BACHELOR THESIS

Ondřej Kudláček

Extension of RDF data analysis and visualization framework architecture and visualization plugins

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Jakub Klímek Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2017
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ........ date ............ signature of the author
Title: Extension of RDF data analysis and visualization framework architecture and visualization plugins

Author: Ondřej Kudláček

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Klímek Ph.D., Department of Software Engineering

Abstract: This extension of Payola framework focuses on user experience. Payola is a web application which allows to visualize Linked Data via its visual plugins. The data can be displayed as graphs, tables, etc. Payola also allows to construct analysis using SPARQL language and browse the analyzed data structures without any need of deep knowledge of SPARQL. The extension is placed between the Payola server and users web browser to increase the visual plugins responsiveness.

Keywords: Linked Data, visualization, framework, plugin
Contents

1 Introduction 4
  1.1 Payola .................................................. 4
  1.2 Extension .............................................. 5
  1.3 Installation ........................................... 5

2 Used technologies 10
  2.1 RDF ..................................................... 10
  2.2 SPARQL ................................................ 12
  2.3 Linked Data .......................................... 12
  2.4 OWL 2 .................................................. 13
  2.5 RDFS .................................................... 14
  2.6 Scala and Play ....................................... 14
  2.7 Database technologies ............................... 15

3 Payola structure 16
  3.1 Workflow .............................................. 16
  3.2 Components .......................................... 17
  3.3 Payola extension ..................................... 19

4 Cache 24
  4.1 Requirements ......................................... 24
  4.2 Implementation ....................................... 25
  4.3 Programmatic usage ................................. 26
  4.4 Evaluation ........................................... 28
  4.5 Further extension ................................... 29

5 Transformations 31
  5.1 Requirements ......................................... 31
  5.2 Implementation ....................................... 32
  5.3 Visual plugin implementation ..................... 33
  5.4 Use case scenario .................................... 35
  5.5 Programmers documentation ....................... 36

6 Embedding support 37
  6.1 Requirements ......................................... 37
  6.2 Implementation ....................................... 38
  6.3 Usage .................................................. 38
  6.4 Use case scenario .................................... 39
  6.5 Programmers documentation ....................... 41
  6.6 Further extension ................................... 42

7 Visualization plugins 43
  7.1 Triple table view ...................................... 43
    7.1.1 Use case scenario ............................... 43
    7.1.2 Programmers documentation ................. 44
  7.2 Graph view ........................................... 45
Preface

Payola is a web application for visualization of RDF data. The first release of Payola contained several visualization plugins for displaying Linked Data. Payola brought a simpler way of browsing through graph data storages (SPARQL endpoints) and analyzing their content and relations. It also allowed users to import their own data and share them with other users, create user groups, public data sources, analyses and more.

The motivation behind creating such a tool was to simplify Linked Data browsing that user would not need any deep knowledge of the SPARQL language. It was developed as a team project by Jiří Helmich, Ondřej Heřmánek, Ondřej Kudláček, Jan Široký and Kryštof Váša in 2012.

This work brings an extension of Payola and focuses on the visual plugins part of the project. It extends previously implemented plugins and brings more data browsing options to the user.
1. Introduction

This chapter introduces the origin of Payola its main used technologies. It contains an introduction of the extension of Payola that this work brings and what it focuses on.

1.1 Payola

At the beginning of Payola development, it was intended to be a data-mining software for searching suspicious relations in public contracts. Pointing out, that for example a company executive owns a car showroom that sold the most expensive model to the person in charge of a public contract, that the company is involved in. Payola was meant to ease finding of corruption. Due to legal issues of publishing found data by Payola it had to change to a data analysis software. With the specific suspicious relations finding mechanism removed [2]. On the other hand there is not any technical barrier that would forbid such a usage of Payola.

The structure of data that Payola is able to process consists of triples, relations of two entities. Providing an analysis mechanism user can filter and search for example for dates, names, numbers or entity relations. To input the data, Payola can access multiple SPARQL endpoints, e.g. http://dbpedia.org/sparql. Such an endpoint provides the data in a structured format - HTML, plain XML, JSON, Turtle or RDF/XML formats (see the chapter 2). Payola can analyze these data and visualize them in a human readable way. Triple Table, Chart or a tree structured Graph are few of the implemented visualizations that can be used to view the data.

Besides the data processing, Payola focuses on user experience. Getting data from SPARQL endpoints requires a certain knowledge of the RDF query language. User interested in the data might not understand the language. Payola simplifies the construction of such a query - analysis, to the level where user might not even know anything about work with a database. To provide a common user experience, analyses can be shared between users or user groups and be accessible publicly [2].

To provide a standalone endpoint, Payola has its Virtuoso database server. Thus users can import their own datasets for analyzing and sharing with others. Even import of a Payola processed dataset is possible.

To demonstrate a bit of Payola see the figure 1.1. It shows a visual plugin that displays a result of a http://dbpedia.org/ data analysis. This particular analysis filtered cities with population over 2,000,000 and countries they are in. Notice the lines between the nodes. These lines represent relations of entities. In this example the highlighted node (entity) New York City is in a relation of type Country to entity United States.
1.2 Extension

From the experience of using Payola it appears that there is a bottleneck in the data analyzing section. When user analyses data the whole analysis result has to be transferred to and processed in the memory of the user's web browser. More specifically when an analysis is completed, the data is serialized by Apache Jena framework to be sent to client side JavaScript. This work brings an extension to this part of Payola. It alters the communication between the client and the server sides and optimizes the amounts of communicated data.

The main part of the extension is the Cache, which will provide a temporary memory for analyzed data (chapter 4). The new Transformations section will provide a processing tools for preparing the analyzed data for the visualizations on the server side (chapter 5). Visualizations implementing the Transformations serve as an example of the caching mechanism (chapter 7).

For the Payola’s User and Developer guides and source codes see the enclosed CD ROM or GitHub repository https://github.com/kudlondr/Payola-visual (the content of this repository is a fork from the main Payola repository at https://github.com/payola/Payola). For the installation guide, see the following section, or Installation guide at the GitHub repository or enclosed CD ROM.

1.3 Installation

Payola requires a Scala environment (see the chapter 2) which is supported on platforms capable of running Java code. Supported systems are OS X 10.7+, Windows 7, Windows Server 2008 R2 and additionally Payola was tested on Windows 10 Pro and Ubuntu 16.041. The application requires a relational database
for storing user data, a Virtuoso server for storing RDF data and optionally an SMTP server for the plugin approval process. The Virtuoso has to run on the same server as Payola or at least require to have a shared file system (when uploading private data, a path to a temporary file is passed to Virtuoso). The SMTP and relational database may run on different servers.

To interact with Payola an HTML5 capable web browser is required. Recommended are Webkit-based browsers (e.g. Chrome v. 57, Safari v. 5.1), Firefox v. 53, Opera v. 40 or IE 11.

Follows the steps of Payola required tools installation, source codes compilation and application launching.

1) First step of installation is to get the source codes of Payola. Those are available at git repository [git://github.com/kudlondr/Payola-visual.git](http://git://github.com/kudlondr/Payola-visual.git) or on the enclosed CD ROM.

2) Payola requires Java Developement Kit (JDK) of versions 6 or 7 (JDK 8 does not allow the Scala Actors library).

   - For Windows operating system download JDK7 ([http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html](http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html)) and install it. To use the installed JDK7 alter the `sbt.bat` script, that the second line of the script contains the file system address of the bin folder of the installed JDK7 (similarly to `set JAVA_PATH_TO_USE=C:\Program Files (x86)\Java\jdk1.7.0_79\bin`).

   - For Linux operating system install the OpenJDK7 or Oracle JDK7.

     If there is already a different JDK version installed alter the `sbt.sh` script (to check if any JDK is installed execute `javac -version` from command line to see if or which version is installed). Replace the `java` with path to the installed JDK7 version `java`. The result script might look similarly to the one in the listing 1.1

     ```bash
     /usr/lib/jvm/jdk-7-oracle-x64/bin/java -xmx1024M -XX:MaxPermSize=512M -Xss2M -jar 'dirname $0' /sbt-launch.jar "$@
     
     Listing 1.1: Altered sbt.sh script
     ```

3) Payola requires (see the [section 2.7](#)) Virtuoso server (tested with open-source Virtuoso version 7.2.4 - 2016-04-25) and relational database server (H2 database server versions 1.3.172 - 2013-05-25 and last stable 1.3.176 - 2014-04-05 were tested) installed on the same server as Payola is running (i.e. localhost). To change this configuration edit the `payola/web/shared/src/main/resources/payola.conf` file.
4) Enclosed CD ROM nor the repository does contain executable binaries of Payola. It is necessary to compile it in order to run it. Compilation of Payola is done by the Simple Build Tool (SBT) command line tool which is enclosed in Payola. To launch it open a command line (console, terminal) and navigate to the payola directory of the project, where sbt.sh (for Unix based system) and sbt.bat (for Windows system) scripts are prepared.

It may take even minutes to startup the SBT tool. It loads the structure of the Payola project and checks updates of the required libraries. Sequence of commands in the listing 1.2 compiles Payola and launches it.

```
cp project initializer run
project server run
```

Listing 1.2: SBT Payola commands

The command `cp` executes the compilation of Payola. The `project initializer run` sequence, is necessary when Payola is launched for the first time. The commands remove all data from database if there are any and construct the tables and initial data in the relational database. The `project server run` sequence starts up the Payola application server.

5) The web application is available on port 9000 for example `http://localhost:9000/` according to the application variable `web.url` in the configuration file.

- An `UnsatisfiedLinkError` exception might occur when launching the `run` command. This behavior was observed on Linux systems such as Debian 8. This is caused by a known bug in the Play framework version 2.3.9.

To work around this error compile JNotify library source codes using a prepared script.

   - To compile the JNotify library, the `build-essential` package is required. The compilation is done by provided `sbt_jnotify.sh` script.
     i. Navigate to the `payola` directory.
     ii. Alter the `sbt_jnotify.sh` script to point to the correct installation of java. This can be done just by editing the first line of the script. The result might be similar to the listing 1.3.

```
JAVA_DIR="/usr/lib/jvm/jdk-7-oracle-x64"
...```

Listing 1.3: Altered `sbt_jnotify.sh` script

---

1The scripts can be edited in order to use the correct JDK version and to change the amount of used memory. By default, Payola uses 1GB of memory (set by the `-Xmx` argument), 512MB of memory for permgen (the `-XX:MaxPermSize` argument) and 2MB stack size (the `-Xss` argument).
iii. Launch the script by `./sbt_jnotify.sh`. The script extracts the `payola/lib/jnotify-lib-0.94.zip` library and the source codes in it, compiles the library, cleans up, copies the compiled library to `payola/lib` directory and starts `sbt` console.

- Optionally the JNotify library might be just added to library path, but it is more reliable to compile it.
  i. Extract the `payola/lib/jnotify-lib-0.94.zip` archive.
  ii. Append the `-Djava.library.path=?` option to the `sbt.sh` (on Linux system) or `sbt.bat` (on Windows system) script
  iii. Replace the `?` in the script with the path to the directory in which the extracted library is (`jnotify.dll` for 32-bit and `jnotify_64bit.dll` for 64-bit Windows system or `libjnotify.so` for 32-bit or 64-bit Linux `libjnotify.so` 64-bit Linux system). The possible result command might be similar to one in the listing 1.4.

