Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis No. 3398

Concept and Implementation of a Pluggable Framework for Storage, Transformation, and Analysis of large-scale Enterprise Topology Graphs

Hannes Todenhagen

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Tobias Binz
Commenced:	October 02, 2012
Completed:	March 24, 2013
CR-Classification:	E.1, K.6

Abstract

The addition of on-demand cloud computing offerings increases the complexity of IT systems rapidly. Enterprise Topology Graphs depict all the components of enterprise IT and their relations, to regain insight into enterprise IT. The focus of this work is the research, design and implementation of a framework to store and manage these graphs in an efficient way. The difficulties are the enormous graph sizes and a lot of meta information, leading to a complex design to offer a good performing solution. The framework is based on a graph database to store the Enterprise Topology Graph efficiently and offers a pluggable architecture to be able to extend the functionality, e.g. with transformation operations the graphs. With the reference implementation of this framework, the complex structures of enterprise IT systems can be stored, managed and easily manipulated to gain a more detailed view on the IT components and their dependencies.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

INTRODUCTION

This Chapter gives a short summary of the context of this diploma thesis. It shows the need of the framework and gives an example to show the motivation of the work behind. Also the problem statement and the outline of the document is given.

1.1 Motivating Example

According to surveys executed by the *Cloud Industry Forum*[1], *Gartner*[2] and *Apache CloudStack*[3], a paradigm shift in favor of *Cloud Computing* can be recognized in the 21st century in the IT landscape. Lots of research has been done and is, of course, still going on. Leading companies switch from an own datacenter infrastructure to a cloud based solution. Resources can now be requested on-demand and a lot of infrastructure, platform and software services are already offered [MG09].

As an motivating example for this piece of work, I will take an imaginative medium-sized software firm called MMS Corporation (The Modern Medium-sized Software Corporation). This company has employees in Central Europe, in Asia and North America.

The product of the MMS Corporation is a standard desktop application with a lot of road construction domain based knowledge. This application is mainly developed in China with the architectural knowledge and designs of german software engineers.

To provide the needed infrastructures and servers the company has its own system engineering team and a data-center in the german headquarter. In the past few years the IT department decided not to buy more hardware, but to use cloud offerings. The repertoire of this offerings includes storage, web-servers and application-servers, as well as infrastructure managing servers and a on-premise software solution for the accountancy.

[1]Cloud Industry Forum - http://www.cloudindustryforum.org/
[2]Gartner - http://www.gartner.com/technology/research/cloud-computing/
[3]CloudStack - http://www.cloudstack.org/

Because of the variety of technology the system engineering team almost lost the control over their infrastructure and has a huge problem to maintain the overview of the multi-vendor and grown server landscape.

The system engineering team starts to draw a graph containing all servers, their applications and dependencies, the network infrastructure and the used cloud offerings. The problem with this solution is, that the graph of MMS Corporation is not standardized and has to be administered manually by a system engineer.

To be able to work with this enterprise graphes in an automated way, a formal definition has to be created. This formal definition is offered by the *Enterprise Topology Graph (ETG)* definition [BFL+12]. This standardized graph represents the whole hardware and software infrastructure of a company and their relations. Updates and their side effects can now be planned and rolled out with the knowledge of the whole IT system.

To fully solve the MMS Corporation's problem there should be a way to create, save, modify and export *ETGs* in a computer assisted and automated as possible way.

1.2 Problem Statement

In this work a concept and implementation of a pluggable framework for storage, transformation, and analysis of large-scale *Enterprise Topology Graphs* is created. This task can be divided in six research tasks:

- **Research efficient storage and serving of ETGs**

  A efficient way to store large-scaled graphs has to be found. Therefore, different graph- or document-databases are presented and their pros and cons evaluated. This database solution must serve requirements like a good performance with big graphs, it should be easy to use and scalable. In addition, the database should make use of standards.

- **Concept of the ETG and segment (subgraphs of ETG) store**

  The ETGs and their subgraphs should be stored in the most optimized way. Requirements like the size and the readability should be met. Also some kind of versioning has to be introduced to be able to keep track of changes over time.

- **Concept of an extendable framework for operations on segments**

  The segments of ETGs must be modifiable. The modifications should be executed by Java clients with defined operations on the nodes and the edges of a graph. It should be possible to undo and redo changes to segments and to reintegrate a segment into the source ETG.

- **Creation of "large" example ETGs for testing and performance evaluation**

  To test the resulting framework and storage solution it should be possible to create large graphs with a test contend. For this purpose an additional module should be added to the framework, which can serve the storage with example graphs.

- **Implementation of the framework**

  An implementation of the created concepts should be created. This implementation must be written in *Java* and should use modern techniques to comply with requirements of scalability and modularisation. In addition, some example plug-ins should be implemented, like e.g. a sub graph search plug-in.

- **Implementation of a simple web GUI**

  To manage the content of the storage solution and to provide the possibility to modify the graphs, a simple web GUI should be provided. This GUI should supply the mechanics to easily add more functionality to the interface.

In summary this work is done to design and implement a framework which allows the management of Enterprise Topology Graphs together with their segmentations, transformation operations, and analysis strategies.

1.3 Outline

This document describes the plan, the design and the implementation of a pluggable framework for the storage and modification of Enterprise Topology Graphs.

First, the fundamentals of the Enterprise Topology Graphs will be discussed. The fundamentals contain a short summary of the usage of cloud computing in enterprises in Section 2.1, the *TOSCA* standard in Section 2.2 and the Enterprise Topology Graph definition in Section 2.3. With the fundamentals provided in these Sections, the gathered requirements for the ETG framework will be presented in Section 2.4.

After the fundamentals Chapter, the basic architecture of the framework will be presented. A short overview of the global architecture is given in 3.1. 3.2 includes a more precise view on the backend architecture. The plug-in management architecture will be introduced in Section 3.3.

The following Chapter 4 provides the foundations and use cases for the framework implementation.

The design decisions for the framework implementation will be discussed in the Design Decisions Chapter 5. All challenges for the implementation will be explored and analyzed to provide the most suitable design.

The Chapter 6 describes the implementation. After a short overview, the main components and their implementations will be discussed.

A summary of the work and an outlook is presented in the Chapter 7.

Chapter 2

Fundamentals and State of the Art

In this Chapter the fundamental technology is discussed. It provides the necessary knowledge to understand the meaning and the context of Enterprise Topology Graphs and this work. First this Chapter delivers a short insight into existing cloud computing definitions and standards and shows their usage in enterprises. Afterwards, the TOSCA standard is discussed in detail, because this standard delivers the pre-conditions for the definition of Enterprise Topology Graphs. The Section 2.3 deals with the definition of the Enterprise Topology Graphs. With the information given in these sections, the requirements for the framework are presented in Section 2.4.

2.1 Cloud Computing in Enterprises

Cloud Computing is a widely used term, but what is the standardized definition of it? Wikipedia delivers the following definition:

Cloud computing is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet).[1]

This declaration is right, but does not provide a clear definition how the cloud looks like. A more detailed definition, even with a reference architecture for a cloud system, can be looked up in the NIST Cloud Computing definition [MG09]. In this definition cloud computing is characterized through five main criteria:

- **On-demand self service**

  Computing capabilities like network storage or application functionality can be queried by a customer as needed.

[1]Wikipedia - http://en.wikipedia.org/wiki/Cloud_computing

- **Broad network access**

  The computing capabilities are available within a network (in most cases the Internet) and can be accessed through standard mechanisms.

- **Resource pooling**

  Resources like storage, processing, memory, and network bandwidth are pooled by a provider and the resources a customer uses are well separated from the other resources.

- **Rapid elasticity**

  To provide large scalability, new resources have to be provisioned in a fast and automated way.

- **Measured service**

  The usage of the resources can be controlled, monitored and reported. This allows a usage-based payment bill creation.

In summary, with Cloud Computing it is possible to gain access to computing resources like processing power, storage, middleware systems, fully installed servers or even application functionality on-demand. The costs of these resources are calculated with usage statistics and the providers mostly guarantee high-availability and robustness.

This business model seems to be well suited for companies, to lower their costs of IT infrastructure and application licenses. Actual a lot of enterprises are using cloud offerings nowadays. Of course it is great to use on-demand computing resources, but in the current Software as a Service offerings the service is often bounded to a cloud provider. This leads to really messy infrastructure landscapes, because different cloud offerings with different providers are used all together in an enterprise.

To clean the mess in cloud-aware infrastructures, a lot of research is going on. Different standards in the domain of cloud computing are getting developed. In this work one standard gets really important, *TOSCA*. *TOSCA* provides a standard to decouple cloud services from their hosting cloud. This standard is described in the next Section (2.2: TOSCA).

2.2 TOSCA

To push forward the usage of cloud services a project named *CloudCycle* was started to provide a secure and compliant way to administer and provide portable cloud services. The project has the goal to cover the whole life-cycle of a service.

The idea for this project is raised due to the fundamental problems which come with cloud computing like provider lock-in, no standards, rudimentary interfaces, no compliance, no certifications, and the little acceptance of users. A new cloud standard called *Topology and Orchestration Specification for Cloud Applications (TOSCA)* enables the support for portable cloud services to avoid a provider lock-in. *CloudCycle* uses this standard to implement the first OpenSource container for portable cloud services.

A *TOSCA Archive* includes a service *Topology*, a plan for the infrastructure needed by the service, the *software* which should be deployed according to the *Topology* and *Management plans* to describe e.g. the deploying, the update workflow or the undeployment of the service. With these artifacts it is possible to describe a service in a very abstract way, for example by not defining the actual application server to be used. On the other hand it is possible to describe a service in a very detailed way, e.g. by including all configuration detail for specific application components.

The newest research result, *Enterprise Topology Graphs*, graphs based on the *Topology* definition of *TOSCA* to describe the whole infrastructure of an enterprise, is the foundation of this work. This definition is described in the following Section (2.3: Enterprise Topology Graphs).

2.3 Enterprise Topology Graphs

The already introduced *TOSCA* standard provides a mechanism to describe topologies for cloud services. This seems to be a great foundation for describing the hardware and software infrastructure of an enterprise, formally defined in [BFL+12]. This definition defines a graph as a set of entities with properties. These entities can be divided into nodes and edges. To represent the IT components of an enterprise, a set of node types and a set of edge types can be created. The types include all node and relationship types, e.g. **Application Server**, **Virtual Machine** or **is hosted on** (edge type). Each type has a set of defined properties, so an application server could have a property **port** or even **vendor**. This type system is very similar to the *TOSCA* type system in the topologies using types and type templates. In addition, a segment of an *ETG* is defined. A segment describes a sub-graph, a set of nodes and edges, which are connected. The mechanism of segments is provided to execute sub-graph matching searches.

With this formal definition it is possible to execute graph transformations. To demonstrate the need of transformation capabilities, the following example is given:

A new update is being rolled out for an application server. This update is very important, because it closes a huge security leak in the server. The update process is very complicated, because of the hybrid nature of the enterprises infrastructure. Some servers are running in their own datacenter on real hardware, some on virtualized machines and other ones in the cloud. Now the *ETGs* come into play. A sub-graph matching search can be executed to find all services, which are deployed on the out-dated application server version [BLNSpt]. Now the graph can be manipulated with an automatic transformation to calculate the most preferable way to organize the services in order to avoid dependencies to an out-dated application server. After the re-organization, the application servers can be updated without effecting the execution of the services. After the update process the graph transformation can be reversed.

The transformation capabilities of *ETGs* provide a powerful way to manage a service landscape. This work creates an environment to save and manipulate *ETGs*. In addition, a plug-in mechanism is provided to offer the possibilities to implement different operations on the graphs.

2.4 Framework Requirements

With the *Enterprise Topology Graph* definition in Section 2.3 and the *Problem Statement* in 1.2 the requirements for a managing framework can be formulated. The following requirements should be met by an implementation of the framework and can be used for system tests.

(r1) Saving Enterprise Topology Graphs - The framework should have the ability to save graphs with up to 1 million nodes. Each node can have multiple properties and edges to other nodes. The framework has to be able to store the graphs and provide a way to access the graphs to modify the meta information.

(r2) Entity Modification - The entities of a graph stored with the framework have to be modifyable. The framework must provide an API to modify the properties and meta information of an entity.

(r3) Versioning - Because of the high complexity of graph transformations, the framework should offer a possibility to introduce different versions of an Enterprise Topology Graph. It has to be possible, to create backup copies of existing graphs to be able to make changes undone.

(r4) Type Set Management - The types of an Enterprise Topology Graph have to be stored, too. It has to be possible to add, remove and edit the types used by the entities in a graph.

(r5) Segment Export and Sub-Graph Matching - To use segments of an Enterprise Topology Graph as input for transformations, the framework must be capable to export a set of entities of a graph as segment. The segment export should be possible with sub-graph matching. Therefore, search and matching functionality has to be provided by the framework.

(r6) Plug-In Architecture - The framework has to provide a pluggable architecture to provide a way to extend the functionality with plug-ins.

(r7) Graphical User Interface - To browse the data stored by the framework, a graphical user interface is needed. This user interface should be extendable by the plug-ins.

These requirements have to be met by the frameworks implementation.

CHAPTER 3

FRAMEWORK ARCHITECTURE

This Chapter contains a first sketch of a component based architecture for the framework. This architecture is derived from the requirements described in Section 1.2.

First an Overview (3.1) of the main components is given. This should deliver a first insight into the overall structure. A layer structure of the framework is given in the Section 3.2. This layered architectural overview gives an insight into the encapsulation of the data. After the detailed architecture of the framework, a more abstract view of the whole system is given. Section 3.3 shows a service oriented architecture for involving plug-ins.

3.1 Architectural Overview

This Section shows the framework with a coarse grained view on to the components. This architectural overview can be built with the requirements in 1.2. The components are shown in Figure 3.1.

The architecture is divided into three main components:

- The **DataService**
- The **Plug-Ins**
- The **WebFrontend**

The **DataService** is the component, which is is charge of the data access. Consequently this component is the heart of the framework. It is responsible for all operations on a central data store, like inserting, exporting and modifying the Enterprise Topology Graphs. All other components retrieve their data from the DataService. The DataService also provides a wrapper for the actual data store, so that the WebFrontent and the Plug-Ins don't have any dependencies on a specific data store implementation.

