salient aspect of the KiWi system is its façading mechanism. Façading can be seen as a way of providing different application-dependent Java views on content items. They are thus a way of realising content versatility in Java in a way natural to developers. A KiWi Façade is specified as a Java interface, annotated with Java 5 annotations that map Java methods to RDF properties (see Figure below). When calling an annotated method, an appropriate query to the triple store is issued instead of accessing the Java field.

```
@KiWiFacade
@RDFType( Constants.NS_GEO+"Point" )
public interface PointOfInterestFacade extends ContentItemI {

     @RDF(Constants.NS_GEO+"long")
     public double getLongitude();

     public void setLongitude(double longitude);

     @RDF(Constants.NS_GEO+"lat")
     public double getLatitude();

     public void setLatitude(double latitude);
}
```

Fig. 7: A Java interface annotated as KiWi Façade; the methods get/setLongitude() map to the RDF property `geo:long`, whereas get/setLatitude() map to `geo:lat`. A content item façaded with this interface is automatically assigned the `geo:Point` type.

For a Java developer working with the system, a façaded content item behaves exactly like an ordinary content item with the additional methods specified in the façade interface, and can be used in any context a content item can be used, e.g. as backing bean for user interface components. This functionality is realised using Java's dynamic proxy mechanism (implemented in the KiWi invocation handler). All methods in the KiWi entity manager may optionally take or return a KiWi Façade instead of a content item if they are passed a façade interface as additional argument.

7. KiWi Core System Components

7.1. System, Persistence & Transaction Services

The KiWi system offers a number of core services that are needed in many situations. Of particular importance is the KiWi entity manager (named in analogy with the Java EE entity manager, as it provides the technological background for realising content versatility.

7.1.1. Configuration Service

Interface: kiwi.api.config.ConfigurationService

Description. The configuration service is a central service offering to store configuration data for other components in the system. It manages both, system-wide configuration data and user-specific configuration data.

- a *system configuration* is identified by a string key (preferrably using a dot notation, e.g. “tagit.google_key”) and can contain either a string, double or integer value or a list of such values
- a *user configuration* is identified by a string key (again preferrably using dot notation) and a user object and can contain either a string, double or integer value or a list of such values

The configuration service offers methods to get a configuration by key (optionally with a default string value in case the configuration is still unset), to set a configuration, effectively storing it in the database on transaction end, and to remove a configuration by key, purging it from the database. Configuration values are cached by the service, so they should only be updated through the service interface. Accessing configuration keys should be reasonably fast once they are cached.

7.1.2. KiWi Entity Manager

Interface: kiwi.api.entity.KiWiEntityManager

Description. The KiWi entity manager is a central service providing unified access to data stored in the relational database, in the triple store, and in the fulltext index. It provides functionalities for storing, querying and searching entities (content items, triples, tags) in different languages:

- *Storing* of entities is handled through the methods `persist()` (for new entities) and `merge()` (for updated entities). Both methods take care of forking the data associated with a Java entity appropriately into relational database, triple store, and fulltext index.
- *Querying* of entities is handled by the method `createQuery()`, which takes as argument a query string in either HQL (Hibernate's object oriented query language) or SPARQL and returns a Query object that can be used in the same way as the ordinary Java EE Query object for retrieving results.
- *Facading* of entities is handled by the `createFacade()` methods, taking as argument a content item or list of content items and returning a dynamically generated proxy object that implements the facading functionality and is put in front of the given content item.

All KiWi Entity Manager methods support façading, described below. Additionally, all updates performed through the KiWi entity manager are automatically wrapped in appropriate transactions that support transaction isolation and commit/rollback functionality. Also, KiWi entity

manager transactions are automatically versioned using KiWi's revision service and can be reverted individually. The KiWi transaction system is discussed in \cite{stroka09transactions}.

API. The KiWiEntityManager is implemented as a JavaEE stateless component with the component name “kiwiEntityManager”, under which it can be accessed by other components. Its signature is made to resemble the “normal” EntityManager signature, i.e. offering the persist, merge, refresh and remove operations (described above), as well as a modified “find”.

7.1.3. Triple Store

Interface: kiwi.api.triplestore.TripleStore

Description. The TripleStore is a component that offers operations for working with RDF data, i.e. creating, querying, and removing nodes, triples, and RDF namespaces. The TripleStore is fully transactional, i.e. all changes will only be written to the database when the transaction is finished successfully. It implements the following methods:

- `createUriResource()/createAnonResource()`: create new or retrieve existing URI resources or blank nodes in the triplestore. If the resource already exists, it is fetched from the database and returned. If it does not yet exist, it is created anew, persisted as part of the transaction and returned.
- `createLiteral()`: create a new literal in the triplestore. Unlike other nodes, literals are always created anew to avoid identity vs. equality problems. The new literal is persisted when the transaction commits.
- `createTriple()`: create new or retrieve existing triple in the triplestore. If the triple already exists, it is fetched from the database and returned. If it does not exist, it is created a new and persisted as part of the transaction.
- `removeTriple()`: mark a triple in the triplestore as deleted. Triples are never actually deleted because they might be part of revisions. Instead, their “deleted” flag is set to “true” by this operation.
- `query()`: allows to query for variable bindings (or resources, triples – depending on which query method is called) using one of the KiWi-supported query languages (SPARQL, SeRQL, KiWi Search). The query is passed as string argument. (Note that this functionality is currently in the process of being moved to a separate QueryService).
- `get/set/removeNamespace()`: retrieve, set, or remove a RDF namespace from the triplestore. Changes are persisted when the transaction commits

In the current implementation it synchronously stores triple data in the database as well as in a Sesame triple store for SPARQL support. In the future, it is planned to reimplement the SPARQL querying functionality on top of the database and thus be able to drop the dependency on Sesame.

All changes performed on the triple store using the above-mentioned methods is subject to transaction management and will only be written to the database when the transaction completes successfully. Changes to the triple store will also be covered by the versioning system of KiWi and can be reverted at any time. Actual persistence of entitys to the database is taken care of by the service `TripleStorePersister` in the same package.

API. The triplestore is implemented as a Seam application-scoped component with the component name “tripleStore”, under which it may be accessed by other components. In the future, it might make sense to change the scope to “session” so that every user may have a separate triplestore,

e.g. when using different levels of reasoning. The TripleStore API is available in the source code of interface `kiwi.api.triplestore.TripleStore`.

7.1.4. Transaction Service

Interfaces:

- kiwi.api.transaction.TransactionService (transaction management)
- kiwi.service.transaction.KiWiSynchronizationImpl (transaction processing)

Description. A transaction is an action or series of actions, carried out by a user or an application program, which reads and/or updates the contents of a database. It is often referred as a logical unit of work on the database, where a logical unit of work can cover an entire program, a part of a program, or a single command . Applications that access one or more data storages must ensure that multiple, simultaneously running transactions do not conflict. In KiWi, the transaction service is, thus, responsible for the consistent and correct storage of data stored in the triplestore and the relational database.

Generally, any modifications on the databases are hold in memory until the transaction completes. Afterwards, the modifications are accomplished on both storages, which enables us synchronized updates. In this way, we are able to control failures, which may occur on one of the data stores. Thus, it gets possible to cancel updates on a data store if the other one fails. The data, therefore, remains in a consistent state.

The implementation of the transaction management is provided by the JBoss Seam and EJB 3.0 container. To alternate the container routines (JBoss Seam provides controlled transactions for every JSF lifecycle), transactions can be declaratively managed. This is realized by annotations placed above methods of a class or the class itself, which are provided by the containers and listed below in Table x.x.

As said before, the data modifications are hold in memory until the container completes the transaction. Most of these data modifications are related to a `ContentItem` (just RDF metadata updates cannot always be related to a CI). The methods to add data to a logical unit of work can be found in the `TransactionService` interface.

`getTransactionCIVersionData()` returns a `CIVersionBean` for a certain ContentItem. This `CIVersionBean` is stored in the `UpdateTransactionBean`, which may hold multiple `CIVersionBeans` and is associated with exactly one `javax.transaction.UserTransaction`. If no association exists it will be created and the SeamTransaction (`org.jboss.seam.transaction.Transaction`), which inherits javax.transaction.UserTransaction, is used to register a class that contains methods which will be called when the transaction ends (Listing x.x). This class must implement the `javax.transaction.Synchronization` interface. The `CIVersionBean` can be seen in Listing x.x. Data Updates are stored by retrieving the CIVersionBean and using the setter-methods to add update data to the CIVersion.

```
( (org.jboss.seam.transaction.UserTransaction)
        Component.getInstance("org.jboss.seam.transaction.transaction",
                        ScopeType.EVENT) ).registerSynchronization(sync);
```

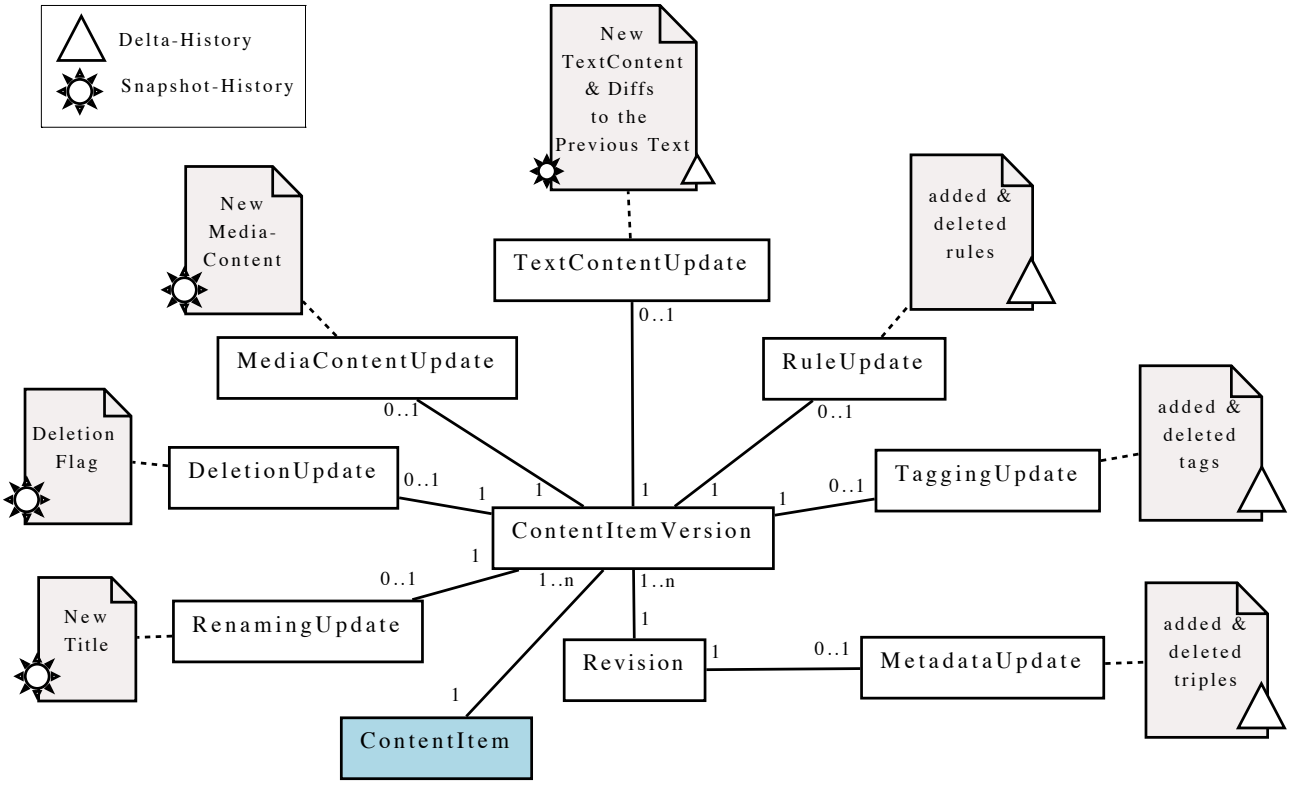
To add data updates that cannot be associated with any ContentItem, TransactionService offers the method getCurrentTransactionData(). This method returns the UpdateTransactionBean, which is also used on the getTransactionCIVersionData.

KiWiSynchronizationImpl is used as the synchronization class that implements the javax.transaction.Synchronization interface. As the interface specifies, the methods beforeCompletion() and afterCompletion() are implemented. These methods will be called directly before and after the transaction completes. The beforeCompletion() method is of primary interest, since it does cleanups in the updates (e.g. removing of modified RDF triples or tags that were deleted and added again), creates missing update objects, collects the CIVersions and builds the Revision (which consists out of CIVersions, ref. section x.x).

The afterCompletion() method notifies the system whether the transaction could successfully complete or not. As mentioned above, the time of synchronization is also used to create the Revision object. The next section will go into detail how this is realized.