```
java -Xmx1024M -XX:MaxPermSize=512M -Xss2M
   -Djava.library.path=././jnotify\lib
   -jar 'dirname $0' / sbt-launch.jar

"%JAVA\PATH\TO\_USE\%\textbackslash java"
   -Xmx256M -XX:MaxPermSize=256M -Xss2M
   -Djava.library.path=././jnotify\lib
   -jar "%SCRIPT\_DIR\%sbt-launch.jar" %*
```

Listing 1.4: Used JNotify library

The Payola configuration file contains following options. It is an application properties file, with key-value options separated by equals sign (=) and may contain commentaries with hash symbol (#) at the beginning of a line.

- `virtuoso.server` - address of the Virtuoso server
- `virtuoso.endpoint.port` - port of the Virtuoso server’s SPARQL endpoint
- `virtuoso.endpoint.ssl` - enter true if the connection to the Virtuoso SPARQL endpoint should use SSL
- `virtuoso.sql.port` - port of the Virtuoso server’s SQL database
- `virtuoso.sql.user` - SQL database login name
- `virtuoso.sql.password` - SQL database login password
- `database.location` - JDBC URL of the relational database
- `database.user` - relational database login name
- `database.password` - relational database login password
• admin.email - Email of the admin user. A user with this email address also gets created when the initializer project is run. When a user uploads a plugin an email is sent to this address for the admin to approve the plugin, as well.


• web.mail.noreply - Email that will be used as a no-reply email address of the web application.

• mail.smtp.server - Mail server.

• mail.smtp.port - Mail server port.

• mail.smtp.user - Mail server username.

• mail.smtp.password - Mail server password.

• lib.directory - storage for 3rd-party libraries

• plugin.directory - where to store plugins uploaded by users
2. Used technologies

Following section introduces technologies used in Payola. It begins with the RDF related technologies. These are in the section 2.1 RDF, 2.2 SPARQL, 2.3 Linked Data and 2.4 OWL 2 and 2.5 RDFS. The other technologies related to the development of Payola are in 2.6 Scala and Play and 2.7 Database technologies sections.

2.1 RDF

Resource Description Framework (RDF) \[3\] is a general purpose language for representing entities (information) and their relations. These entities are uniquely identified by internationalized resource identifiers (IRIs). To express relation between two entities a property is used. It defines a relation of one entity - a subject, to another entity - an object. Properties are also identified by IRIs and with entities they form triples - two entities related to each other by a property. For example the relation that the Charles University has a faculty of Mathematics and Physics could be expressed as seen in the listing 2.1.

```
http://www.mff.cuni.cz
http://www.example.org/terms/faculty
http://www.cuni.cz
```

Listing 2.1: Example of a triple.

The relations of entities form a directed linking structure - a graph. In this graph entities are nodes and properties are edges connecting them. The properties can also be referred to as predicates, which bind two entities together. Moreover this binding can exist between an entity and a literal value as seen in the listing 2.2.

```
http://www.mff.cuni.cz
http://www.example.org/terms/founded
"1348-04-07"
```

Listing 2.2: Example of a entity, property, literal relation.

To serialize the RDF data several common formats are used, standards by World Wide Web Consortium \[4\]:

- JSON-LD - a JSON based notation (listing 2.3),

```
{
   
   "@id":"http://www.example.org/terms/school",
   
   "http://www.example.org/terms/founded":[

   "@value":"1348-04-07"
   
   ]
}
```

Listing 2.3: RDF data in JSON-LD format.
• N-Triples - a line-based format (listing 2.4).

```xml
<http://www.example.org/terms/school>
<http://www.example.org/terms/founded>
"1348–04–07".
```

Listing 2.4: RDF data in N-Triples format.

• N-Quads - N-Triples extended by an optional blank node or IRI labeling to what graph in a dataset the triple belongs to,

• RDF/XML - a XML based syntax (listing 2.5).

```xml
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:terms="http://www.example.org/terms/"
   rdf:about="http://www.example.org/terms/school">
   <terms:founded>1348–04–07</terms:founded>
</rdf:RDF>
```

Listing 2.5: RDF data in RDF/XML format.

• Turtle - a textual compact format (listing 2.6),

```turtle
@prefix terms: <http://www.example.org/terms/> .
terms:school terms:founded "1348–04–07".
```

Listing 2.6: RDF data in Turtle format.

• TriG - a Turtle syntax extension, providing various alternative ways to write graphs and triples, containing default and named graphs,

• RDFa - a XHTML structure allowing to add RDF data into HTML pages (listing 2.7).

```html
<html>
<head>
... 
</head>
<body>
... 
<h2 property="http://example.org/terms/name">Charles University</h2>
<p>Date: <span property="http://example.org/terms/founded">
1348–04–07</span></p>
... 
```

Listing 2.7: RDF data in RDFa format.
2.2 SPARQL

The standard SPARQL query language serves for retrieval and manipulation of data stored in RDF [5]. The language specification allows to define graph patterns contained in the RDF data. Most SPARQL queries are based on a set of triple patterns similar to RDF triples [listing 2.4] where each of the subject, the predicate or the object may be a variable. The pattern is matched to a subgraph of the RDF data when the subgraphs terms can be substituted with variables. The result of the pattern is then a graph equal to the subgraph [5]. For example the following query in the listing 2.8 applied on the data from the previous listing 2.4 would have a result value ”April 7, 1348”.

```
SELECT ?date
WHERE {
  <http://www.example.org/school>
  <http://www.example.org/terms/founded> ?date .
}
```

Listing 2.8: SPARQL select example.

The SELECT query from the example 2.8 is used to extract values from RDF data. The most used query in Payola is the CONSTRUCT for getting RDF graphs. A graph can be built based on a template similarly to SELECT, see the listing 2.9. The result of such a query is a list of resources.

```
CONSTRUCT {
  ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns/type>
  ?type
}
WHERE {
  ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns/type>
  ?type
}
```

Listing 2.9: SPARQL construct example.

2.3 Linked Data

Linked Data [9] is a way of publishing structured data that are interlinked. Common format technologies (section 2.1) have to be used to achieve Linked Data. To allow access to the data a query endpoint has to be set up. W3C
provides technologies for that purpose [7]. According to Linked Data note from 2006 [10] if data meet following conditions they are linked

1) Use URIs to name (identify) informations.
2) Use HTTP URIs so that these informations can be looked up.
3) Provide useful information about what a name identifies when it’s looked up, using open standards such as RDF, SPARQL, etc.
4) Refer to other informations using their HTTP URI-based names when publishing data on the Web.

For example a person named John was born in Liverpool 10.09.1940 and has a friend George. Notice the use of different datasets (Facebook and DBPedia) in the listing 2.10.

```xml
<http://facebook.com/John>
<http://dbpedia.org/ontology/birthPlace>
<http://dbpedia.org/page/Liverpool>

<http://facebook.com/John>
<http://dbpedia.org/ontology/friend>
<http://facebook.com/George>

<http://facebook.com/John>
<http://dbpedia.org/ontology/birthDate>
"1940-09-10"
```

Listing 2.10: Example of linked data sources.

This is the data structure part of Linked Data (N-Triples format). The other part is data access. Linked Data have to be accessible via a query-able endpoint. To meet the definition of Linked Data the data from the example have to be accessible via a standard e.g. SPARQL endpoint.

A typical case of Linked Data set is DBPedia [http://wiki.dbpedia.org/] which provides Wikipedia (https://www.wikipedia.org/) data in RDF and includes links to other data sets. Applications using DBPedia might follow these links and provide possibly more precise knowledge and better user experience [7].

**2.4 OWL 2**

As RDF is for representing information, OWL2 is an RDF model that describes the RDF data formally - for example extending it by data types [6]. An ontology is an RDF data description that is based on the OWL2 RDF Semantics. The OWL2 is an extension of the Web Ontology Language (OWL) which is a semantic markup language for publishing and sharing ontologies. These are formalized vocabularies of terms. It is developed as a vocabulary extension of RDF [3] where any OWL 2 ontology can be viewed as an RDF graph. OWL 2 is a backwards compatible extension of OWL 1. It serves as a definition of a structure of knowledge. It can be serialized into various syntaxes.
• RDF/XML
• OWL/XML - a XML based syntax (listing 10.1)
• Turtle
• Manchester - a textual syntax (listing 10.2)
• Functional - a textual syntax (listing 10.3)

2.5 RDFS

The Resource Description Framework Schema (RDFS) is an extension of RDF. It provides a mechanism for describing groups of resources (information) and relations between them. It contains a class and property constructs, where a resource is an instance of class and each class can be an extension of another class (forming a parent-child hierarchy)[8]. The root of the hierarchy is rdfs:Resource class. The rdfs:Class resource is a parent class for resources that are classes, e.g. foaf:Person.

Properties are instances of class rdf:Property and describe relations between resources - a subject and an object by

• rdf:domain - a class of the subject,
• rdf:range - a class of the object.

This structure allows to define triples according to the property definition. For example to expresses the information that Faculty of Mathematics and Physics (ex:FacultyOfMathematicsAndPhysics a type of foaf:Faculty) is a part of Charles University (ex:CharlesUniversity a type of foaf:University) it could be described by structure in the listing 2.11.

