|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  РОССИЙСКОЙ ФЕДЕРАЦИИ  Федеральное государственное бюджетное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | | |  | Кафедра математического обеспечения вычислительных систем | | | **декомпиляция машинного кода архитектуры x86-64 на язык ПРОГРАММИРОВАНИЯ c** | | | | *Курсовая работа* | | | |  | | | |  | | | |  | | Работу выполнил студент  группы ПМИ-4 3 курса механико-математического факультета  Пухов Н.А. | |  | | Научный руководитель:  старший преподаватель кафедры МОВС  Лядова Л.Н.  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г. | |  | |  | | Пермь 2021 | | | | |
|  |  |

**Аннотация**

текст.

**Оглавление**

[Введение 5](#_Toc70015557)

[Глава 1. Анализ предметной области 6](#_Toc70015558)

[1.1 Понятия и способы представления онтологий 6](#_Toc70015559)

[1.1.1 Виды онтологий 6](#_Toc70015560)

[1.1.2 Способы представления онтологий 6](#_Toc70015561)

[1.2 Анализ онтологических ресурсов 8](#_Toc70015562)

[1.2.1 Разработка критериев оценки ресурсов 8](#_Toc70015563)

[1.2.2 Поиск и оценка онтологических ресурсов 9](#_Toc70015564)

[1.3 Методы сравнения онтологий 10](#_Toc70015565)

[1.3.1 Количественные методы оценки 10](#_Toc70015566)

[1.3.2 Качественные методы оценки 11](#_Toc70015567)

[Глава 2. Проектирование системы 12](#_Toc70015568)

[2.1 Требования к системе 12](#_Toc70015569)

[2.2 Выбор технологий 13](#_Toc70015570)

[2.2.1 Выбор языка программирования 13](#_Toc70015571)

[2.2.2 Выбор фреймворка 13](#_Toc70015572)

[2.2.3 Работа с онтологиями 13](#_Toc70015573)

[2.4 Проектирование поведения системы 14](#_Toc70015574)

[2.5 Проектирование запросов 15](#_Toc70015575)

[2.5.1 Получение списка и числа классов 16](#_Toc70015576)

[2.5.2 Получение списка и числа сущностей 17](#_Toc70015577)

[2.5.3 Получение списка и числа свойств 17](#_Toc70015578)

[2.5.4 Получение глубины онтологии 18](#_Toc70015579)

[Глава 3. Разработка микросервиса 19](#_Toc70015580)

[3.1 Проектирование API 19](#_Toc70015581)

[3.1.1 Поиск онтологий 19](#_Toc70015582)

[3.1.2 Запрос метрик онтологии 20](#_Toc70015583)

[3.2 Программирование микросервиса 21](#_Toc70015584)

[3.2.1 Установка необходимых пакетов и настройка среды разработки 21](#_Toc70015585)

[3.2.2 Создание структуры приложения 24](#_Toc70015586)

[3.2.3 Программирование метода поиска 24](#_Toc70015587)

[3.2.4 Программирование метода получения метрик с помощью API 25](#_Toc70015588)

[3.2.5 Программирование метода получения метрик с помощью запросов к онтологии 27](#_Toc70015589)

[3.2.6 Программирование обработчиков запросов 27](#_Toc70015590)

[3.3. Тестирование и отладка 28](#_Toc70015591)

[3.4 Размещение на хостинге Heroku 32](#_Toc70015592)

[Заключение 34](#_Toc70015593)

[Библиографический список 35](#_Toc70015594)

[Приложение А. Листинг 36](#_Toc70015595)

Введение

На сегодняшний день существует большое количество разнообразных программ с открытым исходным кодом. Исходный код таких программ написан с использованием высокоуровневых языков программирования, среди которых C, C++, Java, Python и другие. Мотивация использования языков программирования с высоким уровнем абстракции обусловлена желанием иметь понятный, расширяемый код, который имеет возможность исполняться на разных видах операционных систем и архитектур процессора. Однако существует множество открытых программ, которые были написаны под определенную архитектуру процессора для конкретной операционной системы с использованием языка ассемблера. Среди такого рода программ можно выделить драйверы устройств или прошивки для микропроцессорной техники. Более того, программисты часто используют вставки кода на ассемблере в своих программах, написанных на высокоуровневых языках.

Использование ассемблера как низкоуровневого кода, который плохо понятен человеку, необходимо с целью повышения производительности программ. Более того, ассемблерный код имеет больше возможностей по оперированию устройством, чем код, написанный на языке высокого уровня. Но возникает большая проблема, связанная с чтением такого низкоуровневого кода другими людьми, которые могут даже не знать ассемблер. Решение данной проблемы состоит в использовании декомпилятора, который транслирует код из низкоуровневого представления в высокоуровневое. Среди существующих декомпиляторов можно выделить: IDA Hex-Rays Decompiler, Ghidra Decompiler, RetDec, Snowman, однако все они имеют определенные недостатки и не всегда генерируют высокоуровневый код в понятном человеку виде.

Объектом данного исследования является низкоуровневый код в ассемблерном или машинном представлении для архитектуры процессоров X86‑64. Предмет исследования – представление такого кода в высокоуровневой форме.

Цель данной работы – декомпиляция кода на языке ассемблера архитектуры X86‑64 в код высокоуровневого языка C в понятном человеку виде.

Для достижения поставленной цели должны быть решены следующие задачи:

1. Проанализировать существующие бесплатные программные реализации, которые позволяют декомпилировать код на языке ассемблера архитектуры X86‑64 в высокоуровневое представление. Выявить критические проблемы этих реализаций, а именно понять, какие примеры низкоуровневого кода они не могут должным образом декомпилировать.
2. Разработать собственные пути решения выявленных проблем. Для этого создать свой собственный программный продукт или взять готовую реализацию и исправить эти проблемы в ней.
3. Анализ существующих решений

В данной главе производится критический анализ существующих программных реализаций декомпиляции кода по следующим критериям:

1. Бесплатность и открытость исходного кода
2. Возможность встраивания
3. Высокая скорость работы
4. Декомпиляция заданного участка машинного кода
5. Использование промежуточного языка виртуальной машины
6. Высокое качество генерируемого кода
7. Полноценная поддержка векторных и математических операций

На данный момент разработано несколько декомпиляторов, которые, в частности, позволяют транслировать код с языка ассемблера или непосредственно машинный код архитектуры X86-64 в высокоуровневый код языка C. Среди популярных декомпиляторов можно выделить:

1. IDA Hex-Rays Decompiler. Данный декомпилятор является платным с закрытым исходным кодом. Он является плагином к программе IDA Pro. Имеет поддержку архитектур процессора x86-64, ARM, MIPS. Отличительная особенность – наличие отладчика, благодаря которому можно в динамике увидеть работу сгенерированного кода.
2. RetDec. Данный декомпилятор был представлен компанией Avast. Он является бесплатным с открытым исходным кодом и имеет поддержку множества архитектур процессоров, среди которых x86-64, ARM, MIPS. Использует промежуточный язык виртуальной машины LLVM. Имеет возможность восстановления иерархии классов и виртуальных таблиц. Генерирует код на языках C и Python. Основной недостаток – генерация кода уступает по качеству коду, который генерирует Ghidra Decompiler. Это в первую очередь проявляется в использовании плохо читаемых предопределенных функций и макросов вместо стандартных операторов языка C. Еще одним недостатком является отсутствие возможности декомпиляции заданного участка кода, а не всего программного образа.
3. Snowman. Особенностью данного декомпилятора является возможность трансляции кода в язык C++. Также он может быть легко встроен в любую программу и может декомпилировать заданный участок машинного кода. Однако он имеет такой же недостаток, что у декомпилятора RetDec, а именно плохо читаемый код во многих случаях.
4. Ghidra Decompiler. Данный декомпилятор был представлен агентством национальной безопасности США. Использует промежуточный язык PCode абстрактной виртуальной машины. Главное его отличие от всех остальных реализаций заключается в использовании виртуальной машины Java в качестве среды для выполнения основной программы Ghidra, хотя он сам написан на языке C++. Отсюда вытекает главный недостаток – низкая скорость работы декомпилятора. Еще один недостаток – это отсутствие возможности встроить данный декомпилятор в свою программу, а также декомпилировать заданный участок машинного кода. Однако главное преимущество данного декомпилятора – генерация им хорошо читаемого кода в большинстве случаев. Это достигается за счет более совершенных эвристических алгоритмов генерации высокоуровневого кода.

