POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Embedded robotics (AER)

PROJECT

System Linked Data dla kursow akademickich

Linked Data system for academic courses

AUTOR:
Eng. Cezary Dynak

SUPERVISOR:
M.Eng., Ph.D. Witold Paluszynski, K-7

GRADE:

WROCŁAW 2016

Contents

Chapter 1

Problem

1.1 Task

Goal of this project is to design basic unit of knowledge and create web application with visualization and editing interface. This should be done by developing ontology compatible with meta-data formats (i.e. RDF) and make it possible to query in standardized way (i.e. SPARQL). Another aspect is to connect it with functionality of package manager like dpkg with deb packages or yum with rpm packages in popular Linux distributions.

1.2 Theory

1.2.1 Ontology in philosophy

Ontology is the philosophical study of the nature of being, becoming, existence, or reality, as well as the basic categories of being and their relations. Traditionally listed as a part of the major branch of philosophy known as metaphysics, ontology often deals with questions concerning what entities exist or may be said to exist, and how such entities may be grouped, related within a hierarchy, and subdivided according to similarities and differences. Although ontology as a philosophical enterprise is highly theoretical, it also has practical application in information science and technology, such as ontology engineering.

1.2.2 Ontology in science

In computer science and information science, an ontology is a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse. It is thus a practical application of philosophical ontology, with a taxonomy.
An ontology compartmentalizes the variables needed for some set of computations and establishes the relationships between them.
The fields of artificial intelligence, the Semantic Web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture all create ontologies to limit complexity and to organize information. The ontology can then be applied to problem solving.

1.3 Similar solutions

1.3.1 European Qualifications Framework

The European Qualifications Framework (EQF) acts as a translation device to make national qualifications more readable across Europe, promoting workers' and learners' mobility between countries and facilitating their lifelong learning. The EQF aims to relate different countries' national qualifications systems to a common European reference framework. Individuals and employers will be able to use the EQF to better understand and compare the qualifications levels of different countries and different education and training systems. Since 2012, all new qualifications issued in Europe carry a reference to an appropriate EQF level.
The core of the EQF concerns eight reference levels describing what a learner knows, understands and is able to do - 'learning outcomes'. Levels of national qualifications will be placed at one of the central reference levels, ranging from basic (Level 1) to advanced (Level 8). This will enable a much easier comparison between national qualifications and should also mean that people do not have to repeat their learning if they move to another country.

