Санкт-Петербургский государственный университет
Кафедра компьютерного моделирования и многопроцессорных систем

Бурмистров Валерий Дмитриевич

Выпускная квалификационная работа бакалавра

Создание структурированных хранилищ данных на основе свободно распространяемых пакетов

Направление 010400.62
Прикладная математика и информатика

Научный руководитель,
Старший преподаватель
Кулабухова Н. В.

Санкт-Петербург
2016

Оглавление

Введение

В настоящее время происходит переход от традиционного хранения данных к цифровому — информация сегодня хранится, зачастую, не в традиционном виде, а в цифровом виде. В семидесятых годах двадцатого века впервые были предложены концепции проектов, реализующих модель приложений, при которых обработка информации осуществлялась бы не на компьютере пользователя, а на удалённых серверах. Отсутствие в то время глобальной сети Интернет делало почти невозможным реализацию подобного рода концепций.

Сейчас же такая «облачная» модель обработки данных является перспективной и быстроразвивающейся. Облачной моделью хранения данных называется модель онлайн-хранилища данных. В ней данные хранятся в распределённой системе на основе множества серверов, связанных общей сетью. Данная модель хранения имеет множество преимуществ, таких как:

1. Доступ может предоставляться любому устройству, имеющему подключение к сети Интернет.
2. Сохранность данных в случае сбоя, отказоустойчивость.
3. Неограниченная масштабируемость.

Зачастую, процесс обучения тесно связан с изучением дополнительных материалов текстового вида (статьи, книги, учебные пособия и т. д.). В настоящее время большая часть этих материалов хранится в электронном виде. Кроме того, число разного рода документов, книг и прочих полезных источников данных неуклонно растёт. К сожалению, почти все источники, в данный момент, не имеют централизованности и упорядоченности в хранении, а значит, работа с ними неудобна. Возникает проблема хранения большого объёма разнородных информационных ресурсов, являющих собой файлы, содержащие текстовые данные.

На фоне этого актуальна задача разработки программного комплекса, решающего проблему хранения и извлечения необходимой информации.

В процессе решения этой задачи необходимо провести анализ современных технологий хранения данных и разрешить следующие вопросы: выбор метода хранения файлов, проектирование и реализация программного комплекса, соответствующего требованиям.

Таким образом, целью данной научно-исследовательской работы является проектирование и последующая разработка системы, реализующей функции хранения данных, содержащих данные в виде текста.

Постановка задачи

В связи с активной научной и образовательной деятельностью кафедры Компьютерного моделирования и многопроцессорных систем факультета ПМ-ПУ СПбГУ, в распоряжении кафедры скопился огромный объём различных документов, используемых в работе сотрудников. Потому остро стоит вопрос об организованном хранении данной документации. Решение данной проблемы — это создание программного продукта, реализующего систематизированное хранение файлов, содержащих текстовую информацию, так как именно в текстовом виде хранится подавляющая часть всех используемых материалов: книги, статьи, электронные письма.

Предназначение данного программного средства: хранение файлов, содержащих данные в виде текста, с возможностью последующего извлечения и полнотекстового поиска, разбиение их на группы в соответствии с содержимым.

Предполагается, что программный продукт будет внедрён для внутреннего использования, учащимися, преподавателями, научными работниками и сотрудниками, имеющими доступ к ресурсам суперкомпьютерного вычислительного центра СПбГУ.

Помимо самих информационных ресурсов (файлов) продукт должен хранить также и техническую информацию об этих файлах: имя, размер, расширение, дату создания и изменения. Кроме того, необходимо учесть возможность обновления файлов.

Общую задачу по созданию данного продукта можно разделить на три подзадачи:

1. Реализация пользовательских интерфейсов
2. Задачи поиска и работы с базами данных
3. Задачи размещения файлов, а также аутентификации и авторизации в

системе

Ранее был реализован прототип данного программного продукта с незаконченным функционалом, потому доработку системы было решено разделить между тремя разработчиками, поставив перед каждым одну из описанных ранее задач.

В рамках данной научно-исследовательской работы рассматривается задача размещения файлов, аутентификации и авторизации в системе, таким образом, в рамках данной работы ставятся задачи по расширению функционала существующего прототипа.

Готовый функционал программного продукта, в рамках данной научной работы, сводится к следующим требованиям:

Задачи размещения файлов:

1. Создание директории размещения файла;
2. Изменение атрибутов созданной директории;
3. Удаление директории и её содержимого с сервера;
4. Копирование директории вместе с содержимым на сервере;
5. Перемещение директории вместе с содержимым на сервере;
6. Загрузка файла на сервер в выбранную директорию;
7. Скачивание файла с сервера из выбранной директории;
8. Обновление существующих файлов;
9. Удаление файлов;
10. Изменение атрибутов файла (имя файла, флаг файла — публичный/частный);
11. Перемещение файлов на сервере;
12. Копирование файлов на сервере;
13. Публикация файла в общий пул документов, доступных в поиске;
14. Просмотр списка файлов в директории;

Задачи по реализации перечисленных выше функций являются основными для данной работы, так как обеспечивают необходимый для работы приложения минимум, в данной проблемной области.

Задачи аутентификации и авторизации:

1. Аутентификация учётной записи пользователя в системе с учётом пары «учётная запись — пароль»;
2. Авторизация учётной записи пользователя при попытке выполнения определённых действий;
3. Отправка заявки пользователей на регистрацию;
4. Регистрация учётной записи пользователя с учётом кодов приглашения;
5. Изменение параметров профиля пользователя;
6. Отправка уведомлений пользователю о состоянии учётной записи пользователя;

Некоторые из документов не предназначены для распространения, а значит важным моментом является безопасность. Для её обеспечения необходимо реализовать возможность отслеживания использования предоставленного функционала, другими словами должна быть организована система управления группами пользователей.

Кроме того, предполагаемый программный продукт должен быть реализован с использованием свободно распространяемого программного обеспечения.

Глава 1. Лицензирование программного обеспечения

Поскольку в рамках данной работы, для решения поставленных задач, должны использоваться свободно распространяемые программные инструменты, важным вопросом является вопрос о лицензировании программного обеспечения.

Прежде всего, лицензия программного обеспечения — это правовой инструмент, регулирующий распространение и использования программного обеспечения, защищённого авторским правом.

Каждый программный продукт распространяется под определённой лицензией. Тип лицензии определяет ограничения в использовании разработке, накладываемые на продукт.

Ниже будут приведены данные о лицензиях на продукты, используемые в ходе разработки или упоминающиеся в работе.

GNU General Public License (Универсальная общественная лицензия, также GNU GPL) — Лицензия программного обеспечения, созданная в рамках проекта GNU в 1988 г [3].

Универсальная общественная лицензия GNU предоставляет пользователю право на копирование и распространение программы, с гарантией того, что пользователи всех программ, созданных на основе данной, также будут обладать обозначенными правами.

GPL позволяет пользователям компьютерных программ обладать ниже перечисленными правами и свободами:

- Свобода запуска программы с любой целью;
- Свобода изучения работы программы, а также её модификации;
- Свобода распространения копий исходного кода и производного кода;
- Свобода улучшения программы, а также выпуска результатов улучшения в публичный доступ, при условии доступа к исходному коду;

GNU Lesser General Public License (Стандартная общественная лицензия ограниченного применения GNU), ранее — GNU Library General Public License (Стандартная общественная лицензия GNU для библиотек) - данный вид лицензии применяется к специальному ПО, чаще всего к библиотекам, права на которое принадлежат Free Software Foundation (Фонд свободного программного обеспечения) [4].

К большинству программных продуктов GNU, в том числе и к библиотекам, применяется обычная Стандартная общественная лицензия GNU. Данная лицензия применяется к библиотекам, обеспечивая возможность их связи с несвободным ПО.