В табл. 1.1 представлено соотнесение декомпиляторов изложенным выше требованиям.

Таблица 1.1. Соотнесение декомпиляторов требованиям

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Требование** | **IDA Hex-Rays Decompiler** | **RetDec** | **Snowman** | **Ghidra Decompiler** |
| Бесплатность и открытость исходного кода | − | + | + | + |
| Возможность встраивания | − | + | + | − |
| Декомпиляция заданного участка машинного кода | − | − | + | − |
| Высокая скорость работы | + | + | + | − |
| Использование промежуточного языка виртуальной машины | − | + | − | + |
| Высокое качество генерируемого кода | + | − | − | + |
| Полноценная поддержка векторных и математических операций | − | − | − | − |

Важно отметить, что общая проблема всех перечисленных декомпиляторов – плохая работа с тем кодом, где преобладают векторные или математические операции и где наличествует пересылка данных из одного участка памяти в другой. Также большинство декомпиляторов могут анализировать и декомпилировать только цельные программные образы вместо заданных участков машинного кода.

Для решения двух названных выше основных проблем существует два варианта: улучшение готовой реализации или создание своей собственной с нуля. Рассмотрим первый вариант.

1. IDA Hex-Rays Decompiler. Данный декомпилятор не имеет открытого исходного кода, поэтому не может быть улучшен.
2. RetDec. Данный декомпилятор не подходит по многим причинам, в частности он не имеет возможности качественной декомпиляции заданного участка машинного кода. Таким образом улучшение данного декомпилятора приведёт к необходимости фундаментальных изменений, что нецелесообразно с точки зрения временных затрат.
3. Snowman. Данный декомпилятор не использует промежуточный язык виртуальной машины, что является критическим недостатком, поскольку не дает возможности виртуально вычислять те константные выражения, которые используют память компьютера. Таким образом улучшение данного декомпилятора приведёт к необходимости фундаментальных изменений, что нецелесообразно с точки зрения временных затрат.
4. Ghidra Decompiler. Данный декомпилятор не может встраиваться в другие программы, имеет привязку к виртуальной машине JVM, поэтому не может быть улучшен.

На основании вышеприведенных умозаключений по каждому декомпилятору было принято решение создать собственную программную реализацию с нуля.

1. Теоретические основы декомпиляции

В данной главе описываются теоретические основы процесса декомпиляции и его этапы.

* 1. Постановка задачи декомпиляции

Пусть - отображение, которое преобразует высокоуровневый код  на некотором языке программирования  в машинный код  некоторого процессора . Будем называть отображение  компилятором. Тогда отображение  будет являться идеальным декомпилятором.

Для компилятора, транслирующего код на языке C в машинный код архитектуры процессора X86-64, соответствующего идеального декомпилятора не может существовать. Это связано с тем, что в процессе компиляции безвозвратно теряется информация о названии переменных и функций, удаляются комментарии в коде. При использовании оптимизатора в компиляторе, теряется первоначальная структура исходного кода.

* 1. Конвейер декомпилятора

Процесс декомпиляции можно представить в виде нескольких основных этапов преобразования данных из низкоуровневого кода в высокоуровневый код.

2.1.1 Декодирование

Первым этапом является декодирование машинного кода, который представляет из себя последовательность байт, в структуры описания инструкций языка ассемблер. Структуры описания инструкций содержат информацию о типе инструкции, значениях и типах операндов, их количестве и т.д. Например, из инструкции “mov [rsp+0x10], 0x1” можно извлечь следующую информацию:

1. Тип инструкции – mov.
2. Первый операнд имеет указатель, который состоит из базового адреса, хранящегося в регистре “rsp”, и числового смещения “0x10”.
3. Второй операнд имеет числовое значение “0x1”.

2.1.2 Транслирование инструкций языка ассемблер в P-код

Вторым этапом декомпиляции является преобразование структур описания инструкций языка ассемблер в аппаратно-независимый P-код. P-код – концепция аппаратно-независимого исполняемого кода в программировании, часто его определяют как «Ассемблер для гипотетического процессора», который служит необходимым промежуточным абстрактным слоем между аппаратно-зависимым языком ассемблер X86-64 и высокоуровневым языком C. P-код содержит минимальный набор команд, благодаря которому можно реализовать любой алгоритм, таким образом P-код имеет полноту по Тьюрингу.

2.1.4 Построение графа потока управления

Третьим этапом является построение ориентированного графа потока управления из листинга инструкций P-кода. В качестве вершин графа выступают блоки инструкций P-кода, которые не содержат команд условных или безусловных переходов за исключением последней команды, которая всегда осуществляет переход в другую область P-кода. В качестве рёбер ориентированного графа используются направления условных или безусловных переходов. Из одной вершины может выходить до двух ребер.

Вершина, из которого нет выходящих ребер, является конечной и всегда содержит последней инструкцией команду возврата, представляющую из себя безусловный переход в другую область программы. Конечных вершин может быть несколько. Вершина, в которую не входит ни одно ребро, является начальной вершиной в графе потока управления и всегда представлена одна.

2.1.5 Построение абстрактного синтаксического дерева из P-кода

Четвертым этапом является построение абстрактного синтаксического дерева из сгенерированного на предыдущем этапе листинга команд P-кода. Абстрактное синтаксическое дерево – это промежуточное представление высокоуровневого кода.

2.4 Проектирование поведения системы

Система может получать на вход онтологию в виде файла и в виде URL. В зависимости от этого обработка может происходить по-разному. Покажем работу системы с помощью UML диаграмм активности (Activity diagram).

На рисунке 2.2 изображено поведение системы при получении на вход адреса до онтологии. На рисунке 2.3 показано как ведёт себя система при получении онтологии.