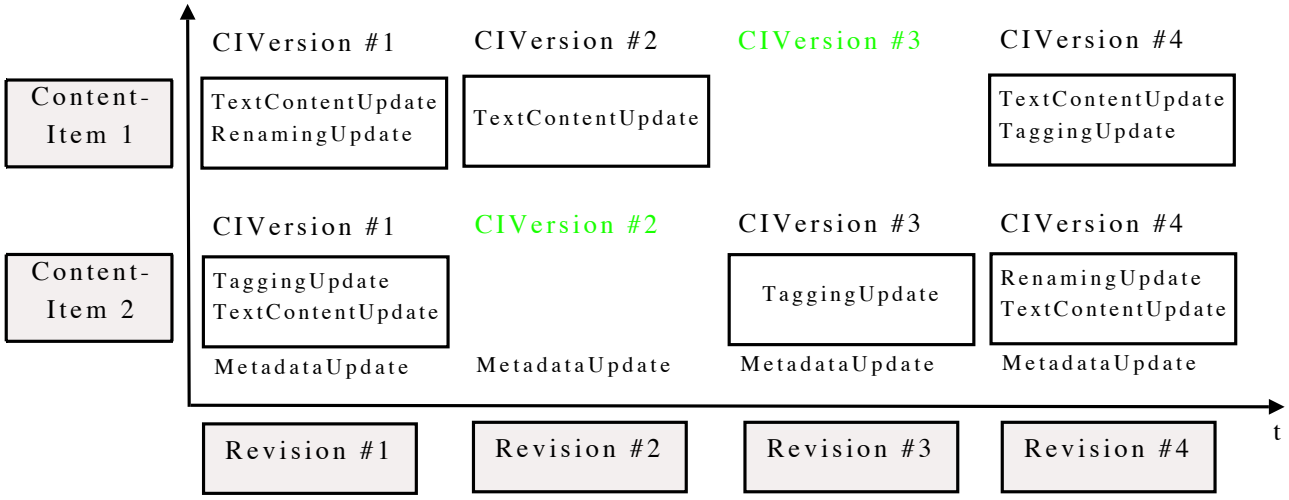
7.1.5. Revision Service

The revision service is responsible for offers non-destructive updates. This enables that data modifications, provided by the users of KiWi, will be traceable and removable. Revisions will contain data that has been changed during one transaction. The overall structure of the revision configuration can be seen in Figure x.x



Every Revision may contain multiple CIVerions, which may contain multiple Updates, as can bee seen in the Figure. MetadataUpdates are directly related to the revision as they cannot always be referred with a ContentItem.

The meaning behind this configuration structure is to allow automatic updates of composed contents. It gets more clear if we have a look at the time-dependent progress of creating versions of ContentItems in Diagram x.x.



This Diagram shows two composed ContentItems (e.g. a parent-child relationship), which have been modified over the time (CIVersions). Since both ContentItems are dependent on each other, every modification results in a new version of every ContentItem, even if one of them did not actually change. These CIVersions must be stored in a common object, which is the Revision itself. This dependent storage is important when a ContentItem version must be restored. Lets assume that version 3 is the currently valid state and we want to restore the first version of ContentItem2. To restore version 1 of ContentItem2, the modifications done to ContentItem1 must also be undone, because otherwise we would lose the consistent composition of both ContentItems. If we would not store CIVersion3 of ContentItem1 in Revision3, we would not be able to check whether ContentItem1 has a dependency on ContentItem2.

Another argument why we store multiple CIVersions in one Revision is not to lose the reference between updates and ContentItems. In a future version of KiWi we would like to allow the user to decide whether a restore of an older version should include all ContentItems, or whether the ContentItems should get independent from each other.

The restore function, displayed in Listing x.x, provides the functionality to restore a certain Revision. As one can see, this function delegates to the CIVersionService methods restore and undo. The distinction was made, because snapshot updates (discrete updates) can be recovered in one step, whereas delta updates (continuous updates) must be restored by resetting every single modification until the version that should have been restored has been reached.

The restore method in CIVersionService contains a loop in which every discrete update is restored:

```
for(KiWiUpdateService kus : updateServicesDiscrete) {
    kus.restore(vers);
}
```

The undo method, on the other hand, delegates to the undo method of each continuous update service:

```
for(KiWiUpdateService kus : updateServicesDiscrete) {
    kus.undo(vers);
}
```

7.2. Context Services

The context of a user's interaction with the KiWi system is defined by the currently selected content item, by the user himself, and by the currently active application. The three context

components have different scopes (current content item and current application are valid within a conversation, user is valid within a session) to reflect that the same user might participate in several interactions with the KiWi system. Each of the components is automatically created on demand by an appropriate factory service. All factory services are located in the package kiwi.action.context.

7.2.1. Current Content Item Factory

The CurrentContentItemFactory is responsible for loading and making available the content item selected by the user in the current conversation. The current content item is available to other components under the component name "`currentContentItem`" and is stored in the current conversation so that different interactions with the system (e.g. in different browser tabs) may select different content items.

The CurrentContentItemFactory loads the selected content item from the database based on the following parameters (in order of precedence), which may be passed to /home.seam as query parameters (e.g. /home.seam?uri=http://...):

- *currentItemId*: the internal database id (id field) of the content item to load; this is the most efficient way to retrieve the content item because it makes use of the database primary key, but it is "unstable" and not portable to other systems as the id might change
- *currentItemKiWiId*: the KiWi identifier of the content item to load; the KiWi identifier is either the URI of the corresponding URI resource or the anonymous ID of the corresponding blank node; the KiWi id is stable for URIs and unstable for anonymous Ids
- *currentItemUri:* the URI of the resource associated with the content item to load; the URI is stable over different installations of the KiWi system.
- *currentItemTitle*: the title of the content item to load; the title might be ambiguous (two content items with the same title may exist) in which case a random matching content item is chosen; the title is thus not a stable way to identify a content item, but it can be useful in many situations (e.g. for tag recommentation or for wiki links).

In case there is no content item corresponding to the given parameter, a new content item is created from it and returned when requested by another component. The new content item is persisted in the database when the transaction commits successfully. In case none of the parameters is given, the start page configured in the system is returned.

The corresponding configuration for setting the parameters when accessing /home.seam is defined in resources/WEB-INF/pages.xml and looks as follows. Note that none of the parameters for /home.seam is required, in which case the start page is returned. Note also that /home.seam is purely a redirector that sets the parameters and then loads the home page of the currently active application.

```
<page view-id="/home.xhtml" >
    <param name="uri" value="#{currentContentItemFactory.currentItemUri}"
           required="false"/>
    <param name="kiwiid" value="#{currentContentItemFactory.currentItemKiWiId}"
           required="false"/>
    <param name="id" value="#{currentContentItemFactory.currentItemId}"
           required="false"/>
    <param name="title" value="#{currentContentItemFactory.currentItemTitle}"
           required="false"/>

    <begin-conversation join="true"/>

    <!--  redirect to current application -->
    <action execute="#{kiwi.ui.redirectAction.redirect()}"/>
</page>
```

7.2.2. Current User Factory

Similar to the CurrentContentItemFactory, the CurrentUserFactory is responsible for loading the currently active user and making it available under the component name "`currentUser`" to other components of the KiWi system. The currentUser is stored in the session scope, so the same user is shared between different interactions with the system that originate from the same browser (or other client).

The currentUser is loaded from the database based on the information provided by the authentication and authorization mechanism offered by Seam:

- if a user has logged in using a valid UserAccount, the User object with the same login name is loaded from the database and offered as currentUser
- if no user is logged in, the User object corresponding to the system-wide anonymous user is loaded from the database and offered as currentUser. Note that the anonymous user object only has limited functionality as it may be shared between many different users.

7.2.3. Current Application Factory

The CurrentApplicationFactory provides the descriptor for the application active in the current conversation under the component name "`currentApplication`" in the conversation context.

The currentApplication is set based on parameters passed to calls of the method `switchApplication()` of the CurrentApplicationFactory. An application is active as long as it has not been changed by a subsequent call to `switchApplication()`. If `switchApplication()` has never been called in the currently active conversation, the system takes "wiki" as default, or whatever default value has been configured in the system wide configuration (configuration key "kiwi.app.default").

The method `switchApplication()` is called automatically by page actions configured for most applications. For example, the page actions for all pages in the TagIT application is as follows:

```
<page view-id="/tagit/*">
    <action execute="#{currentApplicationFactory.switchApplication('tagit')}"/>
</page>
```

Whenever a page in the TagIT application is accessed in a conversation, the currentApplication is

thus switched to “tagit”.

7.3. Content-Related Services

Besides persistence and context, the KiWi system offers a number of core services that simplify the most frequent operations in the system or offer additional functionality. We describe some of these services in the following. This section contains an overview over services that are useful for retrieving and manipulating content. Subsequent sections describe services for user management, for recommendations, for reasoning and querying, and for ontology and thesaurus management.

7.3.1. Content Item Service

The ContentItemService offers under the component name “contentItemService” methods for creating, loading, and manipulating content items. The following groups of methods are implemented in ContentItemService:

- `createContentItem()`/`createExternContentItem()`: creates a new content item and returns it. If no argument is given, a random local (i.e. with the URI prefix of the system) URI is generated; if a postfix is given, a local URI with the given prefix is created, or if a content item with this URI already exists it is returned; createExternContentItem() allows to create content items with non-local URIs, but be aware that these cannot be accessed as linked open data. Created content items will be persisted when the transaction commits.
- `createMediaContentItem()`/`createTextContentItem()`: creates a new content item with the media or text content passed as argument to the method
- `getContentItemXXX()`: retrieve a content item based on KiWi identifier, title, URI, or anon id; in case no matching content item exists, null is returned. Used e.g. by CurrentContentItemFactory.
- `updateMediaContentItem()`/`updateTextContentItem()`: this method allows to do a safe update of the text content associated with a content item where the previous content is versioned as part of the revision. It takes as argument the content item to update and a string value containing HTML as produced e.g. by the editor component, which is then processed in the storage pipeline (extracting wiki-style links, performing automatic metadata extraction, …).
  Note: Always use this method instead of ContentItem.setTextContent if you need version-safe changes of the text content.
- `updateTitle()`: version-safe update of the title of the content item.
- `saveContentItem()`: persists the content item in the KiWi system. Delegates to the appropriate merge/persist methods in KiWiEntityManager.
- `removeContentItem()`: mark a content item as deleted. Note that a content item is never deleted completely in KiWi due to versioning. Rather, this method sets the “deleted” property of the content item to “true” and removes it from search index and triple store so that it cannot be retrieved directly any longer.

7.3.2. Tagging Service

The TaggingService offers methods related to KiWi's tagging mechanism:

- `createTagging()` allows to create a new Tag object representing a tagging activity based on the passed parameters. Label is the label used for tagging, taggedItem is the content item that is tagged, taggingItem is the content item used as tag, and taggingUser is the user who performs the tagging operation and owns the tag.
- `addTagging()` actually persists the tag object created by createTagging.
- `removeTagging()` removes a tagging from the database (marked as deleted, versioning applies)
- `hasTag()` checks whether the content item taggedItem has been tagged with the given label

7.3.3. Comment Service

Interface: kiwi.api.comment.CommentService

Description. The Comment Service facilitates managing comments to content items inside the KiWi system. It offers methods to create a reply to a content item (createReply) and list all comments that are replies to a content item (listComments()).

Internally, a comment inside the KiWi system is simply a content item of type kiwi:Comment. When creating a new comment using the createReply() method, a RDF relation between the parent content item and the comment is created with the property type kiwi:hasComment, which is a subproperty of sioc:has_reply.

7.3.4. Rating Service

TODO (Szaby)

7.3.5. Rendering Service

The RenderingService is used for rendering XML content that comes from the database and transforming it into the presentation displayed to the user. The RenderingService currently differentiates between rendering for HTML presentation, rendering for the editor, and rendering for preview, e.g. in the search results, in tooltips, or in RSS feeds. The actual rendering is performed by small stateless components called “Renderlet”.

```
public interface Renderlet<C> {

	public C apply(KiWiResource context, C content);

}
```

A renderlet takes the rendered KiWiResource as context and the content object to be rendered and returns a new modified content object. There are currently renderlets for the XML XOM representation of text content and for the source (string) representation of text content. Typical renderlets are:

- HtmlLinkRenderlet, which looks up internal wiki links and renders them blue or red, depending on whether the page linked to exists

- EditorLinkRenderlet, which renders internal wiki links and external links in wiki syntax instead of as HTML links
- QueryRenderlet, which evaluates embedded queries when rendering the page and adds the results to the rendered content
- RdfaRenderlet, which takes metadata from the TripleStore and uses it to automatically annotate the HTML content with RDF/A information

The different rendering methods use different renderlet configurations. For example, the normal HTML presentation makes use of the HtmlLinkRenderlet, while the editor presentation makes use of the EditorLinkRenderlet.

The renderlets are configured using the ConfigurationService under keys starting with "renderlets.*". Note that extensions may modify the configuration to add additional renderlets if they need to.