Apache License - лицензия свободного программного обеспечения организации Apache Software Foundation [6]. Соответственно условиям данной лицензии, каждый пользователь программного продукта получает право на воспроизведение, изменение, публичную демонстрацию и исполнение, а также сублицензирование и распространение программы и её производных. Пользователь может воспроизводить и распространять копии программного продукта, а также его производные при выполнении нижеперечисленных условий:

- Предоставление всем последующим пользователям программного продукта и его производных копии данной лицензии;
- Обеспечения модифицированных файлов уведомлении об изменении (в явном виде);
- Сохранение в распространяемых производных продуктах всех; авторских прав, патентов, торговых марок из исходного продукта;
- В случае, если исходная программа имеет текстовый файл NOTICE, любые распространяемые производные программы также должны включать копию этого файла;
- При распространении ПО следует поместить файл LICENSE в корневую директорию;

Файл LICENSE содержит копию лицензии Apache. Файл NOTICE

содержит все библиотеки под лицензией Apache, а также имена их разработчиков [6].

Лицензия MIT (MIT License) — лицензия за авторством Массачуссетского технологического института, разработанная в 1988 г. Лицензия MIT предоставляет пользователю следующие права: использование, копирование, модифицирование, объединение, объединение, публикацию, распространение, сублицензирование, продажу производного ПО. Для распространение конечного продукта под данной лицензией необходимо лишь приложить к ней текст данной лицензии. Кроме того, лицензия является GPL-совместимой, т.е. разрешает программистам комбинировать и распространять GPL продукты с софтом, использующий MIT License.

Eclipse Public License (EPL) - лицензия, разработанная организацией Eclipse foundation в 2004 г. Используется для любых работ, предоставляет пользователю возможность воспроизведения, создание производных работ, публичную демонстрацию и исполнение, распространение и сублицензирование производного продукта.

EPL 1.0 является несовместимой с GPL таким образом, работы, созданные на основе продуктов как с EPL, так и с GPL не могут быть распространены законно. Согласно лицензии GPL, если в продукте используется хотя бы один компонент с лицензией GPL, то цельный продукт следует лицензировать под GPL. Кроме того, GPL требует от пользователя программного кода, не наложения на лицензируемый продукт дополнительных ограничений. В то же время EPL требует от каждого распространителя возможность соблюдения прочих используемых лицензий и патентов и предоставляет право лицензировать продукт под другой лицензией. Такая же проблема у лицензии GPL обстоит и с другими лицензиями.

Проприетарное программное обеспечение (англ. Proprietary software) - является частной собственностью авторов или правообладателей, не удовлетворяющее критериям свободного программного обеспечения (наличие открытого кода недостаточно). Правообладатель такого ПО является

носителем монополии на распространение, копирование и изменение исходного продукта, что означает, что для пользования проприетарного программного обеспечения, чаще всего, взимается плата.

Таким образом, исходя из описанной выше информации о различных лицензиях, учитывая условия распространения используемых инструментов разработки, можно сделать вывод, что лицензия Apache 2.0 наиболее подходящая для лицензирования готового продукта.

Глава 2. Архитектура разрабатываемого программного комплекса и инструменты разработки

2.1 Архитектура программного комплекса

Архитектура данного программного комплекса представляет из себя многоуровневую архитектуру.

Многоуровневая архитектура (Multi-tier architecture) - клиент-серверная архитектура, в которой процессы управления, представления и обработки логически разделены.

Наиболее распространённым вариантом такого типа архитектуры является трёхуровневая архитектура, предполагающая наличие:

- Клиентского слоя, то есть компонента продукта, представленного пользователю;
- Связующего слоя, в данном слое располагаются серверы приложений, к которым подключено клиентское приложение, обеспечивающее контроль потоков данных в программном решении;
- Слой данных, то есть сервер баз данных, а также файловые системы, обеспечивающие хранение данных, используемые сервером приложений;

Зачастую, с целью снижения уровня нагрузки, разделяют на несколько частей логику и серверное приложение, реализующее, выполняя в различных узлах, таким образом происходит увеличение числа архитектурных слоёв. На рисунке 1 представлена конечная архитектура программного комплекса.

Как видно, серверная часть продукта разделена на шесть модулей, реализующих однотипные функции.

- Warehouse, задачей которого является хранение и предоставление

файлов;

- Auth, ответственный за аутентификацию и авторизацию пользователей в системе;

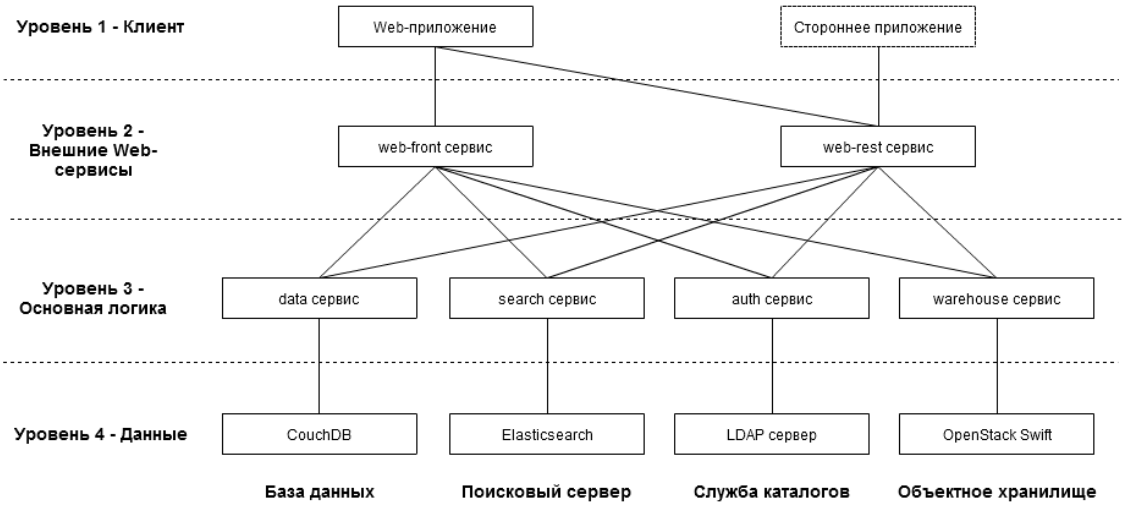


Рис. 1. Архитектура программного комплекса

- Data, хранящий дополнительную информацию о пользователях, а также совершаемых в системе действиях;
- Search, формирующий и осуществляющий поисковые запросы;
- Web-front, обеспечивающий взаимосвязь между вышеназванными приложениями и пользователем, посредством HTML-страниц, представляемых в виде пользовательского интерфейса в браузере;
- Web-rest, выполняющий аналогичные функции, но передающий пользователю данные в виде JSON-объектов, то есть позволяет выполнять асинхронные обращения к серверу, создавая возможность для реализации динамичного интерфейса;

Такое разделение на отдельные модули обработки данных открывает широкие возможности по масштабированию системы. Каждый модуль может быть легко реализован иначе, с использованием других инструментов.