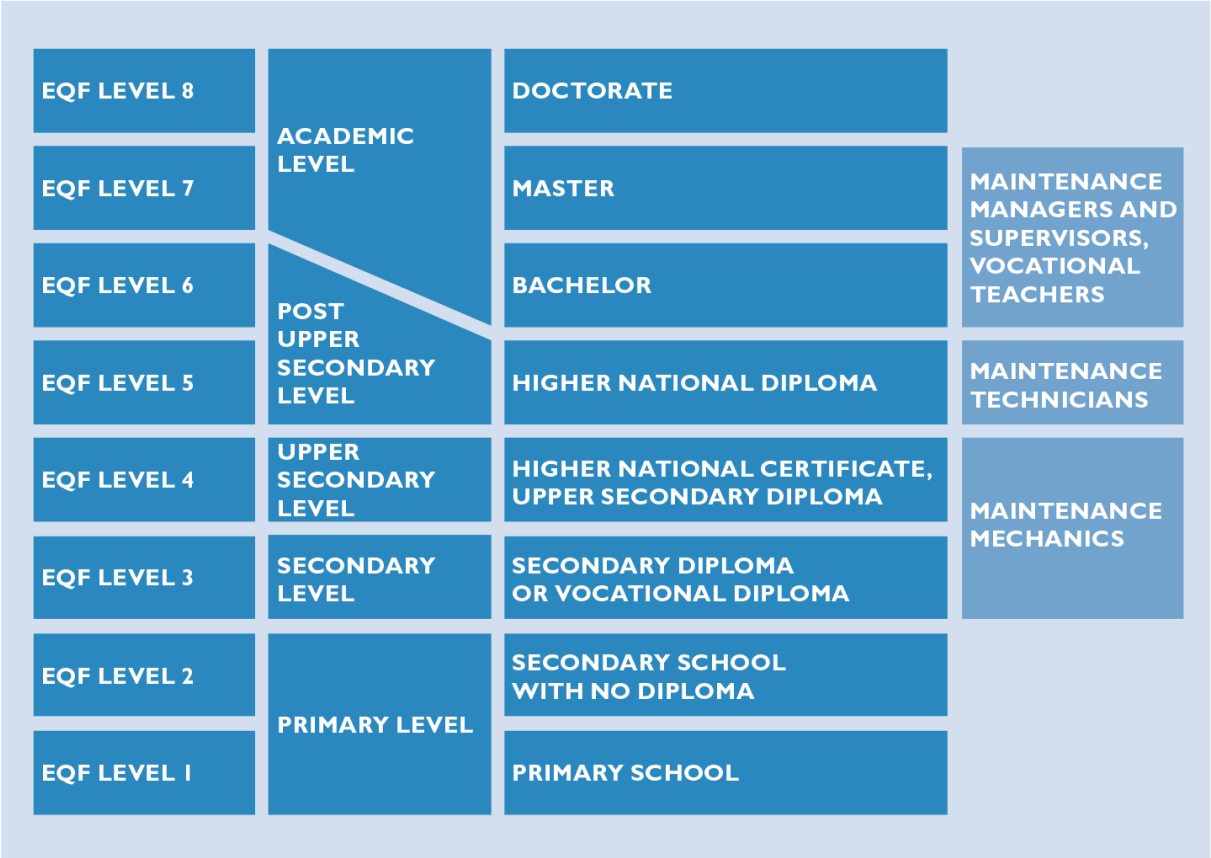

Figure 1.1 European Qualifications Framework general scheme

1.3.2 Package manager

A package manager or package management system is a collection of software tools that automates the process of installing, upgrading, configuring, and removing computer programs for a computer's operating system in a consistent manner. A package manager