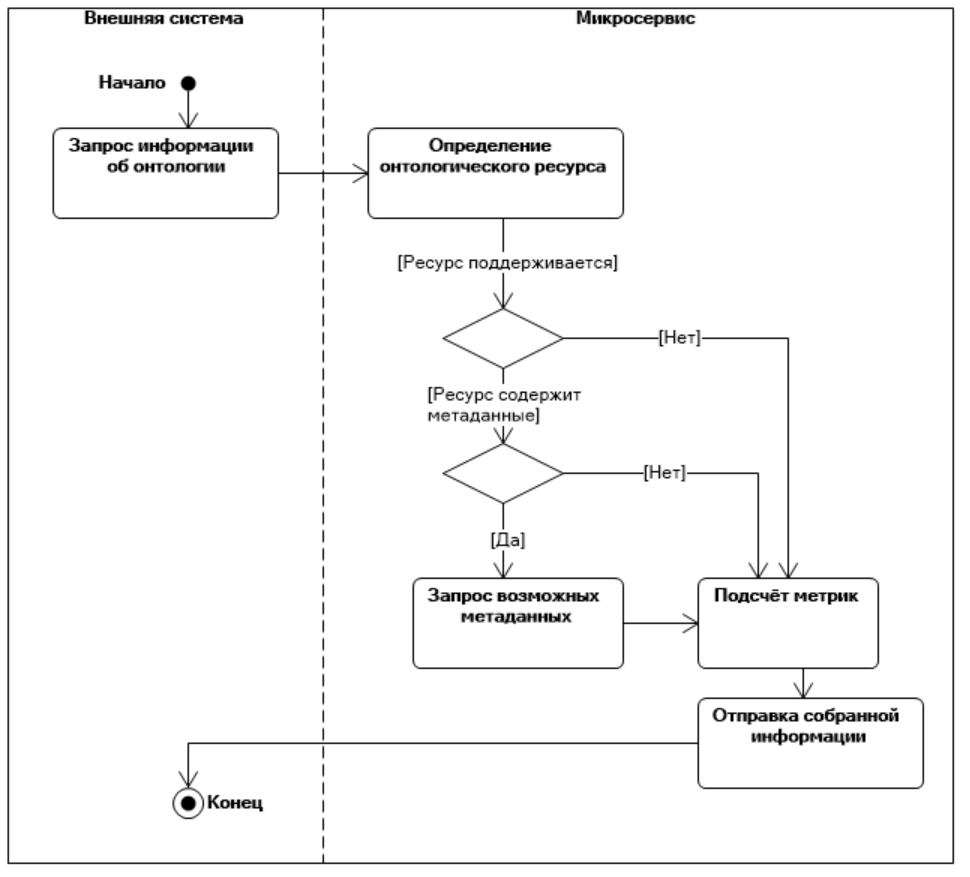


Рисунок 2.2 «Поведение системы при получении URL»

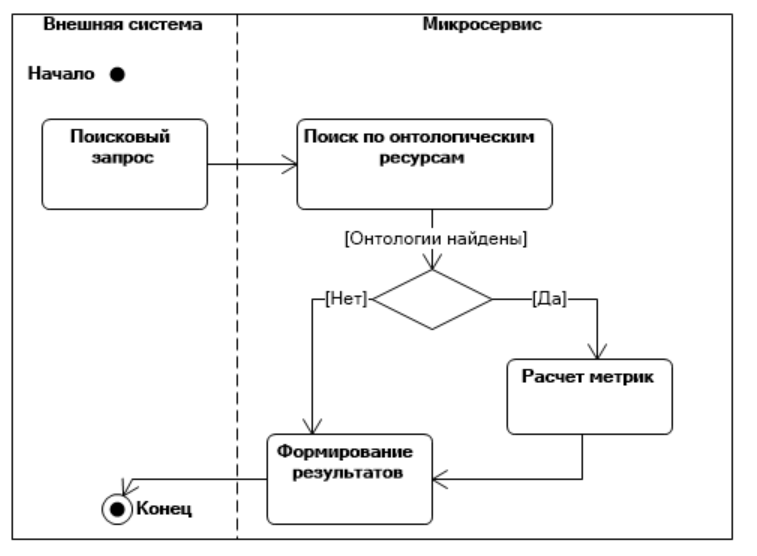


Рисунок 2.3 «Поведение системы при получении поискового запроса»

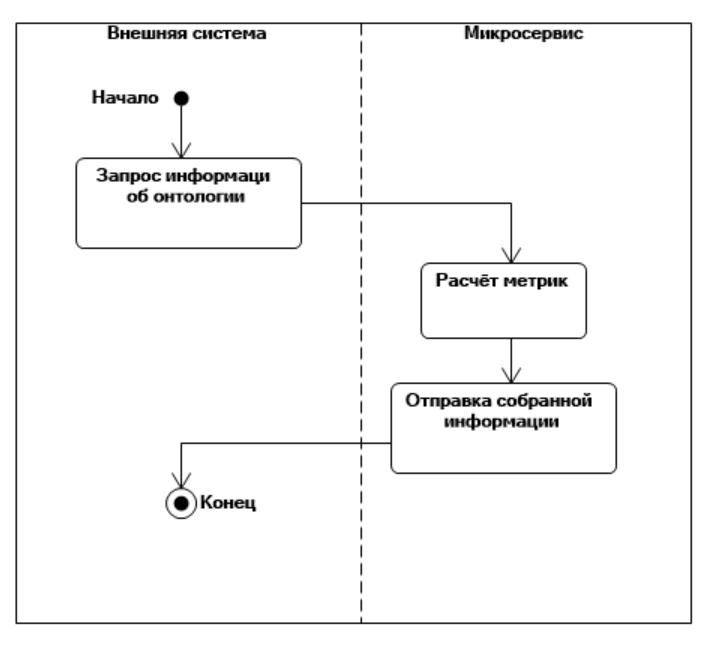


Рисунок 2.3 «Поведение системы при получении онтологии»

2.5 Проектирование запросов

Для того, чтобы получить описанные ранее метрики для онтологий, необходимо осуществлять запросы к онтологиям на языке запросов SparQL и к API онтологических ресурсов с помощью HTTP-запросов.

2.5.1 Получение списка и числа классов

Для того, чтобы получить список класса посредством SparQL необходимо осуществить следующий запрос:

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT distinct ?class

WHERE {

?class a owl:Class

}

order by ?class

Для получения количества классов в онтологии, описанной на языке OWL, необходимо осуществить запрос следующего вида:

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(?s) AS ?totalNumberOfInstances)

WHERE { ?s a owl:Class }

Запросы к онтологии, описанной на языке RDF осуществляются аналогично, но префикс owl: изменяется на rdf:.

Получение списка классов:

SELECT distinct ?class

WHERE {

?class a rdf:Class

}

order by ?class

Получение количества классов:

SELECT (COUNT(?s) AS ?totalNumberOfInstances)

WHERE { ?s a rdf:Class }

Из рассматриваемых онтологических ресурсов полноценно поддерживает API лишь NCBO Bioportal. Рассмотрим запросы на примере этого портала.

Для того, чтобы получить список классов, необходимо обратиться к адресу http://data.bioontology.org/ontologies/%ontology\_name%/classes?apikey=%apikey%&format=json с помощью GET-запроса. Вместо параметра %ontology\_name% необходимо подставить имя онтологии на портале, вместо параметра %apikey% необходимо указать ключ доступа к API NCBO BioPortal, который выдаётся каждому пользователю при регистрации. Параметр format=json указывает, что сервер пришлёт ответ в формате json. При необходимости, можно использовать и другие форматы, например, XML.

Для получения количества классов необходимо отправить GET-запрос на адрес http://data.bioontology.org/ontologies/%ontology\_name%/metrics?apikey=%apikey%&format=json

2.5.2 Получение списка и числа сущностей

Для того, чтобы получить список всех сущностей, можно воспользоваться следующим запросом на языке SPARQL:

SELECT DISTINCT ?entity

WHERE {

{ ?property a owl:NamedIndividual .}

}

Запрос, подсчитывающий количество индвидов выглядит следующим образом:

SELECT (COUNT(DISTINCT ?entity) as ?count)

WHERE {

{ ?property a owl:NamedIndividual .}

}

Стоит отметить, что в обоих запросах присутствует модификатор DISTINCT: он необходим для того, чтобы избежать повторного подсчёта индивидов. Повторный подсчёт может значительно сказаться на искомых метриках.