The **Plug-Ins** provide extensions for the functionality of the DataService. They can provide algorithms, transformations or simple search functionality for Enterprise Topology Graphs and use the data and functionality of the DataService. The Plug-Ins not only provide new

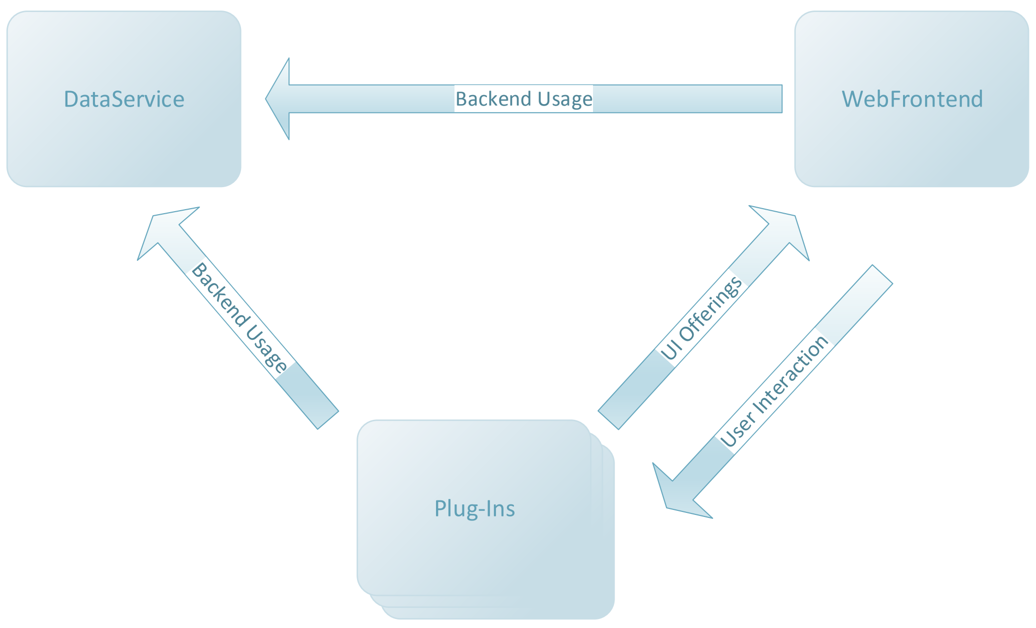


Figure 3.1: Architectural Overview

functionality, but also their own user interface parts for interaction with the framework user. These Plug-Ins will be included into the framework using a flexible plug-in system.

The **WebFrontend** provides a graphical user interface for the framework. This frontent makes it possible to browse through the content of the DataService and to trigger its functionality. It facilitates the work with the framework and offers a good entry point. The WebFrontend also features a simple way to extend the views with plug-in functionality. So the Plug-Ins can implement their own GUI parts and implant them into the frontend.

3.2 Layered Architecture

This Section gives more insight into the treatment of the graph data. Section 3.1 has given a coarse grained overview over the main components, this Section will give more information on how the data is processed. Therefore, the DataService is split into three layers. These layers are shown in the Figure 3.2. Each layer will be described in detail, starting from the topmost layer.

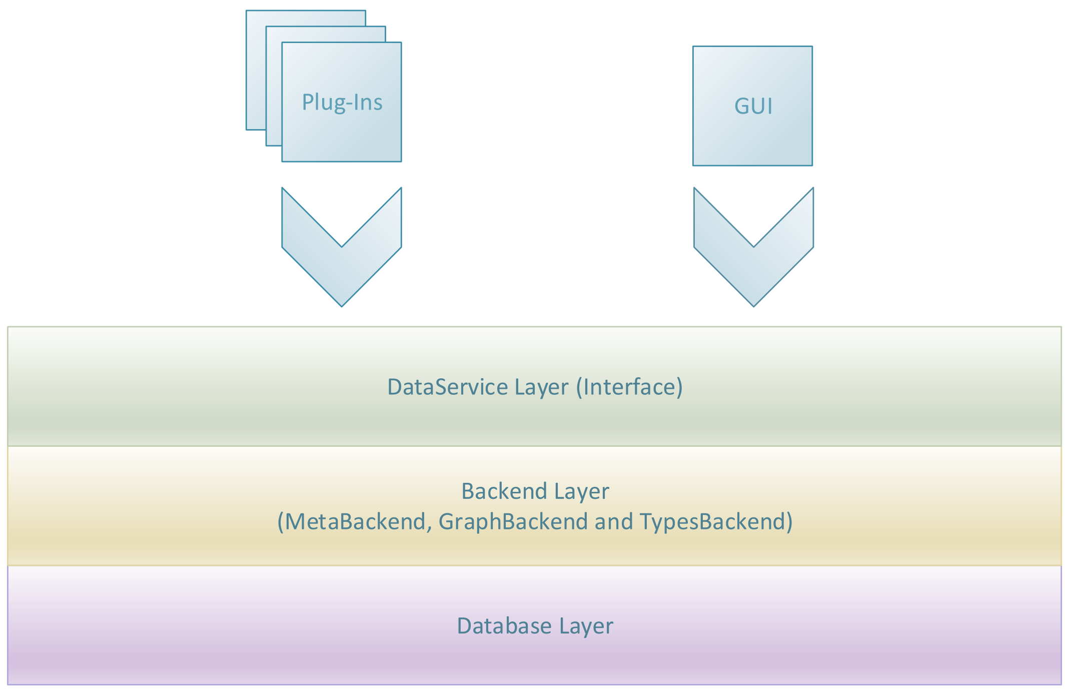

Plug-Ins
GUI
DataService Layer (Interface)
Backend Layer
(MetaBackend, GraphBackend and TypesBackend)
Database Layer

Figure 3.2: Layered View of the Architecture

The first layer is named **DataService Layer**. In this layer the main operations of the framework like *storeETG*, *getETG* or *getETGStructure* are offered. These operations directly handle the ETG graph model as described in Section 2.3. A more detailed view onto the data model will be given in Section 5.2.2. The DataService Interface is the entry point for all participants in the framework like the WebFrontend and the Plug-Ins and is therefore defined as interface. It processes the graphs and uses the underlying layer to store the different information a graph contains.

The second layer is called **Backend Layer**. It is used by the Dataservice Layer and provides three backends: The *MetaBackend*, the *GraphBackend* and the *TypesBackend*. Each backend directly communicates with the Database Layer and offers a specific set of operations in an own context.

The **MetaBackend** operates in a global context and stores all types of meta information, like the creation date of a graph, or even different versions and sub-graph information. It is responsible for a proper Id creation and the storage structure in the database.

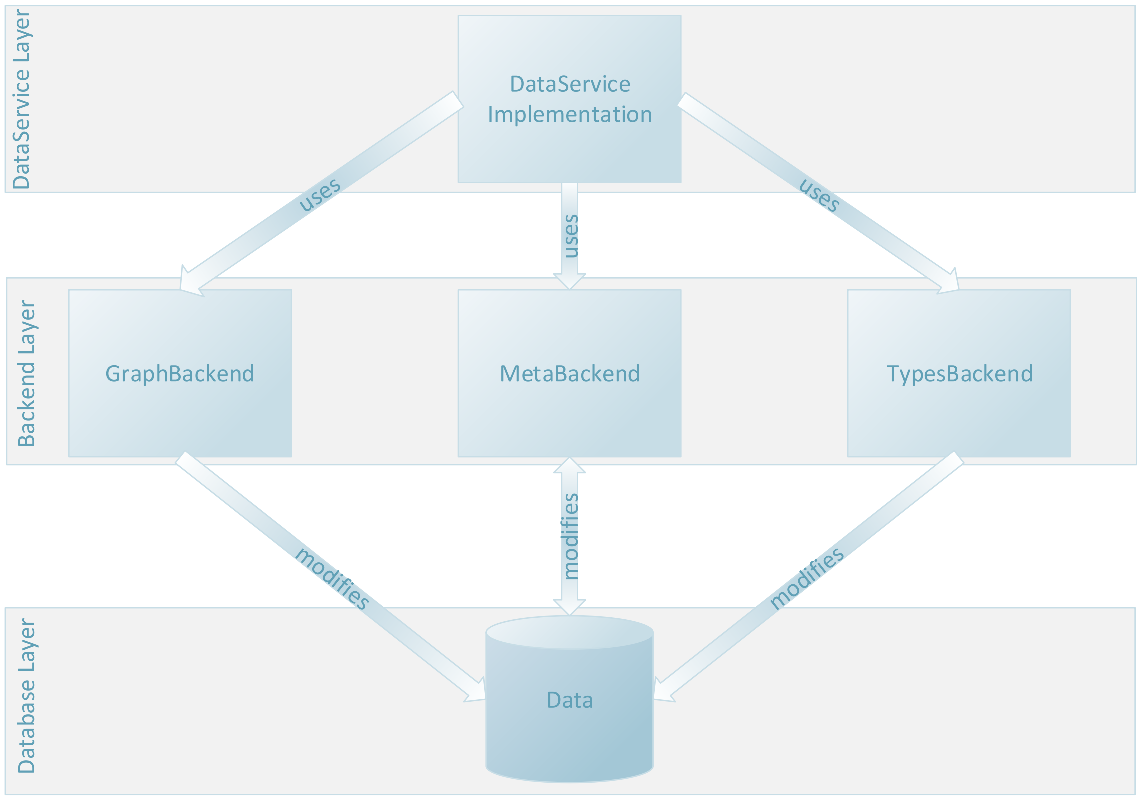


Figure 3.3: Backend Architecture

The **GraphBackend** operates in the context of a specific graph and handles the modifications. With this backend it is possible to insert, modify and delete nodes and edges and their properties.

The **TypesBackend** also operates in the context of the meta information's type branch of a specific graph. It only modifies the type set of an Enterprise Topology Graph. With the TypesBackend it is possible to insert, modify and delete types of a graph.

The third and last layer is called **Database Layer**. This layer heavily depends on a database solution and the interfaces for interaction with it. The layer translates the information into the language of the database solution and stores the data persistently.

A more fine grained view onto the different layers is given in Figure 3.3.

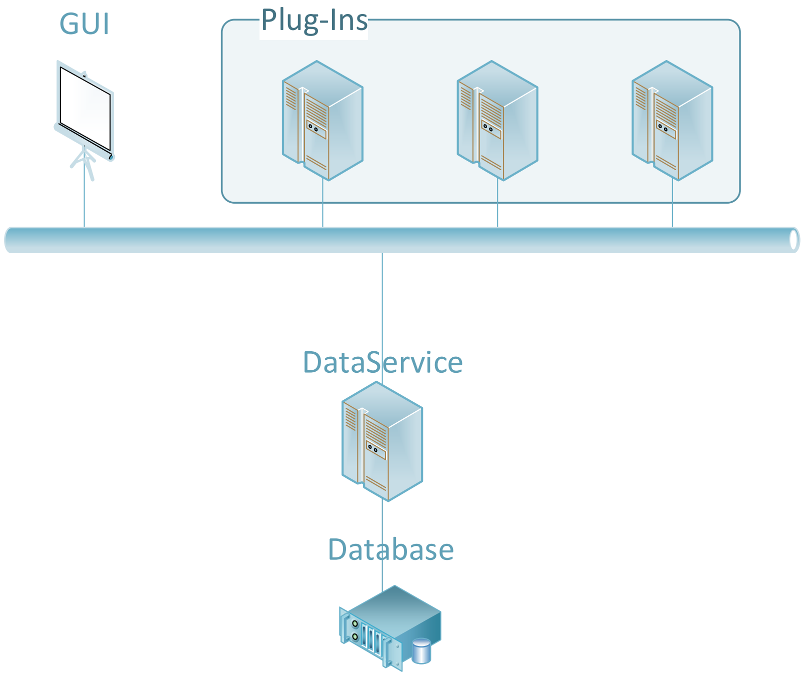


Figure 3.4: Dynamic Plug-In Communication Structure

3.3 Plug-Ins

After the architecture of the DataService and the Backends, this Section presents a more abstract view of the framework. The focus is on the plug-ins.

Figure 3.4 shows a very dynamic structure. This illustration shows, that not only the DataService can be used by the GUI and the Plug-Ins, but also the Plug-Ins themselves. With this architecture it is possible to create and deploy Plug-Ins, which have a dependency onto the DataService and other Plug-Ins.

To give an example: A Plug-In offers a search algorithm for sub-graph matching. This Plug-In can be used by another Plug-In, which searches all outdated application servers of a specific vendor with their hosted services and performs calculates a plan to host all services on servers, which are up-to-date. The outdated servers now can be updated without an impact on the services.

If the Plug-Ins couldn't use the functionality of other Plug-Ins, the search algorithm had to be implemented twice. This would be more effort and, of course, more code to administer.

Chapter 4

Framework Use Cases

Related to the main goal and the resulting architecture, there are three main parts of the framework:

- **DataService** - Save the Enterprise Topology Graphs and their meta information.
- **DataModel** - Represent the Enterprise Topology Graphs and provide a way to modify them.
- **Plug-In Architecture** - Provide a way to extend the framework.

The **Graphical User Interface** enables browsing and manipulating the stored Enterprise Topology Graphs. This component will not be covered by use cases. For more information about the **Graphical User Interface**, the Section 6.3 in the Implementation Chapter can be consulted.

4.1 DataService

The *DataService* is the core component of the ETG framework. It provides a way to store Enterprise Topology Graphs, administrates the meta information and implements all interfaces for ETG manipulation. Because of that it is the interface to a database solution.

The *DataService* has to provide a way to import, export and create Enterprise Topology Graphs and their segments.

4.1.1 Actors and Usage

The *DataService* builds the backend of the framework. It is used to trigger core functionality by clients like a graphical user interface, the plug-ins and other pieces of software. This clients are used as actor in the following use cases.

4.1.2 Use Cases

Import an ETG

Name	Import an ETG
Goal	Import an ETG to the database.
Actor	Client or Plug-In
Pre-condition	The database is running.
Post-condition	A new project is created with an initial graph snapshot and the id of the new project is passed to the actor.
Post-condition in Special Case	No project and no graph is created and *null* is returned.
Normal Case	The client calls the `importETG` method and passes a segment or an ETG, as well as the used set of types as parameter. The contained graph is used as initial snapshot and the set of types is stored within the project.
Special Cases	1. There is not enough remaining space to store the imported ETG.

Table 4.1: Import an ETG

Create an ETG

Name	Create an ETG
Goal	Create a new ETG in the database.
Actor	Client or Plug-In
Pre-condition	The database is running.
Post-condition	A new project is created with an initial empty graph snapshot and the id of the new project is passed to the actor.
Post-condition in Special Case	No project is created and *null* is being returned.
Normal Case	The client calls the `createETG` method and passes the meta information for a project as parameter.
Special Cases	1. There is not enough remaining space to store create the new ETG.

Table 4.2: Create an ETG

Retrieve a List of All ETGs

Name	Retrieve a list of all ETGs
Goal	Get a list of all ETGs saved in the database.
Actor	Client or Plug-In
Pre-condition	The database is running.
Post-condition	A Java list containing the meta information of all ETGs saved in the database is returned.
Post-condition in Special Case	An exception is thrown.
Normal Case	The client calls the `retrieveAllETGs` method without any parameter.
Special Cases	1. There is not enough remaining space to store create the new ETG.

Table 4.3: Retrieve a List of All ETGs

Get a Segment

Name	Get a segment
Goal	Retrieve a segment from a specified ETG as Java object.
Actor	Client or Plug-In
Pre-condition	The database is running.
Post-condition	A Java object of the type *Segment* containing the specified nodes and edges is returned.
Post-condition in Special Case	An exception is thrown.
Normal Case	The client calls the `getSegment` method and passes an id of a stored snapshot or segment, as well as a list of node ids and a list of edge ids as parameter.
Special Cases	1. The ETG can not be found in the database. 2. A node or an edge can not be found in the specified ETG. 3. There is not enough space in the memory to create the *Segment* Java object.