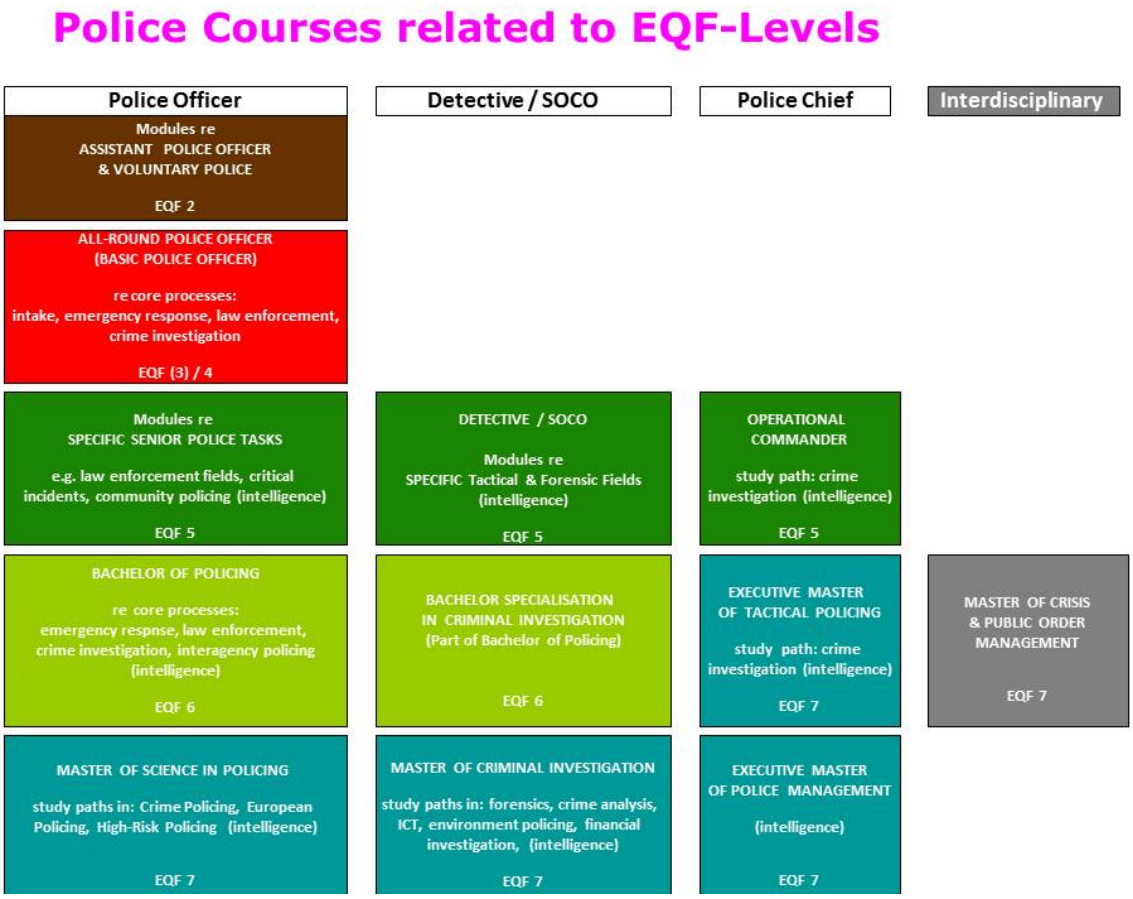

Figure 1.2 EQF implementation example at cepol.europa.eu

deals with packages, distributions of software and data in archive files. Packages contain metadata, such as the software's name, description of its purpose, version number, vendor, checksum, and a list of dependencies necessary for the software to run properly. Upon installation, metadata is stored in a local package database. Package managers typically maintain a database of software dependencies and version information to prevent software mismatches and missing prerequisites. They work closely with software repositories and app stores.

Package managers are designed to save organizations time and money through remote administration and software distribution technology that eliminate the need for manual installs and updates. This can be particularly useful for large enterprises whose operating systems are based on Linux and other Unix-like systems, typically consisting of hundreds or even tens of thousands of distinct software packages; in the former case, a package manager is a convenience, in the latter case it becomes essential.

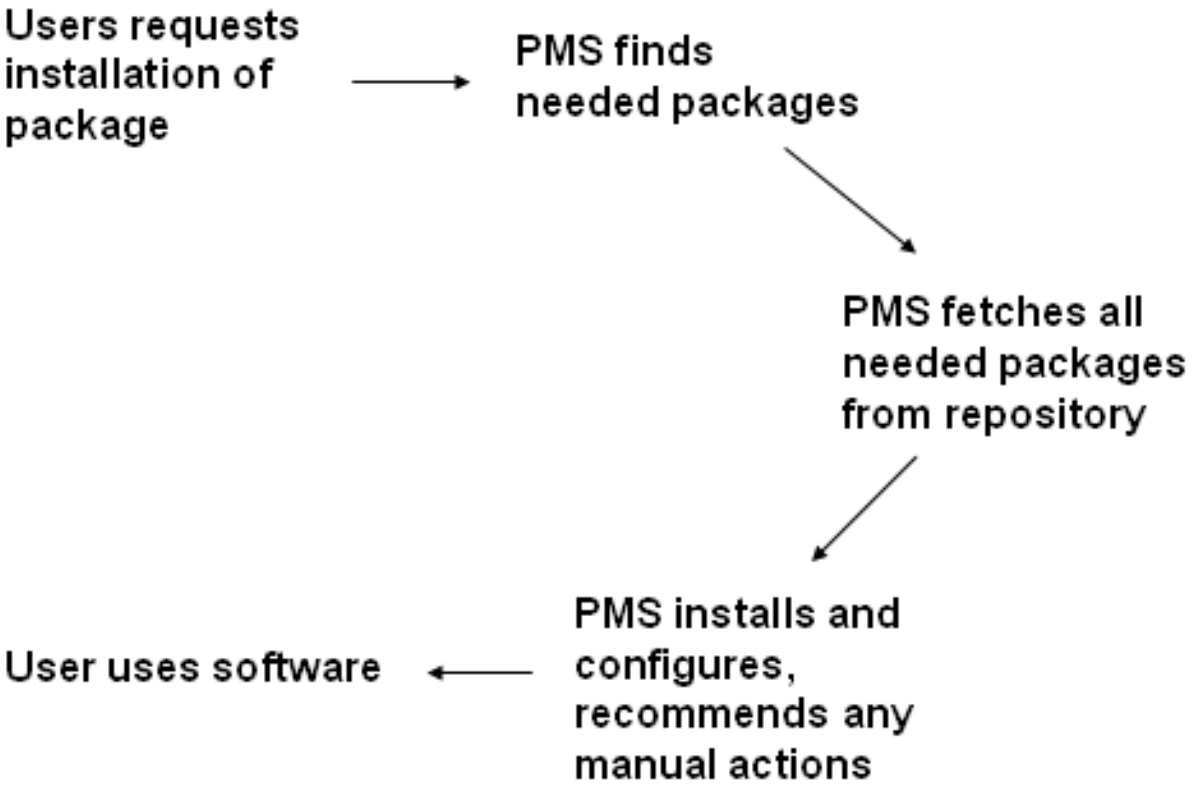

Figure 1.3 Illustration of a package manager being used to download new software.