```
ex:isFacultyOf rdfs:domain foaf:Faculty
ex:isFacultyOf rdfs:range foaf:University
ex:FacultyOfMathematicsAndPhysics
  ex:isFacultyOf
  ex:CharlesUniversity
```

Listing 2.11: RDFS property example.

2.6 Scala and Play

The original Payola started as a .NET software programmed in the C# language. But with the shift to an abstract data analysis tool a functional programming language was chosen and at that time the base of Payola was reconstructed in Scala programming language.

FOAF is a Friend of a Friend vocabulary built upon RDF vocabulary.
Scala is a functional object-oriented language. Conceptually, every value in it is an object and every operation is a method-call. Moreover every function is an object. It supports component architectures through classes and traits\(^2\) and has a native support for design patterns such as singletons or visitors. Scala written code is executed in Java Virtual Machine and it can use Java written frameworks and libraries \([12]\).

Play \([11]\) is an open source web application framework for Java and Scala with support of HTML5 \([13]\). It follows model-view-controller architecture. The framework provides stateless client-server communication based on JBoss Netty web server. It provides its interfaces in both Java and Scala to fit conventions of both languages. Its core is written in Scala language and provides its native support since version Play 2. It uses Simple Build Tool (SBT) for dependency management, hence the usage of SBT for Payola dependency management and compilation.

When Payola was rewritten in Scala a technological experiment was conducted. The user interface code is written in Scala language and during a compilation a Payola component - S2JS, translates it to JavaScript. This way all source code was written in Scala, even user interface. S2JS serves as a code transformation tool, that translates Scala code of the client side application to JavaScript. It is used during application compilation by SBT and constructs the JavaScript from the Scala code.

### 2.7 Database technologies

To communicate with database server Payola uses Squeryl library \([14]\). It is an ORM for Scala that allows compilation time database statements check. A database query is defined using functions without plain SQL written in string.

Squeryl is used to communicate with application database. H2 is a relational database management system (RDBMS) used in Payola. It is open source, Java written and can be embedded into an application or run in the client-server mode. In Payola, it serves as a storage for application data \([15]\).

Virtuoso is a database engine combining RDBMS, RDF, XML or file server functionality in a single system \([16]\). Open source variant (OpenLink Virtuoso) is used in Payola to provide a standalone RDF data endpoint and container.

\(^2\)A class in Scala is a static template that can be instantiated at runtime. A trait is a similar construct as an interface in the Java language. It is used to define object types by specifying the signature of the supported methods with its partial implementation \([12]\).
3. Payola structure

An introduction to the structure of Payola follows to describe where this extension of Payola is placed and how it interacts with the rest of the application. The sections Workflow (section 3.1) and Components (section 3.2) introduce the previous state of Payola. The section 3.3 Payola extension introduces the motivation to create the extension and describes where it is placed into Payola.

3.1 Workflow

Payola can be divided into following logical parts.

- Client Web - the user interface,
- S2SJ - compilation tool transforming the program code written in Scala code into JavaScript, used for Client Web,
- Model - inner data structures,
- Server - business logic which can be divided into Analyses, Data Sources, Plugins, Users, Prefixes and Customizations,
- Database - processing and fetching of model objects.

The following example (figure 3.1) shows the usual workflow reflected in the application structure. User inputs the address of Payola into a web browser. Payola web interface uses the Play framework which creates a response and sends the main page of the application to the client. Then navigating to the login page user can send the credentials. The server side User component verifies the user and returns model objects, that the user owns or has available in Payola. The web interface returns a web page with public endpoints - data sources, public analyses and the analysis that the user has created or has access to (see figure 3.2). Navigating to one of the analysis details performs a call to the Analyses component (see figure 3.3). Launching the analysis starts an asynchronous evaluation of a SPARQL query (one or multiple queries), that the analysis represents. Client side - user’s web browser, calls periodically the Analysis server component to check the status of the evaluation. When it is completed, the analysis detail page is switched to the visualization page. User selects a visualization plugin suitable for the data. The chosen plugin receives the whole graph constructed by the analysis and prepares it for visualization.

When user selects a visual plugin the Presenter has already received the data from the server and it passes them to the selected plugin, for example the Graph plugin. It allows the user to view the data as a graph, connecting RDF entities (vertices) by RDF properties (edges) showing the graph as a tree or circled structure (shown in the figure 3.4). The interaction with the visual plugin is performed solely in the Client Web without any communication to the server.
All the graph visualizations are JavaScript implemented Payola plugins, that are part of the Client Web - running in the user’s web browser. Transferring the graph to the visualization and processing it in Javascript prolongs the time in which user is able to see and work with the analysis result.

The Triple Table View plugin processes graphs analogously to the Graph view. User can choose it, to visualize the result of the analysis. Plugin receives the whole graph to process it and show it to the user. The graph is transferred from the server to the web browser into an internal JavaScript structure, where it is transformed for presentation. Preparing pages of the Table view, creating a HTML table elements for the visualization.

### 3.2 Components

The whole application is spread into eight main components. Whose structure is shown in Figure 3.5.

- Common - abstract entities for the whole project,
- Domain - basic implementation of Common,
- Data - database layer,
- Model - application business logic,
- Shared - server and client common tools,
- Server - server side of web application,
• Client - client side of web application,
• Initializer - initialization tools for application startup.

The component Common contains classes that are available to the whole application, an abstract definition of entities shared and transmitted between server and client side. Those are for example User, DataSource, Analysis, Ontology-Customization, Graph or Privileges. This sub-project focuses on definition of objects with a basic and common interface for all parts of the application and their relations.

The Domain project is an extension and an implementation of the Common. It adds functionalities independent on the Payola project only using the Common component. This project adds a plugin manipulation and creation tools - it contains analysis logic, plugin definition tools and plugin compiler. For details see [1].

The Data component contains definitions of database layer. DataContextComponent is the interface for persisting and fetching data into and from the Virtuoso graph storage and the application entity database. This component contains implementation of the Squeryl ORM and also java.sql.Virtuoso library.

Model represents the business logic of the application. Using dependency injection from Play framework it represents the communication interface between the Data and Sever, Client and Shared components.

The next component is Web, which is a container for the Server, Client, Shared and Initializer components. It is a representation layer implementing the Play
framework. The Server subcomponent contains the server side part of the web application. Providing remote process called functions to the client side. Client defines the web application which is an user interface of Payola. It defines use case Presenters and graph visualization plugins. The code from this sub-project is converted via S2JS to JavaScript to serve as plugins in the Play framework built web presentation. The Initializer is responsible for application database initialization. It can be used during application installation or to cleanup its database by the SBT tool (see [2]). Finally the Shared component contains remote tools for both the server and the client sides, with the code being executed on the server via RPC. It also contains the Payola object which is an access point to the application model. This object allows the client side to access remote functions and data. For the server contained components it allows to access Model and the Data component.

### 3.3 Payola extension

The structure of Payola components that are related to this extension are AnalysisRunner, Presenter and View. User interacts with the application via View in the client side. This uses the Presenter to create analyses and receive data from completed analysis executions (see figure 3.6).

This extension focuses on the server-client communication which reflects on the application model. As Presenter communicated with AnalysisRunner in the previous version of Payola in the modified one it communicates with the Transformations component. It contains preprocessing tools of data for View.

The workflow modification begins at View requesting an analysis creation or execution via Presenter. When an analysis execution is completed the data (result...
Figure 3.4: Graph plugin showing analysis result

to the Cache component and AnalysisRunner returns only the identification of the result. This identification is afterwards used by Presenter to get the stored data via Transformations. It fetches the graph using Cache and prepares it for the Presenter (figure 3.7).

To loosen the requirements on the memory of the user’s computer and the web browser a server side pre-processing layer is suitable. This results into the Transformation component (chapter 5) - a server side visualizations support. Adding such a layer allows visual plugins to perform their graph processing operations remotely on the server. But having only this Transformation rises immediately the following question. As this component provides graph fetching and manipulation, from where it would take the data? If a visualization should show a result of an analysis, it will need the result to be available for each preprocessing request? For example when user lists through pages of the Triple Table view, the layer might provide an operation for fetching another page of an analysis result. Not having the result graph prepared and stored in memory, it would require the analysis to be repeatedly performed. Requiring the user to wait after requesting different page of a graph in the Table view.

Another application layer resolves this problem. A layer that provides a storage functionality and persist graphs after analysis evaluation and returns stored data when necessary. For example when a transformation requires a graph to return its next page for the Triple Table view. This is close to a caching mechanism - temporary storage of computed data that allows the future requests for the same data to be served faster. Hence the name of the layer is Cache.

The desired parts should extend the Model, Shared and Data Sources components and alter Analyses. The final workflow of user executing an analysis with

\(^1\) Other visual plugins are Graph view, Sigma2 view or Column chart. More detailed description of visual plugins is in chapter 7.
Cache and Transformations is shown in figure 3.9 (previous workflow shows figure 3.8). First user executes an analysis. The result of the execution is stored into the cache and forwarded to Transformations to determine which Transformer and Visual Plugin is available to visualize the analysis result (graph). User selects a visualization which calls one of Transformers to prepare the graph. Cache returns the graph to Transformer which prepares it and sends it to the visualization.
Figure 3.6: Communication diagram between user and analysis without cache.

Figure 3.7: Communication diagram between user and analysis with cache.
Figure 3.8: Communication diagram workflow.

Figure 3.9: Communication diagram workflow with Cache and Transformations.
4. Cache

This chapter describes the Cache extension, as introduced in the previous chapter 3. The first motivation to implement the caching extension was to increase visual plugins responsiveness. When visualization processes a graph consisting of over hundreds of thousands of nodes without caching mechanism, the whole graph has to be transferred to the user’s web browser and prepared in its memory for presentation. With caching the graph can be transferred by parts on demand, which would allow more responsive user interface.

The second reason that was suggested in chapter 3 is that a server side layer could provide a data preprocessing for visualizations. A layer that would filter, transform or even extend the data of an analysis result depending on the visualization plugin. It might provide data listing - sending sections of the graph on demand. Without temporary memory for graphs every request would require another analysis execution repeatedly.

Adding the visualizations server side support layer would move the data processing from the visual plugins to the server. Lowering memory demands from the user’s web browser making the visual plugins a plain tool for visualization - see chapter 5.

4.1 Requirements

The required Cache extension should meet the following points. To see a use case describing application with Cache and Transformations (the chapter 5) navigate to the section 5.4.

1. Provide a layer over the Virtuoso database for analysis result graphs storage.
2. Add an application database layer extension for persisting descriptive data about graphs, that are stored in the Virtuoso database.
3. Uniquely identify each graph that is stored in the Virtuoso database.
4. Alter Analyses component that when an analysis execution is completed the result graph is stored into the Virtuoso database.

Expected workflow with Cache starts at analysis being completed. The mechanism of graph data processing and analysing in Payola may be performed by a single SPARQL query working with only one data endpoint. On the other hand to support gathering data from multiple sources one might want to construct a single query for example with Open Data (http://linked.opendata.cz/) and DBpedia (http://wiki.dbpedia.org/) data joined in it. This feature is supported in Payola by processing the graph in application memory. When such an analysis evaluation is finished the whole result graph is kept as a data structure in Payola’s memory. At that point the graph can be stored in the graph database as an identified RDF data to an instance of a Virtuoso database. When the graph is
stored via the Cache layer, the Analysis layer can return only the evaluation id to
the client side - unique analysis execution identification. This id has to be unique
even for different executions of the same analysis, since the data in the analysis
sources might change in time. Once the client side possesses the evaluation id
the graph can be fetched from Cache.

As the analysis result is stored in the database, there has to be an information
about the graph also persisted with it to keep track of the cached graphs. These
information might consist of the evaluation id, time of the analysis execution - to
determine its age, or whether the graph has to be kept longer in the cache than
usually. This requirement might seem unnecessary, but since the stored graphs
can be vast it would not be possible to fetch all the stored graphs into application
and tell one from another. To lower requirements on the graph storage database,
splitting these information from the graph and persisting them in the application
database is a more transparent way of handling the stored data.

4.2 Implementation

The first step for the Cache extension is a transparent interface for commu-
nication between Payola’s components. The new layer has to be part of the
data.DataContextComponent trait - a partially implemented Scala class or an
object. This trait is implemented via data.squeryl.SquerylDataComponent
object that is used in the web.shared.Payola model object, that represents the
access point for remote process calls from the client side and communication be-
tween layers of Payola. This layer can also be called from the AnalysisRunner
when an analysis execution is completed a the result transferred to the Virtuoso
database (requirement item no. 4).

The new model.components.AnalysisResultStorageModelComponent trait
provides access to the Virtuoso storage (required item no. 3). It identifies the
stored graphs uniquely by the evaluation id which is generated by the Analysis
component (required item no. 3). It is accessible via the DataConetextComponent
trait. It is an interface for work with graphs in the Virtuoso database. It en-
capsulates also the mechanism to store graph processing related data using the
AnalysisResultRepositoryComponent trait from package
data.squeryl.repositories and it is a part of DataContextComponent. This
provides functions for the analysis result storage to persist information about the
user, who executed the analysis, the date the graph was stored, the number of
nodes, the graph contains (required item no. 2). This part of the cache allows to
monitor which user has how many stored graphs, which graphs are too old to be
cached and what sizes do the graphs have.

The AnalysisResultRepositoryComponent trait has to access the SQL ap-
lication database. That is allowed by implementing and using Squeryl func-
tions in the data.squeryl.TableRepositoryComponent trait. Using the Squeryl
the information are stored into the H2 database. When another layer of Pay-
ola, for example the client side, requests graph from the Cache, the informa-
tion about the graph are fetched and added to the retrieved cached data - domain.entities.AnalysisResult entity.

Because of the Embedding extension [chapter 6](#), it was decided during the development to leave the graph database management to the users and add an administrative page for the Cache management and configuration in the future.

### 4.3 Programmatic usage

This extension can be accessed only via its functions provided to other components of Payola. The AnalysisResultStorageModelComponent trait from package model.components defines the Cache API provider. Follows a list of its main functions.

- **saveGraph**
  ```scala
  def saveGraph(graph: Graph, analysisId: String, 
                evaluationId: String, user: Option[User] = None, 
                embeddedHash: Option[String] = None): Unit
  ```
  Inserts a Payola model Graph structure by VirtuosoStorage the database. It identifies the graph by the provided evaluationId and it saves the rest of the information via AnalysisResultRepositoryComponent to the application database.

- **exists**
  ```scala
  def exists(evaluationId: String): Boolean
  ```
  Checks the existence of a graph in the application database.

- **analysisId**
  ```scala
  def analysisId(evaluationId: String): String
  ```
  Gets the identification of an analysis from the application database whose result is identified by the evaluationId.

- **getGraph**
  ```scala
  def getGraph(evaluationId: String): Graph
  ```
  Gets a stored graph from the Virtuoso database by performing a construct query. This function is polymorphic and can be called with different parameters.

- **getGraph**
  ```scala
  def getGraph(sparqlQuery: String, evaluationId: String): Graph
  ```

- **getGraph**
  ```scala
  def getGraph(sparqlQueryList: List[String], 
               evaluationId: String): Graph
  ```

- **removeGraph**
  ```scala
  def removeGraph(evaluationId: String, analysisId: String): Unit
  ```
  Deletes the graph from the Virtuoso by VirtuosoStorage and its information by AnalysisResultRepositoryComponent.
The inner class `data.virtuoso.VirtuosoStorage` accesses the Virtuoso graph database and encapsulates SPARQL queries for graph insertion, deletion and selection. It uses the `virtuoso.jdbc3.Driver` library for accessing the database.

- **storeGraphGraphProtocol(graphURI: String, graph: Graph)**

  Converts the graph into RDF/XML representation and stores the graph into the Virtuoso database at the specified IRI.

- **deleteGraph(graphURI: String)**

  Performs deletion of a graph stored at a specified IRI using SPARQL.

- **executeSPARQLQuery(query: String): Graph**

  Performs a query based on its type Construct or Select and converts the result data to model.Graph representation from RDF/XML or RDF/N3 (non-XML data) using Jena library.


  Performs the same operation as the previous function but converts the result Graph into Jena library Dataset structure.

The trait `AnalysisResultRepositoryComponent` added into the database package `data.squeryl.repositories` contains definition of function to access data inside the application database. It is focused on work with a single entity (as the other Repository Components are) which is `AnalysisResult`. This entity contains information about graphs stored inside the Virtuoso database. The Repository Component allows to work with the `AnalysisResult` using following functions.

- **def storeResult(analysisDescription: domain.entities.AnalysisResult, uriHash: Option[String] = None): Unit**

  Inserts or updates the entity inside the application database.

- **def getResult(evaluationId: String, analysisId: String): Option[AnalysisResult]**

  Gets the entity from the database using the analysis and evaluation ids.

- **def deleteResult(evaluationId: String): Unit**

- **def updateTimestamp(evaluationId: String): Unit**

- **def exists(evaluationId: String): Boolean**

- **def getAllAvailableToUser(userId: Option[String]): Seq[AnalysisResult]**

  Gets all entities that are owned by the user or are without a owner (public).
When a graph is stored in Cache a new description `AnalysisResult` entity is persisted and when a graph is removed so is the entity. The `AnalysisResult StorageModelComponent` does not allow for a stored graph to exist without its description entity. It contains the following information about a graph.

- `analysisid` - graph analysis identification whose execution created the described graph,
- `owner` - which user created the graph (if logged into Payola during analysis execution),
- `evaluationid`,
- `verticescount` - number of vertices to quickly determine the size of the stored graph,
- `touched` - time stamp of last update of the data.

### 4.4 Evaluation

Follows a comparison of response times and amounts of transferred data between Payola with and without Cache. To compare its full functionality the measurements include Transformations and the Triple Table visual plugin. The Triple Table plugin was chosen since its user interface has not changed compared to the previous version of Payola.

All the measurements were conducted using following configuration of a single computer:

- Intel i5-2410M CPU
- 2 modules of 4 GBytes 1333 MHz SDRAM DDR3
- Windows 7 Professional with Service Pack 1 OS
- Firefox web browser version 47.0
- H2 database server version 1.3.172
- Openlink Virtuoso open-source edition version 6.1.6

DBPedia.org was used as a datasource to fetch maximum of 10000 entities of `Person` and `City` type ([listing 4.1] show the SPARQL queries).