Как было описано ранее, для работы с онтологями на языке RDF необходимо изменить префикс с owl: на rdf:.

Чтобы получить эти данные с помощью API NCBO BioPortal, необходимо обратиться посредством GET-запроса на следующий адрес: http://data.bioontology.org/ontologies/%ontology\_name%/metrics?apikey=%apikey%&format=json

Для выполнения запроса необходимо подставить имя онтологии и ключ API. Портал возвращает json-объект, включающий в себя число индивидов. Индивиды находятся в поле individuals.

2.5.3 Получение списка и числа свойств

Для получения числа свойств можно воспользоваться запросом, аналогичным запросу в п. 2.5.2. Разница заключается в том, что в этом случае мы считаем количество предикатов ?prop.

SELECT (COUNT(DISTINCT ?prop) as ?count)

WHERE {

{ ?prop a owl:ObjectProperty .}

}

Для того, чтобы получить список свойств, необходимо изменить запрос – убрать агрегатную функцию COUNT():

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(DISTINCT ?property) as ?res)

WHERE {

{ ?property a owl:ObjectProperty .}

}

Чтобы запросить число с портала NCBO BioPortal, необходимо совершить GET-запрос к всё тому же адресу, подставляя необходимые для запроса параметры: пользовательский ключ API NCBO BioPortal и имя анализируемой онтологии. Итоговая строка запроса выглядит следующим образом: http://data.bioontology.org/ontologies/%ontology\_name%/metrics?apikey=%apikey%&format=json

После исполнения запроса API возвращает json-объект с информацией о метриках онтологии. Необходимая информация содержится в поле properties.

2.5.4 Получение глубины онтологии

Для получения глубины онтологии можно воспользоваться запросом на языке SparQL. Стоит отметить, что необходима замена префиксов owl: в зависимости от языка описания, на котором представлена онтология.

select ?class (count(?mid)-1 as ?depth) {

{

select ?root {

?root a owl:Class

filter not exists {

?root rdfs:subClassOf ?superroot

filter ( ?root != ?superroot )

}

}

}

?class rdfs:subClassOf\* ?mid .

?mid rdfs:subClassOf\* ?root .

}

group by ?class

order by ?depth

Данную метрику можно получить с помощью запроса к API NCBO BioPortal. Для этого необходимо осуществить GET-запрос к URL http://data.bioontology.org/ontologies/%ontology\_name%/metrics?apikey=%apikey%&format=json. В результате выполнения запроса сервер отправит json-объект со всеми метриками. Глубина онтологии находится в поле maxDepth.

1. Разработка микросервиса

В данной главе описывается процесс разработки микросервиса с использованием языка программирования Python и веб-фреймворка Flask.

3.1 Проектирование API

Разрабатываемая система выполняет функцию поиска онтологий по запросу. Для этого она выполняет поиск по запрограммированным онтологическим ресурсам. После нахождения онтологий система возвращает для найденных по запросу онтологий метрики, описанные ранее. Кроме этого, должна быть возможность передать любую онтологию в систему, чтобы поиск не выполнялся, но метрики были посчитаны.

Исходя из этого, можно выделить две функции: поиск онтологий и расчёт метрики. Обе функции должны быть доступны пользователю и иметь раздельные методы. Результаты работы методов будут возвращены в формате json, так как он является наиболее универсальным, легко читается как человеком, так и различным программным обеспечением вне зависимости от языка программирования программы-получателя.

Пусть веб-сервис размещен по адресу localhost:5000/api/onto. При запросе на этот адрес система будет возвращать список доступных методов и описание параметров.

3.1.1 Поиск онтологий

Метод поиска онтологий размещен по адресу localhost:5000/api/onto/search/%keyword%. Вместо параметра %keyword% необходимо указать ключевое слово или предложение, которое будет искаться на онтологических ресурсах.

В результате выполнения запроса сервер возвращает json-объект с полями, описанными в табл. 3.1.

Таблица 3.1 «Структура ответа на поисковый запрос»

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| Keyword | Ключевое слово, по которому осуществлялся поиск | Строка |
| Ontologies | Массив объектов, описывающих онтологии и их метрики. | Массив |

В случае возникновения ошибки или ненахождения онтологий по указанному запросу, сервер возвращает json-объект с теми же полями, но в поле Ontologies будет пустой массив [].

* + 1. Запрос метрик онтологии

Для запроса метрик заданной онтологии необходимо осуществить GET-запрос на URL localhost:5000/api/onto/metrics/%URL%. Вместо параметра %URL% необходимо указать адрес до онтологии, метрики которой требуется получить.

В результате запроса сервер возвращает json-объект с полями, представленными в табл. 3.2.

Таблица 3.2 «Структура ответа на запрос метрик»

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| URL | Переданный в метод URL онтологии, для которой осуществляется расчёт метрик | Строка |
| Metrics | Объект, содержащий информацию об онтологии и рассчитанные метрики. | Объект |

В объекте Metrics содержится информация об онтологии, в том числе метаданные онтологии. Данные система получает с помощью запросов к онтологии и с помощью обращения к онтологическим ресурсам. Структура объекта представлена в таблице 3.3.

Таблица 3.3 «Структура объекта Metrics»

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| Metadata | Объект, содержащий метаданные онтологии | Объект |
| MetricsCounted | Объект, содержащий метрики онтологии, рассчитанные с помощью запросов к онтологии | Объект |
| MetricsFromResource | Объект, содержащий метрики онтологии, полученные из онтологического ресурса | Объект |

Объекты MetricsCounted и MetricsFromResource одинаковы по структуре, она описана в табл. 3.4. Разница состоит в том, что в объекте MetricsCounted размещены данные, полученные посредством запросов к онтологии на языке SparQL, а в объекте MetricsFromResourse размещаются данные, полученные из запросов к API онтологического ресурса, где онтология размещена. Если такого ресурса нет, разрабатываемое API возвращает пустой объект.

Таблица 3.4 «Структура объекта Metrics»

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| Classes | Массив, содержащий перечисление всех классов в онтологии | Массив строк |
| classCount | Объект, содержащий число классов в онтологии | Число |
| Individuals | Массив, содержащий перечисление всех индивидов в онтологии | Массив строк |
| indivindualsCount | Объект, содержащий число индивидов в онтологии | Число |
| Property | Массив, содержащий перечисление всех свойств в онтологии | Массив строк |
| propertiesCount | Объект, содержащий число всех свойств онтологии | Число |
| Depth | Число, характеризующее глубину онтологии | Число |

В случае отсутствия значений какого-либо вида возвращается пустой массив, и, соотвественно, число ноль.

3.2 Программирование микросервиса

Микросервис разработан на языке программирования Python версии 3.7. Для разработки используется редактор и отладчик Visual Studio Code. Для работы с API использована среда Postman.

3.2.1 Установка необходимых пакетов и настройка среды разработки

Для того, чтобы система работала в качестве веб-сервера, а также, могла работать с онтологиями на языках OWL и RDF, необходимо установить в язык Python заранее разработанные пакеты. Установка дополнительных пакетов для Python производится с помощью менеджера пакетов pip. Менеджер пакетов – это специально приложение, которое позволяет устанавливать пакеты в систему, а также, контролировать версии пакетов и их зависимостей. Pip распространяется вместе с языком Python, поэтому отдельная установка менеджера пакетов не требуется.