7.3.6. Storing Service

The StoringService is similar to the RenderingService, but is called when updating the content of a content item using e.g. ContentItemService.updateTextContent(). Similar to the former, it processes so-called "Savelets" in the form of a pipeline, which transform the input string content into a proper XML document in canonicalized XML form conforming to KiWi's internal XML format (XHTML plus some special attributes). The following Savelets are currently used:

- *ExtractLinksSavelet*: parses the HTML content passed to the savelet for occurrences of wiki syntax links, i.e. [[ ]] and similar links. Occurrences of wiki links are replaces by appropriate HTML anchor elements with kiwi attributes detailing the relation (kiwi:kind with values "imagelink", "videolink", "intlink", "extlink", kiwi:target containing the title of the item linked to – later dereferenced to the URI identifying the item more precisely). The ExtractLinksSavelet also takes care of identifying bookmarks for fragments (marked up with kiwi:bookmarkstart/kiwi:bookmarkend elements) and compounds (marked up with kiwi:compoundstart/kiwi:compoundend elements).
- *HtmlCleanerSavelet*: cleans up dirty HTML code so that it becomes proper XHTML, ready for parsing and storing in the database as canoncial XML.
- *NavigationalLinksSavelet*: scans the textual content for all navigational links to other pages (i.e. HTML anchor elements) and adds appropriate triples to the triple store, explicitly linking the two resources with the relations kiwi:internalLink and kiwi:externalLink. Also, of the kiwi:target attribute refers to a content item title instead of a resource KiWi identifier, the resource is resolved using ContentItemService and the attribute replaced with the unique KiWi identifier of the linked resource.
- *RdfaSavelet:* Extracts the RDFa metadata from the XHTML content passed as argument and stores it as triples in the TripleStore. The RdfaSavelet currently only supports the RDFa features implemented by the KiWi editor, and not full RDFa.
- *FragmentsSavelet:* Updates the content of the fragments inserted into a content item. (TODO: Marek)
- *ComponentSavelet*: (TODO: Marek)

7.3.7. MultimediaService

Interface: kiwi.api.multimedia.MultimediaService

Description. The multimedia service offers functionality that is commonly required when working with multimedia content, i.e. determining the MIME type of the content and extracting the metadata out of the content.

- the method getMimeType() tries to automatically determin the MIME type of the content using MIME magic and file extension guessing. If no file name is given, only MIME magic is used, which is expensive and not reliable.
- the method extractMetadata() extracts RDF metadata from content items that contain multimedia content, e.g. EXIF or IPTC information from pictures. In future versions, extractMetadata() might even support other data types like videos, PDF documents or Word documents.

7.3.8. Import/Export Service

TODO (Sebastian)

7.4. User and Profile Management

7.4.1. User Service

Interface: kiwi.api.user.UserService

Description. The UserService provides methods for accessing and manipulating users in the KiWi system. The API interface for the user service is as follows. Method names should be self-explanatory. The two `createUser()` methods first check whether the user already exists and throw an exception if this is the case. The method `createUser()` does not automatically persist the user, this needs to be done manually by calling `saveUser()`.

7.4.2. Profile Service

Interface: kiwi.api.user.ProfileService

Description. The ProfileService offers functionality for managing a user's profile information. The profile representation differs from the user representation in that it does not have much to do with authentication and user management. It simply contains additional content and information about the user, like the birthday, website, location, etc. The ProfileService offers various methods for managing a user's profile: generating/retrieving the profile, getting a user's blog posts, getting a user's city and country, etc.

7.4.3. Activity Logging Service

Interface: kiwi.api.activity.ActivityLoggingService

Description. The ActivityLoggingService captures activity events raised by other components in the KiWi system and logs them in the database. Currently, the ActivityLoggingService covers creation, editing, deleting, and visiting of a ContentItem, adding and removing of a tag, login and logout of a user, searches performed by a user, and commenting activity. Further activity logging will be implemented depending on the functionality of the KiWi system. An activity is always associated with a user (the user performing the activity), and depending on the type of activity with a content

item and/or tag.

7.4.4. Community Equity Service

TODO (Sebastian, Mihai)

7.5. Search and Querying Services

7.5.1. SOLR Service (Search)

The SolrService provides the search functionality in the KiWi system under the component name "`solrService`". It is implemented as a Session-scoped component so that every user has a separate instance of the search service. The rationale for this, however, is primarily efficient indexing, because the service will try to bundle several index updates to batch operations and a scoping to the session appeared reasonable for this task.

As its name says, the SolrService is a service that accesses the Apache SOLR indexing system[9] that ships as a separate web application in the KiWi EAR. In KiWi, we have adapted the SOLR configuration in a way that it can be used for very efficiently querying based on titles, types, tags, and RDF datatype/annotation properties (see lib/solr/solr-home/conf/schema.xml). The SolrService provides methods for indexing as well as for searching using this configuration:

- `persist()`/`persistAll()`: add a KiWiEntity to the search index. The persist methods take the content item associated to the KiWiEntity's resource and indexes title, content, types, tags, and RDF literal properties. `persistAll()` is a batch method that allows more efficient indexing.
- `remove()`/`removeAll()`: remove a KiWiEntity from the search index. The method behaves similar to the persist methods.
- `optimize()`: trigger optimization of the SOLR search index; this operation might be useful when many updates of the index have taken place. It can be triggered from the administration interface.
- `search()`: search the SOLR index according to the given criteria. There are two implementations of this method: one takes a KiWiSearchCriteria object and allows a more high-level searching that is already adapted to KiWi's data model, the other takes a SolrQuery as argument and allows a more fine-grained control over the returned results (e.g. to configure facetted search at a higher level of detail).
- `parseSearchString()`: takes a KiWi search string and creates a KiWiSearchCriteria object out of it; the object is returned for further modification or for passing to the search function.
- `buildSolrQuery()`: takes a KiWiSearchCriteria object and transforms it into a SolrQuery object; may be useful if one wants to make use of the high-level KiWiSearchCriteria but do some more fine-grained configuration for the search on the SolrQuery object.

Indexing is triggered automatically by KiWi's internal event mechanism. An entity is added to the indexing queue when the event `KiWiEvents.ENTITY_PERSISTED` fires. The queue is run when the event `KiWiEvents.TRANSACTION_SUCCESS` fires. These events are raised by the KiWiEntityManager and the KiWi transaction mechanism (KiWiSynchronization).

[9] http://lucene.apache.org/solr/

The KiWiSearchCriteria class allows to pass queries to different properties of content items:

- *keywords* are looked up in the title, content, author, and tags of content items; tag matches are weighted 8 times as high as normal matches to reflect the explicit labelling of a content item with a keyword.
- *keywordSearchFields* may be used to manually configure the fields to search for keywords and the weights to give them if a match is found.
- *tags* are searched for in the set of tags of content items; the passed tags are searched for conjunctively, i.e. all passed tags must occur.
- *types* are searched for in the set of RDF types associated with the content item's resource; the apssed types are again searched conjunctively.
- *person* is the login name of a user; the search will look for all content items authored by the user.
- rdfLiteralProperties/rdfIntegerProperties/rdfDoubleProperties search for certain property values of RDF literal data associated with the content item's resource. The different fields reflect different search types, i.e. string, integer and double, because they are indexed differently.
- *rdfFacets* is a set of RDF property uris on which to build additional facets. Facets are automatically built on tags, types, and authors already. Every RDF property added here will instruct the search system to build a facet over this property as well, if possible.

The SolrService requires some configuration in order to run properly:

- the SOLR home configuration needs to be set up properly (can be triggered by the build process) and the property “kiwi.solr.home” in persistence.xml needs to point to the correct SOLR home path
- the SolrService needs to be configured with the correct URL to access the SOLR webservice. Usually, this value can be guessed automatically, but in some settings (e.g. when running SOLR on a different server) it needs to be changed manually in the administration interface.

7.5.2. Personalized Search Service

The PersonalizedSearchService offers under the component name “personalizedSearchService” methods for performing search based on user interest. The following groups of methods are implemented in PersonalizedSearchService:

- `runPersonilazedSearch()`: it activates the personalized search in KiWi. In case a search takes place, the results are ranked based on user interests inferred from tagging activity.

7.5.3. Query Service

TODO (Sebastian)

7.5.4. SPARQLService

TODO (Sebastian)

7.5.5. KwqlService

The `KWQLService` provides the functionality of querying using KWQL, the KiWi query language. KWQL is a rule-based query language based on the keyword search paradigms which allows for simple as well and advanced querying over all components of the conceptual model of KiWi.

`query(String query)` is used for evaluating embedded queries which are evaluated upon rendering of a content item.

`search(String query)` evaluates queries that are posed via the search bar or from the editor for the visual version of KWQL, visKWQL.

Both query and search take a query string and return a `KiWiQueryResults` object which consists of a list of documents matching the query. Queries are first parsed by an ANTLR-created parser. Consequently, the semantic (and, to some extent, syntactic) validity of the query is verified. If this succeeds, a SOLR query is created from the KWQL query which is then evaluated using `SOLRService.search()` (see above). Not all functionality of KWQL can be realized through SOLR queries. Rather, SOLR functions is employed as an efficient filter which returns a superset of the valid query results. False positives are then removed from this set through further processing and the resulting matches are returned.

7.6. Recommendation Services

7.6.1. Recommendation Service

The RecommendationService offers under the component name “recommendationService” methods for generating recommendations. The following groups of methods are implemented in RecommendationService:

- `getRecommendations()`: generate related recommendation to the currently viewed page taking into account the similarity of tags.
- `getPersonalRecommendations()`: generate personalized recommendations to a give user taking into account the similarity of his/her tags and the tags from existing content items in the system.
- `getMultiFactorRecommendations()`: generate personalized recommendation based on four distinct factors: cosine similarity, tag representativeness, tag popularity and affinity between user and tags.
- `getTagGroupedRecommendations()`: generate tag-based recommendations grouped by the tags of the currently viewed page.
- `getGroupRecommendations()`: generate related recommendation for a given group considering the tags assigned to it.

- `getUsersByGroupTags()`: recommend users which share similar tags in different content items.
- `getUsersByContentItem ()`: list users which have assigned tags to a given content item.
- `createMySimilarRecommendation()`: allow users to determine which items are similar based on their own judgment.
- `getMySimilarRecommendation()`: loads recommendation defined as similar already exist in the system.
- `existMySimilarRecommendation()`: checks whether a recommendation defined as similar already exist in the system.
- `removeMySimilarRecommendation()`: remove recommendations defined as similar by users.
- `removeContactRecommendations()`: allow users to remove the contact recommendations.
- `getContactRecommendations()`: loads the content items recommended by a friend in the contact list.
- `recommendItems()`: allows users to recommend items to friends in their contact list.
- `getSemanticVectorRecommendations()`: generate recommendations considering the most representative terms of a given content item.

7.6.2. Tag Recommendation Service

7.6.3. Information Extraction Service

The purpose of the Information Extraction Service is to extract structured information from unstructured, or semi-structured text sources. It utilizes machine learning methods to classify potential suggestions and uses user feedback as training data. The architecture was designed such as to support different kinds of suggestions. The current version supports content item tags and tagged text fragments.

Information extraction service stores the state of classifiers and suggestions in persistent entities:

- *Instance*: Represents a potential suggestion of some specific type, classification features, with associated tag and an optional text fragment. Multiple classifiers may share the same set of instances, if they share the same type. Instances are produces automatically by instance extractors in the background after each text content change.
- *Suggestion*: An instance with an associated classifier and score.
- *Example*: Represents a training example, either extracted from existing taggings or generated from a suggestion after user feedback.
- *Classifier*: Stores parameters of the classifier and has a type which specifies which

instances are relevant for this classifier.
- *Instance Extractor*: Stores dictionaries of features for each classifier type. These dictionaries are needed for efficient representation of instance features.

The methods of the information extraction service are:

- createClassifier(ContentItem tag, Class ieType) creates a classifier of a specific instance extractor type for a certain tag.
- initTagExamples(ClassifierEntity cls) initializes the positive examples for a classifier based on existing taggings.
- trainClassifier(ClassifierEntity cls) recomputes the parameters of a classifier on the current set of (positive or negative) examples.
- computeSuggestions(ClassifierEntity cls) runs the classifier on all existing instances and produces suggestion entities.

- trainAndSuggest(ClassifierEntity cls) a convenience method calling trainClassifier and computeSuggestions in one transaction in the background.
- trainAndSuggestForItem(ContentItem item) retrains all relevant classifiers on the examples on the specified content item and runs the classifiers on all instances on that item.

- acceptSuggestion(Suggestion suggestion, User user) Produces a positive example for a suggestion.
- rejectSuggestion(Suggestion suggestion, User user) Produces a negative example for a suggestion.

- initAndClassifyInstances(ContentItem item) Runs instance extractors on the content item text content and all relevant classifiers.

These are the currently implemented instance extractors: (all support only english language documents at the moment)
- NounPhraseInstanceExtractor extracts noun phrases as instances with the context words as classification features.
- DateInstanceExtractor extracts dates as instances with the context words as classification features.
- DocumentInstanceExtractor extracts bag of words features from the whole content item for document classification and treats the whole content item as one instance.

7.7. Ontology and Thesaurus Management

7.7.1. Ontology Service

TODO (Rolf)

7.7.2. SKOS Service

TODO (Rolf)

7.8. Reasoning Services

Reasoning services provide reasoning for simple RDF-triple-pattern-based user-defined rules, maintain consistency of the triple store (reason maintenance), and provide simple explanations of inferred triples.
Reasoning and reason maintenance are triggered automatically upon metadata updates at the end of each transaction (specifically, the ReasoningService observes the `KiWiEvents.TRANSACTION_SUCCESS+"Async"` event emitted by KiWiSynchronizatonImpl). The reasoning service then creates and enqueues a new reasoning task and processes the queue. A reasoning task consists of a set of added and a set of removed triples. This whole process is asynchronous, so other KiWi services are not blocked. It is possible to disable online reasoning using the ReasoningService.

7.8.1. Reasoning service

Reasoning consists of two services: the user-programmer facing `ReasoningService` and the actual reasoner accessible via ReasonerService.