В данной реализации программного комплекса используются следующие системы, включенные в слой данных:

- База данных CoachDB, хранящая информацию о пользователях и их действиях;

- Поисковый сервер Elasticsearch, управляющий индексами документов и поиском по ним;
- Компонент Openstack Keystone, отвечающий за предоставление такой функциональности, как токены, политики и каталоги, для управления правами пользователей на обращение к сервисам и действиям;
- Объектное хранилище Openstack Swift, используемое для хранения файлов;

2.2 Поисковый сервер ElasticSearch

Elasticsearch — это поисковый сервер, базирующийся на библиотеке Lucene, предназначенной для поиска и индексации документов. Выбор делается в пользу данного продукта в виду предоставления им ряда важных возможностей:

- Масштабируемость и отказоустойчивость;
- Мультиарендность (то есть возможность организовать множество; поисковых систем в рамках лишь одного проекта Elasticsearch);
- Отсутствие схемы, в Eleasticsearch может быть загружен простой; JSON-объект, с индексацией справится сам сервер, что облегчает задачу структурирования данных;
- RESTful API — Elasticsearch управляется почти полностью по протоколу HTTP, посредством JSON-запросов;

Кроме того, за этот проект говорит его простота в освоении и лидирующие позиции в своей области технологий, конкуренты Elasticsearch - Solr, Sphinx развивают свои продукты, вдохновляясь инновациями именно Elasticsearch.

Именно по перечисленным причинам был выбран данный поисковый сервер, для того, чтобы решить задачи поисковых запросов и индексации документов.

2.3 Система управления базами данных Apache CoachDB

CouchDB - это документо-ориентированная СУБД, использующая NoSQL подход. Сильными сторонами данного продукта являются REST интерфейс, а также то, что данные внутри этой системы хранятся в виде JSON документов. Что более важно — использование принципа Map/Reduce — принципа параллельной обработки данных больших выборок, для реализации запросов этого принципа в CoachDB используется JavaScript.

2.4 Система управления сборкой проекта Apache Maven

Maven — это инструмент управления сборкой Java-проекта. Основными преимуществами этого инструмента являются декларативный способ описания проекта, кроссплатформенность (сборка проекта может быть осуществлена в любой операционной системе), удобное управление зависимостями проекта, интеграция с различными средами разработки, возможность сборки прямиком из командной строки [7].

Для сборки проекта Maven использует главный файл сборки pom.xml, описывающий проект. В этом файле, написаном на языке POM (Project Object Model), содержит последовательность тегов, в которых описываются части проекта. Тэг <project> содержит основную информацию о проекте, такую как artifactId — название проекта, groupId — название организации, version — версию проекта. Следующий тэг <dependencies> - хранит список всех используемых при построении проекта библиотек. Каждая из них, в свою очередь, также характеризуется тройкой тегов: groupId, artifactId,version.

Кроме того, может быть задан необязательный тэг <build>, описывающий информацию, необходимую непосредственно процесса сборки: местоположение исходных файлов, используемые плагины.

В процессе сборки с помощью Maven, выполняются следующие

основные фазы сборки проекта:

1. Compile;
2. Test (тестирование с помощью Junit тестов);
3. Package (генерация .jar, .war, .ear файла);
4. Integration-test;
5. Install (копирование ранее сгенерированных файлов в локальный репозиторий);
6. Deploy (публикация в удалённый репозиторий);

Кроме того, дополнительно, могут быть указаны при запуске сборки две фазы: clean и site. Первая нужна для очистки проекта, вторая — для автоматического создания документации к проекту.

2.5 Компоненты комплекса OpenStack

OpenStack — это open source проект по разработке платформы, позволяющей строить частные и публичные «облака». Технология включает в себя серию взаимосвязанных проектов, обеспечивающих разработку многочисленных составляющих инфрастуктурного решения для «облака» [1].

Как было написано ранее, в рамках данной работы используются два проекта из платформы OpenStack: Swift и Keystone. Swift (OpenStack Object Store) — распределённое объектное хранилище, главными особенностями которого являются высокий уровень надёжности и отказоустойчивости, а также лёгкая масштабируемость системы.

Объектное хранилище OpenStack Swift организованно на базе четырёх основных компонент:

- Proxy Server (сервер, объединяющий остальные компоненты вместе) Отвечает за приём пользовательских запросов HTTP, производит поиск местоположение запрошенного объекта с помощью кольца;
- Object Server (непосредственно хранящий данные);
- Container Server (отдаёт список объектов в контейнере);

- Account Server (листинги контейнеров для конкретного аккаунта) Ослеживает имена контейнеров, принадлежащих определённому аккаунту.

При этом, путь доступа к каждому из объектов хранения в Swift состоит из трёх компонент: аккаунт, контейнер, объект и выглядит следующим образом: AccountName/ContainerName/ObjectName.

Как правило, хранилище данных, созданное с помощью Swift имеет вид масштабного кластера, одна из машин которого выполняет роль прокси-сервера; некоторое количество машин — контейнер-серверы и аккаунтинг-серверы; остальные же — объектные серверы и их количество может исчисляться сотнями и более машин, в зависимости от нужд потребителя [8].

Отдельно стоит упомянуть структуру данных, используемую Swift для поиска положения данных в системе — «кольцо» (ring). По сути, кольцо является базой данных, описывающей местоположение объектов. При любом действии, связанном с записью или удалением данных в хранилище, а также выходе из строя узлов хранилища, кольцо изменяется. Кроме того, предусмотрены разные кольца для контейнеров, аккаунтов и объектов.

Разумеется, важнейшим элементом абсолютно любого хранилища являются объектные серверы, выполняющие функции размещения и отдачи файлов. Объектный сервер — это хранилище BLOB-объектов (большой двоичный объект). Таким образом, каждый объект хранилища оказывается на жёстком диске объектного сервера и хранится на нём в двоичных файлах, сопровождаемых метаданными, которые, в свою очередь, хранятся в расширенных атрибутах файлов (xattr). Swift обеспечивает надёжное хранение данных, благодаря их репликации сразу в несколько серверов. Таким образом, в случае утери данных с одного из серверов при его отказе, система восстановит данные с другого сервера и снова создать копию. По умолчанию, Swift создаёт три реплики файлов.

Также, как было написано ранее, серьёзным плюсом в пользу выбора данного объектного хранилища является лёгкая масштабируемость системы.

Для расширения нужно лишь подключить новый сервер к кластеру, с дальнейшей работой по синхронизацией с хранилищем справится сам Swift.

Для аутентификации в системе используется компонент Keystone, обеспечивающий услуги идентификации. В рамках данной службы, пользователю предоставлены некоторые полномочия над учётными данными, кроме того каждый пользователь обладает некой ролью в системе, содержащаяся в метаданных: User, Admin, Searcher. Данные роли могут дать конкретному пользователю большие, относительно других пользователей, возможности.

2.6 Модульная структура приложения

Настоящий проект разрабатывается на основе вложенной структуры с родительскими и дочерними файлами проектов. Каждый из проектов, представленных в иерархическом древе имеет файл pom.xml, используемый сборщиком проекта Apache Maven.

Модуль Domain-model содержит Java-классы, описывающие сущности, используемые для решения конкретных задач. Этот модуль включен в модели серверов обработки данных, аутентификации и авторизации, размещения файлов, уже на их основе строятся готовые к исполнению целевые приложения, а кроме того, Java-библиотеки, открывающие возможность к выгодной эксплуатации созданных сервисов. Также, кроме внешних интерфейсов, они включают в себя описание дополнительных сущностей, нужных для решения задач в отведенной ветке специфической области. Преимущество такого подхода заключается в том, что разработчик имеет возможность реализовывать один и тот же функционал, пользуясь различными программными средствами, используя их как замену друг другу. К примеру, модуль search может быть реализован для Solr, вместо Elasticsearch.

Составляющие Core не зависят от доменной модели и реализуют

дополнительный функционал для web-сервисов. Используют такой же способ построения модель-приложение-клиент. Предоставляют функции отслеживания состояния приложения.

Ветка Web ответственна за пользовательские сервисы. Соответственно, Web-core включает в себя функционал внутренней сети, в то время, как Web-front и Web-rest предоставляют пользователю интерфейсы для взаимодействия с Web-core, для получения данных в двух форматах: HTML и JSON-документов.