Table 4.4: Get a Segment

Save a Segment

Name	Save a Segment
Goal	Save a segment linked to the source snapshot of the source ETG.
Actor	Client or Plug-In
Pre-condition	The database is running.
Post-condition	The segment is saved as independent graph like an ETG snapshot.
Post-condition in Special Case	The segment is not saved and an exception is thrown.
Normal Case	The client calls the `saveSegment` method and passes the segment as parameter. The source ETG and the source snapshot are found in the database.
Special Cases	1. The ETG can not be found in the database. 2. There is not enough space in the memory to store the segment.

Table 4.5: Save a Segment

Add Node

Name	Add node
Goal	Add a new node to a stored graph.
Actor	Client or Plug-In
Post-condition	The new node is inserted into the graph.
Post-condition in Special Case	The new node is not inserted into the graph.
Normal Case	The `addNewNode` method with a *Node* object as parameter is called. The node is inserted in the node list of the graph.
Special Cases	1. The graph can not be found in the databse.

Table 4.6: Add Node

Add Edge

Name	Add edge
Goal	Add a new edge to a stored graph.
Actor	Client or Plug-In
Pre-condition	The source and the target node is already stored in the database.
Post-condition	The new edge is inserted into the graph.
Post-condition in Special Case	The new edge is not inserted in the graph.
Normal Case	The `addNewEdge` method with an *Edge* object as parameter is called. The edge is inserted in the edge list of the graph.
Special Cases	1. The graph can not be found in the database.

Table 4.7: Add Edge

Modify Node Properties

Name	Modify Node Properties
Goal	Modify the properties of a node.
Actor	Client or Plug-In
Pre-condition	The parent graph of the node is stored in the database.
Post-condition	The properties of the node have been updated.
Post-condition in Special Case	The properties of the node are not modified.
Normal Case	The `modifyNode` method with a list of *Property* objects and a node id as parameter is called. The graph contains the targeted node. The existing properties are updated with the new values and new properties are added.
Special Cases	1. The targeted node is not in the node list of the graph.

Table 4.8: Modify Node Properties

Modify Edge Properties

Name	Modify Edge Properties
Goal	Modify the properties of an edge.
Actor	Client or Plug-In
Pre-condition	The parent graph of the edge is stored in the database.
Post-condition	The properties of the edge have been updated.
Post-condition in Special Case	The properties of the edge are not modified and *false* is being returned.
Normal Case	The `modifyEdge` method with a list of *Property* objects and an edge id as parameter is called. The graph contains the targeted edge. The existing properties are updated with the new values and new properties are added.
Special Cases	1. The targeted edge is not in the node list of the graph.

Table 4.9: Modify Edge Properties

Delete Node

Name	Delete node
Goal	Delete a node.
Actor	Client or Plug-In
Pre-condition	The parent graph of the node is stored in the database.
Post-condition	The node is deleted in the node list of the graph.
Post-condition in Special Case	An exception is thrown.
Normal Case	The `deleteNode` method with a node id as parameter is called. The graph contains the targeted node. The node is dropped in the node list of the graph.
Special Cases	1. The targeted node is not in the node list of the graph.

Table 4.10: Delete Node

Delete Edge

Name	Delete edge
Goal	Delete an edge.
Actor	Client or Plug-In
Pre-condition	The parent graph of the edge is stored in the database.
Post-condition	The edge is deleted in the edge list of the graph.
Post-condition in Special Case	An exception is thrown.
Normal Case	The `deleteEdge` method with an edge id as parameter is called. The graph contains the targeted edge. The edge is dropped in the edge list of the graph.
Special Cases	1. The targeted edge is not in the edge list of the graph.

Table 4.11: Delete Edge

4.2 Plug-In System

The *Plug-In System* delivers a dynamic way to add more functionality to the framework. The *Plug-In System* should manage the load and shutdown process of the plug-ins.

4.2.1 Actors and Usage

The *Plug-In System* is used by the framework to discover the loaded plug-ins and by the plug-ins itself to register themselves.

4.2.2 Use Cases

Load Plug-In

Name	Load Plug-In
Goal	Load a plug-in and integrate the UI parts in the frameworks interface.
Pre-condition	The *.jar* file containing the plug-ins UI parts has been put into the plug-ins folder of the framework.
Post-condition	The functionality of the plug-in can be used through the frameworks interfaces.
Post-condition in Special Case	The plug-in functionality is not accessible.
Normal Case	The plug-in calls the `loadPlugin` method and its UI parts are inserted into the frameworks UI.

Table 4.12: Load Plug-In

Shutdown Plug-In

Name	Shutdown Plug-In
Goal	Shutdown a plug-in and remove the UI parts.
Post-condition	The functionality of the plug-in can not be longer accessed and the UI parts are removed.
Normal Case	The plug-in calls the `shutdownPlugin` method and is shutdown and removed in the graphical user interface.
Special Case	1. The plug-in is not loaded in the framework.

Table 4.13: Shutdown Plug-In

CHAPTER 5

DESIGN DECISIONS

In this Chapter the challenges and the resulting decisions during the development are presented.

5.1 DataService

The *DataService* is the backend of the framework and provides access to a database where the Enterprise Topology Graphs are saved and managed. Therefore, it must provide a good performing and scalable graph storage.

5.1.1 Challenges

The following challenges will be discussed in the design section of the *DataService*:

(i) The *DataService* is the central component of the framework. Plug-Ins and the graphical user interface have to retrieve and store data in the Enterprise Topology Graph store. Because of that, it has to be accessible by the different components of the framework. This challenge is addressed in the Section 5.1.2.

(ii) The second key question with respect to the *DataService* is which database solution to use. Not only a huge number of different database vendors offer their solutions, but also even mainly different database models exist. The challenge is to discover the right model and database solution to store and manage huge Enterprise Topology Graphs. The Section 5.1.3 addresses this challenge and provides an evaluation of different database solutions.

5.1.2 Accessibility

To provide the needed accessibility of the *DataService* some architectural decisions have to be made. In matters of loose coupling, a service like implementation of the *DataService* has to be considered. The functionality of the *DataService* has to be expressed through public interfaces in a common used library. The interfaces offer a implementation and database independent way to store, retrieve and modify the Enterprise Topology Graphs and their nodes and edges. With a loosely coupled service the claimed accessibility in challenge 5.1.1 **(i)** is addressed.

5.1.3 Database Solution

There are a lot of different database solutions on the market. Mainly there are two different groups of databases: Relational databases and Object- or Document-based databases, also called NoSQL databases. Because of the connected pieces of data, as which an Enterprise Topology Graph can be represented, current research in the area of social networking can be adapted. Social networking and their appropriate algorithm theory makes heavy use of modern NoSQL databases, e.g. *Facebook*[1] uses an *Apache Hadoop*[2] database for their site activity analytics and messages. Because of the strong analogy of discovery and transformation techniques in social network graphs and Enterprise Topology Graphs a quite similar storage solution can be considered. Therefore, a NoSQL graph-based database seems to be well suited for the requirements in this project. Also in the area of NoSQL databases there are a lot of very different solutions. According to this diversity of offerings only some popular representatives are discussed as possible solution in this work. The selection of this databases in particular is based on a research conducted as part of this thesis to the topic of graph storage systems, especially a paper, which takes a closer look to large-scale linked data processing in terms of an RDF-graph[3][HGHCM12].

HBase

Apache HBase[4] is a very popular database based on Hadoop[5] and HDFS[6] [SKRC10]. It is modeled after Google's BigTable approach and is well suited for a large number of random read/write access calls on very large structured data [HMP06]. This database solution is open-source, but offers high availability features like scalability through distribution and versioning [KKVR12].

[1]Facebook - https://www.facebook.com/
[2]Apache Hadoop - http://hadoop.apache.org/
[3]Resource Description Framework - http://www.w3.org/RDF/
[4]HBase - http://hbase.apache.org/
[5]Apache Hadoop - http://hadoop.apache.org/
[6]Hadoop Distribuited File System - http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html

A conceptual view on Apache HBase's data model shows the relation to googles BigTable approach [CDG$^+$08]. The data is stored in *tables*, which are made of columns and rows. All columns are part of a *column family* and the content stored in table cells is represented by a byte array. The content can be accessed by their row keys, also a byte array. Physically the content is stored in column family based files. The access information for each column family is stored as *column family anchor* in an extra file. The files are stored in the underlying HDFS and categorized in regions to store the tables in a distributed way. For a closer look at HBases architecture, the *HBase Reference Guide*[7] can be consulted.

CouchDB

Apache CouchDB[8] provides a JSON[9] based, distributed database which can be accessed through a regular HTTP API. This solution stores the data as a set of JSON documents. This solution is well suited to implement document management systems in large enterprises [Qia10]. The approach using JSON documents seems not to fit for the requirement of a graph database, because of a non-existent representation of relationships. But some add-ons are available to extend the functionality with support for RDF graphs or JSON-LD[10].

CouchDB uses a totally different concept as HBase. CouchDB uses a B+ tree as internal structure. A B+ tree is a binary tree with an optimization for the performance of access operations through reduction of the tree depth. The content documents are only stored in the leaves of the tree. Because of that, new documents only have to be appended to the B+ trees leaves. This concept improves the speed of operations, but increases the space on the data storage medium. CouchDB stores one database in one B+ tree database file. For more information about the B+ tree and the modifications CouchDB uses in its implementation the *B-tree Section in the CouchDB Guide in the Draft Edition*[11] can be consulted.

HyperGraphDB

HyperGraphDB[12] is a open-source database solution for a knowledge management formalism known as *directed hypergraphs*. This database solution offers a graph based storage and a out-of-the-box embedded object store for Java. The database stores graphs based on different type sets and uses these as a database schema. Also this solution offers a model for n-ary relationships to support a multi-dimensional hypergraph [HPT11].

[7]HBase Reference Guide - http://hbase.apache.org/book/book.html
[8]CouchDB - http://couchdb.apache.org/
[9]JavaScript Object Notation - http://www.json.org/
[10]JavaScript Object Notation for Linked Data - http://json-ld.org/
[11]CouchDB Guide, B-Tree Section - http://guide.couchdb.org/draft/btree.html
[12]HyperGraphDB - http://www.hypergraphdb.org/

Another graph database concept can be spotted in the HyperGraphDB architecture. HaperGraphDB uses an underlying Key-Value store (Current solution: Oracle BerkeleyDB[13]), to store the graphs structure and its content. The model is separated into two layers: The *Primitive Layer* and the *Model Layer*. In the Primitive Layer consists of only two associative arrays, a link store and a data store. The link store only contains identifiers, the data store only contains raw data. The Model Layer contains so called *hypergraph atoms*. An atom can represent a type, a value or a target. With this separation all atoms can be directly stored as byte array or as identity tuple. The whole information can be read in the paper [Ior10].

Neo4j

Neo4j[14] is a NoSQL graph based database solution. This solution is the most popular property graph database implementation with native graph support [SWX12]. It provides three different editions and includes high performance and enterprise support[15]. High availability, performance and even ACID transactions are offered by Neo4j [VMZ$^+$10].

In the internals of Neo4j the graphs are broken down to a set of linked lists of fixed size records. Therefore, a set of files is created: A node store file, a relationship store file and a relationship types file and a bunch of property store files. Each node is stored in the node store file with a reference to their first property record and a reference to their first relationship record. The relationships are stored in the relationship store file with references to their start and end node and references to the next and previous relationship regarding to their start and end node. All properties are stored in the property store files. The main property store file realizes a key value store. Besides the key value store, Neo4j uses files for long strings and long arrays. To offer a good performance, Neo4j uses different methods to hold parts of the files in the cache. For more information the presentation *Neo4j Internals*[16] can be consulted.

5.1.4 Evaluation

To choose the best suited solution for the framework implementation some requirements have to be set up.

First, the frameworks focus is the *Enterprise Topology Graph* storage. The framework has to be able to store very large graphs. The solution should be scalable and good performing on huge data sets. This is reached through mechanisms to provide scalability and a native graph support.

[13]Oracle BerkeleyDB - http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html
[14]Neo4j - http://www.neo4j.org/
[15]Neo4j Licnsing - http://www.neo4j.org/learn/licensing
[16]Neo4j Internals Presentation - http://de.slideshare.net/thobe/an-overview-of-neo4j-internals

Second, the database solution should provide a Java API to interact with. Also the solution should provide an API to interact with a standalone server. This could be a API based on REST, a Web Service or some kind of socket communication.

Third, the framework should be easy to deploy and scalable. Because of that, the database should offer an embedded server solution and a standalone version. So the framework can be deployed all together on a server without installing an additional database or it can be installed in a network with a dedicated database server.

Fourth and last, the APIs of the database solution have to be well documented. The framework is the foundation for plug-ins and other software according to Enterprise Topology Graphs. Because of that, maintainer and developer should be able to gather good information about the database quickly. A big and active community, many examples and of course a professional documentation about the database solution should be available.

This requirements lead to an evaluation table for the database solution show in Table 5.1.

	Scalability	Graphs	Java API	Embedded/ Standalone	Standalone API	Docs
HBase	yes	no	yes	no	yes	yes
CouchDB	yes	no	yes	no	yes	yes
HyperGraphDB	yes	yes	yes	no	no	yes
Neo4j	yes	yes	yes	yes	yes	yes

Table 5.1: Evaluation of the database solutions

The headers of the Table are shortened. The criteria are: Mechanisms for scalability, native graph support, a Java API offering, the availability of an embedded and standalone mode, a communication API for the standalone mode and the availability of an active community, examples and a professional documentation.

HBase and CouchDB do not use native graphs for their database model. It is some work to represent graphs in the solutions.

The solutions HBase and CouchDB do not offer an embedded mode for Java applications. HyperGraphDB only offers an embedded mode and cannot be used as standalone server. An additional peace of software should have been build to offer the installation of a dedicated database server with HyperGraphDB.

According to the Table Neo4j is most suited for the implementation of the ETG framework. In addition, Neo4j offers a very easy to use different, very powerful query APIs, like *Gremlin*[17]

[17]Gremlin - https://github.com/tinkerpop/gremlin/wiki

or *Cypher*[18]. With Neo4j as database solution in the backend, the challenge 5.1.1 **(ii)** is addressed.

5.2 DataModel

After the formal definition of the Enterprise Topology Graphs and the design of the framework backend, the underlying data structure has to be defined. This Section shows all design decisions with regard to these structures and their challenges.

5.2.1 Challenges

The following challenges will be discussed in the design section of the *DataModel*:

(i) The *DataModel* basically has to provide a data structure to process the data of an Enterprise Topology Graph in the DataService. This data structure must represent an ETG with all properties defined in the formal definition given in [BFL$^+$12]. The concept of the data structure is presented in the Section 5.2.2.

(ii) In addition to the information contained in an Enterprise Topology Graph, meta information has to be stored in the database, too. This meta information e.g. consists of a database id, a creation or import timestamp, the number of nodes and edges or a version of the graph. Because of the high importance of the data in ETGs, an incorrect modification could be very costly and irreparable. To prevent the loss of data, the framework has to introduce a versioning management for Enterprise Topology Graphs. To realize this requirement, the *DataModel* has to be extended with a meta information layer. The description and storage of this layer turns out to be the second challenge regarding to the *DataModel* and is addressed in Section 5.2.3.