```
CONSTRUCT { 
  ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> 
  <http://dbpedia.org/ontology/Person> 
} WHERE { 
  ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> 
  <http://dbpedia.org/ontology/Person> 
```
Listing 4.1: SPARQL queries used on DBPedia endpoint.

The [figure 4.1] shows measured response times. These values are sums of processing times between analysis completion and visual plugin displaying analysis result. The variant without contains following steps

1. analysis result data transmission from server to client,
2. data processing in visual plugin and
3. data visualization.

For the variant with Cache the steps are

1. analysis result data storage into Cache,
2. processing in Transformations,
3. data transmission to client and
4. data visualization in a plugin.

In the [figure 4.1] are shown sizes of transferred data between server and client - Transformations and visual plugin. Since the previous version of Payola transferred all of the analysis result data to client the difference is significant. On the other hand using Cache and Transformations Triple Table plugin sends a request to Transformations every time user navigates to different page of the list. In the previous version of Payola in this scenario no data are transferred between server and client.

4.5 Further extension

For the possibility of future development of Payola the creation of an administrative web page was left. The Embedding Support extension [chapter 6] adds a simple usecase of Cache monitoring. Since it displays only graphs created by the logged in user and serves for the embedding mechanism no administrative options are available in it. An application web page for monitoring Cache content, size and data removal times would complete this extension.
Figure 4.1: Response times comparison.
5. Transformations

The Cache extension does not provide any user interaction tools. It only provides API for other Payola components. To use this component directly from visual plugins would not accomplish the required behavior. Implementing SPARQL queries into visual plugins would deny the Payola’s cache extension being transparent to the client side. Adding the queries directly into the Cache would deny the Payola’s transparent architecture. Transformations layer serves as a unified interface between visual plugins and Cache. It serves as a server side communication interface, separating the data preprocessing tasks from the visualizations.

The Transformations part is a container for multiple Transformer elements. Each of them is a toolbox providing functions to fetch RDF data from Cache or any different datasource for visual plugins. First intention was to create a server side support tools for every visual plugin. But since users can create their own visual plugins in Payola, creating more abstract independent tools, that can be reused by all visualizations, follows the Payola’s architecture.

5.1 Requirements

Transformations layer should serve as a communication tool between Presenter and Cache components. Hence that the extension should firstly focus on defining an API. Secondly it should replace graph preprocessing functions implemented in the visualization plugins. Follows a list of requirements, to see the implementation of each requirement go to the section 5.3 or the section 5.4.

1. Define a common interface for individual transformation elements (Transformators), that allows to separate data preprocessing algorithms from visual plugins.

2. Allow each Transformator to determine whether it can or can not transform data - simulating the same behavior when a visual plugin can not display a graph, e.g. because of its structure.

3. Alter current visual plugins, that each can implement one or multiple transformations.

4. Define a container which will provide a list of available transformations.

The transformations should be small bundles of specialized or common tools for visual plugins. Any visualization should be able to use any of the transformations and any number of them (requirement item no. 3). For example in a case, that a transformation cannot be used for a specific data structure outputted by an analysis, the visualization might use another available transformation. If all of visual plugin’s required transformations are unavailable (can not process a certain graph) the visual plugin will be disabled. Possibly all visual plugins could be disabled this way. For this situation there will be added an empty visual plugin, that will inform the user that there is no available visual plugin to display data.
The implementation of each transformation should meet the main point of creating the Transformations extension. It should be able to send sections of data or subgraphs to visualizations - subsets of the whole graphs with limited number of entities. Getting subgraphs from transformations would allow visualizations to display only sections of the entire graphs. User could then specify which section of the graph is required. For the case that user would require the whole graph an identity transformation should be provided for such visualizations.

Finally the Transformations should not forbid a data extending functionality. For example having an analysis result containing list of cities, one might want to fetch GPS coordinates of those cities if the analysis datasource does not provide them. This could be a visual plugin specific transformation.

5.2 Implementation

The visual plugins access the transformations via analysis results. When an AnalysisRunner object finishes an execution and performs a success callback the result graph is stored into the cache. All GraphTransformator trait implementations contained in the TransformationManager object check its availability and it returns those transformations that can be used for the cached graph. More specifically all GraphTransformator implementations call asynchronously a success callback from its isAvailable function implementation.

The AnalysisRunner then returns a container with the Transformators and an analysis execution identifier to the PluginSwitchView. That is a client side class which is a container of all visual plugins. The PluginSwitchView is switched to from the completed analysis view - presenter AnalysisRunner, and calls the setAvailablePlugins function of PluginSwitchView to update the list of visual plugins that can display the graph. Each implemented visual plugin checks its availability by verifying whether any of the required Transformators is available. The PluginSwitchView then initializes a preferred visual plugin or the first one from the list of available ones.

At this point the visual plugin calls its Transformator and gives it the analysis execution id - depends on the implementation of the specific plugin. The Transformator calls the Cache to fetch the graph using the execution id and prepares a sub graph, that is returned to the visual plugin. Finally the plugin visualizes the graph. The components interaction is show in the figure 5.1.

\[\text{Callback is an asynchronous way of returning data from a remote process call. When a remote object function is called from the client side, the call is also provided with a success and error callback functions, that are performed when the remote call is completed successfully or if an error occurs otherwise. For more details see the Payola Developer Guide.}\]

\[\text{Preferred plugin can be selected for embedded analysis results (see the chapter 6).}\]
5.3 Visual plugin implementation

In this Payola’s extension the following visual plugins were extended of the Cache and Transformations implementation as an example:

- Triple Table View
- Graph View
- Sigma2 View

To describe the inner mechanics of Transformations the Triple Table and its graph pagination was chosen. To see the inner mechanics of the implementation of the other plugins see the chapter 7.

The Triple table visualization (figure 5.2) plugin displays the graph as a paginated table of triples (for details see the section 7.1). Without the caching extension, the list has to be created on the client side. Even though that the visualization’s data preprocessing consists only of counting and sorting of the triples and creating a HTML table listing a subset of them. Transferring a graph consisting of hundreds of thousands or even millions of relations could prolong
displaying a single page of the list. The plugin has to construct all the HTML tables of the individual pages and keep them in memory or keep the graph in memory and construct the pages on demand.

The pagination algorithm is reimplemented in the `TripleTableTransformator` object, an implementation of `GraphTransformator` trait (the requirement in the item no. 1). Since this visual plugin does not require the visualized graph to be of any specific structure, this transformation is available for any analysis result graph. When an analysis execution is successfully completed the result graph is stored into the Cache. `AnalysisRunner` object then calls `TransformationManager getAvailableTransformations` (the requirement in the item no. 4) function to determine which transformations can be applied for the graph (code detail is in the listing 5.1). Its functions `isAvailable` are called and depending on the implementation and the graph’s structure the specific transformation returns true or false - if it can be used for the graph or not (the requirement in the item no. 2). The `TripleTableTransformator` is available for all graphs thus it is contained in the set returned by `getAvailableTransformations` function. `PluginSwitchView` than allows to display the visual plugin as available and all other plugins that implement the `TripleTableTransformator` (the requirement in the item no. 3).