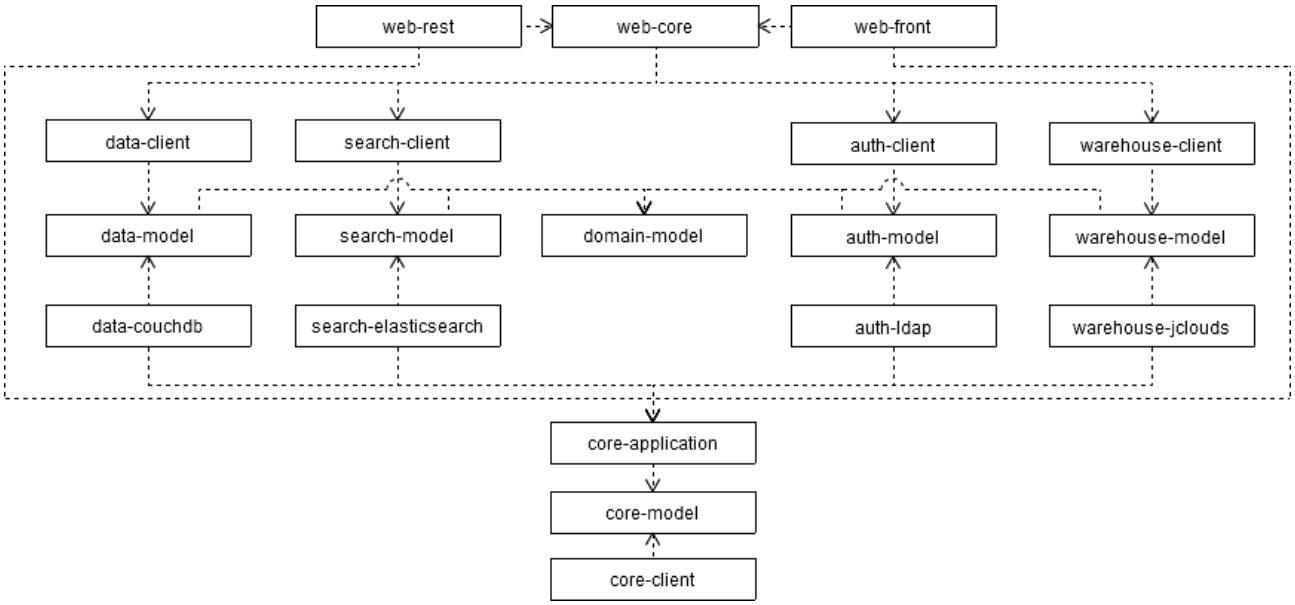


Рис. 2. Модульная структура приложения

Глава 3. Программная реализация поставленной задачи

Для реализации модулей программного продукта, необходимо реализовать элементы MVC-модели приложения на языке программирования Java. Для этого используется фреймворк Spring, предоставляющий архитектурный каркас для лёгкой реализации следующих компонентов системы:

- Классы модели представления, получаемые при получении запроса от пользователя и формировании ответа на него, с целью объединения экземпляров классов доменной модели, с последующей передачей пользователю;
- Контроллеры, обрабатывающие объекты, представляющие пользовательский запрос, передавая их в соответствующие сервисы;
- Сервисы, реализующие типовые функции, используемые контроллером или другими сервисами;

Кроме того, используется библиотека-провайдер Apache jclouds, обеспечивающая доступ к функциям, предоставляемым облачными средствами, в том числе Swift API [5], позволяющий управлять аккаунтами, контейнерами и объектами в системе объектного хранения.

3.1 Задачи размещения файлов

Таким образом, для решения поставленных задач, связанных с хранением файлов, нужно создать классы, описывающие сущности необходимые для выполнения заданных функций, таких как местоположение файла, его атрибуты, а также, на основе этих классов, сервисы и контроллеры, которые в свою очередь реализуют непосредственно логику осуществления функции и их выполнение посредством обращения к API.

На рисунке 3 продемонстрирована схема взаимодействия данных элементов системы.

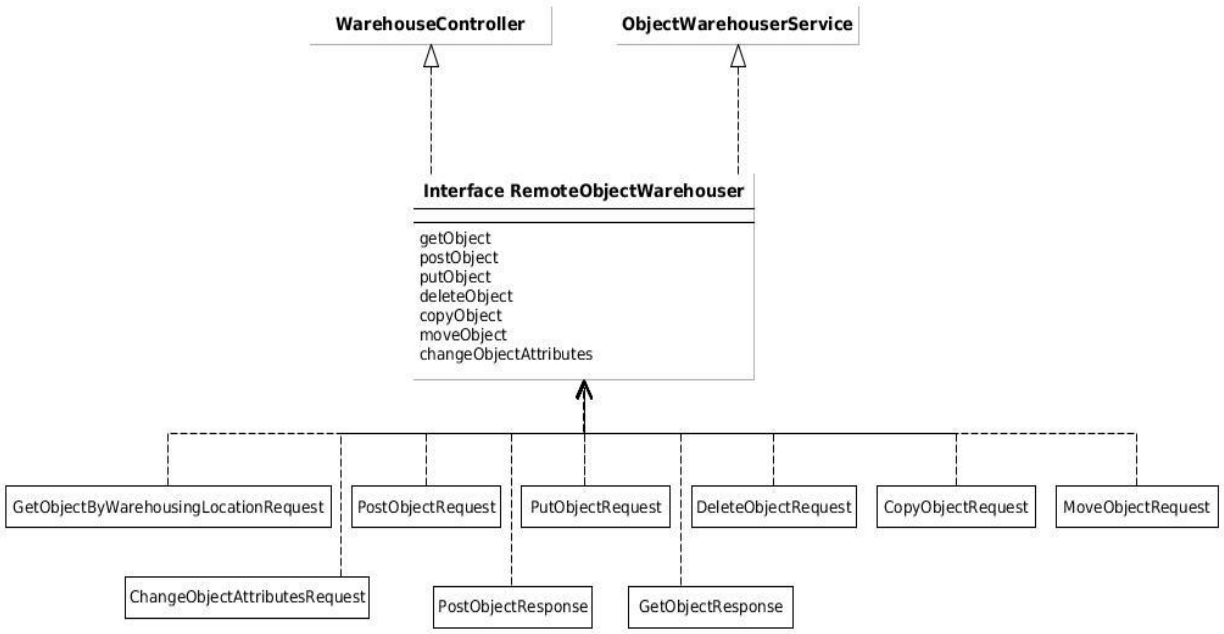


Рис. 3. Взаимодействие классов warehouse-model

Как видно из диаграммы, контроллер WarehouseController и сервис ObjectWarehouserService реализуют методы интерфейса RemoteObjectWarehouser, используя классы модели сервера хранилища данных (warehouse-model), а также классы доменной модели (domain-model). Причём контроллер генерирует соответствующие запросу JSON-запросы, а сервис реализует логику исполнения этих запросов.

Методами интерфейса RemoteObjectWarehouser описываются запросы для взаимодействия с объектами хранения и ответы на эти запросы.

На рисунках 4 и 5, в качестве примера, представлены диаграммы классов для класса GetObjectByWarehousingLocationRequest, описывающий запрос обращения к объекту хранения и ChangeObjectAttributesRequest, описывающий запрос на изменение атрибутов объекта (имя, публичность доступа). С остальными диаграммами можно ознакомиться в приложении 1.

Для выборки хранящегося объекта необходимо установить его местоположение, кроме того, это объект может быть предназначен для публичного доступа, либо для личного, потому необходимо также установить

является ли этот объект публично доступным. Эти два требования ясно видны из рисунка 4.