(iii) Concerning the possible huge size of Enterprise Topology Graphs, problems with oversized objects are bound to occur. To offer a convenient way to process graph information in the graphical user interface or plug-ins, a solution for splitting the enormous amount of data has to be found. An established mechanism in the area of database management is pagination. The result of a database query is divided into multiple sets with a defined size. The third design challenge regarding to the *DataModel* is to conceptualize a paging mechanism for graphs. This challenge is addressed by the Section 5.2.4.

5.2.2 Enterprise Topology Graph Representation

The representation of Enterprise Topology Graphs is strongly geared to the model in the formal definition described in 2.3: Enterprise Topology Graphs. To visualize the concept, Figure 5.1 will be described in detail.

[18]Cypher Query Language - http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html

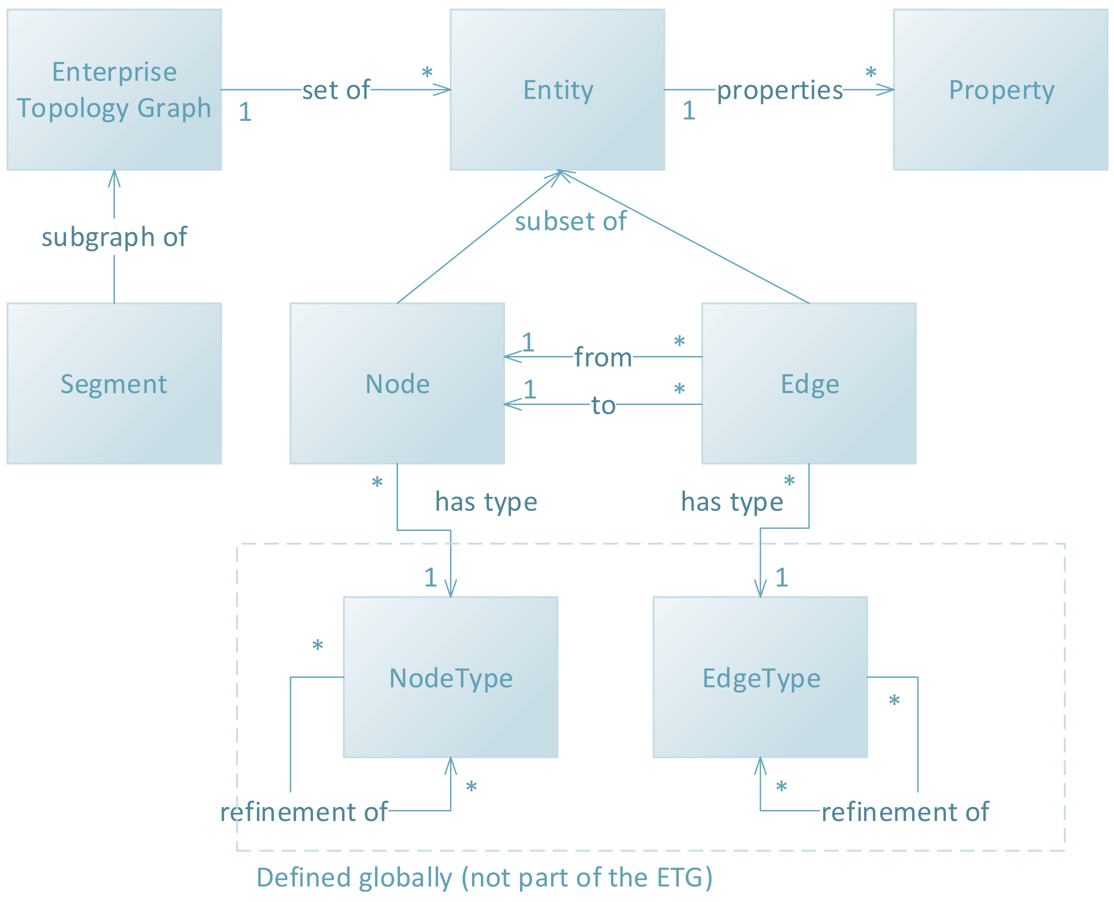


Figure 5.1: Conceptual ETG Model (Adapted from: [BFL+12])

Because of the possibility of different implementations the model will be realized as a set of interfaces. This framework offers a default implementation with a solution for pagination (5.2.4: Custom Iterator (Pagination)). This features may not be needed if an Enterprise Topology Graph is built without using the framework. Therefore, a new implementation could be added.

5.2.3 Meta Information Representation

The framework adds a version control system to the formal ETG definition. To add these features an additional model has to be defined and implemented.

The top level elements are called **ETGProject**. These elements are holding all meta information like the name or the description. An *ETGProject* also references a set of types. All properties of an Enterprise Topology Graph are stored in the ETGProject.

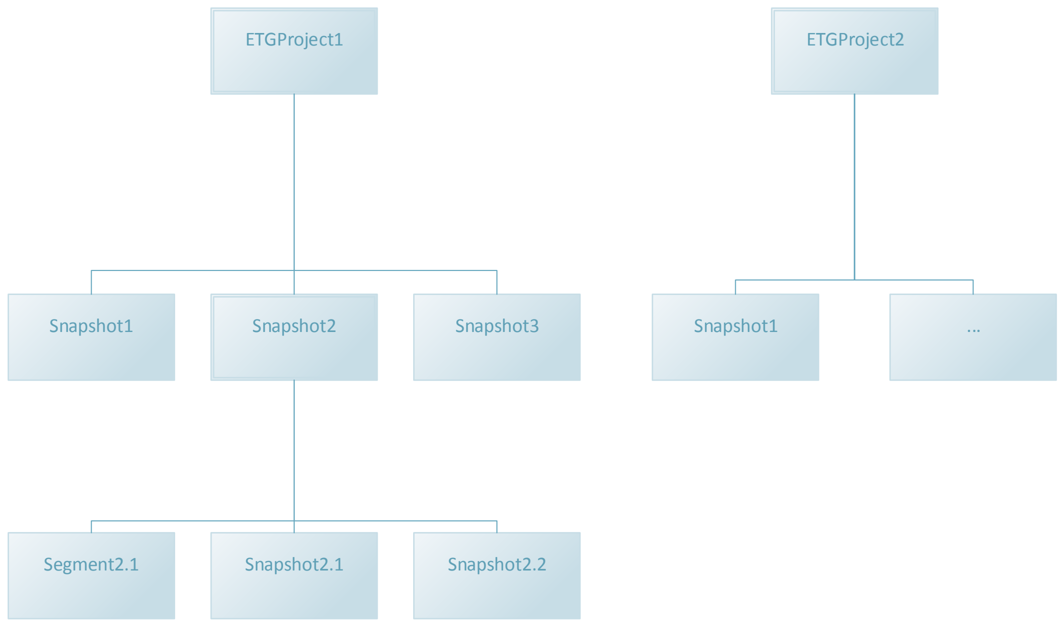


Figure 5.2: Example Project Structure

Snapshots represent different versions of an ETG. A *snapshot* contains all nodes and edges of the ETG in this specific version. By nesting *snapshots*, it is possible to create new versions in an hierarchical way. *Snapshot* include additionally meta information, like a database id, a creation date and the number of nodes and edges.

The **segments** are sub-graphs of an ETG. These *segments* consist of a (sub-)set of nodes and edges of a snapshot or another segment.

Snapshots and *Segments* are obviously the same, except from some different properties. Because of that, both can be represented in the framework by a type called **GraphMetaInformation**. This type can be nested and contains properties with an unique identifier.

To save all meta data in the database with the version relationships, a additional meta-graph is created for each *ETGProject*. The differentiation between *snapshots* and *segments* is solved with different relationship types. An example meta-graph is showed in Figure 5.3.

For an example, the import of an *EnterpriseTopologyGraph* into the framework in a detailed view:

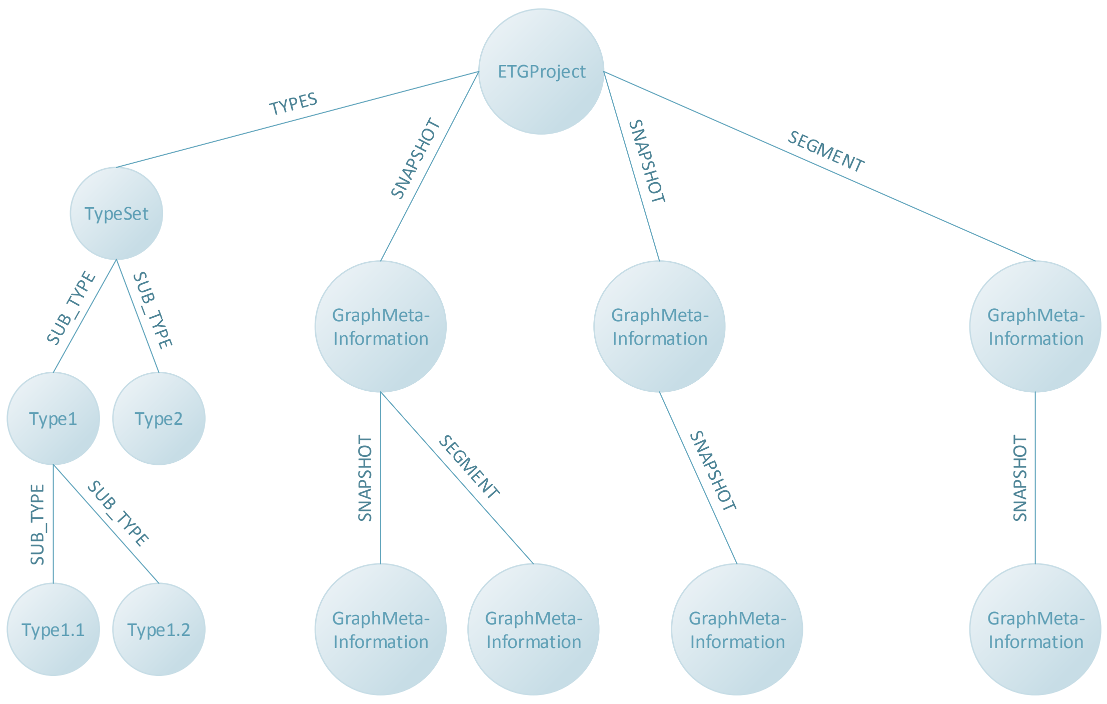


Figure 5.3: Example Meta-Graph

1. A new *ETGProject* is created and all properties of the `EnterpriseTopologyGraph` object are stored in the *ETGProject* node.
2. A new *GraphMetaInformation* is created with the import timestamp and is connected to the ETGProject node via a *SNAPSHOT* relationship.
3. The nodes and edges are stored in the database, tagged with the id of the *GraphMetaInformation* node.

With the strict differentiation between meta-information and the actual nodes and edges of a graph, fast browsing through the database content can be offered and a solution for the challenge 5.1.1 **(ii)** is provided.

5.2.4 Custom Iterator (Pagination)

To introduce a paging mechanism in the DataModel different requirements for this feature have to be checked. First, paging should be possible with the used database solution. According to the evaluation in 5.1.3, Neo4j will be the the choice. Consulting the Neo4j API reference, a *PagingIterator*[19] class can be found. This class uses a default iterator to offer a paging mechanism. Because of the huge size of Enterprise Topology Graphs, each graph will likely be put into an own index. So, an *IndexHits*[20] object will be used as data source for the *PagingIterator*. This interface provides a lazy loading iterator for index search results. According to this two classes, Neo4j offers a built-in paging support.

The second step for offering a paging mechanism is to implement an own *PageableList* for the ETG *Entity* items. This list should contain all information about the containing Entities, about the overall size of the result and an automatic refresh mechanism for pages. For this purpose, the list can only be used with a connection to the DataService.

5.3 Plug-In System

The ETG framework should be easily expandable and offer a plug-in mechanism to plug in new functionality, as well as new UI features. This Section will describe the different challenges and the corresponding solutions to implement the plug-in architecture shown in Figure 3.4. The architectural concept shows a service bus used by the plug-ins and the frontend to communicate with the backend and with other plug-ins.

5.3.1 Challenges

The following challenges will be discussed in the design section of the *Plug-In System*:

(i) The first challenges is to provide an easy way to register new plug-ins. It should be possible to deploy and start plug-ins during runtime without restarting or even recompiling the entire ETG framework.

(ii) Plug-Ins also can use other plug-ins already installed in the framework. A mechanism to detect the availability of other plug-ins has to be added. Also the plug-ins should be able to use the already provided functionality without changing already deployed parts of the software.

(iii) It should not only be possible to add new functionality to the framework, but also new GUI parts. This turns out to be a hard challenge, because plug-in parts of the frontend have to be in the same context like the main applications frontend and it has to be possible to update these parts with the plug-in.

[19]PagingIterator - http://api.neo4j.org/current/org/neo4j/helpers/collection/PagingIterator.html
[20]IndexHits - http://api.neo4j.org/current/index.html?org/neo4j/helpers/collection/PagingIterator.html

5.3.2 Solutions

With the challenges described in the previous section, different solutions can be considered. The first solution is to implement a *Service Bus* [Cha04]. Another possibility is to use the *OSGi platform* to provide an architecture for the modular framework [The11]. Of course, a *custom plug-in mechanism* can also be implemented.

5.3.3 Service Bus

A *service bus* is the central part in software system using a service oriented architecture [CLS+05]. The functionality of the bus is to discover, select and invoke services. Because of the requirement, that all plug-ins should be accessible as Web Service, a service bus seems to be the most likely solution.

OSGi-Platform

The OSGi-platform offers a dynamic and hardware independent platform to implement modularized software in Java. For this purpose, OSGi defines a component model containing *Bundles* and *Services*, and comes with a *Service Registry*.

The two most popular implementations of the OSGi-platform are the two frameworks: *Equinox (Eclipse)*[21] and *Felix (Apache)*[22]. The frameworks can be installed in an application server for the usage of the framework. When using OSGi the software and plugins have to be implemented and packaged as bundles. These bundles contain dependencies to other bundles and the implemented functionality and UI parts.

Simple Plug-In Interface

A *simple custom plug-in interface* and mechanism is the third opportunity to solve the challenges described in Section 5.3.1. This solution could be the trade-off between a very loosely coupled implementation with a service bus and a very organized way with the OSGi-platform.

5.3.4 Evaluation

A *service bus* comes with a huge implementation effort. And GUI objects cannot be passed through a messaging mechanism, because only the state can be serialized. That is a huge disadvantage, because the UI of a plug-in has to be updated by the plug-in itself and has to

[21]Eclipse Equinox - http://www.eclipse.org/equinox/
[22]Apache Felix - http://felix.apache.org/

be used by the framework UI to insert it in the main frontend. A new two-way refresh and update mechanism would be required.

A solution with the OSGi-platform can be considered, because the OSGi-application uses the same context. The update and refresh problems can be solved with this mechanism. Also OSGi makes it possible to deploy bundles containing Web Services as so called *Service Bundles*.

Implementing a custom plug-in mechanism seems to be connected with a huge effort. But on closer examination a custom interface can implemented very easily, because all components are implemented as a self-contained unit with a remote accessibility. These features can be used to implement a central service registration as Java Bean or even Web Service. The UI parts can be loaded manually with the main applications classloader to be put in the right context. A hot-deploy mechanism of plug-ins could easily be realized without using a OSGi-framework.