Для работы разрабатываемого приложения в качестве веб-сервера необходима установка фреймворка Flask и его зависимостей. Для этого, в терминале PowerShell необходимо выполнить команду pip install Flask. После установки проверим корректную работу пакета и настроим среду разработки Visual Studio Code для работы в режиме отладки. Для этого, запустим редактор, перейдём в настройки конфигурации отлажки, создадим конфигурацию, указывая на установленный экземпляр Python и с параметрами, необходимыми для корректной работы в режиме отладки.

В файле конфигурации launch.json укажем:

{

"name": "Python: Flask",

"type": "python",

"request": "launch",

"module": "flask",

"env": { “FLASK\_APP": "test.py" },

"args": [

"run",

"--no-debugger",

"--no-reload"

],

"jinja": true

}

После этого, в настройках запуска добавлена необходимая для работы конфигурация. Создадим файл test.py со следующим содержанием:

from flask import Flask

import sys

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return sys.version

Теперь при переходе по адресу localhost:5000/ будет отображаться установленная версия Python. Убедимся, что всё работает корректно, для этого отправим GET-запрос через Postman. Запрос успешно отправлен, система отправила в ответ версию Python, см. рис. 3.1.

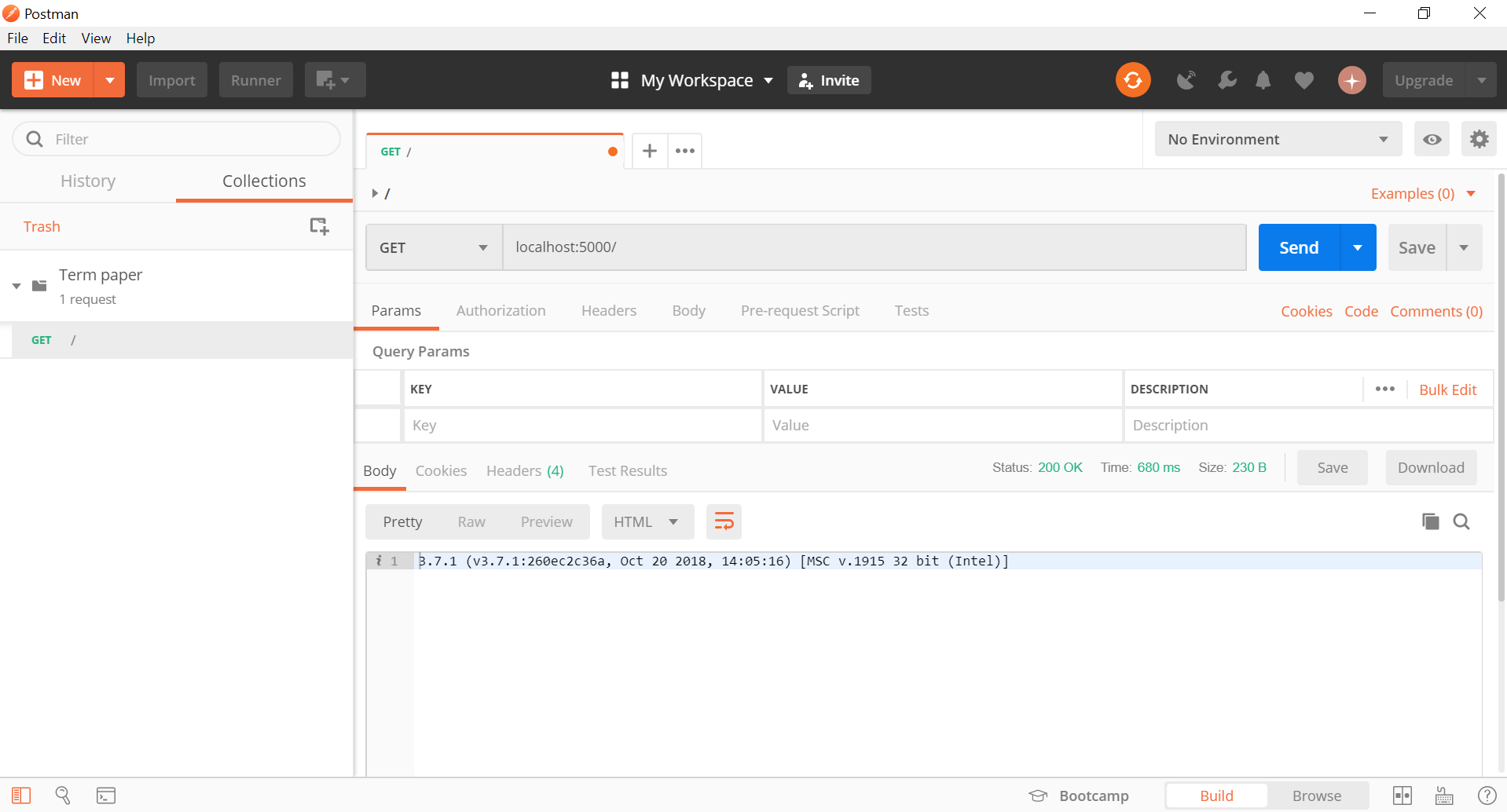


Рисунок 3.1 «Отправка тестового запроса»

После этого, можно установить аналогичным образом остальные пакеты. В ходе работы потребуются:

1. Owlready2 – пакет, позволяющий работать с онтологиями на языке OWL, преобразовывая их в объект языка Python.
2. Rdflib – пакет, позволяющий аналогичным образом работать с онтологиями на языке RDF, а также, отправлять запросы на языке запросов SparQL. Стоит отметить, что этот пакет необходим для работы предыдущего.
3. Requests – пакет, предназначенный для работы с http-запросами в Python. С помощью этого пакета будет осуществляться работа с внешними API.

Не требует установки Json – пакет, позволяющий сериализовать объекты Python в формат json и делать обратное действие – десериализовать json в объекты. Он пригодится как для работы с внешними API, так и для формирования результатов запросов к разрабатываемой системе.

Для того, чтобы система легко разворачивалась, создадим в корне проекта файл requirements.txt, в котором будут записаны все необходимые библиотеки, указав их конкретные версии:

Owlready2==0.14

requests==2.19.1

Flask==1.0.2

rdflib==4.2.2

Теперь их можно установить одной командой pip install -r requirements.txt вместо пяти раздельных. Это пригодится в дальнейшем.

* + 1. Создание структуры приложения

Для удобства программиста необходимо разработать структуру приложения и разместить разные фрагменты программы в разные файлы. Такой подход позволит повысить читаемость и поддерживаемость разрабатываемого микросервиса.

При разработке будем придерживаться следующей структуры:

1. config.py – файл конфигурации, содержащий константы, например, ключ API NCBO BioPortal, количество рассматриваемых онтологий в поисковой выдаче и т.д.
2. help.py – модуль программы, содержащий вспомогательные функции, например, определение типа онтологии, запросы метрик из онтологии RDF, OWL, из API и т.д.
3. main.py – модуль программы, содержащий непосредственно входную точку приложения. В модуле размещены функции, отвечающие за обработку вызовов методов API.

Такое разграничение зон ответственности позволяет сделать файл main.py более понятным и читаемым, а файл help.py облегчить от использования методов роутера Flask, чем также повысить читаемость.

Кроме этого, при необходимости отказаться от использования Flask не придётся значительно переписывать программный код – методы, работающие с внешними API и с онтологиями не зависят от веб-фреймворка.

3.2.3 Программирование метода поиска

Как было описано ранее, система должна осуществлять поиск онтологий для замера метрик. Поиск онтологий на ресурсах с API (например, NCBO BioPortal) и без него значительно отличается. Если у онтологического ресурса есть API, необходимо отправить заданный запрос и получить искомые данные. Если API нет, задача становится менее тривиальной: приходится разбирать HTML-страничку, которые могут значительно друг от друга отличаться по структуре.