```scala
val resultResponse = 
  ...
Payola.model.analysisResultStorageModel.saveGraph(...)
```
The `TripleTablePluginView` can perform a remote call to `getCachedPage` function of `TripleTableTransformator` when user requests another page of the entities of the visualized graph (notice the controls at the bottom of the figure 5.2). Each request sends evaluation id to identify the analysis result stored in the cache. The Transformator performs a request to the cache getting the specific part of the graph by a SPARQL query, leaving the sub graph processing on the Virtuoso database. The graph division is performed in the database and the application server does not need to manipulate the data. The result can be sent to the plugin as the database returned it.

### 5.4 Use case scenario

To continue with the example of the Triple table plugin from the section 5.3 from user’s perspective a use case example follows to present the extension for users.

1) From the Payola dashboard navigate in the **Accessible analyses** list to **Big cities with countries** analysis detail page (see the figure 3.2).

To use the Cache and Transformation extensions, user does not need to be logged into Payola.

2) Launch analysis by **Run analysis** button (see the figure 3.3).

At this point the Cache and Transformations extensions are used. When the execution is completed the analysis result is stored into the Virtuoso database by the Cache extension and all the transformations determine, whether they can or can not be applied to the execution result graph. The list of available transformations determines which visual plugins are available. User can choose only from those.

3) Select the Triple Table plugin from the list of available plugins (**Change visualization plugin** button; see the figure 3.4). The visualization displays the result of the executed analysis.

At this point the extensions are used again. Selecting the Triple Table the visual plugin performs a remote call to the transformation which fetches a subgraph using the Cache extension and finally the result of the analysis execution is displayed (as shows the figure 5.2).

```java
    getAvailableTransformations(r.outputGraph)

EvaluationCompleted(  
    availableTransformators, r.instanceErrors)
...
successCallback(resultResponse)
```

Listing 5.1: AnalysisRunner code snippet.
4) Using the pagination buttons (the Next, Previous, etc. buttons) the extensions are used again. For each pagination request a communication to the Transformator and Cache extensions is performed and fetches a subgraph from the Virtuoso database.

### 5.5 Programmers documentation

The transformations are remote (server side) objects, that provide data pre-processing functions for visual plugins. Each transformation has to implement the API of `cz.payola.web.shared.transformators.GraphTransformator` trait. The implementations of the trait can provide a visual plugin specific functions. The trait defines the following interface:

- **def transform(evaluationId: String)**
  
  (successCallback: Option[Graph] => Unit)
  (errorCallback: Throwable => Unit): Unit

  Via remote call SuccessCallback returns a graph identified by the evaluationId, that is fetched from the Cache or other datasource. If an error occurs remote call ErrorCallback is performed.

- **def isAvailable(input: Graph)**
  
  (successCallback: Boolean => Unit)
  (errorCallback: Throwable => Unit): Unit

- **def isAvailable(input: Graph): Boolean**

  Visual plugins might require a specific input graph structure. This function checks the graph structure or its properties, that the visual plugin requires. A remote (called from client) and a server side only variants of this function are available.

- **def getSampleGraph(evaluationId: String)**
  
  (successCallback: Graph => Unit)
  (errorCallback: Throwable => Unit): Unit

  Returns by a remote call an initial graph. For example for the Triple table it is the first page of the table.

The `web.shared.transformators.TripleTableTransformer` implementation is an example to the available functions of the transformations API. It performs a remote call to function `getCachedPage` which encapsulates a SPARQL query call to the Cache and paginates the graph data. This is a visual plugin specific operation and substitutes the graph pagination implemented in the Triple table plugin of the previous version of Payola.
6. Embedding support

The Cache extension serves as a temporary graph database. When an analysis execution is completed and its result is ready for visualization, the graph is stored into the Cache and accessed via Transformations from visual plugins. Storing the graph only for the time it is required. But since the database cleaning is left to be managed by users, the Cache can be used in a different way. Storing the graphs in the database and providing direct links to them can be used to embed analyses results in other web pages and applications.

Users can generate with such a tool unique URIs to stored graphs and use those as a link not only to Payola but to an already executed analysis result and a specific visual plugin previously selected by user. Visiting this link would then directly fetch the graph from the cache without any need of repeating the analysis execution and for the user to wait for the data to be prepared. A use case example is described in the section 6.4.

6.1 Requirements

Follows a list that describes user and application requirements of this Payola’s extension.

1. Create a web page for logged in users to manage cached graphs.
2. For each stored graph display links to its analysis, evaluation and the embedded variant of the graph.
3. Allow deletion of stored graphs and updating of evaluated data for embedded ones.

To support embedding of Payola analysis results a management page should be added and accessible for logged in users (the requirement item no. 1). Those then could view their analyses results currently stored in the Cache. The page should display for each graph a direct link to the analysis which created that result also with a link to that execution result. There should be an option to remove a cached graph and its description info from the application database.

The most important part of the management page should be an option to generate an embedding URI for particular stored graph (item no. 2). This URI should be shorter than the full link to the execution graph result. Also when the URI is generated it should be possible to set a preferred visual plugin that would be used to display that analysis result by default. Also a default public customization should be possible to pick. Finally for embedded graphs there should be a way to update an analysis result - keeping the generated URL but executing the analysis and linking the new result with that URL (item no. 3).
6.2 Implementation

This Payola extension adds a management page with options for controlling data of the Cache extension (chapter 4). The web page is defined by the list.scala.html file used by the Play framework. The page contains a list of AnalysisResult entities stored in the application database. Only users logged into Payola can display this page and the entities on this page are only those created by the user. Displayed stored graphs are sorted by time stamps of their last update.

User can create an embedding URI for every stored graph by selecting the Create link (see the figure 6.1). This generates a unique hash code used as a shorter URI for the full link to the graph. This operation is done in the CacheStore object, which also provides a getter function of the stored graphs and a delete operation for removing the graph from Virtuoso and application databases. For every embedded URI, user can set a preferred visual plugin that is switched to directly, when the embedding link is used. Setting such a plugin is done via CacheStorePresenter remote object that calls EmbeddingDescriptionRepositoryComponent functions to persist AnalysisResult entities to the application database. The object are used when user select a default visual customization. Only public visual customizations can be selected for embedded graphs. The selected customization is used for visualization of the data.

The embedded graphs can change using the Update option making an analysis result more recent. When the graph changes in such a way that a selected preferred visualization could not display it the embedded link might become invalid. That is why the PluginSwitchView switches between the visual plugins. The first plugin in this scenario is the selected one for the embedded URI. But when this plugin can not display the graph, another plugin - next in the order, is selected. This leads to leaving the preferred visualization of the embedded graph to the user.

To enable the embedding URIs to link to a particular analysis result graph the application routes configuration file is altered. Linking an embedded URI to the CacheStore embed function, which reconstructs the hash URL to full link. Since the embedded URI is unique, the EmbeddingDescriptionRepositoryComponent searches it in the database and the full link is reconstructed and a web page redirection is performed. The full link contains information about the analysis result - the evaluationId and the selected visualization plugin if any.

6.3 Usage

The embedding support is available for logged in users. In the main menu of the application is a My Stored Results link to the page. This page shows a table
of analyses execution results that the user performed. The list contains links to the executed analyses, last update timestamps and links to the result graphs (see the figure 6.1). These links are all the execution results stored inside the Cache database that the user created. The embedding mechanism is accessible via the Create button which generates a unique code that is a shorter version of the full link to a result graph. For each stored graph with an embedded address a default visual plugin selector is available. When a plugin is chosen the graph is visualized using that plugin when the embedded address is visited. Additionally for each embedded graph a default public visual customization can be selected. This customization is applied for Graph or Sigma2 plugins. Each graph with an embedded address has also an option to refresh itself by Update button. It reruns the analysis execution and replaces the old result with the new one. The page also allows to remove stored execution results via Delete buttons.

6.4 Use case scenario

The embedding feature is available only to logged in users. If user is already logged in, follow the scenario from step 1.

i) From Payola dashboard navigate to the login page Sign in, in the top right corner of the page (see the figure 6.2).

ii) Input user email and password and click Sign in or create an account by clicking Sign up.

1) In the header menu click the My Stored Results link to navigate to the list of stored graphs.

This page displays all graphs stored in cache that the logged in user has created (see the figure 6.1). A new stored graph is added when user executes an analysis.

2) To execute an analysis go back to the dashboard, or click the Payola logo or Dashboard in the header menu.

![Figure 6.2: Payola login page.](image-url)
3) Navigate to the **Big cities with countries** analysis in the **Accessible analyses** box and press **Run Analysis**.

This adds the analysis result to the cache which is now visible in the stored results list.

4) Navigate to the **My Stored Results** via the header menu.

The **My Stored Results** list is sorted by the date of last update of the stored graph. The result of the just executed analysis is the first from the top. The name of the analysis is a link to the analysis detail. The link in the **Result Url** column is a direct link to the analysis result.

5) To get a link for embedment click the **Create** in the **Embedded Url** column.

6) To set Triple Table visualization as a default visual plugin select it from the list in the **Default visual plugin** column.

The embedded URL now points to the stored analysis result and when the URL is used, it is visualized by the Triple Table plugin.

7) When the embedded graph should be updated (for example when the data in the datasource are changed), use the **Update** button in the **Actions** column.

This button executes the analysis and updates the stored result. The embedded URL is redirected to the updated graph and the **Result Url** is changed.

8) To remove the graph from the cache and disable the URLs pointing at it use the **Delete** button in the **Actions** column.
6.5 Programmers documentation

Since Payola uses the Play framework the structure of this part of the extension is based on it. The file `sever\app\conf\routes` specifies the URI patterns with HTTP methods associated to Action calls. Each class defining actions implements `play.api.mvc.Controller`, in this case `controllers.CacheStore` Scala object which defines following methods.

- **def list(page: Int = 1)**

  Using `Payola.model.analysisResultStorageModel` it gets all user accessible data and visualizes the `views.html.cachestore.list` web page.

- **def delete(id: String)**

  Allows to remove a stored result from virtuoso and application databases using `Payola.model.embeddingDescriptionModel.removeByAnalysisId` function.

- **def create(id: String)**

  Via `Payola.model.embeddingDescriptionModel.createEmbedded` creates an URI for embedding.

- **def embed(uriHash: String)**

  Translates short embedded URI by `Payola.model.embeddingDescriptionModel.getEmbeddedAnalysisResult` to a full URI and redirects to it.

The functions of `CacheStore` object are used from the web page definition in `web\views\cachestore` folder in `list.scala.html` file. It specifies the structure of the web page similarly to JavaServer Pages (JSP) or JavaServer Faces (JSF). `web.client.presenters.entity.cachestore.CacheStorePresenter` is used for rendering individual records of the list. It is initialized by a page request and during Payola compilation it is translated by S2JS to JavaScript. Its functions construct HTML elements for each record of the list and binds a remote call to a default visual plugin selection events. It uses function `setViewPlugin` from `EmbeddingDescriptionManager` object.

The analysis result updating mechanism uses `EmbeddedUpdater` class from package `web.client.presenters.entity.cachestore`. It is translated by S2JS and serves as a JavaScript binding library. Its function `updateEmbeddedResult` launches an analysis via a remote call of `AnalysisRunner.runAnalysisById` and schedules a periodical polling of `AnalysisRunner.getEvaluationState` to check the result of the analysis execution. When the remote call returns a successful result, the embedding description is updated with the new graph via `Payola.model.analysisResultRepository.storeResult`. 
The `EmbeddingDescription` entity serves as a container of the stored graph result. It is a content of `AnalysisResult` entity. It is persisted as other entities into the application database with its repository component trait implementation, `data.squeryl.repositories.EmbeddingDescriptionRepositoryComponent`. It contains following fields.

- `urihash` - short URI used to redirect to the full URI of an analysis result.
- `defaultvisualplugin` - determines a visual plugin, that is used directly to display the graph. It is not mandatory.
- `defaultcustomization` - determines a visual customization, which is applied to the graph. It is not mandatory.
- `analysisresultid` - binding identification to the `AnalysisResult` entity.
- `evaluationid` - identity of the analysis execution.

### 6.6 Further extension

Since the graph cache storage management is left for the users, an administration web page could be added. It was discussed during the development phase and left for the future Payola developers to consider. The page could provide a way to set cached graphs timeouts - defining how long one cached graph can be contained in the database. When the timeout is reached an automated task would remove such graphs. Also an administrative user could view which users have the most stored graphs and it could allow the administrator to set a constant limit - for example how many not embedded graphs can be stored in the database.
7. Visualization plugins

This chapter is about extensions and modifications of the previously implemented visual plugins in Payola. Triple table plugin is described in the section 7.1 and Graph view in the section 7.2. One new plugin Sigma2 is described in the section 7.3.

7.1 Triple table view

The default visual plugin of Payola provides a structured view of data - in the same way as the OpenLink Virtuoso Faceted Browser does. It displays list of RDF classes related to their properties. The visualization splits the whole graph into pages where one page contains maximum of 50 entities and each of those maximum of 5 properties - if there are more properties for one entity they are hidden and can be shown by the Show more button (see the figure 5.2). The Triple table plugin is also described in the section 5.3 with a use case scenario in the section 5.4.

The pagination of the graph was also implemented in the previous version of Payola. But the page navigation was performed in memory of web browser (client side of Payola). With implementation of this extension the plugin performs a remote call to the server whenever user navigates to a different page.

7.1.1 Use case scenario

To try out the Triple table visual plugin see the section 5.4 or follow the next steps.

1) From the Payola dashboard navigate to the Populated cities analysis detail in the Accessible analyses box.

2) Launch the analysis by the Run analysis button.

3) Press the Change visualization plugin and select Triple Table.

The plugin displays the analysis result as a table of entities with list of properties and their values. If an entity has more than 5 properties, they are hidden and a Show more button is displayed under the last property of that entity. When this button is clicked all the properties are displayed.

4) There is a pagination tool at the bottom of the page (see the figure 7.1). It allows to navigate to the first page by pressing 1 or previous page by pressing Previous. Depending on the number of pages, it is possible to navigate to the last page by pressing the button with the highest number or to go to the following page by pressing Next.

5) If there are more than 3 pages, it is possible to navigate directly to desired page, by clicking one of the ... buttons and writing the number of the page to the displayed input and pressing Enter key or the Go button. To hide the input click again one of the ... buttons.
The pagination tool also displays an information of how many entities are currently visible and how many entities does the visualized graph contain.

![Figure 7.1: Triple table plugin pagination tool.](image)

### 7.1.2 Programmers documentation

Programmatically the enhancement of this visualization lies in the implementation of the trait `GraphTransformator` - `TripleTableTransformator`, as an example of splitting a visual plugin into plain user interface generator and data preprocessing. As it is described in the Transformations chapter 5 the `TripleTableTransformator` prepares the graph stored in the cache. It splits the graph by sorting the entities - using `ORDER BY` SPARQL clause, and getting limited number of them simulating pagination - by `OFFSET` and `LIMIT` clauses. `TripleTableTransformator` returns with the graph also a number of entities in the whole graph, this allows the plugin to display the number of the page that is currently visible. The listing 7.1 shows a code snippet which performs an RPC call (this code is translated by S2JS to JavaScript and is a part of the visual plugin).