Interface: `kiwi.api.reasoning`.ReasoningService

Description:

```
public Iterable<KiWiTriple> getBaseTriples();
public Iterable<KiWiTriple> getInferredTriples();
       Convenience methods that provide access to a list of all base
       triples and all inferred triples. Base triples are triples that are
       not inferred.

public int inferredTriplesCount();
       Returns the number of inferred triples after an explicit call to
       runReasoner().

public void runReasoner();
       Runs the reasoner on the whole triple store. It doesn't check
       whether the reasoner is already running. The intended use is for
       the case when online reasoning is disabled. It does not process the
       reasoning task queue.

public void setOnlineReasoningAllowed(boolean allowed);
       Enables or disables online reasoning. If the reasoning is currently
       running and it is being disabled it will stop running after the
       queue is processed.

public void runEnqueuedTasks();
       To be used after online reasoning is re-enabled to process the queue created while online reasoning
       was disabled.
```

Interface: `kiwi.api.reasoning`.ReasonerService

Description:

ReasonerService manages the actual reasoner and reason maintenance. Currently it only calls the reasoner and reason maintenance as necessary to process the given ReasoningTask. In the future it should provide a means to configure reasoning features.

```
public void processTask(ReasoningTask t);

       Called by ReasoningService. Calls the actual reasoner and reason
       maintenance as needed.
```

7.8.2. Rules

The reasoner processes rules which are currently stored and loaded from kiwi.service.reasoning.rules.txt. On the one hand, the file is loaded and parsed each time reasoner is run making it easy to test different rules, on the other hand changes in rules are not reflected yet by reason maintenance. Therefore only adding new rules is safe. Modifying current rules or deleting them may result in an inconsistent triple store.

Rule format:

The currently supported rules are simple RDF-triple pattern based rules with variables and construction in the rule head. Each rule has the format

body -> head

where both body and head are a conjunction of triple patterns separated by commas. Triple pattern is of the form

(subject predicate object)

where subject can be a URI, or a variable, predicate can be a URI or a variable, object can be a URI, a variable, or a literal. Variable is of the form

$<identifier>

literal is of the form

"<string>"

Rule body must contain at least one variable. Variables in the rule head which do not occur in the rule body are replaced by newly constructed URIs unique to the rule match. It is possible to specify the beginning part of the URI that should be used using the following syntax:

uri(<uri>)@$<identifier>

The newly created resource will then begin with the <uri> and will be appended with a generated identifier unique to the rule match. Rule match consists of the triples that matched the rule body (in the order they matched it) and the rule itself.

Currently the rules.txt file contains rules that derive the RDFS subClassOf hierarchy and RDF types and rules that represent the meaning of owl:sameAs property. The format of the file is one rule per line.

Examples[10]:

($1 rdf:type $2), ($2 rdfs:subClassOf $3) -> ($1 rdf:type $3)

Derives resource types according to their current type and the type hierarchy.

($1 rdfs:subClassOf $2), ($2 rdfs:subClassOf $3) -> ($1 rdfs:subClassOf $3)

Derives the type hierarchy.

($1 rdf:type kiwi:Tagging) -> ($1 http://kiwi.eu/core/sth $x)

An example rule that shows the $x variable in the rule head which does not occur in the rule body. Therefore when the rule matches a triple a new resource will be created and placed in place of $x. This is useful for the derivation of new taggings for example.

[10]Note that namespace prefixes are not implemented yet. They are used in the examples to make them more legible.

($1 kiwi:hasTagging $2), ($2 kiwi:hasTag ”a”) -> (uri(kiwi:inferredTag)@$x kiwi:hasTag ”b”)

This rule shows the use of literals and the uri-construction feature.

7.8.3. Reason maintenance service

Interface: `kiwi.api.reasoning.reasonmaintenance.`ReasonMaintenanceService

Description:

Reason maintenance service maintains consistency of the triple store. It makes sure that any inferred triple in the triple store can in fact be inferred given the current state of the triple store.

```
public void addJustification(Justification justif);
        Called by the reasoner to add justifications for newly inferred
        triples.

public List<Justification> getJustificationFor(Long tripleId);
        Provides a list of justifications of a triple with tripleId.

public void removeTriples(Set<KiWiTriple> triples);
        Processes removed base triples. As a result any inferred triples
        that depends only on the base triples will be removed.
```

7.8.4. Explanation service

Currently explanation service provides access to justifications of a given triple so that a justification graph can be rendered on the client side.

Interface: kiwi.api.explanation.ExplanationService

Description:

```
public List<Justification> explainTriple(Long id);

        Pulls justifications that explain the triple with the given triple
        id.
```

Interface: `kiwi.widgets.explanation.ExplanationWidgetWebService`

Description:

```
public List<NodeJSON> getSubtree(@QueryParam("id") Long id)

        Returns a JSON representation of the immediate justification
        subgraph of the triple with the given id. It is used for the
        interactive animated tree in Explanation in the Inspector. It is
        called using:

        /KiWi/seam/resource/services/widgets/explanation/explain?
        id=<KIWI_TRIPLE_ID>
```

7.9. Authentication & Authorization

7.9.1. Basic Authentication

…

TODO (Sebastian)

7.9.2. Authentication with Facebook

...

TODO (Sebastian)

7.9.3. Authentication with FOAF+SSL

Interface: kiwi.api.user.FoafSSlLoginService

FOAF+SSL is a authentication protocol that links a *Web ID* (which is a URI in the web that identifies a person) to a public RSA key, thereby enabling a global, decentralized/distributed, and open yet secure social network. It functions with existing browsers.

It uses PKI standards — usually thought of as hierarchical trust management tools — in a decentralized "web of trust" way. The web of trust is built using semantic web vocabularies (particularly FOAF) published in RESTful manner to form Linked Data.

FOAF+SSL enables authentication via client certificates that are stored in the users` browser. The authentication process requires that SSL is enabled and configured for client authentication on the application server. The authentication itself is done via a SSL handshake. Therefore the client sends a request to the server, which itself returns a server certificate to the user that identifies that the server is what it claims to be. After that, the server requests the client certificate, which is illustrated in Figure x.x for Mozilla Firefox 3.

The application server must be configured to allow all kinds of client certificates, even self-signed, because otherwise the client certificates must be signed by a known CA, which will not be accepted by any user.

The application server configuration for JBoss AS 4.2.3 is explained in the following:

First of all we will have a look into the server.xml file, where the application server connector configurations can be found. You will find that file in the ${JBoss.home}/server/default/deploy/jboss-web.deployer directory. The jboss-web.deployer is the Tomcat 6 bundle that JBoss AS integrates. Inside of the server.xml you'll find a Connector tag that defines the SSL connections. If you haven't changed the default configuration yet, the paragraph should be commented out. So the first step for you is to uncomment it and configure it in the following way:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" maxThreads="150"
   scheme="https" secure="false" strategy="ms" address="${jboss.bind.address}"
   keystoreFile="${jboss.server.home.dir}/conf/server.keystore"
   keystorePass="changeit" sslProtocol="TLS"
   truststoreFile="/usr/lib/jvm/java-6-sun-1.6.0.10/jre/lib/security/cacerts"
   truststorePass="changeit"
   SSLImplementation="org.jsslutils.extra.apachetomcat6.JSSLutilsImplementation"
   acceptAnyCert="true" clientAuth="want" />
```

The default https port is 443, but ports below 1024 require that the server is started with root privileges, so we take the JBoss default port 8443 for testing purposes instead. As we do not only want to allow server-side authentication, but mutual authentication clientAuth must be at least set to "want". There are three values for clientAuth: Set it to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to want if you want the SSL stack to request a client certificate, but not fail if one is not presented. A false value (which is the default) will not require a certificate chain unless the

client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.

Servers authentify themselves by sending the client a certificate signed by a well known Certificate Authority (CA) whose public key is shipped in all browsers. Browsers use the public key to verify the signature sent by the server. If the server sends a certificate that is not signed by one of these CAs (perhaps it is self-signed) then the web browser will usually display some pretty ugly error message, warning the user to stay clear of that site, with some complex way of bypassing the warning, which if the user is courageous and knowledgeable enough will allow him to add the certificate to a list of trusted certs. This warning will put most people off. It is best therefore to buy a CA certified certificate (for example this one for €15 at trustico.) Usually the CA's will have very detailed instructions for installing the cert for a wide range of servers. Anyway, for testing purposes it is fair enough to use self-signed certificates.

To enable https connections and persist server, as well as client certificates, it is important to specify the directory of the keystore and the truststore, which will be build in the following. The JVM comes together with a truststore, so you do not have to create another one. However, you have to specify the directory of the JVM truststore.

The next step to complete is to build the keystore in the ${JBoss.home}/server/default/conf directory. This can be done with the help of the keytool program, which can be found in the ${JAVA.HOME}/bin directory. To create a keystore, export its certificate and import it into the JVM truststore, type the following command into the console:

```
.../server/default/conf$ keytool -genkey -alias server -keyalg RSA -keystore server.keystore
.../server/default/conf$ keytool -export -alias server -keystore server.keystore -file server.cer
.../server/default/conf$ keytool -import -v -trustcacerts -alias server -file server.cer -keystore ${JAVA_HOME}/jre/lib/security/cacerts.jks -keypass changeit -storepass changeit
```

After the first command you'll be asked to fill out some information, for example the keystore password, the first name and last name, the organisation, the city, state, etc.. Note that you should write the host address (www.host.de, 127.0.0.1, etc.) instead of your first and last name for the server keystore, because otherwise the browser complains that the address produced in the keystore is not matching the address that a user typed into the browser.

Another important configuration of server.xml is the link to the custom SSLImplementation (`SSLImplementation="org.jsslutils.extra.apachetomcat6.JSSLutilsImplementation"`). This library has been provided by Bruno Harbulot and enables to accept any client certificates (when acceptAnyCert is set to true), not just those where the issuer has been trusted before.

KiWi uses the foaf-ssl service as an alternative way to sign on. It, therefore, connects to an IDP servlet, which provides the forwarding to a passed URL and the passing of the users webid and the current timestamp via HTTP GET parameters, if the authentication process succeeded. The IDP server uses an HTTPS connection, which prevents malicious attacks (e.g. man-in-the-middle attack, other eavesdropping) and signes the resulting url to which it forwards. The signature is appended as a GET parameter to the url.

The following provides an example on how the authentication process could look like. Let's assume that the IDP servlet is called in the following way:

https://foafssl.org/srv/idp?
authreqissuer=http://www.kiwi.eu/seam/resource/restv1/FOAFSSLAuthentication

The IDP server then uses the `net.java.dev.sommer.foafssl.verifier.DereferencingFoafSslVerifier` from the foaf-library to verify the client's certificate and to return the web id:

```
X509Certificate[] certificates = (X509Certificate[]) request
    .getAttribute("javax.servlet.request.X509Certificate");

if (certificates != null) {
    X509Certificate foafSslCertificate = certificates[0];
    try {
        verifiedWebIDs =
           FOAF_SSL_VERIFIER.verifyFoafSslCertificate(foafSslCertificate);
  }

}
```

If the certificate could be verified, the IDP server creates a signed response URL with the given authreqissuer URL, similar as shown below (3):

http://www.kiwi.eu/seam/resource/restv1/FOAFSSLAuthentication?webid=http://test.webid.com/12345&ts=2009-07-08T12:00:21-0700&sig=af12786de34789a345978fcd78f86e8903c87979a7896435

```
// String authnResp = authrequissuer URL

URI webId = verifiedWebIDs.iterator().next().getUri();
authnResp += "?" + WEBID_PARAMNAME + "="
    + URLEncoder.encode(webId.toASCIIString(), "UTF-8");

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ");
authnResp += "&" + TIMESTAMP_PARAMNAME + "="
    + URLEncoder.encode(dateFormat.format(Calendar.getInstance().getTime()),
     "UTF-8");

String signedMessage = authnResp;
Signature signature = Signature.getInstance("SHA1withRSA");
signature.initSign(privateKey);
signature.update(signedMessage.getBytes("UTF-8"));
byte[] signatureBytes = signature.sign();
authnResp += "&" + SIGNATURE_PARAMNAME + "="
    + URLEncoder.encode(new String(Base64.encode(signatureBytes)), "UTF-8");
```

KiWi then has to check whether the request to http://www.kiwi.eu/seam/resource/restv1/FOAFSSLAuthentication really came from the IDP server. This is done by verifying the signed request URL WITHOUT the signature part against the public key of the IDP server. The verification process is available in the `validateIDPrequest()` method, which can be found in the `FOAFSSLAuthentication` bean.