Создадим функцию, осуществляющую поиск по NCBO BioPortal. Как следует из документации портала, для осуществления поиска необходимо отправить GET-запрос следующего вида: http://data.bioontology.org/search?q=%query%&apikey=%apikey%&format=json. Вместо параметра %search% необходимо передать поисковый запрос, а вместо параметра %apikey% необходимо указать ключ доступа к API NCBO BioPortal, который выдаётся каждому пользователю при регистрации. Параметр format=json указывает, что сервер пришлёт ответ в формате json. При необходимости можно использовать и другие форматы, например, XML.

Важным является то, что такой поиск ищет не только по названиям онтологий, но и по классам, поэтому поисковая выдача получается очень объёмной. Для целей исследования я ввёл в файле config.py переменную, ограничивающую размер выдачи: RESULTS\_COUNT.

В результате исполнения запросов, сервер отправляет ответ фиксированной структуры. Наиболее интересные для нас поля – массив collection, содержащий непосредственно результаты поиска в виде объектов. В кажлом объекте содержится ссылка на онтологию: links.ontology. В результате выполнения поиска, метод микросервиса возвращает объект с массивом адресов онтологий для дальнейшей обработки. Код метода приведён далее.

def searchNCBO(query, count):

searchUrl = f'http://data.bioontology.org/search?q={query}&apikey={c.NCBO\_API\_KEY}'

r = requests.get(searchUrl)

res = {

'ontologies': []

}

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()['collection']

if (count > len(items)):

count = len(items)

for item in range(count):

res['ontologies'].append(items[item]['links']['ontology'])

return res

3.2.4 Программирование метода получения метрик с помощью API

Для онтологий, полученных с помощью онтологического ресурса NCBO BioPortal можно легко получить рассчитанные порталом метрики. Это делается с помощью запросов к API, описанных во второй главе работы. Рассмотрим подробнее реализацию метода.

def getNCBOMetrics(name):

res = {

'classes': [],

'classCount': -1,

'individuals': [],

'individualsCount': -1,

'properties': [],

'propertiesCount': -1,

'depth': -1

}

# Get class list

requestUrl = f'http://data.bioontology.org/ontologies/{name}/classes?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

for item in items['collection']:

res['classes'].append(item['prefLabel'])

# Get properties list

requestUrl = f'http://data.bioontology.org/ontologies/{name}/properties?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

for item in items:

res['properties'].append(item['label'])

# Get all numbers

requestUrl = f'http://data.bioontology.org/ontologies/{name}/metrics?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

res['classCount'] = items['classes']

res['individualsCount'] = items['individuals']

res['propertiesCount'] = items['properties']

res['depth'] = items['maxDepth']

return json.dumps(res)

Данный метод отправляет запросы к API и считывает результаты, преобразовывая их к выходному формату разрабатываемого микросервиса. В результате работы метода возвращается JSON-объект описанной ранее структуры. Стоит отметить, что на вход метод может принимать только идентификаторы онтологий, которые были загружены на портал NCBO BioPortal.

3.2.5 Программирование метода получения метрик с помощью запросов к онтологии

Приложение позволяет рассчитать метрики с помощью запросов к онтологии на языке SparQL. Для выполнения запросов, разработанных во второй главе, были установлены пакеты Owlready2 для работы с онтологиями в формате OWL и RDFlib для работы с онтологиями в формате RDF. Для того, чтобы определить, какого типа онтология поступила в систему, приложение пытается считать её с помощью методов для OWL-онтологий. При возникновении ошибки приложение пытается считать онтологию как описанную с помощью RDF. Фрагмент кода приведён ниже:

def isRDF(uri):

try:

onto\_path.append("/onto/")

my\_world = World()

onto = my\_world.get\_ontology(uri)

onto.load()

return False

except:

return True

Затем, в зависимости от типа поступившей онтологии, выполняется соответствующий метод расчёта метрик, выполняющий SparQL-запросы. Для выполнения запросов, необходимо загрузить онтологию методом соответствующего пакета, получить объект типа Graph и выполнить метод объекта .query(%запрос%) или .query\_owlready(%запрос%) соответственно.

В результате выполнения метода расчета метрик, возвращается объект описанной ранее структуры. В случае возникновения ошибок, в соответствующие поля объекта записываются сообщения об ошибке.

* + 1. Программирование обработчиков запросов

Для работы с запросами необходимо объявить соответствующие методы в файле main.py, а также, внести соответствующие маршруты в Route приложения. Для этого, для каждого маршрута объявляется свой метод, ему в соответствие ставится запись в маршрутизаторе (роутере), например:

@app.route("/api/onto/metrics/<path:uri>")

def metrics(uri):

if (h.isRDF(uri)):

return h.getRDFMetrics(uri)

else:

return h.getOWLMetrics(uri)

В представленном фрагменте кода показан метод API, возвращающий метрики онтологии по её URI. Как видно из кода, метод размещен по адресу /api/onto/metrics, а в качестве параметра принимает URI онтологии. Аналогичным образом запрограммированы и метод поиска онтологии: он размещён по адресу /api/onto/search/. В качестве параметра он принимает поисковый запрос.

Методы API возвращают строку – JSON-объект с результатами работы, структура объектов была описана ранее.

* 1. Тестирование и отладка

Для того, чтобы протестировать микросервис, воспользуемся онтологией с заранее расчитанными метриками, например, онтологией BEVON – онтологией спиртных напитков. Метрики онтологии описаны по адресу http://rdfs.co/bevon/latest/html#sec-classes. Указано, что в онтологии существует 66 классов и 18 свойств.

Проверим работу микросервиса, передав онтологию в метод /api/onto/metrics. Получен следующий результат:

{

"classes": [

"Aging",

"AlcoholicBeverage",

"Ale",

"AmericanWhiskey",

"Beer",

"BeerBottle",

"BeerCocktail",

"BeerKeg",

"Beverage",

"BeverageCan",

"BourbonWhiskey",

"Brandy",

"Brewery",

"CanadianWhisky",

"CarbonatedWater",

"Cocktail",

"Cognac",

"Container",

"DilutedSoju",

"DistilledBeverage",

"DistilledSoju",

"Distillery",

"DryStout",

"Dunkel",

"FermentationBase",

"FermentedBeverage",

"FruitWine",

"Gin",

"GlassBottle",

"Highball",

"IrishRedAle",

"IrishWhiskey",

"JapaneseWhisky",

"Keg",

"Lager",

"Liqueur",

"Longneck",

"MaltWhisky",

"MineralWater",

"MixedDrink",

"Mocktail",

"NonAlcoholicBeer",

"NonAlcoholicBeverage",

"PaleAle",

"PaleLager",

"Pilsner",

"Pisco",

"PlasticBottle",

"Porter",

"RedWine",

"Rum",

"Sake",

"ScotchWhisky",

"Shandy",

"SingleMaltScotch",

"SingleMaltWhisky",

"Soju",

"SpeysideSingleMalt",

"Stout",

"Tequila",

"TequilaBlanco",

"Vodka",

"Water",

"Whisky",

"WhiteWine",

"Wine"