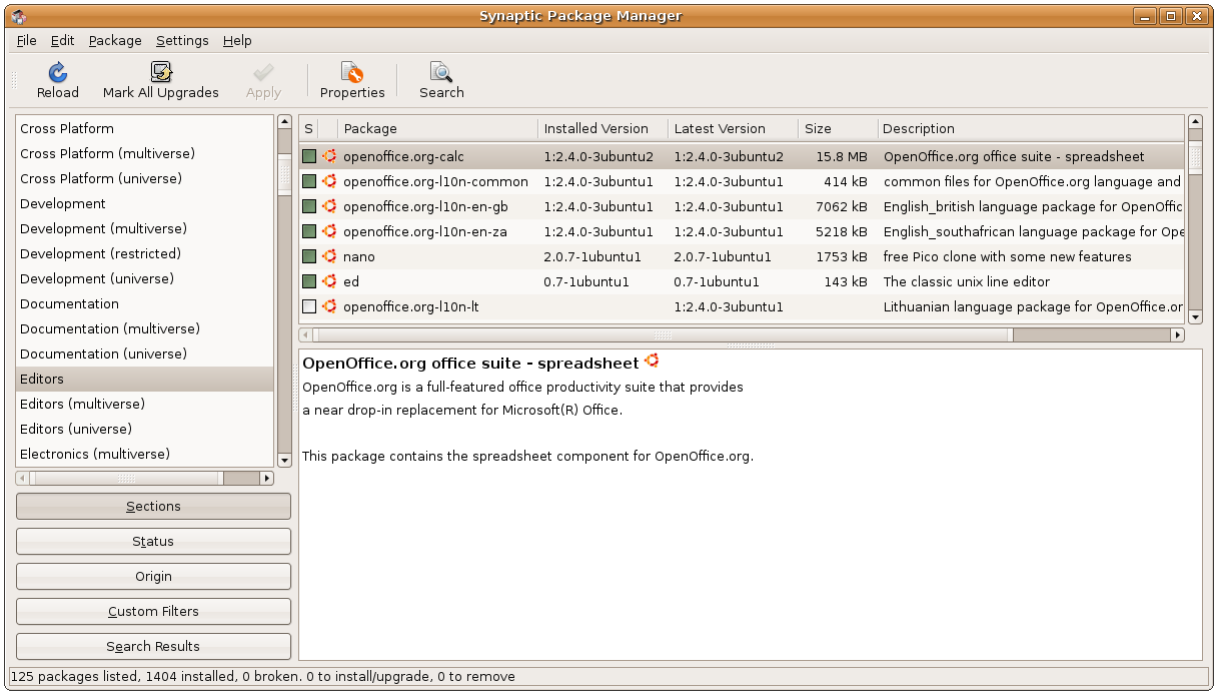

Figure 1.4 Synaptic, a GUI for dpkg

Chapter 2

Implementation

2.1 Package management

2.1.1 npm

npm (the node package manager) is the default package manager for the JavaScript runtime environment Node.js.
As of Node.js version 0.6.3, npm is bundled and installed automatically with the environment. npm runs through the command line and manages dependencies for an application. It also allows users to install Node.js applications that are available on the npm registry. npm is written entirely in JavaScript and was developed by Isaac Z. Schlueter as a result of the frustrations he had experienced while working with CommonJS and with inspiration from similar projects for PHP (PEAR) and Perl (CPAN).
The goal is to create package management tool in node/npm ecosystem, like Bower. Bower is a package management system for client-side programming on the World Wide Web. It depends on Node.js and npm. It works with git and GitHub repositories.