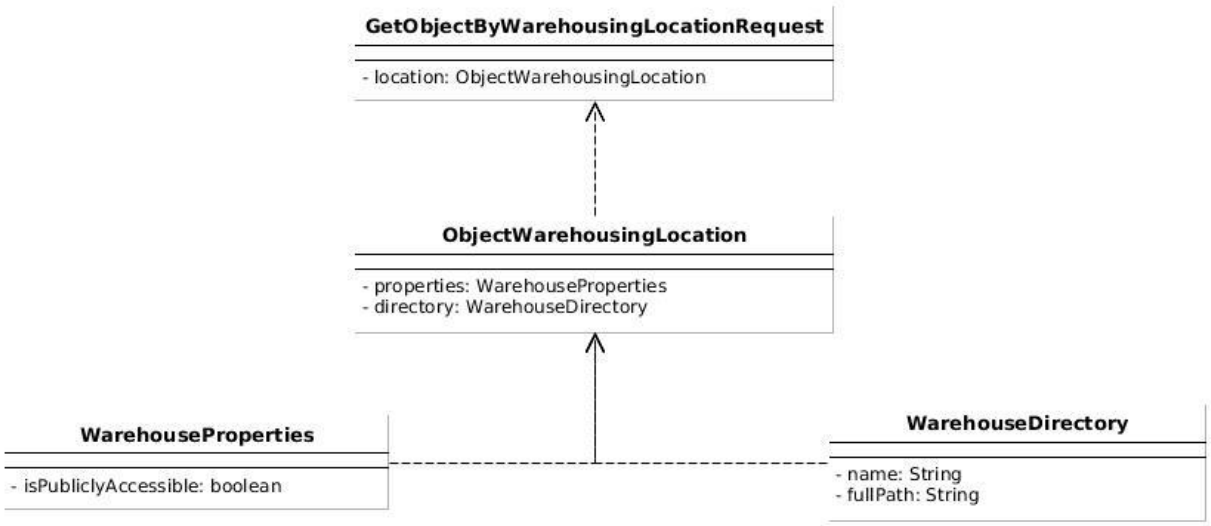


Рис. 4. Диаграмма для класса GetObjectByWarehousingLocationRequest

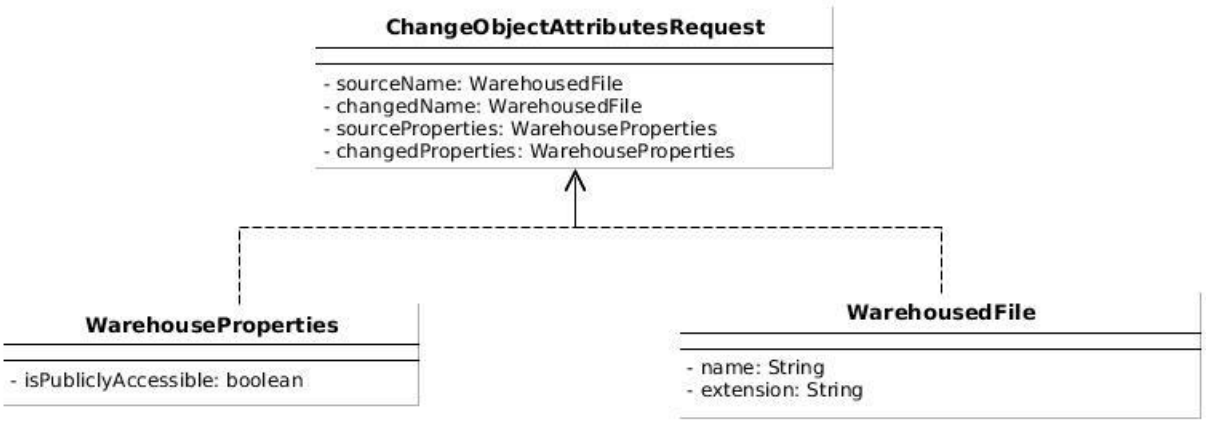


Рис. 5. Диаграмма для класса ChangeObjectAttributesRequest

Ниже приведены краткие описания классов, отвечающих за запросы к объектам хранения:

- GetObjectByWarehousingLocationRequest — запрашивает объект хранения, использует для этого, как было написано ранее, местоположение объекта и данные о его публичности;
- PutObjectRequest — создаёт объект вместе с его метаданными в выбранном контейнере;

- PostObjectRequest — создаёт или обновляет метаданные объекта хранения;
- DeleteObjectRequest — удаляет выбранный объект;
- CopyObjectRequest — создаёт копию объекта в указанном местоположении;
- MoveObjectRequest — перемещает объект в указанное местоположение;
- ChangeObjectAttributesRequest — изменяет атрибуты выбранного объекта (имя, публичность доступа);
- PostObjectResponse — в ответ на произведённое действие возвращает результат в виде выбранного объекта и его обновлённых метаданных;
- GetObjectResponse — по аналогии с предыдущим, возвращает выбранный объект.

Механизм работы большинства из этих функций интуитивно понятен и почти продиктован названием. Swift API предоставляет почти аналогичные по названиям функции, которые реализуют «серверную» составляющую этих запросов, так, например, операция DELETE, с указанием местоположения и названия объекта, удаляет этот объект из контейнера.

Но в Swift не предусмотрены операции перемещения и переименовывания объекта, в привычном понимании. Такого функционала Swift API не предоставляет. Проблема реализации данных функций решается элементарно. Для того, чтобы переместить файл необходимо совместить операции копирования объекта и его удаления, таким образом операция перемещения объекта легко исполняется с помощью последовательности тривиальных функций: копирование существующего объекта, размещение его в новом, указанном местоположении, удаление исходного объекта.

Аналогичным образом решается проблема переименовывания объекта. Для того, чтобы назначить объекту новое имя необходимо сделать копию содержимого объекта, создать новый объект в этом же контейнере, но с новым именем, удалить исходный объект.

Следующая возникающая проблема связана со способом размещения

объектов в Swift. Дело в том, что в отличии от привычных файловых систем, Swift не предусматривает создание директорий - контейнеры не могут быть вложены друг в друга, поэтому директорий, в привычном понимании, в Swift нет. Объектное хранилище Swift не может быть интерпретировано в смысле иерархической структуры, вместо этого оно может быть представлено псевдо-бесконечной картой «ключ-значение», где ключи – это строки, упорядоченные в алфавитном порядке, а значение – есть бинарные объекты.

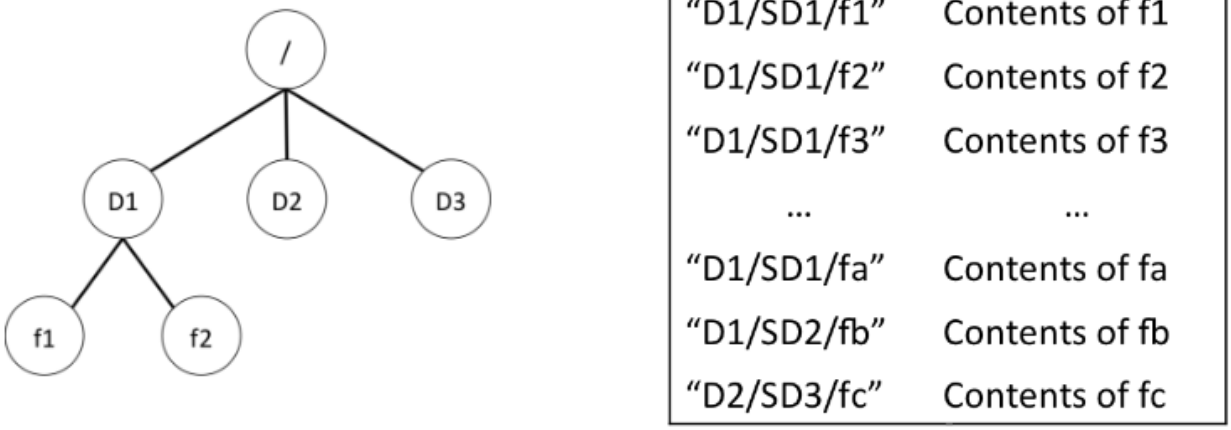


Рис. 6. Представление хранимых объектов в ОС и в Swift

Взамен реального создания директорий, Swift предлагает механизм имитации директорий, то есть позволяет создавать псевдо-директории.