A solution with a custom plug-in interface seems to be best suited for all challenges described in 5.3.1. A central registration can be set up to easily register plug-ins. To discover the availability of the registry, a system like the *Java Naming and Directory Interface (JNDI)* or a check for a *Web Services Description Language (WSDL)* file on a defined location can be used [TB02] [CCM$^+$01]. The plug-ins UI components can easily be loaded by the applications frontend and can communicate with the plug-ins functionality by calling the remote interface.

CHAPTER 6

IMPLEMENTATION DETAILS

In this Chapter the implementation of the framework will be described. First, an Overview is given in the Section 6.1. After the short overview, the main project of the ETG framework, containing different backends and helper for connection and transaction management, will be discussed in Section 6.2. The graphical user interface of the framework will be shown in Section 6.3. The implementation of two example plug-ins is presented in 6.4. Last, Chapter 6.5 describes the tests for the framework and delivers some perfomance metrics.

6.1 Implementation Overview

This Section will give a short overview of the implementation. The implementation is realized in Java. As application server and server runtime *JBoss 7.1.1*[1] is used. The database solution is the *Neo4j graph database 1.8 community edition*[2]. For developing the frontend the *Vaadin framework 7.0.1*[3] is used. For more information on the versions and used libraries Appendix B can be consulted.

This reference implementation provides all features mentioned in the use cases in Chapter 4.

6.2 Data Service Project

DataService is the central project in the ETG framework. It provides session beans for the different backends, is responsible for the database connection management and offers a helper class for transactions. With the invocation of the different backends, the *DataService* provides a solution for the requirement **(r1)**.

[1]JBoss Downloads - `http://www.jboss.org/jbossas/downloads/`
[2]Neo4j Downloads - `http://www.neo4j.org/download`
[3]Vaadin Downloads - `https://vaadin.com/download`

6.2.1 Database Connection Management

The database connection management is encapsulated by a singleton called `DBConnectionFactory`. This singleton holds all open connections to a specific database. The default implementation creates a new embedded database for a given path, or reuses the already create database as embedded instance. The Neo4j connection object is thread-safe, so only one connection instance has to be hold per database. This connection instance can be shared between the backends and the `DataServiceBean`. To guarantee a proper shutdown of the connection, the `DBConnectionFactory` counts all current users of the database connection. If no more users are registered to a connection, the connection will be properly shutdown. Because of better performance, the `DataServiceBean` always holds a connection during its lifetime. So the backends always can use a initialized connection.

For future work, the `DBConnectionFactory` easily can be modified. It would be possible to offer different implementations for different types of connections, e.g. the default input to create a new connection is a local path to the database folder, but if an URL is passed, a new REST connection to a remote Neo4j server or even cluster can be established. This encapsulation of the connection instantiation offers a easily modifiable way to connect to different Neo4j database scenarios.

6.2.2 Transaction Management

Transaction management is a very important subject when dealing with huge data and databases. Neo4j comes with a built in transaction support and enforces the usage of them. All modifying database operations have to be encapsulated in a transaction. In the framework the creation and state management of transaction are handled by the `ITransactionHelper` interface and the default implementation `TransactionHelper`.

With the `TransactionHelper` it is possible to use nested transactions. If a transaction fails, all parent transactions will fail, too. This is a very helpful mechanism for e.g. importing a huge *Enterprise Topology Graph* in a transaction scope. The graph is only imported if all nodes and edges are imported properly in their own transactions. For the random graph creation a set of consecutive transaction can be used. So if one transaction fails, the graph only does not contain the entities, which should have been created in the scope of the failed transaction.

The transaction can be used by the user, too. The `GraphBackend` passes the responsibility to encapsulate the modifying actions in transactions to the user. So the user can dictate the transaction scopes.

6.2.3 Data Service

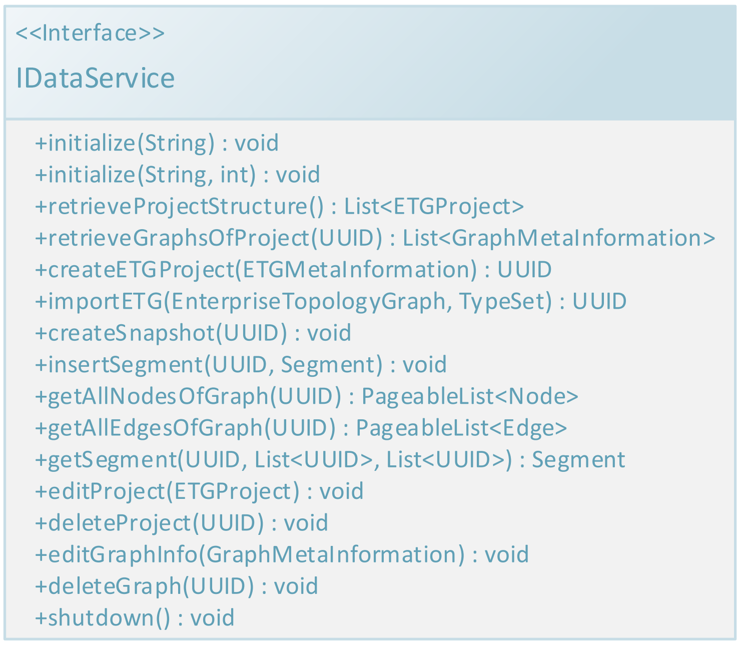
The `DataServiceBean` is the central entry point for the framework usage and implements the `IDataService` interface.

Listing 6.1 ITransactionHelper interface for transaction handling.

```
package de.da_todenhhs.etg.dataservice.interfaces;

import org.neo4j.graphdb.GraphDatabaseService;

/**
 * Helper class to keep on track with the transactions.
 *
 * @author Hannes Todenhagen
 *
 */
public interface ITransactionHelper {

      /**
       * Begins a database transaction.
       */
      public void beginTransaction(GraphDatabaseService graphDb);

      /**
       * Lets a running transaction succeed.
       */
      public void succeedTransaction();

      /**
       * Lets a running transaction fail.
       */
      public void failTransaction();

      /**
       * Finishes a running transaction.
       */
      public void finishTransaction();

}
```

All framework clients always must use this interface to access the functionality. This bean coordinates the invocation of the backends, provides exception handling and a proper logging and initializes the framework configuration.

The Figure 6.1 shows all methods, wich are implemented in the access point to the frameworks functionality. There are two `initialize` methods, because in addition to the database path, the page size of the `PageableLists` can be configured. The default value is 500 entities per page. The remaining methods can be used to retrieve meta information and to create, delete and modify projects and graphs. After the framework usage, for example if the frontends UI is closed, the data service has to be shutdown. The shutdown method closes the remaining database connections and cares about a proper database shutdown. The database will run a recovery mechanism if the shutdown could not be executed without an error. E.g. if a transaction did not finish, the Neo4j database rolls back the already executed part of the transactions and logs a warning.



```
<<Interface>>
IDataService

+initialize(String) : void
+initialize(String, int) : void
+retrieveProjectStructure() : List<ETGProject>
+retrieveGraphsOfProject(UUID) : List<GraphMetaInformation>
+createETGProject(ETGMetaInformation) : UUID
+importETG(EnterpriseTopologyGraph, TypeSet) : UUID
+createSnapshot(UUID) : void
+insertSegment(UUID, Segment) : void
+getAllNodesOfGraph(UUID) : PageableList<Node>
+getAllEdgesOfGraph(UUID) : PageableList<Edge>
+getSegment(UUID, List<UUID>, List<UUID>) : Segment
+editProject(ETGProject) : void
+deleteProject(UUID) : void
+editGraphInfo(GraphMetaInformation) : void
+deleteGraph(UUID) : void
+shutdown() : void
```

Figure 6.1: Class diagram of the IDataService interface.

The other backends provide additional features, e.g. manipulating the graph data directly in the Neo4j store (6.2.6) or execute search queries with an automated result transformation in segments (6.2.7).

6.2.4 Meta Backend

The `MetaBackendBean` implements the `IMetaBackend` interface and administrates the meta graph described in 5.2.3 - Meta Information Representation, except from the types path of the graph. The functionality in the `MetaBackendBean` is needed to provide a solution for the requirement **(r3)**.

This bean is deployed as local bean and is the only bean, which cannot be used by a framework client directly to avoid an inconsistent and broken database state. The meta information is the holds the identifiers for graphs, because of that the `MetaBackendBean` is only used by the `DataServiceBean`.

The diagram in Figure 6.2 shows the implemented `IMetaBackend` interface. The *Meta Backend* provides the methods to insert, retrieve, update and delete all kinds of meta information.

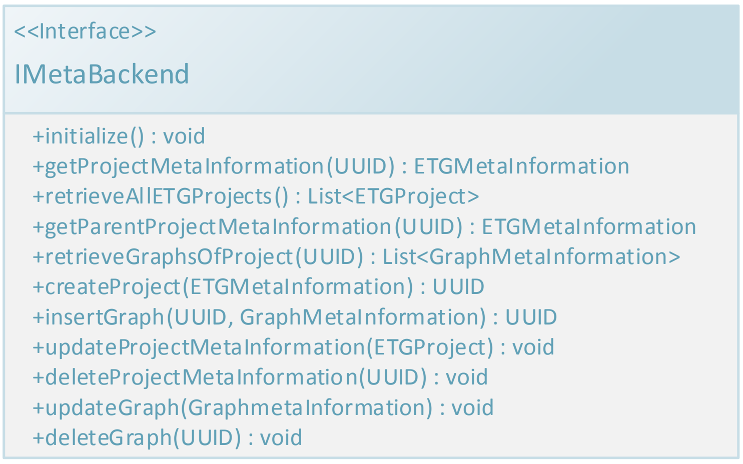

<<Interface>>
IMetaBackend
+initialize() : void
+getProjectMetaInformation(UUID) : ETGMetaInformation
+retrieveAllETGProjects() : List<ETGProject>
+getParentProjectMetaInformation(UUID) : ETGMetaInformation
+retrieveGraphsOfProject(UUID) : List<GraphMetaInformation>
+createProject(ETGMetaInformation) : UUID
+insertGraph(UUID, GraphMetaInformation) : UUID
+updateProjectMetaInformation(ETGProject) : void
+deleteProjectMetaInformation(UUID) : void
+updateGraph(GraphmetaInformation) : void
+deleteGraph(UUID) : void

Figure 6.2: Class diagram of the IMetaBackend interface.

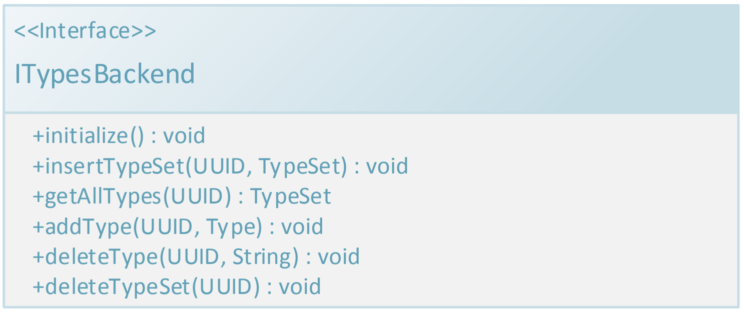

<<Interface>>
ITypesBackend
+initialize() : void
+insertTypeSet(UUID, TypeSet) : void
+getAllTypes(UUID) : TypeSet
+addType(UUID, Type) : void
+deleteType(UUID, String) : void
+deleteTypeSet(UUID) : void

Figure 6.3: Class diagram of the ITypesBackend interface.

6.2.5 Types Backend

The `TypesBackendBean` complements the `MetaBackendBean` and implements the `ITypesBackend` interface. The type management for an ETG project is implemented in the TypesBackend. Types or even full type sets can be added or removed. The `TypesBackendBean` mainly works with the `TypeSet` class. This class represents the strong hierarchical types structure to match the definition in 2.3. A specific `TypeSet` object for an Enterprise Topology Graph can be seen as a dictionary for the type identifiers in the entities. This bean provides a solution to match the requirement **(r4)**.

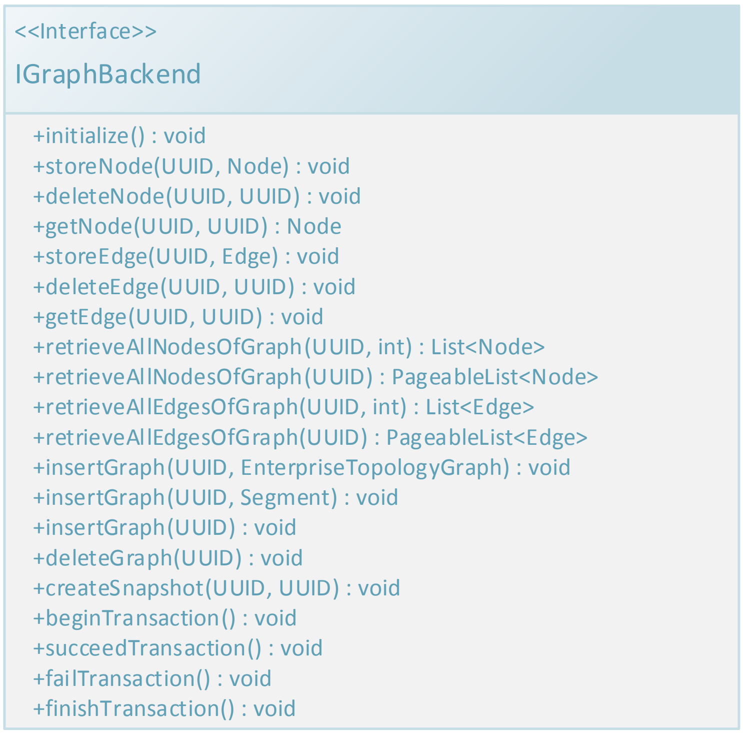

<<Interface>>
IGraphBackend
+initialize() : void
+storeNode(UUID, Node) : void
+deleteNode(UUID, UUID) : void
+getNode(UUID, UUID) : Node
+storeEdge(UUID, Edge) : void
+deleteEdge(UUID, UUID) : void
+getEdge(UUID, UUID) : void
+retrieveAllNodesOfGraph(UUID, int) : List<Node>
+retrieveAllNodesOfGraph(UUID) : PageableList<Node>
+retrieveAllEdgesOfGraph(UUID, int) : List<Edge>
+retrieveAllEdgesOfGraph(UUID) : PageableList<Edge>
+insertGraph(UUID, EnterpriseTopologyGraph) : void
+insertGraph(UUID, Segment) : void
+insertGraph(UUID) : void
+deleteGraph(UUID) : void
+createSnapshot(UUID, UUID) : void
+beginTransaction() : void
+succeedTransaction() : void
+failTransaction() : void
+finishTransaction() : void

Figure 6.4: Class diagram of the IGraphBackend interface.

The `ITypesBackend` (shown in Figure 6.3) interface provides the method to access the types in the ETG project.

6.2.6 Graph Backend

The `GraphBackendBean` implements the `IGraphBackend` interface and is the graph processing unit in the framework. It stores and manages the different entities in the ETGs. The `GraphBackendBean` is used by the `DataService` to store the graph data, but can also be used by the user directly to create and store large graphs, which would not fit into the memory as `EnterpriseTopologyGraph` object. This mechanism makes it possible to store graphs for example in *.xml* files or in a binary format and then import them into the frameworks database. This bean is in charge of the entity manipulation of graphs and provides the solution for the requirement **(r2)**.