```
private boolean validateIDPrequest() {
    final ServletContexts servletContexts = ServletContexts.getInstance();
    inal HttpServletRequest request = servletContexts.getRequest();

    String webid = request.getParameter("webid");
    String timestamp = request.getParameter("ts");
    String sig = request.getParameter("sig");

    /**
     * Testing the timestamp
     */
```

```
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM
    -dd'T'HH:mm:ssZ");
try {
    Date authnDate = dateFormat.parse(timestamp);
    Calendar cal = Calendar.getInstance();
    cal.setTime(authnDate);
    if(cal.before(Calendar.getInstance())) {
        log.debug("Timestamp verification no. 1: Testing validation");
    } else {
        log.error("Timestamp is invalid");
        return false;
    }
    Calendar fiveSecsBef = Calendar.getInstance();
    fiveSecsBef.add(Calendar.SECOND, -5);
    if(cal.after(fiveSecsBef)) {
        log.debug("Timestamp verification no. 2: Testing expiration");
    } else {
        log.error("Timestamp has expired");
        return false;
    }
} catch (ParseException e2) {
    log.error("System failure: Parsing timestamp produces exception");
    return false;
}

KeyStoreServlet ks = (KeyStoreServlet)
    Component.getInstance("keyStoreServlet");

KeyFactory keyFactory = null;
PublicKey pubKey = null;
try {
    keyFactory = KeyFactory.getInstance("RSA");
    pubKey = ks.getPublicKey();
} catch (NoSuchAlgorithmException e) {
    log.error("System failure: No such algorithm");
    return false;
}

/**
  *  converting signature string to byte array
  */
byte[] sigToVerify = Base64.decode(sig);

/**
  * Verifying signature
  */
Signature signature = null;
try {
    signature = Signature.getInstance("SHA1withRSA");
    signature.initVerify(pubKey);
} catch (NoSuchAlgorithmException e2) {
    log.error("System failure: No such algorithm");
    return false;
} catch (InvalidKeyException e) {
    log.error("System failure: Invalid public key");
    return false;
}

StringBuffer bufferedUrl = request.getRequestURL();
try {
    bufferedUrl.append("?webid=");
    bufferedUrl.append(URLEncoder.encode(webid,"UTF-8"));
    bufferedUrl.append("&ts=");
```

```
            bufferedUrl.append(URLEncoder.encode(timestamp,"UTF-8"));
        } catch (UnsupportedEncodingException e3) {
            log.error("System failure: Unsupported Encoding");
            return false;
        }

        try {
            signature.update(bufferedUrl.toString().getBytes("UTF-8"));
            if(signature == null) {
                log.error("Signature is invalid (null)");
                return false;
            } else if (!signature.verify(sigToVerify)) {
                log.error("Signature cannot be verified");
                return false;
            }
        } catch (SignatureException e2) {
           log.error("Signature is invalid");
           return false;
        } catch (UnsupportedEncodingException e) {
        log.error("System failure: Unsupported Encoding");
        return false;
        }
return true;
}
```

After the verification of the IDP timestamp and signature, the webId is used to retrieve the foaf profile and to load the data into the triplestore. If the webID owner does not yet have a KiWi profile, it will be created.

7.9.4. Authentication with Single-Sign On

Besides FOAF-SSL and normal authentication, we would also like to provide another single-sign on solution, which is well-established and can be used in a company-environment. We chose OpenSSO as single-sign on service and will implement it until end of September 2009.

7.9.5. Authorization in JBoss Seam

Besides authentication, which is the process of identifying a person, KiWi also enables authorization, which means that the users, depending on their roles (e.g. *admin*) or predefined rules (e.g. *all friends*), are allowed or restricted to access methods and content. JBoss Seam enables role-based and rule-based authorization through the expression language. The #{s:hasRole('role')} function allows to restrict application functionality to users in certain roles. This can be applied on the view

```
<s:link view=”/admin.xhtml” value=”Admin Login”
   rendered=”#{s:hasRole('admin')}” />)
```

in the navigations, defined in pages.xml

```
<navigation from-action=”home”>
    <rule if=”#{s:hasRole('admin')}”>
        <redirect view-id=”/admin/home.xhtml” />
    </rule>
    <rule if=”not #{s:hasRole('admin')}”>
        <redirect view-id=”/home.xhtml” />
    </rule>
</navigation>
```

in the business logic:

```
if(identity.hasRole('admin')) {
   return “/admin/home.xhtml”;
} else {
   return “/home.xhtml”;
}
```

or as annotation over a method or the whole class

```
@Restrict(#{s:hasRole('role')})
```

Rule-based authorization is provided with the Drools rule-engine, which supports declarative programming. Restrictions can be modified without affecting the business logic. Rules are defined in the security.drl file. An example of a rule is given below:

```
rule CreateUser
  no-loop
  activation-group "permissions"
when
   check: PermissionCheck(target == "seam.user", action == "create", granted == false)
  Role(name == "admin")
then
  check.grant();
end
```

The rule can be used similar as the role-restriction, discussed aboce. An example in given below:

```
@Restrict(#{s:hasPermission('seam.user','create',currentUser)})
public User createUser() {
    ...
}
```

The example rule checks whether a user has the role 'admin' before allowing him/her to create another user.

Authorization is not yet maturely implemented in KiWi.

7.10. Miscellaneous Services

7.10.1. Geo Coding Service

TODO (Sebastian)

8. KiWi User Interface Components

8.1. Editor

The basic KiWi wiki editor is meant as a WYSIWYG (what-you-see-is-what-you-get) editor with some features that don't follow the WYSIWYG principles in order to keep wiki content editable with only a keyboard. The reason for not making it strictly WYSIWYG is to edit purely textual functionalities of a wiki editor without using special GUI to maintain them, such as inserting a wiki link using the "[[resource-name || text-label]]" syntax.

8.1.1. Extending the editor

In most of the editor extensions, we followed the principle of losely coupling the extension with the Editor's JavaScript engine and put as much functionalities to the JSF layer as possible. This gives us independency of the actual editor implementation we used and puts most of the responsibilities to the JSF layer as this provides backing functionalities and is a highly stable part of the KiWi architecture.

In order to extend the editor's behaviour 3 steps are common in all extension cases:

1. Create a JSF dialog using the KiWi PopupAction functionalities,
2. Create a simple MCEditor plugin that provides the activation of your JSF dialog
3. Implement the connection between the editor trigger to get the needed information from the editor to the Plugins backing bean and mechanism sending back the HTML the extension has produced, using a JavaScript callback function.

Creating a popup template

Inorder to implement a popup your Configuration KiWiApplication has to override the `initDialogBoxes` method like in Fig. 1.

```
@Override
public void initDialogBoxes(PopupManagerAction pma) {
     pma.registerDialogbox("wiki.querywizardPanel",
          "Query Wizard",
          "/wiki/dialogs/querywizard.xhtml", 700, 500);
     pma.registerDialogbox("wiki.kiwilinkPanel",
          "KiwiLink Editor",
          "/wiki/dialogs/kiwilink.xhtml", 400, 250);
     pma.registerDialogbox("wiki.imageBrowserPanel",
          "Image Browser",
          "/wiki/dialogs/imageBrowser.xhtml", 700, 500);
}
```

Fig. 8: registering KiWiApplication-specific popup templates provided by the wiki application.

The first argument is a unique name of the popup. This will be used by a javascript call `showKiwiDialogbox(name)` to show the dialog and `hideKiwiDialogbox(name)` to hide it.

The pointed template contains the function-specific part of the popup and the buttons it needs. For an example look at one of the core popup dialogs.

Tiny MCE Editor plugin

In this document we won't describe the details on how to implement a plugin for the Tiny MCE Editor. For a reference take a look at one of the KiWi editor plugins and the corresponding tutorials at the MoxieCode wiki[11].

Connect popup to the backend

In most cases an editor plugin is a functionality that has a state before the plugin gets triggered. This state is to be submitted to the backend so a backed popup can be shown with some state-specific data. After the user interacts with this popup window, a callback actualizes the state of the editor content. Fig. 2. shows a way to send the state to the backend using the `<a:jsFunction />` tag that can be triggered by the editor plugin and fed with state-specific parameters.

```
<a:form>
    <a:jsFunction name="wizardSetQuery"
        oncomplete="showKiwiDialogbox('wiki.queryWizard');"
        reRender="fmt,#{queryWizardAction.renderedSearchPageParts()}"
        action="#{queryWizardAction.searchEngine.runSearch}">
        <a:actionparam name="param1"
                  assignTo="#{queryWizardAction.format}" />
        <a:actionparam name="param2"
                  assignTo="#{queryWizardAction.query}" />
        <a:actionparam name="param3"
                  assignTo="#{queryWizardAction.lang}" />
     </a:jsFunction>
</a:form>
```

Fig. 9: Example showing how to store the editor state before displaying the dialog.

Using the `oncomplete` parameter of the `jsFunction` we can activate the dialog box (see Fig 2.). When the user interaction is done and the user clicks the OK button an editor-specific callback has to be called with the parameters the user interaction results in order to replace or insert the new wikitext. See Fig. 3. for an example.

```
<a:commandButton value="Ok" id="queryWizardOk"
     action="#{queryWizardAction.submit}"
     oncomplete="insertQuery();hideKiwiDialogbox();" />
<a:commandButton value="Abort" id="btnquerywizardAbort"
     oncomplete="hideKiwiDialogbox();" />
<a:jsFunction name="insertQuery"
     data="#{queryWizardAction.queryParameters}"
     oncomplete="queryWizardJSLib.createQuery(data);"
      />
```

Fig. 10: Example of how to trigger callback after the user finished the interaction to the dialog.

8.2. Type Templates (for Editor, Search, View)

KiWi is made for a wide range of different applications with the idea to give a simple basic layout and a flexible way to show different kinds of content items differently. The presentation is realized through JSF templates coupled with java action beans. The extension mechanism of the KiWi system offers the possibility to specify templates for editing, representing in a result list, and

[11]http://wiki.moxiecode.com/index.php/TinyMCE:Create_plugin/3.x

viewing in the wiki of specific content types the KiWi Extension implements. In case of a project management application, this content types would be *Project*, *Task*, *Milestone*, etc. The default content template has only some simple and generic properties to show that are common for most applications. These could be *Title*, *Description*, *Tags*, *Author* and *Creation date*. A project management extension for example would implement a new content type *Project* (using the facading mechanism, see where?). This type has an *rdf Type* like *sample:Project* and a set of project-specific properties and relations like *Project leader*, *Budget*, *Partners*, *Milestones*, *Tasks*, etc.

The layouting system decides in the following kind of methods about the template to use:

```
public String getContentItemTemplate(ContentItem ci) {
        String res = null;
        ExtensionService es = (ExtensionService) Component.getInstance("extensionService");
        Collection<KiWiApplication> apps = es.getApplications();
        // ask all applications if they have a template other then the default one to use
        for (KiWiApplication app : apps) {
                res = app.getContentItemTemplatePath(ci);
        }
        // If no application has implemented a better visualisation,
        // use the default one.
        if (res == null) {
                res = "typetemplates/defaultcontentitem.xhtml";
                // Find out if the contentItem is a user (hasType kiwi:User)
                for (KiWiResource type : ci.getTypes()) {
                        String seRQLID = type.getSeRQLID();
                        log.debug("type: #0", seRQLID);
                        if (seRQLID.contains(Constants.NS_KIWI_CORE + "User")) {
                                res = "typetemplates/userprofile.xhtml";
                        }
                }
        }
        log.debug("getContentItemTemplate: using #0 to display #1", res, ci
                        .getTitle());
        return res;
}
```

Fig. 11: The LayoutAction. GetContentItemTemplate method deciding which template to use.

To notify the KiWi layouting system to use a specific template for editing, view or to display the item as a search result the extension has to override the following methods

```
public String getContentItemTemplatePath(ContentItem ci);
public String getEditTemplatePath(ContentItem ci);
public String getSearchResultItemTemplatePath(ContentItem ci);
```

The String returned has to be the absolute path to the template in the view folder of the application like `"/myApp/templates/projectViewTemplate.xhtml"`. A sample implementation could look like this:

```
@Override
public String getContentItemTemplatePath(ContentItem ci) {
      String res = null;
      for( KiWiResource type : ci.getTypes() ) {
            String seRQLID = type.getSeRQLID();
            if(seRQLID.contains(MyNamespace + "Project")) {
                  res = "/myApp/templates/projectViewTemplate.xhtml";
            }
      }
      return res;
}
```

Fig. 12: Overriding a `KiWiApplication` method in order to redirect the layout engine for rendering a specific content type.

In order not to modify the default rendering template this method is supposed to return `null`.

The other templates are to be modified on the same way.

The template itself is based on JSF and is allowed to reference the services and actions of the KiWi system using the Unified Expression Language and the component names. The most important context variables are `contentItem` referring to the `ContentItem` to be displayed, `currentUser` refferring to the logged in user. Furthermore, all loaded action components are reachable.

In case the template needs extra backing functionalities providing business logic holding the state of the user interface, the extension has to implement an Action Bean component in the extensions action package.

8.3. Rendering and Saving

8.4. Widgets

8.5. RDFa and RDForms

9. KiWi Development Environment

9.1. Putting KiWi to work (get/compile/deploy KiWi)

What do I need to be able to compile/deploy/run and debug the KiWi project ? The answer is :

•Java JDK (JRE) 1.6.0 .**

•Eclipse 3.4.1. (or NetBeans).

•JBoss application server 4.2.3 or 5.0.0, please take care you need a JBoss version which runs with java 6.0.

•Apache ant 1.7.0 or bigger.

•JBoss Seam 2.1 or bigger (optional).

System requirements : at least : Pentium III with 2 GB RAM.

9.1.1. How you can obtain the KiWi project ?

The KiWi project can be found in the SVN repository : *https://svn.salzburgresearch.at/svn/kiwi/KiWi*

Because the big variety of SVN clients I will not explain in detail the SVN check out steep from the point of view of a specific SVN client. The SVN command line look like :

svn -co 'https://svn.salzburgresearch.at/svn/kiwi/KiWi/trunk myDir

the *myDir* is the directory were the entire kiwi project will be extracted.

Note : the KiWi project has approx. 600 Mb, so be sure that you have enough available storage place. The reason for this huge pack of data are the ontologies files(the ontologies files and other related subjects will explained in detail later in this document).

9.1.2. What you obtain form the repository ?

The answer is pretty simple: an eclipse project. With some small adjustments you can be able to compile/deploy/run and debug it.

The first steps are :

1.create a directory kiwi.ws , this will be project workspace (after the eclipse workspace philosophy).

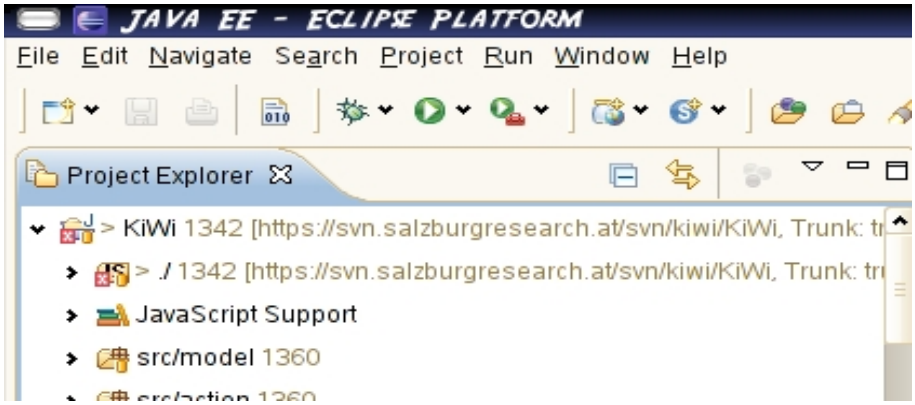
2.check out the KiWi project in this directory - the SVN trunk contains a KiWi directory - this is your project directory.

3.go in the KiWi directory and run from the console : *ant -buildfile build.xml prepareProject*

4.start eclipse and change the workspace in to the new created kiwi.ws directory

If all the previous steeps was successfully, executed the eclipse shows the KiWi project (see the

next screen shot).

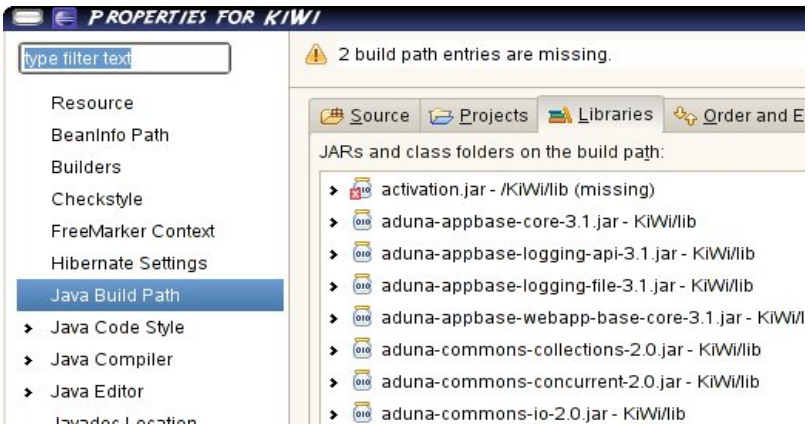


Troubleshooting : in this stage the the eclipse can report a class not found exception (see the next screen shot). The reason could be, that the library list (stored in the .classpath) is not up to date anymore. To handle this, you must be sure that the eclipse's projects classpath includes all the jars from the KiWi/lib directory: select the project properties (project menu->properties) go in the Java Build Path and select the libraries tab, here take care that no jar is marked as missing jar

Sometimes is possible that a needed class is packed into a jar file that is not part of any library. In this case you must locate the jar which contains the needed class and add it to the libraries jars.

If the prepare step was successful, you can develop your own classes, but after this you must be able to run/debug them together with the rest of the KiWi project. To be able to do this, you must know more details about the KiWi project, like configuration files, application server, bootstrap,data base, etc.

KiWi is an enterprise application. It was developed to run together with JBoss and in the future it will be also possible to run it on other application servers. Like most of the EE application it has 4 layers (presentation, business, persistence and data base) and each of them must be configurated to run properly. Later in this document I will provide more technical details about this layers, but for the moment the only one thing that you need to know is that you have to do whit a EE application. For the first steps we included a HyperDrive database (details on http://www.h2database.com/) in the KiWi package, so you don't have to be worried about this layer at the beginning. But, to mention it, The KiWi system is able to run with various databases, which is described later in this document.



Eclipse - missing resource

Sometimes is possible that a needed class is packed into a jar file that is not part of any library. In this case you must locate the jar which contains the needed class and add it to the libraries jars.

If the prepare step was successful, you can develop your own classes, but after this you must be able to run/debug them together with the rest of the KiWi project. To be able to do this, you must know more details about the KiWi project, like configuration files, application server,

bootstrap,data base, etc.

KiWi is an enterprise application. It was developed to run together with JBoss and in the future it will be also possible to run it on other application servers. Like most of the EE application it has 4 layers (presentation, business, persistence and data base) and each of them must be configurated to run properly. Later in this document I will provide more technical details about this layers, but for the moment the only one thing that you need to know is that you have to do whit a EE application. For the first steps we included a HyperDrive database (details on http://www.h2database.com/) in the KiWi package, so you don't have to be worried about this layer at the beginning. But, to mention it, The KiWi system is able to run with various databases, which is described later in this document.

9.1.3. Configure the KiWi project and run it together with the apache ant build.xml file.

As I mention before, you deal with a EE application, so in a very simplistic way your classes (together with all the other KiWi classes) will run on a application server inside of a enterprise container (more details about this on h*ttp://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Overview3.html*). So before a class runs, it must be deployed on the application server. The deploy is done with an ant script named build.xml, which is placed in the Kiwi root directory. Under normal circumstances you are not allowed to modify this file. The *build.xml* has a configuration file named : *build.properties* - its content looks like :

```
# @@warning@@

jboss.home = /jBoss.4
jboss.version = 4.2.3
solr.home = /tmp/kiwi/solr

#jboss.version=5.0.0.GA
#jboss.version=4.2.3
```

The content for build.properties

There you have to specify location and version of your application server. The "#" is used to comment(ignore) a line.

9.1.4. The first KiWi start

If the configuration is done then you are ready for the first test. For this you need to :

1.deploy the kiwi project, to do this run from the command line : *ant explode*

2.start the database (remember we had included one because it is necessary for a EE application). To start the hiperdrive database go in the KiWi/lib and run from the command line java -jar h2-1.1.100.jar, this will also start the h2 client (which appears like a yellow icon in your notification area) in your default web browser - this client requires java script.

3.open a termianl and go in to your JBoss directory and run from the command line : run.XXX (XXX - this depends on which operation system you are). The JBoss will produce a lot of messages on the console, but the server is completely started after you see the an output similar to :

"[ServerImpl] JBoss (Microcontainer) [5.0.0.GA (build: SVNTag=JBoss_5_0_0_GA date=200812042120)] Started in ...."

4.open your web browser and enter the URL : *http://localhost:8080/* to see the JBoss start page (a prove that the JBoss is running)

5.after you enter the URL : *http://localhost:8080/KiWi/* you should see the kiwi start page (this page

requires java script)

To stop the JBoss server (and together with it KiWi) you must interrupt the JBoss process in the terminal where you start it; or you can call the shutdown.XXX script from the server bin directory. Alternative to run the JBoss in to terminal you can start it from the eclipse (See the JBoss and Eclipse).

By now you are able to deploy and run the KiWi project. Unfortunately for developing you need a little bit more. In the next section we will describe in detail the configuration and the deployment features.

9.1.5. Start KiWi Troubleshooting

Under normal circumstances the The JBoss server runs with its default configuration, unfortunately during the development process you need more. One of the most common problem here is the *OutOfMemoryException* which occurs when you start the server or when the server runs and you do some action with it. To solve this you need to improve your actual server configuration placed in the *run.conf* (the file is in the your JBoss directory *bin* directory). You can experiment until you get the wished behaviour or you can replace the JAVA_OPTS with the following line :

```
JAVA_OPTS="-Xms128m -Xmx1024m -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000
-XX:MaxPermSize=512m -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

JBoss JAVA_OPTS parameter

9.1.6. Advanced configurations

The KiWi developing environment contains a lot of configuration files are are in the KiWi/resource directory. The most important are :

The DataSource configuration files

This files are located in the *KiWi/resource* and they follow the pattern KiWi-*-ds.xml. There are three of them - dev, prod and test. Each of this covers a certain application life cycle aspect (e.g. developing). The DataSorces are used to configurate the connection with data base. Every time you persist an instance of an entity the persistence provider will open (or reuse) a pooled connection to the database - in the majority of cases each data source is commonly associated with a connection pool that contains the information for connecting the database.

Usually you won't interact with any of this files. But you have to change it if you want to use other databases (e.g. postgres instead of H2). By default this files are using the H2 database.

The EE configuration files

This files are placed in the *KiWi/resources/META-INF* directory. Some of the most important files are:

•application.xml - the deployment descriptor - it describes which Java EE modules are placed in which ear(jar or war) file. For the moment there is only one web module and only one ejb module. As far as you don't plan to add/remove new modules you don't need to access this file.

•ejb-jar.xml - deployment descriptor for only one ejb module. In this file you can add new interceptors, filters, environment variables, etc. if you need them.

•jboss-app.xml - the JBoss deployment descriptor, here you can configure JBoss vendor specific

issues, this can differ from a server version to an other. Please consult the JBoss documentation for more details.

•persistence-*.xml - the persistence descriptor is used to configurate the Entity Manager. A reason to change this file can be to change the database dialect or to specify some vendor specific database properties (hibernate properties in this case). Don't mix the *persistence-*.xml* files with *DataSource configuration files*! Instead of DataSource configuration files this file configure the entity manager (like which dialect the Entity Manager uses) and not te way how the Entity Manager connect to the database.

•MANIFEST.MF - this is the jars manifest file for the kiwi.ejb-jar. This file contains basic information about the KiWi project like version, vendor or dependencies. The dependencies can be considerate the main information contained in this file, practically the classpath dependencies or which classes(packed in jars) are required to run KiWi.

The Web / presentation layer configuration files

This files are are placed in the *KiWi/resources/WEB-INF* directory. Some of the most important are :

•web.xml - the web container deployment descriptor - mostly you don't interact directly with this file. Because the JBoss seam and Facelets use this file for framework configuration don't change it without a serious reason.

•pages.xml - the Facelets configuration files - it contains the main navigation rules for most of the presentation layers. A navigation rule describes in which condition the user can navigate between two pages. A reason for altering could be a major change in a navigation rule. Complementary with this file are the individual page configuration files (a.k.a. .page.xml files)- these are in the same directory with the xhtml files (KiWi/view) and will be described in more detail later in this document.

•faces-config.xml - the configuration for Facelets - usually you don't interact with this file.

9.1.7. Advanced compile and deploy using the build.xml apache ant file

The compile and deploy task is accomplished with a ant file named build.xml, placed in the KiWi project root directory. With this ant file you are able to execute the compile&deploy tasks as well as the code generation tasks. In this section we will describe the compile&deploy tasks.
I organized the tasks after them importance.

•**explode** - by far the most important tasks - it is used to compile, pack and deploy the KiWi application to the application server. Basically this task only performs an incremental hot deployment of web artifacts and java classes. For the compile and pack it uses other existent ant targets like jar, war, ear or datasource. One important thing to notice here: this task deploys the exploded archives to the application server, it does not pack the application in ear files.
•**explodeandtest** - similar to the explode task, but it also compiles and prepares the test classes. One important thing to notice here: this task is designed to work from eclipse, if you use this script from the command line (using a normal ant syntax) you may run in problems (TODO : I am not shore here I still must do some experiments). This is the default task (TO Discuss this default task bring a strong eclipse dependency so I am not shore that it must be default).
•**cleanExplode** - it cleans the entire environment for classes and other generated files produced with a previous (ant) session. After that it compiles and explodes the application and also explicit informs the server that new resources are available. You should use this task when you need to be sure if your classes (and other resources) are deployed and loaded from the application server. Please note that this task is more resource expensive than the explode task.
•**clean** - it cleans all the compiled classes, the test output and the test report directories. You should use this task if you have doubts about a compiled class file.

•**unexplode** - this is the complement to the explode task - more precisely it removes your application from the application server (deployed with a previous session)
•**doTest** - prepares the test environment, runs the tests and creates test reports for the entire project. This task requires a running HiperDrive database. You can start it manually : *java -jar h2-1.1.100.jar* (the jar is in the lib direcotry). This task can take a while so don't run it unless you don't really need it.
•**test** - it only builds the test and runs them, but it does not prepare (or clean) the test environment. This can cause various problems if the test environment is not yet prepared (more details about the test environment later in this document). Because of this task assumes that you already have a clean and ready to go test environment it is also difficult to use, that is the reason why the *doTest* task is preffered.
•**validateConfiguration** - this task is used to prove that your changes in the xml (configuration) files are also correct. Use this task if you have doubt about your changes. If this task fails, that the application server will crash your application very likely(during deployment or run time).
•**prepareProject** - this task prepares your project for developing after you check out the project from SVN. Take care, this task is designed to work one time when you just checked out the project from SVN, don't run it after.
•**deploy** - it copies the actual (compiled classes) and copy them on the JBoss deployment directory, the previous version is removed. Please note that this task uses your already compiled classes and because the previous version is removed you must be shore that all the existed source files are compiled . Use this task when you already compiled (all) your classes and you only needs to deploy them on the server.
•**undeploy** - it removes the already existent KiWI from the application server.
•**dist** - creates a kiwi distribution using the actual system. Please note that this task can take some minutes. Use this task when you need to build "out of box" kiwi system based on your local version. More precisely this task will deploy your actual kiwi version on a JBoss server and then it packs this server in to a zip file. If you unzip this file you can start the server and you will have a running kiwi system (see the distribute kiwi section for more details).