Как было указано ранее, доступ к объекту хранения в Swift осуществляется с помощью трёх компонент: аккаунта, контейнера и объекта. Запрос к хранимому объекту выглядит следующим образом: https://domain.name/{APIversion}/{accountname}/{containername}/{objectname}

Для создания псевдо-директории необходимо добавить символ «/» в имя объекта, к примеру, файл text.txt можно поместить в каталог texts путём добавления соответствующей строки в имя объекта: texts/text.txt. Таким образом, доступ к файлу text.txt будет осуществляться с помощью запроса https://domain.name/{API version}/{accountname}/{containername}/texts/text.txt Фактическое местоположение объекта при этом — контейнер. Можно создать

множество псевдо-директорий и даже «вкладывать» друг в друга, но фактическое место хранения при этом не изменится — все объекты располагаются внутри контейнера.

Для того, чтобы обратиться к объектам, располагающимся «внутри» какой-либо директории при формировании запроса используется вспомогательные параметры «delimeter» и «prefix». Для примера, предположим, что в контейнере с названием container находятся объекты folder1/text1.txt и folder2/text2.txt, тогда для обращения к объекту text1 путь к объекту в запросе должен быть сформирован следующим образом:

{API version}/{accountname}/container?prefix=folder1/&delimeter=/

Соответственно, для получения данных о всех хранящихся в контейнере файлах следует формировать GET-запрос без вспомогательных параметров.

Псевдо-директории при этом не являются реальными объектами, Swift ссылается на реальный объект, название которого не заканчивается символом «/». Псевдо-директории не имеют типа содержимого, но каждая псевдо-директория имеет свою SUBDIR запись в JSON-ответе.

Таким образом, проблемы создания директории и перемещения файлов в директориях, а также изменения атрибутов директории, решаются изменением имени группы объектов. Решение задачи переименовывания было описано ранее.

3.2 Задачи аутентификации и авторизации

Прежде чем приступать к описанию модуля аутентификации и авторизации необходимо пояснить разницу между этими определениями.

Аутентификация — это процесс удостоверения личности пользователя.

Авторизация — процесс проверки наличия у пользователя прав на выполнение определённых действий.

То есть процедура аутентификации проводится при входе пользователя в систему, для этого используется, например, пара «логин — пароль». С

помощью логина пользователь совершает заявление о том, кем он является, а с помощью пароля доказывает истинность этого заявления. Процедура авторизации пользователя же производится после запроса на определённое действие, например, при запросе на создание директории будет проведена авторизация и, если пользователь не обладает ролью, которой разрешается выполнение данного действия, пользователь получает отказ.

Аутентификация в Keystone проводится посредством проверки мандатов — данных принадлежащих только пользователю, в данном случае это пара «логин — пароль».

После проверки идентичности, Keystone предоставляется пользователю токен, подтверждающий идентичность. В токене хранится список ролей, доступных для пользователя, таким образом, при попытке выполнения некоторого действия в системе, Keystone сопоставляет список доступных пользователю ролей и список ролей, допускаемых для выполнения операции и по результатам сравнения запрещает или разрешает доступ.

В таблице 1 приведено соответствие между ролями пользователя и доступными ролями.

Роль в системе	Доступный функционал
ANONYMOUS	1,3,4 из блока аутентификации
USER	2, 5 из блока аутентификации
SEARCHER	Выполнение полнотекстового поиска по документам; Просмотр тематического каталога, сформированного по критерию принадлежности документов к определённой группе
UPLOADER	1-12,14 из блока хранения
TRUSTED_UPLOADER	13 из блока хранения
USER_MANAGER	Управление статусом приглашений

	на регистрацию; Управление статусом заявок; Управление учётными записями; Управление персональными профилями.
UPLOAD_MANAGER	Управление загруженными файлами
AUTHORITY_MANAGER	Управление учётными записями (с ограничениями)
ADMIN	Управление учётными записями

Таблица 1 Соответствие доступного функционала ролям в системе

Токены имеют срок действия, поэтому после успешной аутентификации пользователя в системе производится также процедура проверки токена. Алгоритм данной процедуры представлен на рисунке 7.

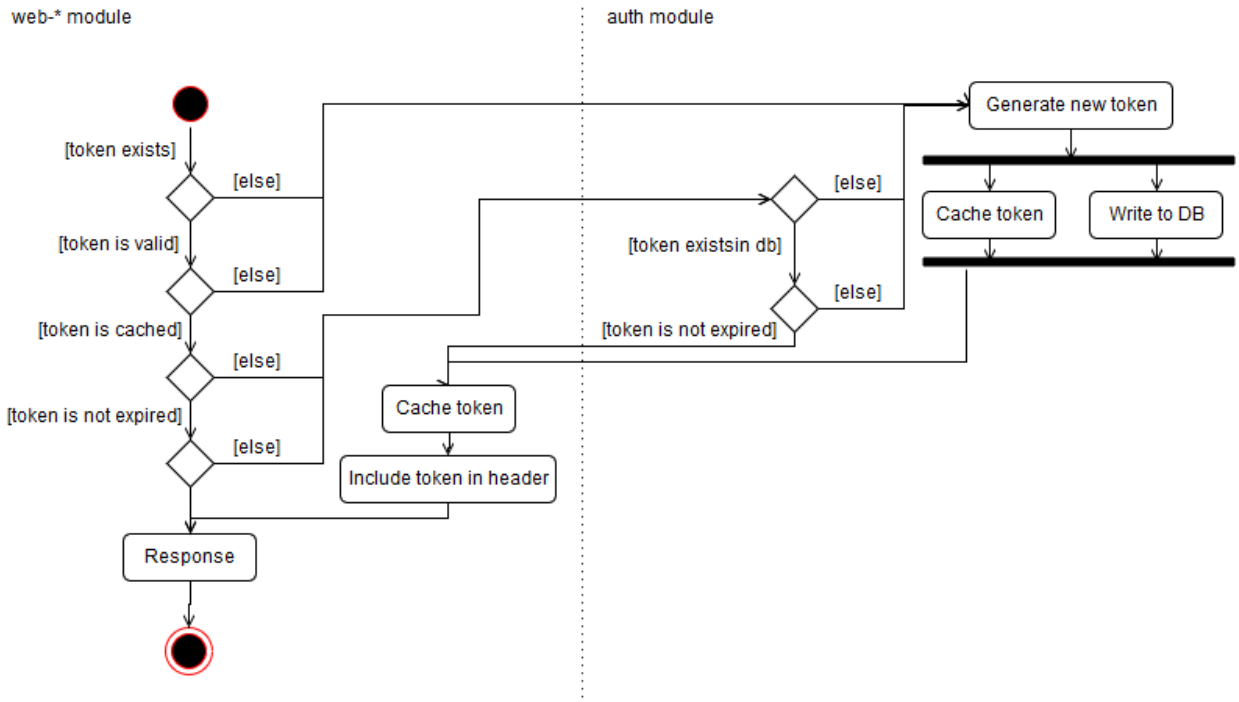


Рис. 7. Схема проверки токена

То есть для реализации модуля аутентификации и авторизации необходимо реализовать классы модели представления, контроллеры и сервисы,

описывающие данный алгоритм. В частности, процедуры генерации токена, записи токена в базу данных и его кэширование.

Тестирование программного комплекса

Для тестирования разрабатываемых модулей использовалось альфа-тестирование. Был произведён запуск программного продукта на исходных данных и выполнена проверка результатов на правильность.