The diagram in Figure 6.4 shows all available method in the `IGraphBackend` interface. All entity modifying methods have to be executed in the scope of a transaction.

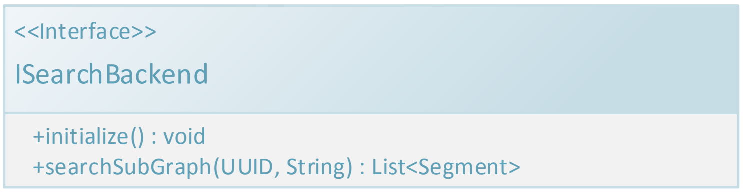


Figure 6.5: Class diagram of the ISearchBackend interface.

Indices

The `GraphBackendBean` manages the huge Enterprise Topology Graphs with the help of indices. Each ETG has its own `ETGIndexManager` to hold an index for the graphs nodes and an index for the graphs edges. The `ETGIndexManager` cares about the indices names and about the frequently used queries, like the *Get All Nodes*- or the *Get All Edges*-query.

6.2.7 Search Backend

The framework offers the possibility to query the ETG database with **Cypher Queries**[4] in combination with the **Apache Lucene Index Queries**[5]. The `SearchBackendBean` can be used by the frameworks clients to execute a query on the graph and implements the `ISearchBackend` interface. This bean is part of the solution for the requirement **(r5)**.

The `SearchBackendBean` is a very powerful component of the framework, because not only *SELECT* queries can be executed, but also *CREATE* or even *DELETE* queries. The `SearchBackendBean`

As seen in the Figure 6.5, the `ISearchBackend` provides only one productive method. With this method, all kinds of queries can be executed in the scope of a specific stored snapshot or segment.

ETG Queries and Index Names

This Section will give some example queries and will show the usage of **Cypher** and **Lucene**. The query has to be passed to the `SearchBackendBean` as String. The Syntax evaluation will be executed by the Neo4js Cypher parsing library. The `searchSubGraph` method throws a

[4]Cypher Query Language - http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html
[5]Apache Lucene Core - http://lucene.apache.org/core/

`QuerySyntaxException` if the String can not be parsed. This exception contains more information about the wrong syntax.

Cypher Queries can be a combination of the following distinct clauses:

- **START** - Defines the start nodes or edges for the query. Can be obtained through an index lookup with a *Lucene Query*.
- **MATCH** - The sub graph pattern to match. The graph exploration will be started on the *START* entities.
- **WHERE** - Defines filtering criteria for the different entities in the sub graph.
- **RETURN** - The entities which should be returned. The `SearchBackendBean` only will return Segments. With this clause the user can define which entities should be contained in the resulting segment. The ETG framework ignores properties in the return clause, because only full entities or lists of entities can be added to a segment.
- **CREATE** - With this clause entities can be created.
- **DELETE** - Clause for deleting entities.
- **SET** - Values of properties can be set with this clause.
- **FOREACH** - Loop clause for update operations on a set of result entities.
- **WITH** - Divider to split the query in multiple, distinct parts.

A standard sub-graph matching query looks like this:

```
START first=node:<indexId>("id:*")MATCH first-[edge]->second RETURN first, edge,
second
```

This query return a list of sub-graphs with two nodes, connected by a directed edge.

This queries can be extended by the use of aggregations like `COLLECT`. This aggregation can be used to retrieve a list of entities as `RETURN`. A sub-graph representing the neighborhood of a node with the depth one can be gathered with the following query:

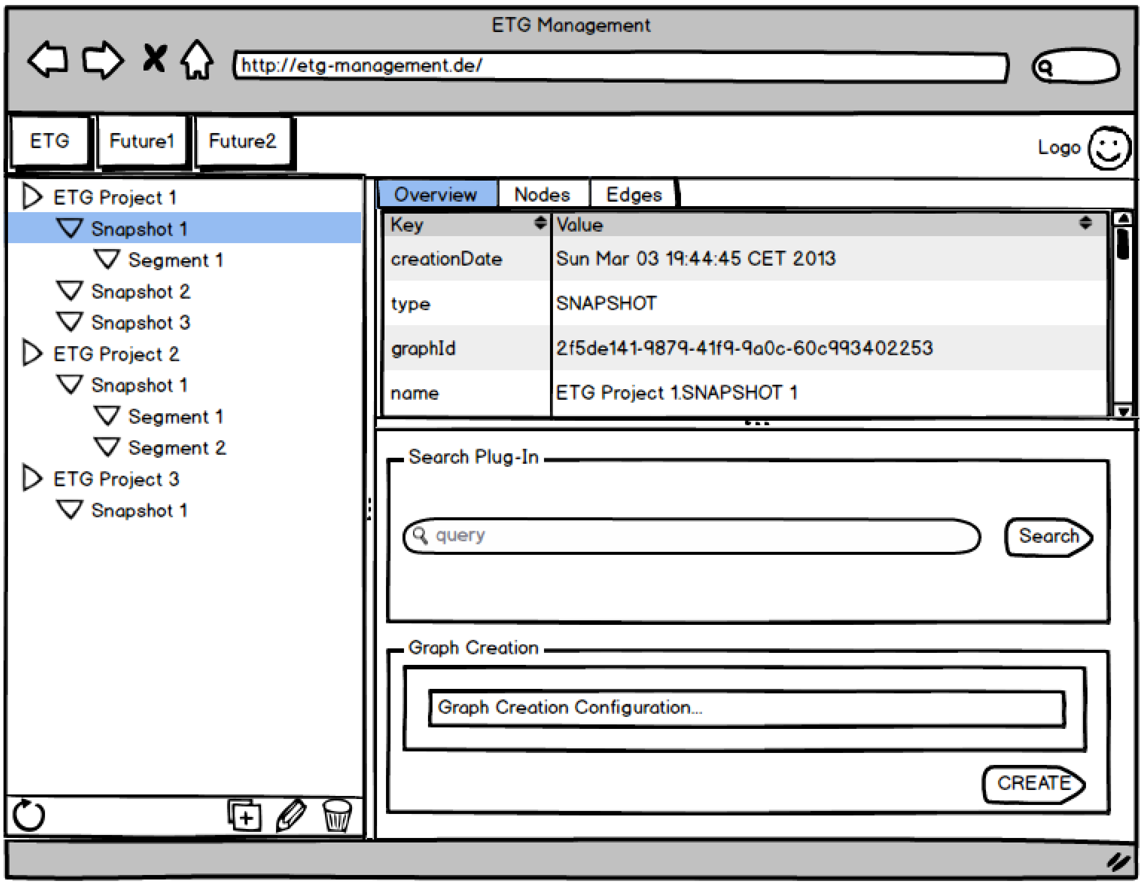
```
START first=node:<indexId>("id:<nodeId>")MATCH first-[edge]->second RETURN first,
COLLECT(edge), COLLECT(second)
```

This query results in a sub-graph containing all outgoing edges of the node with all target nodes.

The pattern, used in the `MATCH` clause can be extended through filter options. The following query collects the hosted nodes of a *JBoss 7.1.1* node:

```
START first=node:<indexId>("id:<nodeId>")MATCH first<-[edge:HOSTED_ON]-second WHERE
second.name = 'JBoss 7.1.1'RETURN first, COLLECT(edge), COLLECT(second)
```

The result of this query is a sub-graph containing the *JBoss 7.1.1* node and all outgoing edges with the type `HOSTED_ON` and their target nodes.



Key	Value
creationDate	Sun Mar 03 19:44:45 CET 2013
type	SNAPSHOT
graphId	2f5de141-9879-41f9-9a0c-60c993402253
name	ETG Project 1.SNAPSHOT 1

Figure 6.6: Mock-up of the Frontend.

6.3 Frontend Project

The *Frontend* project implements a basic graphical user interface for the ETG framework, to provide browsing functionality to the user. With this simple interface, the ETG projects, the graphs, and their entities can be explored and manipulated. The functionality of the *Frontend* project matches the requirement **(r7)**. Because of the pluggable UI architecture, it also provides a part of the solution for requirement **(r6)**.

The frontend is implemented using the *Vaadin 7*[6] framework. This framework is based on the *Google Web Toolkit (GWT)*[7] and provides basic GUI components and widgets like buttons, windows, and even tables or lists.

[6]Vaadin - https://vaadin.com/home
[7]GWT - https://developers.google.com/web-toolkit/

The GUI is divided in two main areas: The navigation tree to navigate through the meta graph of the ETG framework (described in 5.2.3) and the main area to display the properties and environment information of the projects, graphs or even entities (see Figure 6.6). A list of all nodes and all edges of a selected graph is shown as tab in the main area, too. The toolbar on the top of the screen is designed to extend the web interface with other management framework interfaces.

6.3.1 Plug-In Management

A main feature of the frontend is the possibility to extend the interface by plug-in specific GUI components. These components are delegated to the frontend, when the plug-in is deployed and registered. The registration of the plug-ins is managed by the `PluginRegistrationBean`, which implements the `IPluginRegistration` interface. The plug-ins have to call the `registerPlugin` method in their start-up sequence (more information in 6.4). This method inserts the plug-in in the singleton `PluginManager`, which holds all registered plug-ins and provides a way to register `IPluginLoadListener`s for status updates of the plug-in set. With this listener interface UI components can react on plug-in registrations and can load the appropriate plug-in UI components.

The plug-in components are used by the main frontend. They have to be updated and placed in different containers. To make this usage possible, the components have to be loaded in the same classloading context as the main frontend interface. To achieve this goal, the `registerPlugin` method expects a path to a *.jar* archive containing the UI components. This *.jar* file is scanned for a specific plug-in class, which implements the `IPlugin` interface.

Each plug-in can define a plug-in panel, which is inserted in the plug-in view of the specified context (see 6.3.2). In addition, the plug-in can define a set of actions. Each action has to extend the abstract `ETGAction` class and must define a target class. With the help of the target class the frontend determines the objects, on which the actions can be performed.

For more information on how to implement a plug-in, the registering and the loading sequence, the next Section (Plug-Ins) can be consulted.

6.3.2 GUI Context

A GUI context defines constants and methods, which can be used in the interface implementation of a plug-in. Every GUI context implementation has to implement the `IGuiContext` interface. A context is passed to the plug-ins interface object, when it is created by a frontend view. That's because different views can be created in a different context (e.g. a property view can be used to display properties of a snapshot or an entity.). The context holds the id of the object in the scope, e.g. the graph id of a snapshot. In addition, the `IGuiContext` interface provides a default method to create closable tabs in the main area of the frontend. With this method plug-ins can create their own tabs.

The `IGuiContext` interface can be extended in the future. If different requirements occur in the different contexts, additional interfaces and implementations can be added. For example, if the entity context needs to extend a property view, a new interface for an entity context can be created. This new interface has to extend the `IGuiContext` and can provide the information and methods to manipulate the property view. The implementation of this new context has to be added to the frontend project. With this possibility for extension, the context can easily used to manipulate the frontend UI in a controlled way and the framework's interface can be modified to match future plug-ins requirements.

6.4 Plug-Ins

This Section will describe the process to implement a plug-in to extend the functionality and user interface of the ETG framework in 6.4.1. After that, in 6.4.2 the registering and loading process of plug-ins and their corresponding UI components will be shown. At the end, two example plug-ins will be shown: The *Random Graph Creator Plug-In* (in 6.4.3), which creates random ETGs with direct usage of the **Graph Backend** (see 6.2.6), and the *Search Plug-In* (in 6.4.4), which provides base search functionality and neighborhood discovery for entities using the **Search Backend** (see 6.2.7).

6.4.1 Plug-In Implementation

To implement a plug-in for the ETG framework, basically three parts have to be created:

- The interfaces
- The functionality
- The graphical user interface

First, **the interfaces** have to be defined. These interfaces describe the usage of the plug-ins functionality and are used to define session beans or even Web Services. These interfaces have to be provided in a *common* project of the plug-in, because the UI objects have to use them and this components are in a different deployment unit.

Second, **the functionality** must be implemented using session beans. The bean project can be implemented without caring about the ETG framework. The features, provided by the framework, can be accessed through the frameworks public beans.

If the implementation of the beans is finished, a start-up bean has to be added with a post-construct method, to trigger the registration process after the plug-ins deployment. In the post-construct method the `PluginRegistrationBean` (described in 6.3.1) has to be used to register the plug-in. To deregister the plug-in also a pre-destroy method has to be added. Application servers differ in the mechanism to define and use start-up beans.

Listing 6.2 Example implementation of a *PluginRegistrator* for a plug-in.

```java
/**
 * Session Bean implementation class PluginRegistrator
 */
@Singleton
@Startup
public class PluginRegistrator {

      private static final Logger logger = Logger
                  .getLogger(PluginRegistrator.class);

      /**
       * Executed at the startup of this module. Registers the plug-in in the
       * framework.
       */
      @PostConstruct
      public void onStartup() {

            try {
                  Thread.sleep(500);

                  Context c = new InitialContext();
                  ((IPluginRegistration) c
                              .lookup("java:global/ETGFrameworkEAR/Frontend" +
                                          "/PluginRegistrationBean"))
                              .registerPlugin(Config.UNIQUE_ID, Config.JAR_NAME);
            } catch (NamingException e) {
                  logger.warn("Could not find the PluginRegistration service.");
                  logger.trace(e.getMessage(), e);
            } catch (InterruptedException e) {
                  logger.error(e.getMessage(), e);
            }
      }

      /**
       * Deregisters the plug-in in the framework on shutdown.
       */
      @PreDestroy
      public void onShutdown() {
            try {
                  Context c = new InitialContext();
                  ((IPluginRegistration) c
                              .lookup("java:global/ETGFrameworkEAR/Frontend" +
                                          "/PluginRegistrationBean"))
                              .deregisterPlugin(Config.UNIQUE_ID);
            } catch (NamingException e) {
                  logger.warn("Could not find the PluginRegistration service.");
                  logger.trace(e.getMessage(), e);
            }
      }

}
```

The Figure 6.2 shows an example `PluginRegistrator` bean, using a JBoss 7.1.1 application server runtime. The `Config` holds all plug-in properties and is contained in the *common* project of the plug-in.

The plug-in is now ready to be used. But if the plug-in needs to extend **the graphical user interface** of the framework another project must be added. In the GUI project a panel implementing the `IPluginPanel` interface can be added. This panel behaves like a standard Vaadin panel and can be designed like every Vaadin component. This panel is the main GUI component of the plug-in. In addition, actions can be implemented using the abstract `ETGAction` class. To delegate this UI components to the frontend, a *plug-in class* is needed. This class must extend the `IPlugin` interface.

The listing 6.3 shows an example implementation of the `IPlugin` interface for a plug-in providing a panel and an action for two different targets.

The implementation of the plug-in does not require any more steps. In the Appendix A, more information about the deployment possibilities can be obtained.

6.4.2 Register- and Load-Sequence

After the implementation of a plug-in, this Section gives more insight into the registration and loading process of plug-ins. These processes are illustrated with the UML sequence diagram in Figure 6.7.