```java
private def paginateToPage(goingToPage: Int) {
    if(evaluationId.isDefined) {
        TripleTableTransformator.getCachedPage(
            evaluationId.get, goingToPage, 
            allowedLinesOnPage) {

        paginated =
```
The difference between the previous and the current versions from user point of view has not changed. The graph is displayed as a list of Subjects and Values of Properties. The User can browse through pages of graph using First, Previous, Next or Last buttons or inputting the number of page and pressing the Go button. User can browse the data in datasources by following links in Subject or Value columns. To export the graph into Turtle file (TTL) or RDF/XML formats there is a Download button, which was already available in the previous version of Payola.

### 7.2 Graph view

The Graph view plugin is also a modified version of a previously implemented visual plugin. This visualization displays the graph by graphics representation - the graph is drawn into HTML5 Canvas element and represented as vertices - entities, connected by edges - properties. The plugin is available for any data structure and is split into three following visualizations:

- **Circle**

  The entities are placed in a circled structure, where the main entity is in the center and all entities that are direct neighbors of the first one are in the circle around it - those that are a property of the main entity or the main entity is their property. In the circle around them are entities neighboring with the entities in the first circle and so on.

- **Tree**

  The entities are positioned into a tree structure, where the main vertex is in the root of the tree - top or left side of the whole structure. And similarly as the Circle, each level of the tree contains those entities that are neighbors to entities in the previous level.

- **Gravity**

  In this structure positioning depends on the relations of the entities. The graph is structured into the tree and then a gravitational model is applied in cycles. Each entity is given a force that it pushes away all other entities.
(vertices) and a relation (edge) has an opposing force that brings two entities closer together. Applying this physical model entities are spread from each other when not having a relation and brought together when having more common properties.

The modification of this visual plugin is focused on server and client side communication - as the Triple table is. In the previous version when the displayed graphs were too big - consisting of millions of entities, the structure was too complicated and could get confusing and took a long time to visualize. Splitting the visual and data preprocessing parts allows the user to browse an analysis result without any need to fetch the whole graph from the server.

7.2.1 Use case scenario

To try out the Graph visualization follow the next steps.

1) Navigate to the Payola dashboard and go to the Big cities with countries analysis detail in the Accessible analyses box.

2) Launch the analysis by the Run analysis button.

3) Press the Change visualization plugin and select Circle Visualization from the drop down menu (see the figure 3.4). This visualization presents the entities of the graph as circles and property relations as lines between the circles. To present a smaller section of the graph, there are entities grouped in rectangles.

4) To fetch entities with property rdf:type http://dbpedia.org/ontology/Country click by left mouse button on the square that is connected to the entity http://dbpedia.org/ontology/Country (left hand side of the graph). In the popup box click Unpack all (see the figure 7.2).

5) Now reposition the graph. Select entity http://dbpedia.org/ontology/Country and from the Visual tools drop down select Set main vertex.

6) To fetch a single city from the other group, click the same way on the remaining square and in the popup box click the icon in front of the dbpedia:Amman.

7) Entities can be grouped. Select (left mouse click) dbpedia:Amman hold Shift key and select dbpedia:Jordan From Visual tools drop down select Group vertices.

8) To disable graph repositioning when manipulating groups uncheck Reposition when group changed.

9) A group can be labeled, that it is easier to determine what contains. Click the new group and write in the input in the popup box Jordan.
10) To get the graph as an image click the Download as PNG in the Visual tools drop down.

11) To see properties and their values select an entity, for example **dbpedia:Canada**.

To reposition manually entities or groups select one or multiple by holding **Shift** and drag them using left mouse button. To deselect an entity click the background.

If repositioning algorithm takes too long **Stop animation** button in the header can be pressed or the background of the graph can be clicked.

---

Figure 7.2: Graph view with popup window displaying the content of a group.

### 7.2.2 Programmers documentation

Programmatically the **VisualPluginView** contains the basic structure of the Graph view plugin implementation. It defines action listeners and remote call functions. Its only implementation is the **BaseTechnique** abstract class which defines positioning and drawing operations for the **TreeTechnique** (snippet of code visible in the [listing 7.2]), **GravityTechnique** and **CircleTechnique** classes. These classes contain only sequences of positioning operations defined by **Animation** class. That is a tool for defining a sequence of animating the drawn graph in the HTML5 Canvas element.

```java
class TreeTechnique(prefixApplier: Option[PrefixApplier] = None)
  extends BaseTechnique("TreeVisualization", prefixApplier)
```
The HTML5 Canvas drawing tools are defined in the data containers - View trait. The chosen graphics rendering approach is that every drawn object is responsible for drawing itself. The View trait implementations containing the data are GraphView, VertexViewElement, VertexViewGroup, EdgeView and for text InformationView - entities and properties, using the drawing functions defined in the View the scene is rendered in a cascade. Starting with the GraphView a container of Component objects. The Component is drawn with all its contained VertexView objects and EdgeView objects. These draw InformationView objects, that are contained in them.

Related to graphics there were added two new features to this plugin. The first was already mentioned at the beginning of this section. The groups and grouping mechanism allows users to select multiple vertices and create vertex groups via the menu. Grouped vertices can again be withdrawn from a group via the context menu of the group. This way the less relevant parts of the graph can be hidden the same way as the graph is sent from the server VisualTransformator.

The second visual modification of this plugin lies in the positioning and animations. Using the menu any of the plugin a vertex can be marked as main. This centers the selected vertex in the circular positioning or puts the vertex in the root of the tree in the tree positioning. In case these animated positioning techniques should be stopped when in progress a Stop button was added to the plugin menu.

### 7.3 Sigma2 view

The Sigma2 is a new visual plugin added in this Payola’s extension. It is an implementation of Sigma JavaScript library (http://sigmajs.org/) which
specializes in graph drawing into HTML5 Canvas element. Its implementation does not use any of the specialized transformations but the **IdentityTransformator**, which sends the whole graph as it is stored inside the cache to the plugins. The visualization displays the graph in the same manner as the Graph View - entities are represented by circles or vertices and properties by lines or edges connecting the vertices. Literals are not drawn into the graph representation but are listed in a popup box, which is show when an entity is selected.

### 7.3.1 Users documentation

The visualization features can be seen in the scenario in the subsection 7.3.2 after Payola installation (see the section 1.3).

This plugins allows user to selection an entity by clicking on it. When pointing at an entity all not neighbouring entities and properties of the graph are hidden. The plugin does not support entity dragging. Only the whole graph can be moved by dragging an entity or the background of the visualization.

Positions of the entities are initialized randomly. The final layout can be reached by pressing **Start positioning** button from the menu. The entities are repositioned by ForceAtlas2 plugin of the Sigma.js. It moves the entities away from each other, when not having a property. When entities are in a relation, they are moved closer. It is a similar gravitational algorithm as is implemented in the Graph plugin. The layout algorithm can be stopped by pressing **Stop positioning** button. The positioning result can be seen in the figure 7.3 on a exemplary graph.

Figure 7.3: Sigma2 view.
7.3.2 Use case scenario

To try out the Sigma visualization follow the next steps.

1) Navigate to the Payola dashboard and go to the Big cities with countries analysis detail in the Accessible analyses box.

2) Launch the analysis by the Run analysis button.

3) Press the Change visualization plugin and select Sigma.js from the drop down menu (see the [figure 3.4]).

   This visualization presents the entities of the graph as dots and property relations as lines between them. The entities are initially positioned randomly.

4) To start the entities positioning press the Start positioning button.

   The entities in the graph are spread out by the ForceAtlas2 plugin of the Sigma.js library.

5) Press the Stop positioning button to finish it.

6) To display property literal values of entity dbpedia:Delhi mouse over and click the entity (dot) with the dbpedia:Delhi label next to it (as shown in the [figure 7.4]).

   When mouse hovers over an entity with literal properties a transparent popup window is shown with a list of them. When the entity is clicked the window loses its transparency and the list is fully visible.

7) to hide the popup click the white background of the plugin or mouse over another entity.

8) The graph can moved around by clicking the white background and dragging mouse.

9) The graph can be also scaled up or down using mouse wheel.

7.3.3 Programmers documentation

The implementation of this visual plugin is not similar to the Graph View or Triple Table View although its code is written in Scala and it is translated by S2JS into JavaScript. Since this is an implementation of a JavaScript library the objects used at runtime need to be accessible at compile-time that there will not be any compilation errors. This is done by JavaScript object adapters - Scala traits defined in the s2js.adapters package. These traits substitute the runtime objects used in the Scala code of the visual plugin which are replaced by the JavaScript objects of the library during S2JS translation. To use features of the

---

1When S2JS transformed code is being executed in a web browser.
2When Payola is being compiled and S2JS translation is executed.
library the S2JS allows to write pure JavaScript code in Scala using @javascript annotations. Code defined this way is used as a body of the annotated Scala function. In the listing 7.3 is visible the initialization of Sigma2 plugin.