2.1.2 package.json

All npm packages contain a file, usually in the project root, called package.json - this file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package. The package.json file is normally located at the root directory of a Node.js project.
Node itself is only aware of two fields in the package.json:

```
{
  "name" : "barebones",
  "version" : "0.0.0",
}
```

The name field should explain itself: this is the name of your project. The version field is used by npm to make sure the right version of the package is being installed. Generally, it takes the form of major.minor.patch where major, minor, and patch are integers which increase after each new release. For more details, look at this spec: http://semver.org .

For a more complete package.json, we can check out underscore:

```
{
  "name" : "underscore",
  "description" : "JavaScript's functional programming helper library.",
  "homepage" : "http://documentcloud.github.com/underscore/",
  "keywords" : ["util", "functional", "server", "client", "browser"],
  "author" : "Jeremy Ashkenas <jeremy@documentcloud.org>",
  "contributors" : [],
  "dependencies" : [],
  "repository" : {"type": "git", "url": "git://github.com/doccloud/underscore.git"},
  "main" : "underscore.js",
  "version" : "1.1.6"
}
```

As you can see, there are fields for the description and keywords of your projects. This allows people who find your project understand what it is in just a few words. The author, contributors, homepage and repository fields can all be used to credit the people who contributed to the project, show how to contact the author/maintainer, and give links for additional references. The file listed in the main field is the main entry point for the library; when someone runs require("library name"), require resolves this call to require("package.json:main").
Finally, the dependencies field is used to list all the dependencies of your project that are available on npm. When someone installs your project through npm, all the dependencies listed will be installed as well. Additionally, if someone runs npm install in the root directory of your project, it will install all the dependencies to ./node_modules.
It is also possible to add a devDependencies field to your package.json - these are dependencies not required for normal operation, but required/recommended if you want to patch or modify the project. If you built your unit tests using a testing framework, for example, it would be appropriate to put the testing framework you used in your devDependencies field. To install a project's devDependencies, simply pass the –dev option when you use npm install.
For even more options, you can look through the online docs or run

```
npm help json
```

2.2 Ontology

2.2.1 json-ld

Generally speaking, the data model used for JSON-LD is a labeled, directed graph. The graph contains nodes, which are connected by edges. A node is typically data such as a string, number, typed values (like dates and times) or an IRI. There is also a special class of node called a blank node, which is typically used to express data that does not have a global identifier like an IRI. Blank nodes are identified using a blank node identifier. This simple data model is incredibly flexible and powerful, capable of modeling almost any kind of data. For a deeper explanation of the data model, see section 7. Data Model. Developers who are familiar with Linked Data technologies will recognize the data model as the RDF Data Model.
JSON-LD is a concrete RDF syntax as described in [RDF11-CONCEPTS]. Hence, a JSON-LD document is both an RDF document and a JSON document and correspondingly represents an instance of an RDF data model. However, JSON-LD also extends the

RDF data model to optionally allow JSON-LD to serialize Generalized RDF Datasets. The JSON-LD extensions to the RDF data model are:

- In JSON-LD properties can be IRIs or blank nodes whereas in RDF properties (predicates) have to be IRIs. This means that JSON-LD serializes generalized RDF Datasets.
- In JSON-LD lists are part of the data model whereas in RDF they are part of a vocabulary, namely [RDF11-SCHEMA].
- RDF values are either typed literals (typed values) or language-tagged strings whereas JSON-LD also supports JSON's native data types, i.e., number, strings, and the boolean values true and false. The JSON-LD Processing Algorithms and API specification [JSON-LD-API] defines the conversion rules between JSON's native data types and RDF's counterparts to allow round-tripping.

Summarized, these differences mean that JSON-LD is capable of serializing any RDF graph or dataset and most, but not all, JSON-LD documents can be directly interpreted as RDF as described in RDF 1.1 Concepts

2.2.2 schema.org

Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet, on web pages, in email messages, and beyond.
Schema.org vocabulary can be used with many different encodings, including RDFa, Microdata and JSON-LD. These vocabularies cover entities, relationships between entities and actions, and can easily be extended through a well-documented extension model. Over 10 million sites use Schema.org to markup their web pages and email messages. Many applications from Google, Microsoft, Pinterest, Yandex and others already use these vocabularies to power rich, extensible experiences.
Schema.org is sponsored by Google, Microsoft, Yahoo and Yandex. The vocabularies are developed by an open community process, using the public-schemaorg@w3.org mailing list and through GitHub.
A shared vocabulary makes it easier for webmasters and developers to decide on a schema and get the maximum benefit for their efforts. It is in this spirit that the sponsors, together with the larger community have come together, to provide a shared collection of schemas.