The registration process starts with the invocation of the post-construct method of the `PluginRegistrator` of the plug-in. This method calls the `PluginRegistrationBean` and passes the plug-ins unique identifier and the name or even the path to the *.jar* file containing the plug-ins UI classes. The `PluginRegistrationBean` tries to load the UI from the given path with the help of the `PluginLoader`. If no path is given or the file can not be found on the given path, the `PluginLoader` searches for a matching *.jar* file in the default plug-ins folder. The default plug-in folder can be configured in the *Data Service*. If the `PluginLoader` finds the classes, a new classloader will be created. This classloader is a child loader of the frontends classloader, because of that, the new classloader is aware of the existing context and all classes in the frontend and common project can be linked properly. With this new classloader the plug-ins implementation of the `IPlugin` interface is instantiated and passed to the `PluginManager`. In the manager, the new plug-in is added to the list of initialized plug-ins and all registered `IPluginLoadListeners` are informed by calling their `pluginInitialized` method. The registration process is completed.

The plug-in loading process can be started on two different ways: First, if a new `IPluginLoadListeners` is added to the `PluginManager`, for each plug-in in the list of initialized plug-ins the listeners method `pluginInitialized` is called. The view, which has implemented the listener, is then aware of the plug-in. The view can instantiate a new plug-in panel and can attach the plug-ins actions to the target objects. The second way to start the plug-in

Listing 6.3 Example *IPlugin* implementation.

```
/**
 * The entry point for the example plug-ins GUI.
 *
 * @author Hannes Todenhagen
 *
 */
public class ExampleGuiPlugin implements IPlugin {

      @Override
      public String getName() {
            return "Example Plug-in";
      }

      @Override
      public String getDescription() {
            return "Example Plug-in showing the "
                  + "implementation of a plug-in";
      }

      @Override
      public String getUniqueId() {
            return "de.da_todenhhs.etg.example";
      }

      @Override
      public ContextType getContext() {
            return ContextType.GRAPH;
      }

      @Override
      public IPluginPanel getUi(IGuiContext context) {
            return new ExamplePanel(context); // The plug-ins panel.
      }

      @Override
      public Collection<ETGAction> getActions(IGuiContext context) {

            List<ETGAction> actions = new ArrayList<ETGAction>();

            if (context != null) {
                  actions.add(new ExampleAction(context, Node.class));
                  actions.add(new ExampleAction(context, Edge.class));
            }

            return actions;
      }
}
```

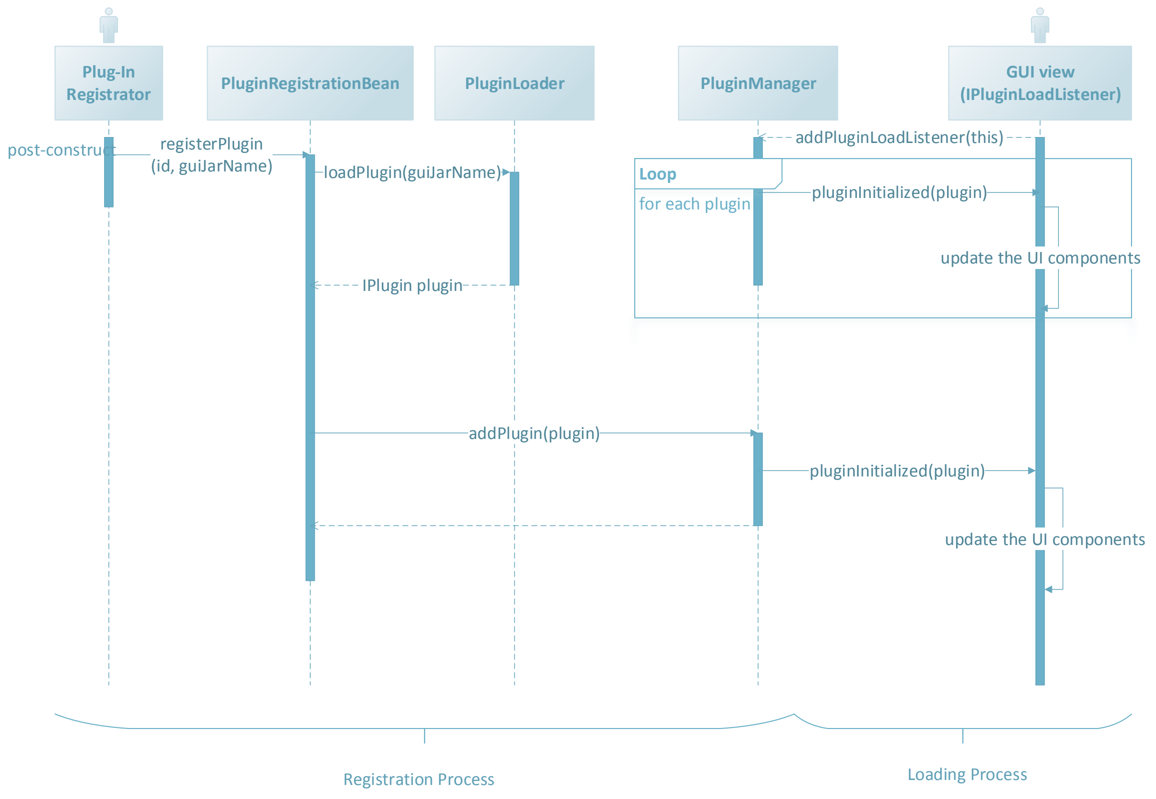


Figure 6.7: Plug-In registration and loading process.

loading process can be triggered by a completed plug-in registration process. If the plug-in successfully got added to the `PluginManager`, all listeners get informed about the new plug-in.

The deregistration process is similar to the registration process. The listeners also will get informed about the deregistration by calling their `pluginShutdown` method. A deregistration process is triggered by the undeployment of a plug-in. This can only happen, if the plug-ins are deployed in their own *Enterprise Application Archive*. More information about the different deployment possibilities can be obtain by reading the Appendix A.

6.4.3 Random Graph Creator Plug-In

The *Random Graph Creator Plug-In* provides a possibility to create new ETG projects with an initial, random snapshot. The snapshot is created randomly and has a defined number of nodes. This plug-in also offers a plug-in panel to trigger the creation process in the user interface. This plug-in directly uses the `GraphBackendBean` in the *Data Service*, to be able to create very big graphs without running the risk of a `OutOfMemoryException`.

6.4.4 Search Plug-In

The *Search Plug-In* only extends the user interface and does not offer new framework functionality. The plug-in offers a panel, to send a query to the framework. The resulting segments are displayed in a table with collapsible items. The plug-in also offers actions to execute a neighborhood discovery for entities. This action executes a pre-defined query.

```
START first=node:<graphIndex>("id:<entityId>")MATCH first-[rel]-second RETURN first,
COLLECT(rel), COLLECT(second)
```

This query results in a segment containing the neighborhood of the specified entity with depth one. The `COLLECT` aggregation joins the different paths on the first node. If the `COLLECT` is removed, the result of the query would be a list of segments, each with two nodes and one edge.

6.5 Test and Performance

The tests for the implementation are described in Section 6.5.1. The tests cover the functionality described in Chapter 4. In Section 6.5.2, some diagrams are shown to give a short overview of the performance of the framework.

6.5.1 Test

The main functionality of the framework is realized with Java Beans. Therefore, it is very difficult to test the methods with the testing framework *JUnit*[8], because the Beans are executed within a Context. There are some different techniques to test Java Enterprise Applications. For example, *Mocks* can be used, simplified implementations of unavailable dependencies or frameworks, that execute tests in an enterprise context (e.g. *Arquillian*[9]). But these solutions need a lot of extra effort to realize significant tests. Fortunately, because the ETG frameworks *Backends* do not depend on each other, the tests can be realized with standard *JUnit* tests. With a following system test of the *DataService* the tests are complete, because the *DataService* administers the *Backend* calls. This system test has been written as framework client.

[8]JUnit - `http://junit.org/`
[9]Arquillian - `http://arquillian.org/`

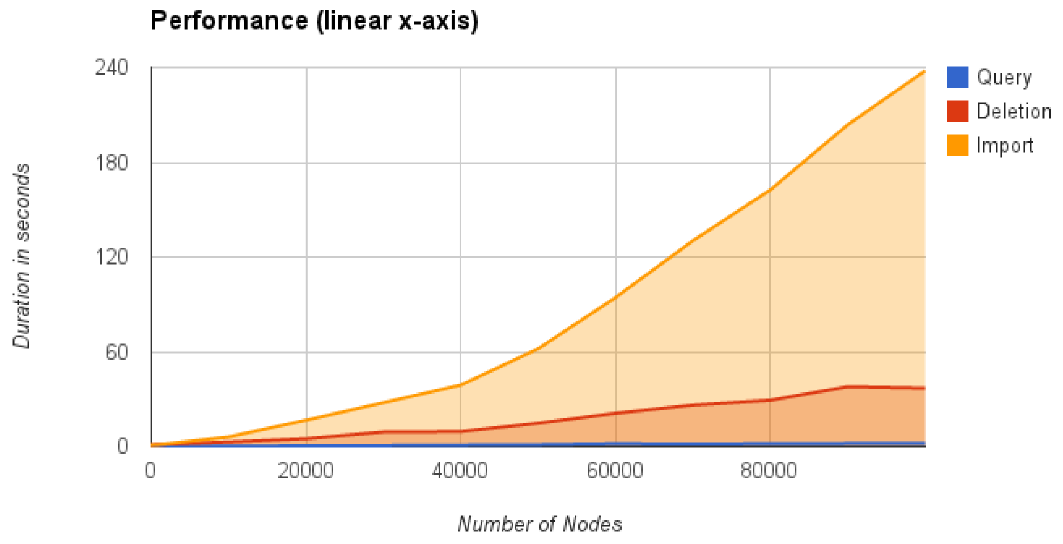


Figure 6.8: Performance diagram with linear x-axis

6.5.2 Performance

This section shows some diagrams to give an overview of the frameworks performance. The measurements are executed in the test environment to get a direct view on the *Backends* performance. Performance data for the three most expensive operations has been captured: Importing an Enterprise Topology Graph, executing a query to get all nodes and deleting an ETG. These trigger the most invocations on different *Backends* and are well suited for an evaluation. The number of entities of the graph was increased up to 150.000. This number can be processed without the direct usage of the *GraphBackend*. Five properties have to be stored for each entity. To get the duration of a real database query, the `retrieveAllNodes`method of the *DataService* is used. This method returns a `PageableList` with a static page size of 500 items. Each measurement was executed 10 times for all graphs and the diagrams show the average values.

Figure 6.8 and Figure 6.9 show the results of the performance measurement.

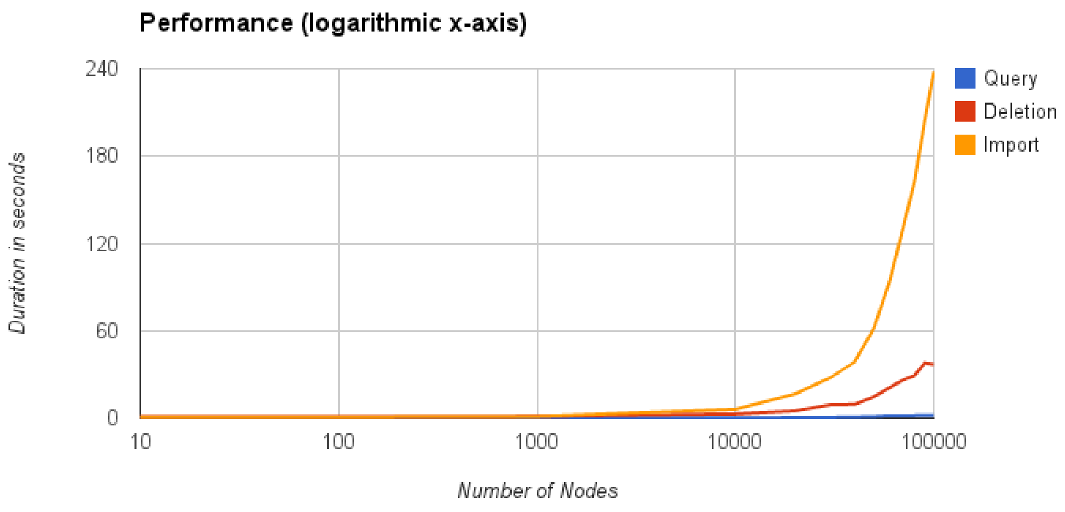


Figure 6.9: Performance diagram with logarithmic x-axis

CHAPTER 7

SUMMARY AND OUTLOOK

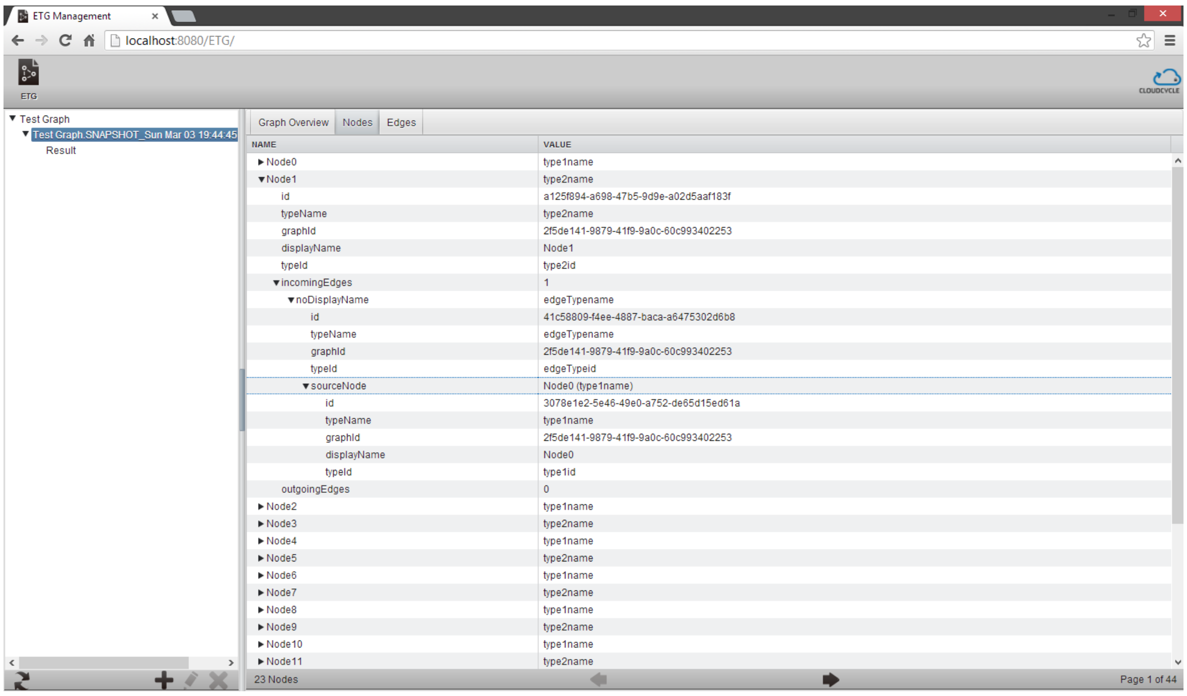
The usage of the Cloud Computing paradigm is getting more and more accepted and on-demand infrastructure, platform and software offerings are mixed with traditional datacenters. The systems structures are getting more complex. The optimization of the performance and costs of the IT components is getting more difficult due to this rising complexity. To be able to properly manage the IT landscape of a company a mechanism to describe the whole system, including all on- and off-premise components, is needed.