```scala
@javascript(""
    return new sigma({
        graph: {
            nodes: nodeList,
            edges: edgeList
        },
        container: wrapper,
        settings: {
            drawEdges: true,
            defaultNodeColor: '#0088cc',
            labelThreshold: 0,
            edgeLabels: true
        }
    });
""

protected def initSigma(nodeList: List[NodeProperties],
                        edgeList: List[EdgeProperties],
                        wrapper: Element):
    sigma.Sigma = null
```

Listing 7.3: Sigma2 initialization code snippet.

To make the graph positioning as quick as possible the vertices of the graph are placed randomly when the plugin is initialized. Since the plugin does support
only mouse over events to highlight neighborhood of a vertex and no custom vertices positioning as the Graph View, the library contains another plugin - the ForceAtlas2. This plugin handles layout of the drawn graph. It implements a force-directed layout algorithm that positions the vertices in a gravitation-like fashion. The layout animation and algorithm is not launched automatically but can be started and stopped by a plugin menu button. For more details about the plugin see [18]. The algorithm is described in [19].
8. Visual customizations

Visual plugins in Payola provide browsing tools for RDF data. If it is the Triple Table plugin, Graph View or Column chart the visualized graph is always represented by the data it contains. Visual customizations bring a way of modifying the displayed graph based on its content. In the previous release Payola allowed to customize graphs based on their structure and entity datatypes. In this Payola extension these customizations are slightly changed and user defined customizations are introduced. Follows a list of required modifications:

- Alter Ontology customizations that it modifies the entities or properties that are related to the selected entity.
- Introduce customization defining without ontologies.
- New customizations would allow to modify nodes, edges or groups without any conditions based on graph structure or entity relations.
- Create a customization definition that will modify Nodes that have a certain property and optionally the property has a specified value.

8.1 Ontology customizations

In the previous version of Payola and in this extension, customizations are applied to Graph View and Sigma View plugins. An ontology customization in these plugins can be used via header menu. Ontology customization is defined by specifying a URI of an ontology and a name. The customization loads the ontology from the provided address. It constructs a customization with all entities and properties that the ontology defines. Via the Edit ontology customization popup form user can specify, how the graph will be customized.

In the previous version of Payola this customization was defined differently. User could choose display options of entities and properties separately - the relation between an entity and a property was not used. In this modified version properties are strictly assigned to entities (figure 8.1).

The customization of an entity (changed color, vertex size or displayed glyph over a vertex) is applied to all vertices in the displayed graph that are of the same rdf:type that is the entity of the customization. For example a customization with an entity [http://dbpedia.org/ontology/City](http://dbpedia.org/ontology/City) (for short City) will customize all vertices that have a property rdf:type with object City. Customizations of properties (edges) is then applied only to those properties that relate to the customized entity and that have the same IRI as the property in the customization. For example an entity [http://dbpedia.org/page/Prague](http://dbpedia.org/page/Prague) (for short Prague) is present in the customized graph and it has a property [http://dbpedia.org/ontology/country](http://dbpedia.org/ontology/country) (Country) with an object [http://dbpedia.org/page/Czech_Republic](http://dbpedia.org/page/Czech_Republic) (CZ). The ontology contains entity City with a property Country, then the property customization will be applied to the edge between Prague and CZ, since Prague has a rdf:type City.
8.2 User customization

This customizations variant is compared to Ontology customization purely user defined. A new User customization does not define any visual modifications to a displayed graph. User can specify Node settings, Group, Edge and Node based on edge customizations. The first Node allows user to visually alter any entity contained in the current graph or even specify a custom entity IRI. Group modification sets options to contained groups of entities. Edge is a setting for any property in the graph not depending on its neighboring entities. And Node based on edge modification is for setting visual options to entities based on their property value. That is for example when an entity has a \texttt{rdf:type} property of value \url{http://dbpedia.org/ontology/country}, the customization is applied only to those entities. Moreover if there is not only one graph modification of this kind the customizations are applied in the order that they are defined in. From the figure 8.2 a graph altered by this customization would have its entities modified first by the \url{http://dbpedia.org/ontology/country} customization and all the entities that would satisfy the condition of \texttt{rdf:type} would be modified afterwards replacing the settings of \url{http://dbpedia.org/ontology/country}, in case of a conflict.

Although the User customization is ontology independent the Node visual options can work with OWL2 data types. By specifying the order of labels for an entity user can define a label that is displayed over it. A custom label can be set but more importantly label literal values can be used. If an entity has a property of \texttt{skos:prefLabel} its value would be applied as a label for the entity \texttt{dbpedia:Nigeria} from the example in the figure 8.3. Though if there is no
8.3 Use case scenario

To try out the ontology based customization follow the next steps.

1) From the dashboard log into Payola. Click Sign in and input user email and password or create a new account by clicking Sign up. Navigate to the Big cities with countries analysis detail page from the dashboard (see the figure 3.2).

2) Execute the analysis by the Run analysis button (see the figure 3.3).

3) Select the Circle visualization plugin from the list of available plugins (Change visualization plugin button; see the figure 3.4).

4) Click on both squares (by left mouse button) and in the context menu press Unpack all. This will fetch the whole graph and the following customization will be more visible (see the figure 8.4).

5) Click Change appearance button, which loads customizations available to the logged in user.
6) From the Change appearance menu, select Create New Ontology Customization and write DBPedia ontology into the Name field and http://downloads.dbpedia.org/2015-10/dbpedia_2015-10.owl into Ontology URLs.

7) From the left hand side list select http://dbpedia.org/ontology/Country and click into Fill color input and select green color.

8) Again in the left hand side list select http://dbpedia.org/ontology/City and click into Fill color input and select blue color.

The result graph might look like the one in the figure figure 8.5.

To try out the manually constructed customization follow the first 5 steps of the ontology customization scenario, to log in, execute analysis and load customizations. After that follow the next steps.

1) From the Change appearance menu, select Create New User Customization and write User customization into Name field.

2) Press Node and select dbpedia:Ibadan to add customization of the Country entity. Click the new Node in Node, edge, group customizations box and select red color by clicking into Fill color input.

3) Press Node based on edge and add rdf:type. Click the new customization in the Edge Based node customizations and select green color in the Fill color input and check Use property value as label.
4) Press again **Node based on edge** and add [http://dbpedia.org/ontology/country](http://dbpedia.org/ontology/country). Click it and then press **Select** next to the **Property value** input and select **dbpedia:Nigeria**. Next select yellow color for this customization in the **Fill color** input and check **Use property value as label**.

Now when the **Done** button is pressed, the visualization consist of fewer entities (nodes). Most of them are green with [http://dbpedia.org/ontology/city](http://dbpedia.org/ontology/city) label. Those are the city entities, that have a **rdf:type** property (altered by first customization). Entities without this property are hidden. Only one entity is red with label **dbpedia:Nigeria**. This entity is altered by the [http://dbpedia.org/ontology/country](http://dbpedia.org/ontology/country) customization. The **Node based on edge** customizations are applied from first to last. Notice that the Node customization for **dbpedia:Ibadan** is applied. The **Node based on edge** customizations gradually and after those the Node customization was applied (as seen in the figure 8.6).

### 8.4 Programmers documentation

The implementation of the Customizations depends on the Play Framework since Payola implements it. The button Change appearance in the menu of the visual plugins is defined as a **DropDownButton** element in **PluginSwitchView** class. When user presses the button a remote call is performed to the **Model** object to fetch customizations available for the current user by following methods.

- def ontologyCustomizationsByOwnership
  (successCallback: OntologyCustomizationsByOwnership => Unit)
Figure 8.5: Graph plugin view with ontology defined customization.

(errorCallback: Throwable => Unit)

Returns via successCallback method call a list of available ontology based customizations.

- def userCustomizationsByOwnership
  (successCallback: UserCustomizationsByOwnership => Unit)
  (errorCallback: Throwable => Unit)

Returns via successCallback method call a list of available manually defined customizations.

In the PluginSwitchView class the remote call fills the list of available customizations and additional buttons are added (customization creation buttons and a Disable customizations button).

A creation of a customization opens an input form (created by JavaScript code) with ontology URL and customization name inputs. Submitting the form opens a customization editor form UserCustomizationEditModal or OntologyCustomizationEditModal, edit forms of the customizations. All of the inputs are binded to remote calls that persist inputted data into application database via the Model object methods.

- def createUserCustomization(name: String)
  (successCallback: UserCustomization => Unit)
  (errorCallback: Throwable => Unit)
Figure 8.6: Graph plugin view with applied user defined customization.

- def createOntologyCustomization
  (name: String, ontologyURLs: String)
  (successCallback: OntologyCustomization => Unit)
  (errorCallback: Throwable => Unit)

Remote methods for creation of ontology based and user defined customizations.

- def deleteUserCustomization
  (customization: UserCustomization)
  (successCallback: () => Unit)
  (errorCallback: Throwable => Unit)

- def deleteOntologyCustomization
  (ontologyCustomization: OntologyCustomization)
  (successCallback: () => Unit)
  (errorCallback: Throwable => Unit)

A remote methods for deletion of ontology and user customizations.

- def changeCustomizationName
  (customization: DefinedCustomization, newName: String)
  (successCallback: () => Unit)
  (errorCallback: Throwable => Unit)

A remote method for renaming of customizations.

- def forceCustomizationsByOwnershipUpdate
  (successCallback: (UserCustomizationsByOwnership,
A remote method for updating ontology and user defined customizations.

Updating any of the attributes of customizations triggers a remote call with a success callback method binded to the `onCustomizationsChanged` function of the `GraphPresenter` class. This class is a container for the `PluginSwitchView` which contains and manages the visual plugins. The call triggers `updateCustomization` method and executes the update mechanism of the visualization. The currently used visual plugin receives the data of the edited customization and it is applied depending on its implementation. For example the Graph view applies the customization in a cascade. First the vertices are customized by the entities defined in the customization, then the edges and finally the labels from user customizations are applied.
9. Related work

Follows a list of tools solving similar problems as the Cache and Transformations layers in this Payola’s extension.