Тест 1

Тестируемые функции: просмотр списка загруженных файлов в выбранной директории.

Тестируемый элемент: отображение содержимого директории.

Ввода: на навигационной панели приложения выбрать «Загруженные» файлы.

Вывод: выбранный пункт меню будет подсвечен, на новой загруженной странице отображена структура директории со всеми вложенными элементами

Тест 2

Тестируемые функции: загрузка файла на сервер в выбранную директорию.

Тестируемый элемент: сохранение файла на сервере вместе с выбранными атрибутами.

Ввод: на панели управления директории нажать кнопку «Загрузить сюда файл»; на открывшейся форме загрузки нажать кнопку «Обзор»; с помощью открывшегося диалогового окна браузера выбрать файл test.txt; отметить поле формы «Доступный публично» флагом; нажать кнопку «Загрузить».

Вывод: происходит загрузка файла на сервер, после чего страница перезагружается и новый файл отображается в списке элементов каталога.

Тест 3

Тестируемые функции: скачивание выбранных файлов с сервера;

Тестируемый элемент: сохранение выбранного файла на компьютер пользователя.

Ввод: выбрать из списка документ, нажать на ссылку с именем файла, в открывшимся диалоговом окне браузера выбрать пункт
«Сохранить файл», нажать кнопку «ОК».

Вывод: В папке «Загрузки» на компьютере пользователя проявится файл, идентичный файлу на сервере.

Тест 4

Тестируемые функции: аутентификация учетной записи пользователя в системе с использованием пары имя учетной записи и пароль.

Тестируемый элемент: ввод и отправка данных аутентификации учетной записи с использованием формы входа в систему.

Ввод: в поля соответствующие поля формы аутентификации ввести Имя пользователя и Пароль, нажать кнопку «Вход».

Вывод: произойдет переадресация на главную страницу, вместо пунктов меню «Вход в систему» и «Заявка на доступ»
появится меню пользователя, обозначенное именем учетной записи.

Результаты проведённого тестирования совпали с предполагаемыми результатами, отмеченными в пункте вывода каждого из тестов, подтвердив корректность работы системы.

Выводы

В ходе работы был реализован программный модуль, осуществляющий задачи хранения файлов в распределённой системе. Также частично реализован модуль отвечающий за аутентификацию и авторизацию в системе.

Следующие функции, заявленные в требованиях к системе являются реализованными:

1. Создание директории размещения файла;
2. Изменение атрибутов созданной директории;
3. Удаление директории и её содержимого с сервера;
4. Копирование директории вместе с содержимым на сервере;
5. Перемещение директории вместе с содержимым на сервере;
6. Загрузка файла на сервер в выбранную директорию;
7. Скачивание файла с сервера из выбранной директории;
8. Обновление существующих файлов;
9. Удаление файлов;
10. Изменение атрибутов файла (имя файла, флаг файла — публичный/частный);
11. Перемещение файлов на сервере;
12. Копирование файлов на сервере;
13. Публикация файла в общий пул документов, доступных в поиске;
14. Просмотр списка файлов в директории.

Реализация данного модуля предоставляет минимально необходимый для работы приложения, в данной проблемной области, функционал, таким образом основные задачи настоящей работы выполнены.

В виду временных ограничений не удалось полностью реализовать модуль аутентификации и авторизации, реализованной на данный момент является лишь функция аутентификации в системе, с использованием пары

«логин – пароль». Для остальных функций данного модуля были разработаны пути их осуществления, то есть спроектирован сам модуль, в дальнейшем планируется его реализация.

Заключение

Целью настоящей выпускной квалификационной работы являлось проектирование и последующая разработка модулей программного комплекса, предназначенного для хранения и анализа текстовых документов, с целью расширения имеющегося функционала разрабатываемого прототипа. Выполнение данной работы предполагало реализацию модулей хранения данных и аутентификации/авторизации в системе.

В ходе работы был произведён анализ предметной области, спроектирован и реализован модуль, ответственный за функции размещения файлов на сервере, а также частично реализован модуль аутентификации и авторизации системы, с использованием языка программирования Java.

Разрабатываемый программный комплекс создаётся с использованием свободно распространяемых средств разработки и библиотек. Используемые средства разработки делают и серверную и клиентскую часть платформонезависимыми.

В дальнейшем планируется продолжение разработки данного программного продукта, окончательная реализация требуемых модулей, а также развёртывание готовой системы на базе РЦ «ВЦ СПбГУ». Кроме доработки разрабатываемых модулей, в дальнейшем, программный комплекс может быть снабжён рядом дополнительных функций, таких как просмотр хранимых в системе документов прямо в окне браузера.

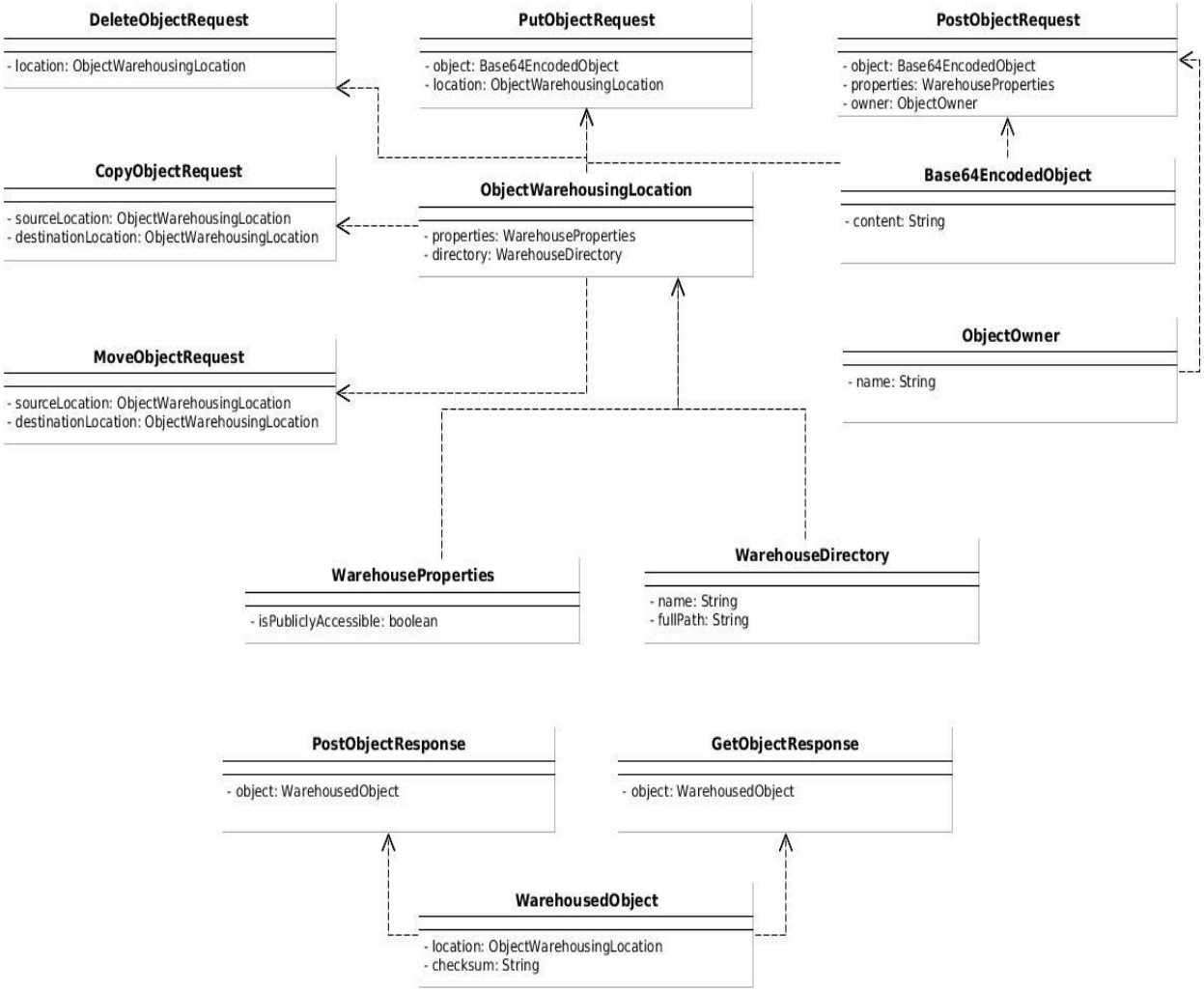
Готовый программный продукт помимо основного способа применения может быть использован и в других областях деятельности, например, в таких востребованных и перспективных областях, как machine learning и data mining.