This are the most common used task, the build.xml contains more, but most of them are hided by one of this main task. Every main task delegates work to other small tasks.

You can find more details about apache ant on (*http://ant.apache.org/*).

Deploy Troubleshooting : The KiWi system is based on java 6.0 (1.6.0), for this the all the ant tasks (specially the one where the java compiler is involved) are expecting that the your system is use java 6. If your system uses an other java version then the compile task will fails, and you will have a similar output :

```
.........
compile:
    [javac] Compiling 511 source files to /Users/mradules/labo/kiwi.ws/KiWi/exploded-archives/KiWi.jar
    [javac] javac: invalid source release: 1.6
    [javac] Usage: javac <options> <source files>
    [javac] where possible options include:
    [javac]   -g                  Generate all debugging info
    [javac]   -g:none               G
.....
```

Ant error output, the system is using a wrong JVM

If you get this exception please be shore that your default JVM is a java 6.0.

9.1.8. Distribute KiWi

Sometimes you need to be able to demonstrate how the kiwi system is working and this outside of

your machine, for this you need to be able to create a running kiwi system based on your local kiwi - this is the purpose for the ant task named : **dist**. After you run this ant task the *dist* directory placed in the KiWi project root contains two zip files :

1.kiwi-jboss-XXXX.zip - it contains the actual kiwi distribution.

2.kiwi-src-XXXX.zip - all the sources for the actual kiwi distribution.

the XXXX depends on the version number, this information is stored in `project.version` ant property (see the build.xml file for more details).

To start the kiwi server go in kiwi distribution directory (this is the directory where you unzip the distribution zip file), in the bin directory and run the run.xxx script (xxx depends on your operating system for linux and mac os run.sh and for windows run.bat), this action will also start the H2 database. If the scripts run successfully then you will notice in your notification area a small kiwi logo, if you click on it (or right click and then "*show kiwi*" pop up menu item) you will start your default web browser with the *URL* for your local kiwi. The URL for your local kiwi server is *"http://localhost:8080/KiWi/wiki/home.seam"*.

The KiWi system requires java 6.0 (1.6.0), if your machine is running a wrong java then the run.xxx script will complain about.

9.2. Directory Structure, where/how to organize your files

9.2.1. KiWi Main Directory Structure

The KiWi project uses a JBoos Seam specific file structure. The KiWi project root directory contains a lot of files and directories the most important are :

•**config** – this directory contains KiWi project specific configuration files, the configuration files are for third party tools (e.g. Checkstyle, Eclipse Code formatter or find bugs). You can new configuration files for other third party tools here in to a separate directory with a meanfull name.

•**dist** – this directory contains the KiWi distribution and it is generated with the dist ant task (see the Distribute kiwi section for more details). This directory is generated – don't sore any kind of information here!

•**docs** – this directory contains all the KiWi related documentation. Here you can find also the java api documentation but please note that this is automated generated.

•**exploded-archives** - this directory is used during the deploy to server process and it contains the kiwi.ear (ear=enterprise archive) expanded. This directory is generated – don't sore any kind of information here!

•**extensions** – this directory contains all the KiWi extensions (more informations about this on the KiWi extension section).

•**lib** all the libraries(packed in jars) used I the KiWi projects. If you want to use other library together with the kiwi system then you need to copy it in this directory. If you need that a certain jar is deploy like together with kiwi on the application server during the deploy process then alter the deployed-jars-ear.list (or deployed-jars-war.list for web applications).

Even if the purpose for this directory looks simple wrong manipulation can bring you a lot of problem. Before you copy any new jars here be shore that the classes contained in the new added jar are not already contained in the already existing jars. More, be shore that the new added jars classes are not based on very old and deprecated APIs or on very new APIs. If your implementation is based on some new implementation which may come in conflict with the actual one you can still use the endorsed solution, this solution implies a special directory where you can place your

jar(s). For JBoss this directory is named endorsed and it is placed in the root directory for your server. For more information please refer also the endorsed section and the *http://java.sun.com/javase/6/docs/technotes/guides/standards.*

◦ Lib/solr – the default solr distribution. Don't alter this directory. Refer the *"Solr section"* for more details about sorl installation and configuration.

•**resources** – this directory contains all the meta information required for an enterprise application (e.g. the datasource files, workflow files). See also the Advanced Configurations section for more details.

•**src** – this directory contains all the java sources files for the Kiwi Core. This directory follows also the seam convention. See the "*KiWi Core directory structure"* section.

•**view** – this directory contains all the view related files (xhtml, javascript files, css) sources files for the kiwi core.

•**build.xml** – ant script used to compile/deploy/test/build distribution for the KiWi system. Please refer the upper section for details about the buid.xml. Its default task is the explodeandtest task. This tasks deploys the actaul KiWi project to the Jboss and run the tests. The build.xml requires a configuration file the build.properties.

•**build.properties** - the actual working copy for the configuration file, this file is based on a template named : **build.properties.tmpl**.

9.2.2. KiWi Core Directory Structure

All the KiWi Core source codes are stored under the **src** directory placed in the KiWi project root. This src directory contains three subdirectories :

•**action** – this directory contains all the *"Action Beans"* used in the core system. An action bean is a seam component used directly from the presentation layer (view) to accomplish view specific actions. In a very generic way we can say that the action beans are used every time when the view wants to communicate with the rest of the system. All action bean must use the "Action" suffix in they name (e.g. PopupManagerAction). It is not desirable to add classes here, if you need to extend the KiWi functionality, use for this the extension mechanism.

Refer the *"From Client Server to Enterprise"* section for more information about this.

•**model** – this directory contains all the classes related with the data manipulation and management. Here you can find the Entities, the EntityManager, the versioning system, the triple store and many others. It is not desirable to add classes here, if you need to extend the KiWi functionality, use for this the extension mechanism.

Refer the *"From Client Server to Enterprise"* section for more information about this.

•**util** – this directory contains a collection of general purposes java classes.

•**test**– this directory contains all the unit test for kiwi and for the kiwi extensions.

9.3. Extension Mechanism

In a very simplistic way the *KiWi system* is formed from two kind of components :

•Core components

•Extensions

An *extension* is a interchangeable piece if software which encapsulate a certain behaviour, design to work together with the KiWi core with the purpose to extend the core functionality. The user can create its own extensions, and it must be able to decide which extension to use(together with the KiWi Core).

An Extension can be :

•a **service** - encapsulate a certain specific behavior (applicable on the core components) and make it available in a friendly way. This extension is similar with the **action** extension from many point of view but what makes it different is the way to way how is applied, the **actions** can be interact only with *content item(s)* and the services can be applied on more than content items. By example a service can be a stateless session beans that provide certain service functionality (i.e. Java methods) to other components.

•an **action** - encapsulate a certain specific behavior applicable on a *content item*. Understand this extension like a simplified kind of service extension (applicable only on *content item(s)*). *NB: the "action" mechanism is not yet available, but foreseen for the future*.

•a **perspective** - a perspective is useful in organizing various KiWi views widgets (and other graphical components) around certain task(s). A perspective can be also helpful in managing menus and tool bars(by group them). The user can use more than one perspective - he can switch between the perspectives. As an KiWi extension developer we can either create a new perspective from scratch or enhance existing perspective. Enhancing existing perspective essentially means to accommodate your view in an existing perspective.

•a **widget** is a small user interface component that can be used by different perspectives to show or edit a certain kind of information, e.g. EXIF metadata, tagging, ...

9.3.1. How it works ?

An extension make sense only in a KiWi system. This means there must be an way to bring the extension and the KiWi core together - this is call extension installation. In the most common cases the extensions are packed in jar files and they are down-loadable from the KiWi portal. You only need to copy them in the extension directory and the KiWi system will install them.

Alternative you can use the *auto install* feature to allow the KiWi system to claim the needed extension by it self - you only must specify the extension name. (still in work)

9.3.2. Develop your own KiWi service extension.

For this you need to add your extension directory to the actual KiWi extension directory (extensions). The directory structure must look like :

```
extensions
+-- your extension
   +-- resources
      +-- META-INF
         +-- components.xml
         +-- faces-config.xml
      +-- messages_en.properties
   +-- src
      +-- exampleplugin
         +-- action
            +-- ...
         +-- api
            +-- ...
         +-- services
            +-- ...
   +-- view
      +-- javascript
         +-- exampleplugin.js
      +-- layout
         +-- template.xhtml
      +-- stylesheet
         +-- exampleplugin.css
      +-- file1.xhtml
      +-- file2.xhtml
      +-- ...
```

KiWi extension directory structure

This directory structure is very important to, otherwise the kiwi build process can not find the your components(extensions); the kiwi build process is responsible for the compile and deploy the KiWi core and the exception.

9.3.3. Directories details

•resources - used to store the META-INF information about the files contained in the extension. Here you can find :

◦ components.xml - seam specific file, it contains a list will all the seam components used in this extension

◦ pages.xml - Facelets specif file - it contains the navigations rules for the views (pages).

◦ messages_XXX.properties - used to the internationalization.

•src - all the java sources

•view all the view files, take care the navigation rules are sored in the META-INF. This directory contains the extension home page.

TODO : In the future will be possible to develop the kiwi extensions outside of the KiWi system - like independent projects. Here we must provide a extension project pattern - together with a kiwi mock. The kiwi mock purpose is to simulate the kiwi system install feature - in this way the user can test it self the plug in before goes official.

The first thing what you must care about is "to present" your extension to the KiWi system, this is done with a loader. A loader is a managed class which has a method triggered by the seam container in the Post Initialization phase. Because you have a managed class you can use the dependency injection. The KiWi system has a service used to register extensions - this is the way how you introduce your extension to the KiWi system. You can inject also other resources (like logger).

```
@Name("myExten.cfg")
@Scope(ScopeType.STATELESS)
public class ExtensionConfiguration {

   @In
   private ExtensionService extensionService;

   @Logger
   private Log log

   @Observer("org.jboss.seam.postInitialization")
   public void initializeExtension() {

      extensionService.registerApplication(new KiWiApplication() {

         @Override
         public String getIdentifier() {
            return "myExtension";
         }

         @Override
         public String getName() {
            return "myExtension";
         }
      });
   }
}
```

Used to register a KiWi extension

This code snippet does :

•injects the ExtensionService in the extensionService field

•annotate the initializeExtension method with the @Observer("org.jboss.seam.postInitialization"), this will inform the Seam container to trier this method in the post initialization phase. Because we are in the post initialization the extensionService was already injected.

•create a KiWiApplication instance (with an anonymous class) which livers

◦ the extension name - a human readable String used to identify the application

◦ the identification - an application unique id for the extension.

◦ FIXME : what happen if two extension have the same id - some exception ?? I must try

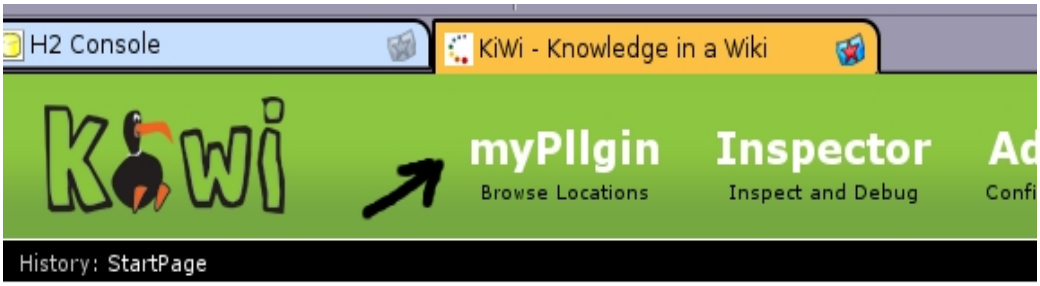
•register this instance in the extensionService.

•you can prove if the extension is already install like in the next snippet :

```
   private void register(ExtensionService service, KiWiApplication app) {
      final String name = app.getName();
      final KiWiApplication getIt = service.getApplication(name);
      if (getIt == null) {
         service.registerApplication(app);
      }
   }
```

Checks if an extension is already install

If you compile and deploy your extension the KiWi main view must look like :



KiWi and the installed extension list

Note for windows users : the windows file system does not make a difference between files names with big and small capitals and because the identifier is use to build the path for the extension home page don't use capital letter for identifier.

So if we try to summarize : to register a KiWi extension you need to provide a KiWiApplication instance to the ExtensionService. Unfortunately this was not all.

Seam note : if everything is clear until here, there is still a open question and is seam related : Why do I use the *@Observer("org.jboss.seam.postInitialization")* why I don't observe something else? The reason is simple if you know how the *@Observer* and the *Seam container* are working. The @Obsever annotation allows you to observe its build in events (or contextual events), and the *org.jboss.seam.postInitialization* is a seam event triggerd when seam has initialized and started up all components. As the name suggest an extension is nothing more then a extension of base functionality (or components) so it is important to start the install process after all the base components are initialized and started up.

The reason why we don't objserve a certain class initialization is the flexibility - we don't want to creat a dependency between the extesnion mechansim and a certain class.

For more information about Seam please go in the Seam section.

Seam note : Don't forget to use the *@Name* annotation - otherwise the seam container will ignore your observer because without this addnotation your class will be not a Seam Component and the Seam Container will not care about it.

9.3.4. Extension advanced.

In the previous section we see how to register a extension, in this section I will try to explain who I can customize the added extension. For this you need to understand how the kiwi main view (header) works.