- LodLive (http://en.lodlive.it/) is a demonstration project of Linked Data standards for RDF data browsing. It is a pure JavaScript HTML Canvas web application. It fetches data from data sources using SPARQL endpoints on demand. Compared to Payola’s extension it works similarly as graph visualization when user requests a neighborhood of graph’s node. It calls Transformation layer (present in LodLive in SPARQL query construction and endpoint communication) and fetches data from Cache.

- Protovis (http://mbostock.github.io/protovis/) is a JavaScript library for graph visualization. It uses unlike HTML Canvas in Payola SVVG graphics. It defines basic principles used in D3.js which is its successor.

- D3.js (https://d3js.org/) is a JavaScript library that visualizes data using HTML, SVG and CSS. It focuses on the basic problematics of data visualization such as data selection, dynamic properties setting (similar to jQuery or Protovis libraries) or animated transitions. Visualizations or data layouts are left to its plugins (Reingold–Tilford Tree, Force-Directed Graph, Fisheye Distortion).

This extension of Payola focuses mainly on Cache and data interchange between client and server to solve user interface response time issues. D3.js solves this problem when visualizing vast graphs by decreasing memory consumption. Each node of a visualized graph contains as less information as possible. For example the nodes are identified using its data content.

Payola on the other hand handles each node as a JavaScript object that contains all of its information (references to its edges and neighboring nodes, its name and properties). This is a consequence of Scala code to JavaScript transformation. To allow the D3.js approach to this problematics a plain JavaScript library and a Scala code adapter would be required.

- Pajek and Pajek-XXL (http://mrvar.fdv.uni-lj.si/pajek/) are desktop applications focused on visualization and graph structure analysis. Even though it is not a web application, it visualizes vast graphs. Its data visualization is based on path finding and force-directed graph drawing algorithms (A-star, Kamada-Kawai and Fruchterman Reingold algorithms) to position graph nodes compared to Payola’s graph visualization.

Regardless of the differences between Pajek and Payola, the data visualization mechanism is similar. For large graphs Pajek-XXL processes data and selects subgraphs. Payola’s Cache stores the whole graph and Transformer selects its subgraph and the graph visualization displays only few neighboring nodes of it.
• GINY ([http://csbi.sourceforge.net/](http://csbi.sourceforge.net/)) is a Java graphing library which makes an extensive use of Piccolo ([http://www.cs.umd.edu/hcil/piccolo/](http://www.cs.umd.edu/hcil/piccolo/)) graphics library. It has been developed primarily for research of biology. Its inner mechanics are separated into Model and View. Where the model section works in the same way as Cache and Transformations in Payola. It loads the whole graph into RootGraph section of Model and GraphPerspective section of the Model then select only a certain subgraph which is visualized [27].

• IsaViz ([https://www.w3.org/2001/11/IsaViz/](https://www.w3.org/2001/11/IsaViz/)) is a Java desktop application for browsing and authoring RDF data using Graphviz library.

• Graphviz ([http://www.graphviz.org/](http://www.graphviz.org/)) is a graph visualization software. It does not contain GUI editors but processes textual represented graphs. It allows to apply layouts, fonts, change node colors and shapes an export graphs into SVG, Postscript, PDF or display an interactive graph browser. It can position graph nodes hierarchically, apply spring model layout. In comparison to Payola it does not focus on caching or processing graphs as a whole but it allows both approaches to its implementation.

• Gephi ([https://gephi.org/](https://gephi.org/)) is a desktop application for graph data visualization and presentation. It does not use any kind of caching but applies View Layers on the whole graph. It is similar to GINY library but without GraphPerspective section.

• Graphopt ([http://www.schmuhl.org/graphopt/](http://www.schmuhl.org/graphopt/)) is a desktop application for graph layout optimization. It positions the graph in a two dimensional space using a physical model applied to graph’s nodes and edges. It does not make use of preprocessing or caching but positions the whole graph.
10. Conclusion

This Payola extension modifies its inner mechanics that users can experience more responsiveness. It alters Payola as a framework by addition of the Cache and Transformations components. But it also changes it as an application extending its visual plugins and adding embedding mechanics.

This extension can be looked at as a research of data partition visualizing. It approaches the problematics of vast data visualization by data sub set selection and caching compared to other similar visualization tools. The extension shows that by implementing the mechanism the data visualizations are more responsive and can display the same amount of information as the previous version of Payola.

During development and completion of this extension, the main development branch of Payola has moved forward. It was presented as a Linked Data analysis and visualization framework [20]. It was differentiated to Linked Data Visualization Model (LDVM) framework and the implementation Payola [21]. A statistics representation a processing means were added as Data Cube [22]. A vocabulary for its components description was introduced [23]. A new implementation of LDVM was presented [24]. And the work continued to Linked Open Data automated visualizers [25].
Attachments

Content of the enclosed CD ROM

With the thesis comes a CD ROM containing source codes of the implementation, compilation tools and runtime environment tools with installation guide and Payola User and Developer guides.

/doc - Contains installation, user and developer guides.

/src - Contains source files of the Payola and the implemented extension.

/tools - Contains compilation and runtime environment tools used and mentioned in the installation guide.
<?xml version="1.0"?>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
xml:base="http://www.w3.org/2002/07/owl#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <Prefix name="schema" IRI="http://schema.org/">
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#/>
    <Prefix name="literal" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#/>
  </Prefix>
  <Declaration>
    <Class abbreviatedIRI="schema:Airport"/>
  </Declaration>
  <Declaration>
    <Class abbreviatedIRI="schema:CivicStructure"/>
  </Declaration>
  <Declaration>
    <Class abbreviatedIRI="schema:Place"/>
  </Declaration>
  <Declaration>
    <Class abbreviatedIRI="schema:Thing"/>
  </Declaration>
  <SubClassOf>
    <Class abbreviatedIRI="schema:Airport"/>
    <Class abbreviatedIRI="schema:CivicStructure"/>
  </SubClassOf>
  <SubClassOf>
    <Class abbreviatedIRI="schema:CivicStructure"/>
    <Class abbreviatedIRI="schema:Place"/>
  </SubClassOf>
  <SubClassOf>
    <Class abbreviatedIRI="schema:Place"/>
    <Class abbreviatedIRI="schema:Thing"/>
  </SubClassOf>
  <Annotation Assertion>
    <AnnotationProperty abbreviatedIRI="rdfs:comment"/>
    <AbbreviatedIRI>schema:Airport</AbbreviatedIRI>
    <Literal datatypeIRI="literal:PlainLiteral">
      An airport.
    </Literal>
  </Annotation Assertion>
  <Annotation Assertion>
    <AnnotationProperty abbreviatedIRI="rdfs:label"/>
    <AbbreviatedIRI>schema:Airport</AbbreviatedIRI>
    <Literal datatypeIRI="literal:PlainLiteral">
      An airport.
    </Literal>
  </Annotation Assertion>
</Ontology>
A public structure, a town or concert hall.

Entities that have a physical extension.

The most generic type of item.

Listing 10.1: Example of OWL/XML ontology syntax.

Prefix : <http://www.semanticweb.org/owl/owlapi/turtle#>
Ontology:

AnnotationProperty: rdfs:label

AnnotationProperty: rdfs:comment

Datatype: rdf:PlainLiteral

Class: schem:Airport

Annotations:
  rdfs:label "Airport",
  rdfs:comment "An airport."

SubClassOf:
  schem:CivicStructure

Class: schem:Thing

Annotations:
  rdfs:label "Thing",
  rdfs:comment "The most generic type of item."

Class: schem:Place

Annotations:
  rdfs:label "Place",
  rdfs:comment "Entities that have a physical extension."

SubClassOf:
  schem:Thing

Class: schem:CivicStructure

Annotations:
  rdfs:label "CivicStructure",
  rdfs:comment "A public structure, a town or concert hall."

SubClassOf:
Listing 10.2: Example of Manchester OWL ontology syntax.

Prefix(:=<http://www.semanticweb.org/owl/owlapi/turtle#>)
Prefix(schema:=<http://schema.org/>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)

Ontology(
  Declaration(Class(schema:Airport))
  Declaration(Class(schema:CivicStructure))
  Declaration(Class(schema:Place))
  Declaration(Class(schema:Thing))
  AnnotationAssertion(
    rdfs:comment schema:Airport "An airport.")
  AnnotationAssertion(rdfs:label schema:Airport "Airport")
  SubClassOf(schema:Airport schema:CivicStructure)
  AnnotationAssertion(
    rdfs:comment
    schema:CivicStructure
    "A public structure, a town or concert hall.")
  AnnotationAssertion(
    rdfs:label
    schema:CivicStructure
    "CivicStructure")
  SubClassOf(schema:CivicStructure schema:Place)
  AnnotationAssertion(
    rdfs:comment
    schema:Place
    "Entities that have a physical extension.")
  AnnotationAssertion(rdfs:label schema:Place "Place")
  SubClassOf(schema:Place schema:Thing)
  AnnotationAssertion(
    rdfs:comment
    schema:Thing
    "The most generic type of item.")
  AnnotationAssertion(rdfs:label schema:Thing "Thing")
)

Listing 10.3: Example of Functional OWL ontology syntax.
Bibliography


# List of Figures

1.1 Analysis result data .................................. 5
3.1 Workflow diagram .................................. 17
3.2 Payola dashboard after user login ..................... 18
3.3 Analysis detail page .................................. 19
3.4 Graph plugin showing analysis result .................. 20
3.5 Payola component diagram [image] ................. 21
3.6 Communication diagram between user and analysis without cache .......................... 22
3.7 Communication diagram between user and analysis with cache ......................... 22
3.8 Communication diagram workflow .................... 23
3.9 Communication diagram workflow with Cache and Transformations ............... 23
4.1 Response times comparison ......................... 30
5.1 Communication diagram of components with Transformations .................. 33
5.2 Triple table view plugin with DBpedia.org datasource .................. 34
6.1 Stored analysis results in the cache .................. 39
6.2 Payola dashboard .................................. 40
7.1 Triple table plugin pagination tool .................. 44
7.2 Graph view with popup window displaying the content of a group .................. 47
7.3 Sigma2 view ..................................... 49
7.4 Sigma graph visualization with literals list ............ 51
8.1 Ontology customization editor ....................... 54
8.2 User customization editor with Node based on edge settings ..................... 55
8.3 User customization editor with Node settings .................. 56
8.4 Graph plugin view with group context menu .................. 57
8.5 Graph plugin view with ontology defined customization .................. 58
8.6 Graph plugin view with applied user defined customization .................. 59
List of abbreviations

**GPS** Global Positioning System, a location providing navigation system

**HTML** HyperText Markup Language, a XML based standard language used for web pages

**IRI** Internationalized Resource Identifier, a URI identifier that can contain Unicode characters

**JSON** JavaScript Object Notation, a lightweight data interchange format

**ORM** Object-relational mapping, a technique for data conversion often used between database and application

**OWL2** Ontology language, a OWL 1 successor model that provides datatypes

**RDF** Resource Description Framework, a directed, labeled graph data format for representing information in the Web

**RDBMS** Relational Database Management System, a database management system, that is based on a relational model

**RPC** Remote Process Call, a procedure execution commonly on another computer via network

**S2JS** Scala to JavaScript translation tool, a part of Payola project

**SPARQL** SPARQL Protocol and RDF Query Language, a semantic language for RDF database querying

**SQL** Structured Query Language, a language for data management in a relational database

**URI** Uniform Resource Identifier, an identity with a specific structure used to represent resources in the Web, consisting of ASCII characters

**URL** Uniform Resource Locator, a location identification part of URI

**XML** EXtensible Markup Language, a machine and human readable data format

**TTL** Turtle (Terse RDF Triple Language) format for expressing RDF data