],

"classCount": 66,

"individuals": [

"Aging",

"AlcoholicBeverage",

"Ale",

"AmericanWhiskey",

"Beer",

"BeerBottle",

"BeerCocktail",

"BeerKeg",

"Beverage",

"BeverageCan",

"BourbonWhiskey",

"Brandy",

"Brewery",

"CanadianWhisky",

"CarbonatedWater",

"Cocktail",

"Cognac",

"Container",

"DilutedSoju",

"DistilledBeverage",

"DistilledSoju",

"Distillery",

"DryStout",

"Dunkel",

"FermentationBase",

"FermentedBeverage",

"FruitWine",

"Gin",

"GlassBottle",

"Highball",

"IrishRedAle",

"IrishWhiskey",

"JapaneseWhisky",

"Keg",

"Lager",

"Liqueur",

"Longneck",

"MaltWhisky",

"MineralWater",

"MixedDrink",

"Mocktail",

"NonAlcoholicBeer",

"NonAlcoholicBeverage",

"PaleAle",

"PaleLager",

"Pilsner",

"Pisco",

"PlasticBottle",

"Porter",

"RedWine",

"Rum",

"Sake",

"ScotchWhisky",

"Shandy",

"SingleMaltScotch",

"SingleMaltWhisky",

"Soju",

"SpeysideSingleMalt",

"Stout",

"Tequila",

"TequilaBlanco",

"Vodka",

"Water",

"Whisky",

"WhiteWine",

"Wine"

],

"individualsCount": 66,

"properties": [

"aging",

"brewery",

"container",

"distillery",

"duration",

"fermentation\_base",

"food",

"ingredient",

"manufacturer",

"material",

"non\_alcoholic\_version",

"origin",

"previous\_content",

"quantity",

"related",

"variant",

"variantOf",

"volume"

],

"propertiesCount": 18,

"depth": 1

}

Как видно из результата, метрики соответствуют указанным автором: 18 свойств, 66 классов.

Тем не менее, при обращении к некоторым онтологиям, запросы могут не работать. Проблема заключается в некорректной работе метода list() пакета Owlready2 или RDFlib, применяемого к результатам запроса.

3.4 Размещение на хостинге Heroku

Для того, чтобы разместить веб-сервис для общего доступа в сети Интернет, был использован бесплатный хостинг для веб-приложений Heroku. Кроме этого, для того, чтобы публиковать приложение на хостинге использован git-репозиторий на портале GitHub.com.

Чтобы разместить приложение на Python с использованием фреймворка Flask на Heroku, необходимо добавить в проект два файла. Во-первых, это файл requirements.txt, который позволяет установить все необходимые зависимости при развертывании приложения на хостинге. Файл имеет следующее содержание, которое частично была описана ранее:

Flask==1.0.2

Owlready2==0.14

rdflib==4.2.2

requests==2.19.1

gunicorn==19.9.0

Кроме этого, необходимо добавить файл Procfile, который указывает Heroku на тип приложения (веб-сервис или фоновое приложение и т.д.), команду, которую необходимо выполнить и на файл с точкой входа. Создадим такой файл, укажем веб-сервер gunicorn и файл main.py.

web: gunicorn main:app

Приложение готово к развёртыванию. Создадим Pipeline – описание процесса развёртывания приложения и создадим само приложение «Term-paper» на Heroku, добавим его в созданный Pipeline. В настройках Pipeline укажем связь между процессом и репозиторием на GitHub.



Рисунок 3.2 «Настройка Pipeline»

После этого, приложение будет автоматически разворачиваться по адресу <https://term-paper.herokuapp.com> при отправке наборов изменений на GitHub.

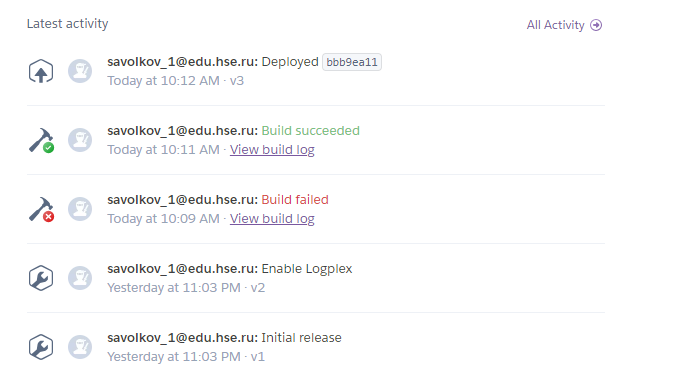


Рисунок 3.3 «История развёртываний»

Заключение

В ходе работы были проанализированы виды онтологий и их метрики, выбраны наиболее подходящие для сравнения онтологий между собой. Затем, проанализированы различные онтологические ресурсы, из них выбраны наиболее подходящие исходя из разработанных требований к ресурсам и проведённой оценки. Кроме этого, были написаны запросы к онтологиям на языке SparQL. Разработанные запросы можно использовать для получения метрик, предлагаемых к использованию для сравнения онтологий между собой.

Был спроектирован и разработан на языке Python прототип микросервиса, позволяющего автоматизировать запросы разработанных критериев и ускорить или автоматизировать процесс выбора онтологии для дальнейшего переиспользования.

В ходе работы возникло множество сложностей. Прежде всего стоит отметить неудобство работы с онтологическими ресурсами – большинство из найденных не имеют открытого API и представляют собой список онтологий с кратким описанием. Это значительно усложняет работу с ресурсами из программы, приходится писать отдельные методы для каждого сайта, что было бы дорого при коммерческой разработке. Кроме этого, многие из отобранных ресурсов имеют узкую профильную тематику – онтологии для медицины и биологии. Разработанные онтологические запросы работают медленно – в дальнейшем их необходимо переписать и оптимизировать. Исполнение запросов как к онтологиям, так и к API онтологических ресурсов может занимать долгое время, поэтому, в будущем необходимо реализовать кэширование полученных данных, чтобы экономить время пользователей системы.

Библиографический список

x

|  |  |
| --- | --- |
| 1. | Голиков В.Н. Применение онтологий [Электронный ресурс] URL: http:/​/​www.ict.nsc.ru/​ws/​YM2006/​10628/​golikov.html (дата обращения: 15.02.2019). |
| 2. | Ankolekar A. et al. DAML-S: Web service description for the semantic web // International Semantic Web Conference. Berlin, Heidelberg. 2002. pp. 348-363. |
| 3. | Talebpour M., Sykora M. D., Jackson T. KEOD 2018 // Ontology selection for reuse: Will it ever get easier? Seville, Spain. 2018. |
| 4. | Константинова Н. С., Митрофанова О. А. Онтологии как системы хранения знаний // Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы", 2008. |
| 5. | Lanin V., Lyadova L., Shalyaeva I. 11th IEEE International Conference on Application of Information and Communication Technologies (AICT) // Ontology-Driven System for Monitoring Global Processes on Basis of Internet News. 2017. pp. 385-389. |
| 6. | Horrocks I. International Conference on Extending Database Technology // DAML+ OIL: a reason-able web ontology language. Berlin. 2002. |
| 7. | Vrandečić D., York S. European Semantic Web Conference // How to design better ontology metrics. Berlin. pp. 311-325. |
| 8. | Yao H., Mark Orme A., Etzkorn L. Cohesion metrics for ontology design and application. // Journal of Computer science, Vol. 1, No. 2005, Jan 2005. pp. 107-113. |
| 9. | Никоненко А.А. Обзор баз знаний онтологического типа // Штучний інтелект, 2009. |
| 10. | Добров, Б. В., Соловьев, В. Д., Иванов, В. В., Лукашевич, Н. В. Онтологии и тезаурусы. Казань, Москва. 2006. |

x

Приложение А. Листинг

import json

import requests

from owlready2 import \*

from rdflib import \*

import config as c

def isRDF(uri):

try:

onto\_path.append("/onto/")

my\_world = World()

onto = my\_world.get\_ontology(uri)

onto.load()

return False

except:

return True

# if (path.find('rdf') == -1):

# return False

# else:

# return True

def searchNCBO(query, count):

searchUrl = f'http://data.bioontology.org/search?q={query}&apikey={c.NCBO\_API\_KEY}'

r = requests.get(searchUrl)

res = {

'ontologies': []