Chapter 3

Examples

3.1 Use case

Inspired by the package.json format prescribed by the npm community and using an ontology described on http://schema.org below is a relatively short JSON-LD document that describes the Fidgit codebase. Let's call it code.jsonld for now.

```
{
  "@context": "http://schema.org",
  "@type": "Code",
  "name": "Fidgit",
  "codeRepository": "https://github.com/arfon/fidgit",
  "citation": "http://dx.doi.org/10.6084/m9.figshare.828487",
  "description": "An ungodly union of GitHub and Figshare http://fidgit.arfon.org",
  "dateCreated": "2013-10-19",
  "license": "http://opensource.org/licenses/MIT",
  "author": {
    "@type": "Person",
    "name": "Arfon Smith",
    "@id": "http://orcid.org/0000-0002-3957-2474",
    "email": "arfon@github.com"
  }
}
```

Note the first two line (@context and @type) defines the context for the key/value pairs in the JSON structure so that name means the name of the codebase. You can see the full ontology for Code here but this should mostly be straightforward to understand.
Once we get to the authors attribute we're now entering a new context, that of an individual. As we're still using the schema.org ontology for type Person we only need to set the @type attribute here.
There are a bunch more attributes that we could set here but this feels like a minimal set of information that is sufficient for citation (and therefore credit and attribution for the author).

3.2 SPARQL and RDF integration

Queries are represented in a JSON structure. The most easy way to get acquainted with this structure is to try the examples in the queries folder through sparql-to-json. All examples of the SPARQL 1.1 specification have been included, in case you wonder how a specific syntactical construct is represented.
Here is a simple query in SPARQL:

```
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
SELECT ?p ?c WHERE {
    ?p a dbpedia-owl:Artist.
    ?p dbpedia-owl:birthPlace ?c.
    ?c <http://xmlns.com/foaf/0.1/name> "York"@en.
}
```

And here is the same query in JSON:

```
{
  "type": "query",
  "prefixes": {
    "dbpedia-owl": "http://dbpedia.org/ontology/"
  },
  "queryType": "SELECT",
  "variables": [ "?p", "?c" ],
  "where": [
    {
      "type": "bgp",
      "triples": [
        {
          "subject": "?p",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "object": "http://dbpedia.org/ontology/Artist"
        },
        {
          "subject": "?p",
          "predicate": "http://dbpedia.org/ontology/birthPlace",
          "object": "?c"
        },
        {
          "subject": "?c",
          "predicate": "http://xmlns.com/foaf/0.1/name",
          "object": "\"York\"@en"
        }
      ]
    }
  ]
}
```

The representation of triples is the same as that of the N3.js library. Queries are represented in a JSON structure. The most easy way to get acquainted with this structure is to try the examples in the queries folder through sparql-to-json. All examples of the

SPARQL 1.1 specification have been included, in case you wonder how a specific syntactical construct is represented.
Here is a simple query in SPARQL:

```
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
SELECT ?p ?c WHERE {
    ?p a dbpedia-owl:Artist.
    ?p dbpedia-owl:birthPlace ?c.
    ?c <http://xmlns.com/foaf/0.1/name> "York"@en.
}
```

And here is the same query in JSON:

```
{
  "type": "query",
  "prefixes": {
    "dbpedia-owl": "http://dbpedia.org/ontology/"
  },
  "queryType": "SELECT",
  "variables": [ "?p", "?c" ],
  "where": [
    {
      "type": "bgp",
      "triples": [
        {
          "subject": "?p",
          "predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "object": "http://dbpedia.org/ontology/Artist"
        },
        {
          "subject": "?p",
          "predicate": "http://dbpedia.org/ontology/birthPlace",
          "object": "?c"
        },
        {
          "subject": "?c",
          "predicate": "http://xmlns.com/foaf/0.1/name",
          "object": "\"York\"@en"
        }
      ]
    }
  ]
}
```

The representation of triples is the same as that of the N3.js library.