With the definition of Enterprise Topology Graph a solution is provided. The focus of this diploma thesis was the research, the design and the implementation of a framework to manage ETGs. The goal was to develop a plugin-based framework with basic storage and manipulation functionality in addition to two prototype plugins, which provide a basic sub-graph matching and an example ETG generation. Other developers can extend the framework by developing additional plug-ins for e.g. specific transformations on ETG sub-graphs.

The first step towards the final software was to gather information to formulate the requirements. Based on these requirements a first architecture was created. This architecture contains three main parts, a backend, plug-ins and a web interface. Afterwards, use cases could be created for the framework. This use cases describe the actual functionality of the different parts of the frameworks core. In the design phase, the challenges of each part were discovered and matching solutions were planned. For example, the underlying data base solution had to be selected. After evaluating multiple database solutions, Neo4j was determined to deliver the most sufficient functionality to build the ETG storage. To store the ETGs, a database model had to be created. The meta information of the ETGs is saved into a meta graph to enable features like the storage of different versions of one ETG. The framework was implemented in Java and offers the full functionality the use cases defined. A custom plug-in mechanism was created to easily add new features to the framework. A web interface, realized with the Vaadin Toolkit, also was implemented to interact with the framework.

A screenshot of the final web interface is shown in Figure 7.1.

The goal of future work is the automatization of the management of complex enterprise IT systems. This diploma thesis is the first step to benefit from the huge power of Enterprise Topology Graphs. In future, more plug-ins will be created to introduce transformation



NAME	VALUE
▶ Node0	type1name
▼ Node1	type2name
id	a125f894-a698-47b5-9d9e-a02d5aaf183f
typeName	type2name
graphId	2f5de141-9879-41f9-9a0c-60c993402253
displayName	Node1
typeId	type2id
▼ incomingEdges	1
▼ noDisplayName	edgeTypename
id	41c58809-f4ee-4887-baca-a6475302d6b8
typeName	edgeTypename
graphId	2f5de141-9879-41f9-9a0c-60c993402253
typeId	edgeTypeid
▼ sourceNode	Node0 (type1name)
id	3078e1e2-5e46-49e0-a752-de65d15ed61a
typeName	type1name
graphId	2f5de141-9879-41f9-9a0c-60c993402253
displayName	Node0
typeId	type1id
outgoingEdges	0
▶ Node2	type1name
▶ Node3	type2name
▶ Node4	type1name
▶ Node5	type2name
▶ Node6	type1name
▶ Node7	type2name
▶ Node8	type1name
▶ Node9	type2name
▶ Node10	type1name
▶ Node11	type2name

Figure 7.1: Screenshot of the final web interface.

operations on the graphs to be able to automatically manipulate graphs. Also a machine-aided discovery of Enterprise Topology Graphs will be a topic in future research.

APPENDIX A

DEPLOYMENT AND INSTALLATION GUIDE

This Chapter offers a guide for the deployment of the ETG framework. The framework can be deployed in two different ways: The framework and the plug-ins in one *.ear* file, or each plug-in in an own *.ear* file.

The two deployment are discussed in the following Sections. A JBoss 7.1.1 server is used in the scenarios as underlying application server. Configurations like the log level or common libraries depend on the application server. If another application server is used, this configurations have to be looked up in the specific server documentation.

The log level configuration can be done in the *standalone.xml* configuration file. To configure the ETG frameworks logger, a new logger has to be added in the `<subsystem xmlns="urn:jboss:domain:logging:1.1">` Section of the config file.

The example configuration in listing A.1 shows a logger for the framework with the level *TRACE*. The different categories, represented by the package names, can be configured differently. E.g. another logger can be added with a different level for a plug-in, if more or less information is needed. Important is, that the log handler the configuration uses is configured correctly. The handlers level must contain the framework log level. The console handler configuration in A.2 defines the *INFO* level. If this log handler is used by the frameworks logger configured in A.1, only *INFO*, *WARN* and *ERROR* messages will be displayed in the console. This mechanism is great to log different levels with different log handlers. E.g. warnings and errors can be displayed on the console to keep on track with the frameworks operations, but debug messages can be written into a log file to be able to gather more specific information in an error case.

Listing A.1 Example configuration for the framework logger.

```
<logger category="de.da_todenhhs.etg">
      <level name="TRACE"/>
</logger>
```

Listing A.2 Example configuration for a console log handler.

```
<console-handler name="CONSOLE">
      <level name="INFO"/>
      <formatter>
            <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t) %s%E%n"/>
      </formatter>
</console-handler>
```

A.1 All-in-One Deployment

The *all-in-one* deployment possibility is easier to deploy, but not as dynamic as the *multiple archives* approach. All projects are build and put together in one archive. The common libraries of the framework and the implemented plug-ins have to be declared as *EAR Libraries*, so that these libraries can be used by all other components. All project can then be added as *.ear* content.

The Vaadin libraries have to be put on the *.ear* level, too. That's because the frontend and the plug-ins GUI components use the classes. If the Vaadin libraries would be deployed within each GUI component, linkage errors would occur during the runtime, because of duplicate class loading processes.

The Neo4j libraries only have to be included into the DataService projects classpath. The libraries will be automatically deployed in the *.ears* `/lib` folder.

If the framework is deployed as one archive, the plug-ins will be loaded after the deployment in the post-construction phase. The plug-ins can not be started or stopped independently. If a new plug-in has to be added, the whole archive has to be rebuilt and deployed again. This redeployment results in a downtime during the deployment. To dynamically deploy additional plug-ins, the *multiple archive deployment* approach has to be used.

A.2 Multiple Archives Deployment

The *multiple archive* approach is harder to deploy, but is much more dynamic. In this approach the framework will be deployed without plug-ins. Each plug-in has to be build as extra archive and can be deployed and undeployed independently.

A.2.1 Framework

The frameworks archive can be built the same way as in the *all-in-one* approach, except the common and the Vaadin libraries. The *common* and the Vaadin libraries has to be installed as global usable library, because this libraries will be used across *.ear* boundaries.

With a JBoss 7 application server the installation of the Vaadin and common libraries is done as global module. Global modules can be used by all programs running on the server. To

Listing A.3 Example modules.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="de.da_todenhhs.etg">
        <dependencies>
        <module name="javaee.api"/>
   </dependencies>
   <resources>
      <resource-root path="Common.jar"/>
            <resource-root path="commons-cli-1.2.jar"/>
            <resource-root path="commons-jexl-2.1.1.jar"/>
            <resource-root path="commons-logging-1.1.1.jar"/>
            <resource-root path="cssparser-0.9.5.jar"/>
            <resource-root path="jsoup-1.6.3.jar"/>
            <resource-root path="sac-1.3.jar"/>
            <resource-root path="vaadin-shared-deps-1.0.2.jar"/>
            <resource-root path="vaadin-server-7.0.1.jar"/>
            <resource-root path="vaadin-client-compiled-7.0.1.jar"/>
            <resource-root path="vaadin-client-7.0.1.jar"/>
            <resource-root path="vaadin-shared-7.0.1.jar"/>
            <resource-root path="vaadin-themes-7.0.1.jar"/>
            <resource-root path="vaadin-theme-compiler-7.0.1.jar"/>
   </resources>
</module>
```

Listing A.4 Example entry for a global module.

```
      <subsystem xmlns="urn:jboss:domain:ee:1.0">
         <global-modules>
            <module name="de.da_todenhhs.etg" slot="main"/>
         </global-modules>
      </subsystem>
```

make the libraries accessible through the module mechanism, all *.jar* files have to be placed in the modules folder, under a specific path. For example, all needed libraries are placed under `$JBoss_Home/modules/de/da_todenhhs/etg/main`. The `main` folder defines a slot. With the slots, different versions of a module can be provided. In addition, a `module.xml` file is needed. Listing A.3 shows an example `modules.xml` file. In this file, all libraries and a name for the module have to be configured.

To make the module globally available, the module has to be declared as global in the `standalone.xml`. Therefore, an `subsystem` entry has to be added under the `subsystem xmlns="urn:jboss:domain:ee:1.0"`. In listing A.4 the example from A.3 is declared as global module.

If the Vaadin and common libraries are provided as global module, the framework archive can be deployed.

A.2.2 Plug-Ins

To be able to dynamically add and remove plug-ins, two steps are required. First the common project of the plug-in has to be build and put into the global module. Therefore, the *.jar* file has to be put into the modules folder and a `resource-root` element has to be added in the `module.xml`. After that the plug-in archive can be built. The bean project has to be contained in the *.ear* file, because this project offers the start-up method to register the plug-in in the framework. The UI project has to be placed regarding to the path configuration in the plug-in. If the UI *.jar* file is in the right place and the common *.jar* is added to the global module, the plug-in *.ear* archive can be deployed. Of course, the framework has to be up and running.

To remove a plug-in, the *.ear* can be undeployed using the application servers management console.

Appendix B

Dependencies and Licenses

This Chapter provides a list of dependencies and their licenses for the ETG framework implementation.

B.1 Vaadin

The implementation uses the *Vaadin 7* framework. This framework is under the Apache License Version 2.0. The definition of the license can be found under `http://www.apache.org/licenses/LICENSE-2.0.html`. More information about the licenses of third-party software used by *Vaadin* is provided on `https://vaadin.com/license`.

The framework depends on following libraries:

- commons-cli-1.2.jar
- commons-jexl-2.1.1.jar
- commons-logging-1.1.1.jar
- cssparser-0.9.5.jar
- jsoup-1.6.3.jar
- sac-1.3.jar
- vaadin-client-7.0.1.jar
- vaadin-client-compiled-7.0.1.jar
- vaadin-server-7.0.1.jar
- vaadin-shared-7.0.1.jar
- vaadin-shared-deps-1.0.2.jar
- vaadin-theme-compiler-7.0.1.jar

- vaadin-themes-7.0.1.jar

This libraries has to be included into the frameworks *.ear* file, if the *All-in-One Deployment* (A.1) approach is used. If the *Multiple Archives Deployment* (A.2) approach is used, these libraries have to be put into the global module.

B.2 Neo4j

The ETG framework implementation uses the *Neo4j Community Edition* as underlying database solution. Neo4j is under the GPL license, which can be found under `http://www.gnu.org/licenses/gpl.html`. More about the different versions of Neo4j and their licensing can be found on `http://www.neo4j.org/learn/licensing`.

In addition, the *Google Concurrent Linked Hash Map* is needed to run *Neo4j* properly. This library is under the Apache License Version 2.0. The license definition can be found under `http://www.apache.org/licenses/LICENSE-2.0.html`.

The framework depends on the following libraries:

- concurrentlinkedhashmap-lru-1.3.1.jar
- geronimo-jta_1.1_spec-1.1.1.jar
- lucene-core-3.5.0.jar
- neo4j-cypher-1.8.1.jar
- neo4j-graph-algo-1.8.1.jar
- neo4j-graph-matching-1.8.1.jar
- neo4j-jmx-1.8.1.jar
- neo4j-kernel-1.8.1.jar
- neo4j-lucene-index-1.8.1.jar
- neo4j-shell-1.8.1.jar
- neo4j-udc-1.8.1.jar
- org.apache.servicemix.bundles.jline-0.9.94_1.jar
- scala-library-2.9.1-1.jar
- server-api-1.8.1.jar

These libraries have to be included in the *DataService* project to make the ETG framework work.

Bibliography

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 742 –749. 2012. doi:10.1109/CLOUD.2012.143. (Cited on pages 7, 10, 15, 36 and 37)

[BLNSpt] T. Binz, F. Leymann, A. Nowak, D. Schumm. Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pp. 61–70. Sept. doi:10.1109/EDOC.2012.17. (Cited on page 15)

[CCM+01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web services description language (WSDL) 1.1, 2001. (Cited on page 42)

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008. doi:10.1145/1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816. (Cited on page 33)

[Cha04] D. Chappell. *Enterprise service bus*. O'Reilly Series. O'Reilly, 2004. (Cited on page 41)

[CLS+05] F. Curbera, F. Leymann, T. Storey, D. Ferguson, S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005. (Cited on page 41)

[HGHCM12] M. Hausenblas, R. Grossman, A. Harth, P. Cudré-Mauroux. Large-scale Linked Data Processing - Cloud Computing to the Rescue? In F. Leymann, I. Ivanov, M. van Sinderen, T. Shan, editors, *CLOSER*, pp. 246–251. SciTePress, 2012. (Cited on page 32)

[HMP06] W. Hsieh, J. Madhavan, R. Pike. Data management projects at Google. In *Proceedings of the 2006 ACM SIGMOD international conference on Management* *of data*, SIGMOD '06, pp. 725–726. ACM, New York, NY, USA, 2006. doi:10.1145/1142473.1142566. URL http://doi.acm.org/10.1145/1142473.1142566. (Cited on page 32)

[HPT11] D. T. A. Hoang, T. Priebe, A. M. Tjoa. Hypergraph-based multidimensional data modeling towards on-demand business analysis. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, iiWAS '11, pp. 36–43. ACM, New York, NY, USA, 2011. doi:10.1145/2095536.2095545. URL http://doi.acm.org/10.1145/2095536.2095545. (Cited on page 33)

[Ior10] B. Iordanov. HyperGraphDB: a generalized graph database. In *Proceedings of the 2010 international conference on Web-age information management*, WAIM'10, pp. 25–36. Springer-Verlag, Berlin, Heidelberg, 2010. URL http://dl.acm.org/citation.cfm?id=1927585.1927589. (Cited on page 34)

[KKVR12] V. K. Konishetty, K. A. Kumar, K. Voruganti, G. V. P. Rao. Implementation and evaluation of scalable data structure over HBase. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pp. 1010–1018. ACM, New York, NY, USA, 2012. doi:10.1145/2345396.2345559. URL http://doi.acm.org/10.1145/2345396.2345559. (Cited on page 32)

[MG09] P. Mell, T. Grance. Cloud Computing Definition. *National Institute of Standards and Technology*, 2009. (Cited on pages 9 and 13)

[Qia10] Y. QianCheng. Metadata integration architecture in enterprise data warehouse system. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pp. 340 –343. 2010. doi:10.1109/ICISE.2010.5690320. (Cited on page 33)

[SKRC10] K. Shvachko, H. Kuang, S. Radia, R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pp. 1–10. IEEE Computer Society, Washington, DC, USA, 2010. doi:10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972. (Cited on page 32)

[SWX12] B. Shao, H. Wang, Y. Xiao. Managing and mining large graphs: systems and implementations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 589–592. ACM, New York, NY, USA, 2012. doi:10.1145/2213836.2213907. URL http://doi.acm.org/10.1145/2213836.2213907. (Cited on page 34)

[TB02] T. M. Thomas, M. Books. *Java data access: JDBC, JNDI, and JAXP*. M & T Books, 2002. (Cited on page 42)

[The11] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4, Version 4.3, 2011. (Cited on page 41)

[VMZ+10] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pp. 42:1–42:6. ACM, New York, NY, USA, 2010. doi:10.1145/1900008.1900067. URL `http://doi.acm.org/10.1145/1900008.1900067`. (Cited on page 34)

All links were last followed on March 24, 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Hannes Todenhagen)