}

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()['collection']

if (count > len(items)):

count = len(items)

for item in range(count):

res['ontologies'].append(items[item]['links']['ontology'])

return res

def getNCBOMetrics(name):

res = {

'classes': [],

'classCount': -1,

'individuals': [],

'individualsCount': -1,

'properties': [],

'propertiesCount': -1,

'depth': -1

}

# Get class list

requestUrl = f'http://data.bioontology.org/ontologies/{name}/classes?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

for item in items['collection']:

res['classes'].append(item['prefLabel'])

# Get properties list

requestUrl = f'http://data.bioontology.org/ontologies/{name}/properties?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

for item in items:

res['properties'].append(item['label'])

# Get all numbers

requestUrl = f'http://data.bioontology.org/ontologies/{name}/metrics?apikey={c.NCBO\_API\_KEY}'

r = requests.get(requestUrl)

if (r.status\_code != 200):

res['error'] = r.status\_code

return res

items = r.json()

res['classCount'] = items['classes']

res['individualsCount'] = items['individuals']

res['propertiesCount'] = items['properties']

res['depth'] = items['maxDepth']

return json.dumps(res)

def getOWLMetrics(uri):

res = {

'classes': [],

'classCount': -1,

'individuals': [],

'individualsCount': -1,

'properties': [],

'propertiesCount': -1,

'depth': -1

}

onto\_path.append("/onto/")

my\_world = World()

onto = my\_world.get\_ontology(uri)

onto.load()

graph = my\_world.as\_rdflib\_graph()

try:

class\_list = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT distinct ?class

WHERE {

?class a owl:Class

}

order by ?class

"""))

for cl in class\_list:

res['classes'].append(cl[0].name)

except:

res['classes'] = 'Error'

try:

res['classCount'] = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(?s) AS ?totalNumberOfInstances)

WHERE { ?s a owl:Class }

"""))[0][0]

except:

res['classCount'] = 'Error'

try:

individuals\_list = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?entity

WHERE {

?entity rdf:type ?type.

?type rdfs:subClassOf\* owl:Class.

}

"""))

for ind in individuals\_list:

res['individuals'].append(ind[0].name)

except:

res['individuals'] = 'Error'

try:

res['individualsCount'] = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(DISTINCT ?entity) as ?count)

WHERE {

?entity rdf:type ?type.

?type rdfs:subClassOf\* owl:Class.

}

"""))[0][0]

except:

res['individualsCount'] = 'Error'

try:

prop\_list = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?property

WHERE {

{ ?property a owl:ObjectProperty .}

}

"""))

for prop in prop\_list:

if (prop == None):

continue

res['properties'].append(prop[0].\_name)

except:

res['properties'] = 'Error'

try:

res['propertiesCount'] = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(DISTINCT ?property) as ?res)

WHERE {

{ ?property a owl:ObjectProperty .}

}

"""))[0][0]

except:

res['propertiesCount'] = 'Error'

try:

depth = list(graph.query\_owlready(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

select ?class (count(?mid)-1 as ?depth) {

{

select ?root {

?root a owl:Class

filter not exists {

?root rdfs:subClassOf ?superroot

filter ( ?root != ?superroot )

}

}

}

?class rdfs:subClassOf\* ?mid .

?mid rdfs:subClassOf\* ?root .

}

group by ?class

order by ?depth

"""

))

for d in depth:

if (res['depth'] < d[1]):

res['depth'] = d[1]

except:

res['depth'] = 'Error'

return json.dumps(res)

def getRDFMetrics(uri):

res = {

'classes': [],

'classCount': -1,

'individuals': [],

'individualsCount': -1,

'properties': [],

'propertiesCount': -1,

'depth': -1

}

graph = Graph()

graph.parse(uri)

try:

class\_list = graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT distinct ?class

WHERE {

?class a rdf:Class

}

order by ?class

""")

for cl in list(class\_list):

res['classes'].append(cl[0].name)

except:

res['classes'] = 'Error'

try:

res['classCount'] = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(?s) AS ?totalNumberOfInstances)

WHERE { ?s a rdf:Class }

"""))[0][0]

except:

res['classCount'] = 'Error'

try:

individuals\_list = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?entity

{

{ ?property a owl:NamedIndividual .}

}

"""))

for ind in individuals\_list:

res['individuals'].append(ind[0].name)

except:

res['individuals'] = 'Error'

try:

res['individualsCount'] = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(DISTINCT ?entity) as ?count)

{

{ ?property a owl:NamedIndividual .}

}

"""))[0][0]

except:

res['individualsCount'] = 'Error'

try:

prop\_list = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?prop

WHERE {

?s ?prop ?o.

}

"""))

for prop in prop\_list:

res['properties'].append(prop[0].name)

except:

res['properties'] = 'Error'

try:

res['propertiesCount'] = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT (COUNT(DISTINCT ?prop) as ?count)

WHERE {

?s ?prop ?o.

}

"""))[0][0]

except:

res['propertiesCount'] = 'Error'

try:

depth = list(graph.query(

"""

PREFIX owl: <http://www.w3.org/2002/07/owl#>

select ?class (count(?mid)-1 as ?depth) {

{

select ?root {

?root a rdf:Class

filter not exists {

?root rdfs:subClassOf ?superroot

filter ( ?root != ?superroot )

}

}

}

?class rdfs:subClassOf\* ?mid .

?mid rdfs:subClassOf\* ?root .

}

group by ?class

order by ?depth

"""

))

for d in depth:

if (res['depth'] < d[1]):

res['depth'] = d[1]

except:

res['depth'] = 'Error'

return json.dumps(res)

from flask import Flask

from owlready2 import \*

import json

from rdflib import \*

import requests

import help as h

import config as c

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return h.getNCBOMetrics('MEDDRA')

@app.route("/api/onto/search/<query>")

def search(query):

ontologiesList = h.searchNCBO(query, c.RESULTS\_COUNT)

res = {

'keyword': query,

'ontologies': []

}

for ontology in ontologiesList['ontologies']:

# try:

# r = requests.get(f"{ontology}/download?format=rdf&apikey={c.NCBO\_API\_KEY}", allow\_redirects=True)

# file = open('test', 'wb')

# file.write(r.content)

# file.close()

ont = {

'url': ontology,

'metrics': {

'metadata': {},

'metricsCounted': {},

'metricsFromResource': {}

}

}

try:

ont['metrics']['metricsCounted']= metrics(f"{ontology}/download?format=rdf&apikey={c.NCBO\_API\_KEY}"),

except:

ont['metrics']['metricsCounted'] = 'Error: cant download'

try:

ont['metrics']['metricsFromResource'] = h.getNCBOMetrics(

ontology[ontology.rfind('/')+1:])

except:

ont['metrics']['metricsFromResource'] = 'Error'

res['ontologies'].append(ont)

# except:

# pass

return json.dumps(res)

@app.route("/api/onto/metrics/<path:uri>")

def metrics(uri):

# if (uri.find('.owl') == -1 and uri.find('.rdf') == -1):

# return json.dumps({'error': 'Not .rdf / Not .owl', 'uri': uri})

if (h.isRDF(uri)):

return h.getRDFMetrics(uri)

else:

return h.getOWLMetrics(uri)