After your extension is registered in the ExtensionService the kiwi main view header ask for all the installed extensions and for the each extension it build a commandLink. This are the big bold white text labels. A commandLink is a link which acts like a button - when the user click on then the view content changes - it loads the *extension home page*, this action is know also like the switch application. The *extension home page* is a xhtml document with the name home.xhtml placed under the path "/identifier/" - the identifier is described with the KiWiApplication instance. In the previous example the identifier is the string returned by the getIdentifier method ("*myExtension*") in this case the *extension home page* is "*/myExtension/home.xhtml*".

In your extension directory structure this file is placed under the */view* directory - so the project absolute path for the myExtension extension : is *.../extensions/myExtensionDir/view/myExtension/home*.

Because the identifier is unique each extension will have its own view home directory.

The home.xhtml is a normal JSF/Facelets file and it must be develop according with the JSF/Facelets rules.

This sequence diagram shows the two main extension related work flows : the extension loading (the upper flow) and the application switch, more the diagram shows the participant type (e.g. XHTML Document).

9.3.5. Navigation rules details

A navigation rule is a rule which describe how the user can navigate between two or more documents (e.g. it goes from home page to its specific application page). The KiWi application is a Seam base application, the Seam provides you various of ways to define navigation rules.

One of this way is the return String method - If a seam component returns a String from method which is called from a GUI component then this String is the new location where the user will navigate. This technique allows you to define dynamic navigation rules on the other side makes your application less flexible because they are hard coded.

by example :

The next line call (from the view layer) the method "switchApplication(app)" on the seam component named kiwi.ui.headerAction.

```
<h:commandLink action="#{kiwi.ui.headerAction.switchApplication(app)}" >
```

this method looks like this :

```
public String switchApplication(KiWiApplication app) {
....
   return "/"+app.getIdentifier()+"/home.xhtml";
}
```

here the return value is used like navigation target - the browser loads the page identified after the dynamic build string.

9.3.6. Navigation constrain

The Seam provide a fine granulate navigation/configuration feature over its XXX.page.xml - with this file you can have a navigation/configuration file for each page. This feature is not supported by the extension mechanism.

All other seam (and JSF) navigation rules are available.

For more details see the Seam section.

9.3.7. Extension via descriptor files

There is an second way to register an extension without to generate a KiWiApplication instance. For remembering the KiWiApplication instance is a class which contains information about your extension (e.g. the name or the unique identifier).

The KiWi system provides a descriptive way to configure your extension via properties files. For this you need two things :

•a package for your configuration

•a properties file

The properties file can contain :

•name - the name for your extension,this is mandatory

•identifier - the unique identifier for your extension,this is mandatory

•version - the version for your extension

•project - the project where your extension belongs

•author - the extension author

•passInConstructor - future feature (see the passInConstructor section)

•presentationType - future feature, it describe which kind of presentation layer part it represents (e.g. the footer).

As you can see only the name and the identifier and mandatory - this two properties correspond to the getName():String and getIdentifier():String from the KiWiApplication interface and they have the same meaning.

All the KiWi descriptor files must follow the "XXX-descriptor.properties" name pattern.

The descriptor file you must place it in classpath. If you use the KiWi ant build file then the build file will take care about this issue for you - otherwise you must handle this problem alone.

If the descriptor exist in the package "kiwi.extension.descriptors" and this package is placed in the classpath then the KiWi system will locate them and the extension will be automatic installed.

Alternative you can install extensions where the package or the descriptor file name pattern or is other. For this you only need to create your descriptor files on other package and place this package in the classpath. After this you need to use the *DescriptorsFactory* class from the package *kiwi.extension* this class will allows you to build KiWiApplication for all your descriptor files, this *KiWiApplication* instances can be installed in the *ExtensionService*.

The following code snippet exemplify this :

```
final List<KiWiApplication> apps;
try {
    final String confgBase = "my.package.cfg";
    apps = DescriptorsFactory.getDescriptors(confgBase); // 1
} catch (final IOException e) {
....
}
...
for (final KiWiApplication app : apps) {
    register(extensionService, app); // 2
}
......
private void register(ExtensionService service, KiWiApplication app) {
    final String name = app.getName();
    final KiWiApplication getIt = service.getApplication(name);
    if (getIt == null) {
        service.registerApplication(app);
    }
}
```

On the **1** you load the all the descriptors files from the package *my.package.cfg* and you transform it in KiWiApplication instances. After this on 2 it install it on the ExtensionService (and check if the extension is not already installed).

9.3.8. Customize the installation with the passInConstructor

The passInTheConstructor property allows you to customize the way how your extension is installed. This property value is a String and this string represents the absolute class name. This class must have a constructor witch accepts two arguments a ExtensionService and a ConfigurationService - this is the only requirement. More you can use the *@kiwiPostInstall*, *@kiwiPreInstall* and *@performInstall* annotations to mark the pre/post install methods.

9.3.9. Why to use this class ?

In most of the classes this class is used to prove the condition under a extension can be install - this because some extensions requires other extensions (or resources). If a descriptor contains this property an instance will be build and then before/after the install process begin/end the method will be trigger. Because the instance get in its parameter the ExtensionService you can check which other extensions are already installed and you can take the proper decision. If the method annotated with *kiwiPreInstall* throw an exception then the extension installation does not take place. You can achieve the same thing id you use the *@performInstall* annotation, and you return false - the *@performInstall* is applicable for a method with no arguments which can return a boolean.

You can do more than check if the pre install condition are satisfied - you can allocate needful resources by install and you release them by uninstall. The uninstall can be use also to remove the possible GUI components from the view - if this is necessary.

9.3.10. Setting the default extension

The default extension is known to the ConfigurationService by the key “kiwi.app.default”. When the system is not running, it can be changed in the database as follows:

update configuration_listvalue set element = 'my_extension' where configuration_id = (select id from configuration where kiwikey = 'kiwi.app.default');

9.4. SORL

9.4.1. What is SORL

SORL is the acronym for Searching On Lucene w/Replication and it is a stand alone enterprise search server which applications communicate with using XML and HTTP to index documents, or execute searches. Find more information about this theme on (*http://lucene.apache.org/solr/*).

9.4.2. SORL and KiWi

The KiWi system uses SORL in embedded mode, that means you need to install it(SOLR) st. If you check up the KiWi project from from the svn then you will also get a SOLR distribution also, this is placed in : *root/lib/solr* , where root is your project root. You must copy this directory in a know location, because KiWi persistence layer will try to nd this location and use it. You can copy it by hand or you can use one the *prepareSolr* ant task to do this for you, this is the recommended way.

The *prepareSolr* requires a property named *solr.home* in the build.properties file, this path is the path for your local SOLR. You also need to configure your persistence unit (persistence-XXX.xml) properly, for this add a new property named *kiwi.solr.home* with a proper value to the KiWi persistence unit. The XXX value (persistence-XXX.xml) depends on which software life stage you are (dev for developing, test for testing etc).

Here is a rule to follow : the *solr.home* from the *build.properties* file must point on the same path with the *kiwi.solr.home* from the *persistence-dev.xml* file.

Now you must redeploy the kiwi application and to restart the server.

9.4.3. Path and relative path

Please take care the path specified with the *build.properties*'s *solr.home* property is a absolute path, in the *build.properties* file you can only define absolute paths.

The *kiwi.solr.home* property from the *persistence-dev.xml* file is more generous, here you can specied absolute and relative paths - the relative paths are relative to the jBoss bin directory (e.g. if you use the solr value for your *kiwi.solr.home* then the kiwi will search the solr in the JBoss/bin/solr directory).

The way to dene absolute or relative paths dier from an operating system to an other. E.g. under ms environment the absolute path starts with a drive letter followed by the double point char or under unix(linux and mac os) with the slash character. All other ways to dene a paths are leading to relative paths.

9.4.4. Troubles with SORL

In most of the cases if the two paths previous described are not pointing to the same location then a run-time exception will raise, the exception will claim a certain SOLR XML specific file. (e.g. "RuntimeException: Can't find resource 'solrconfig.xml' in classpath or ...."). In this case the path where kiwi search the *SORL* is wrong. To solve this problem you need to prove if the property

named *kiwi.solr.home* from the deployed *persitence.xml* (placed in the jBoss.home/server/default/deploy/KiWi. ear/KiWi.jar/META-INF) points to the right path (the path where the SOLR is installed).

The directory where the SOLR is installed must look like :

```
..
README.txt
bin
conf
```

SOLR directory structure

9.5. From Client Server to Enterprise

One of the most intriguing features for one computer program was and is the possibility to communicate with other programs (on the same computer or on other machines). The next sections briefly presents the most popular distributed computing architecture and try to explain the evolution process(from client-server to enterprise application) and its reasons.

9.5.1. Two tier architecture

Knew like Client Server architecture this was one of most primitive and most popular form of distributed computing. It implies two parts : a server and a client, the server is used to serve one or more clients. The big problem with this kind of architecture is the tidy coupling between the two parts, the sever can serve only one kind of clients and the clients must know the server. Even more, because in the most of the cases the server provides access to some (local) resources it tends to encapsulate the way how this resources are managed and manipulated - in this way the server has (at least) two responsibilities : to serve the clients and to manage (and manipulate) local resources and this is a bad design.

We can say that the server tends to become monolithic by assimilating more an more responsibilities. The solution for this can be an other layer, by

adding this we are already to the next section.

9.5.2. Three tier Architecture

In the previous section we infer the necessity for one extra tier but the question is where I to place it ? The Answer is : between the client and the server. In this way the client and the server are not so strong depending on each other. In this case the tiers are : Client, Business (new) and Server. Some responsibilities from the server tier will move to the new tier (business), the purpose of this move action is to create application tiers with a single (or very less different) responsibility. The Business tier will contain the application logic, this logic will be used from the Client tier to accomplish client specific task and the business tier will apply its logic to the Server tier. For an easy understanding we can make an analogy between this architecture and a real life situation; the application logic (business tier) is like a tool box, it contains tools specialized for certain operations. The user (client tier) use this tolls in a convenient way to accomplish his tasks. The user action are applied on some pieces of (raw) material and the final products are stored in a depot managed from an other guy - this guy is the server tier.

This new architecture brings a lot of **advantages :** the server tier is interchangeable - we can change/extend the server implementation in the future without to care about the other tiers, the same logic for the client. Even more you can provide more more than one clients by combining the business tier features.

But with all of this the three tier architecture still has some pay offs, the server tier still has to many responsibilities, it serve the clients and managed and manipulated resources. The solution for this is an extra tier.

9.5.3. Four Tier Architecture

As we discuss in the previous section the server tier is spit it now in two tiers, so per total we have four, this four tiers are Client (presentation), Business, Persistence and Database. The Client tier (knows also like Presentation) is used to capture the user gestures and to transport them on the undying tiers - the client does not posses any kind of logic, the logic is placed on the more deeper tiers. The next tier is the Business tier, here are dened the application logic (or the business logic), this logic needs to be applied on some objects - this objects are named the domain models - the Domain Objects forms the next tier (the persistence tier). In most of the cases the information needs to be persisted, this is happen on the last tier, the database - here the domain objects are serialized persisted and restored from them persistent from. For a better understanding I try to continue the example (real life analogy) from the Three tier Architecture section. In that analogy the user uses a set of tools on some pieces of raw material, the tools and the raw materials was stored and manages

from only one person. In this new architecture there are at least two persons which manage/store the already ready pieces and the row materials, each of them have only one specialization, one only sores the already ready pieces in the depot (the database tier) and the other one will help the user to find the right piece or raw material (the persistence tier).

Advantages : The design is not fragile and easy to use - the sub-systems are now independent within the whole design, changes and updates can be better applied. The tiers (or layers) are abstract dened and this make them interchangeable (or pluggable); in this way the persistence solution (the database) can be change without to affect the upper layer functionality. This kind of architecture is from far one of the most popular architecture in the JEE word.

The kiwi project follows this architecture also.

9.5.4. Sort of conclusion

The four tier architecture is not the last evolution stage. I think that we can consider the way how the client server architecture changes until it reach the four tier form like an evolution. Like the natural evolution where the unicellular organism evolute by creating new (and specialized) organs also the primitive architecture evolute and add new and specialized layer(s). The advantage over the natural evolution is a fast the life cycle - increasing the possibilities to adapt to new conditions.

9.5.5. The KiWi architecture and the Four(Five) Tier Architecture

Kiwi uses a 5 tier architecture, this because the business layer (from the Four Tier Architecture) was spited in two other sub layers. The reason why the business layer is divide in two is because the the system tries to separate the user related actions from the system (application related action), in this way the service layer can be used not only from the presentation layer it can be also used from other applications (e.g. like web services). The actual tiers are :

1.The Presentation Layer - used to capture the user gestures and validate the input. This is the graphical interface - the kiwi uses for this JSF/ Facelets.

More about this on the The Presentation Layer section.

2.The Action (bean) Layer - the gestures captured in the upper layer are transformed in system gestures. The components included in this tier are beans (or more precisely seam components).

3.The Service (bean) Layer - each system gesture can use one or more services to accomplish its task. The components included in this tier are beans also (or more precisely seam components).

4.The Persistence Layer - the task from the upper layer are applied on domain objects, this objects are stored and version on this layer. This layer is also know like the kiwi core, from this point of view we can say that the kiwi core main purposes is to persist and version the domain objects – more details about this on the Domain Object Facade.

5.The Database Layer - the tier stores physical the domain objects and the related meta data.

Appendix I: List of Components

Appendix II: List of Named Queries