Список используемой литературы

1. Кападиа Амар, Варма Средхар, Раджана Крис Реализация облачного хранилища с Openstack Swift. Packt Publishing, 2014. 140 с.
2. Ротон Дж. Знакомство с OpenStack : Storage-компоненты Swift и Cinder – https://www.ibm.com/developerworks/ru/library/cl-openstack-swift-cinder/
3. Free Software Foundation, Inc. Стандартная общественная лицензия GNU 3.0 - Проект GNU - Фонд свободного программного обеспечения – http://www.gnu.org/copyleft/gpl.html
4. Stallman M.R. Various Licenses and Comments about Them – http://www.gnu.org/licenses/license-list.html
5. The Apache Software Foundation. Apache jclouds® :: User Guides – https://jclouds.apache.org/guides/
6. The Apache Software Foundation. Apache License v2.0 and GPL Compatibility - http://www.apache.org/licenses/GPL-compatibility.html
7. The Apache Software Foundation. Maven – Introduction to the Build Lifecycle – https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
8. The OpenStack Foundation. OpenStack Docs: Current – http://docs.openstack.org/index.html

Приложение

Приложение 1. Диаграммы классов модели хранилища



Приложение 2. Листинг — Сервис ObjectWarehouserService

```
1. package ru.spbu.cc.wisefox.warehouse.jclouds.service.implementation;
2.
3. import java.io.IOException;
4. import java.io.InputStream;
5. import java.util.UUID;
6.
7. import com.google.common.io.ByteStreams;
8. import com.google.common.io.BaseEncoding;
9.
10. import org.springframework.beans.factory.annotation.Autowired;
11. import org.springframework.beans.factory.annotation.Value;
12. import org.springframework.stereotype.Service;
13.
14. import org.jclouds.blobstore.domain.Blob;
15.
16. import ru.spbu.cc.wisefox.warehouse.model.entity.request.GetObjectByWarehousingLocationRequest;
17. import ru.spbu.cc.wisefox.warehouse.model.entity.request.PostObjectRequest;
18. import ru.spbu.cc.wisefox.warehouse.model.entity.request.PutObjectRequest;
19. import ru.spbu.cc.wisefox.warehouse.model.entity.request.CopyObjectRequest;
20. import ru.spbu.cc.wisefox.warehouse.model.entity.request.MoveObjectRequest;
21. import ru.spbu.cc.wisefox.warehouse.model.entity.request.DeleteObjectRequest;
22. import ru.spbu.cc.wisefox.warehouse.model.entity.response.GetObjectResponse;
23. import ru.spbu.cc.wisefox.warehouse.model.entity.response.PostObjectResponse;
24. import ru.spbu.cc.wisefox.warehouse.model.behaviour.RemoteObjectWarehouser;
25. import ru.spbu.cc.wisefox.warehouse.model.entity.ObjectWarehousingLocation;
26. import ru.spbu.cc.wisefox.warehouse.model.entity.Base64EncodedObject;
27. import ru.spbu.cc.wisefox.warehouse.model.entity.WarehousedObject;
28. import ru.spbu.cc.wisefox.domain.model.entity.WarehouseProperties;
29.
30. import ru.spbu.cc.wisefox.warehouse.jclouds.service.behaviour.BlobStoreObjectWarehouserService;
31.
32. @Service
33. public class ObjectWarehouserService implements RemoteObjectWarehouser {
34.
35.     @Value("${blobStore.container.public}")
36.     String publicContainerName;
37.
38.     @Value("${blobStore.container.private}")
39.     String privateContainerName;
```

```
40.
41.  @Autowired
42.  BlobStoreObjectWarehouserService blobStoreObjectWarehouserService;
43.
44.  private static final BaseEncoding decoder = BaseEncoding.base64();
45.
46.  public GetObjectResponse getObject(GetObjectByWarehousingLocationRequest request) {
47.     ObjectWarehousingLocation location = request.getLocation();
48.
49.     Blob blob =
50.        this
51.           .blobStoreObjectWarehouserService
52.              .getObject( location.getFullPath() + location.getName(),
53.                       location.getProperties().isPubliclyAccessible() ?
54.                          publicContainerName :
55.                          privateContainerName );
56.
57.     if (blob == null) {
58.        throw ObjectNotFoundException;
59.     }
60.
61.     byte[] objectData = {};
62.     try {
63.        InputStream is = blob.getPayload().openStream();
64.        objectData = ByteStreams.toByteArray(is);
65.     } catch (IOException e) {
66.        throw InternalServerErrorException
67.     }
68.
69.     return new GetObjectResponse( new Base64EncodedObject( location.getName(),
70.                                          this.decoder.encode(objectData) ) );
71.  }
72.
73.  public PostObjectResponse postObject(PostObjectRequest request) {
74.     Base64EncodedObject object = request.getObject();
75.     boolean publiclyAvailable = request.getProperties().isPubliclyAccessible();
76.
77.     String objectName = object.getName();
78.
79.     String path = String.format( "/%s/%s",
80.                          request.getOwner().getName(),
81.                          UUID.randomUUID() );
```

```
82.
83.       String checksum =
84.          this
85.             .blobStoreObjectWarehouserService
86.                .putObject( publiclyAvailable ? publicContainerName : privateContainerName,
87.                      String.format( "%s/%s", path, objectName),
88.                      object.getContent() );
89.       return
90.          new PostObjectResponse(
91.             new WarehousedObject(
92.                new ObjectWarehousingLocation( objectName,
93.                                   path,
94.                                   new WarehouseProperties(publiclyAvailable) ),
95.                checksum ) );
96.    }
97.    public void putObject(PutObjectRequest request) {
98.       ObjectWarehousingLocation location = request.getLocation();
99.       Base64EncodedObject object = request.getObject();
100.
101.            String absoluteObjectPath = location.getFullPath();
102.
103.            if ( location.getName() != null) {
104.               absoluteObjectPath += location.getName();
105.            } else {
106.               absoluteObjectPath += object.getName();
107.            }
108.
109.            String checksum =
110.               this
111.                  .blobStoreObjectWarehouserService
112.                     .putObject( location.getProperties().isPubliclyAccessible() ?
113.                                   publicContainerName :
114.                                   privateContainerName,
115.                                absoluteObjectPath,
116.                                object.getContent() );
117.          }
118.
119.          public void deleteObject(DeleteObjectRequest request) {
120.             ObjectWarehousingLocation location = request.getLocation();
121.
122.             this
123.                .blobStoreObjectWarehouserService
```

```
124.                 .removeObject( location.getProperties().isPubliclyAccessible() ?
125.                                 publicContainerName :
126.                                 privateContainerName,
127.                             location.getFullPath() + location.getName() );
128.         }
129.
130.         public void copyObject(CopyObjectRequest request) {
131.             ObjectWarehousingLocation source = request.getSourceLocation();
132.             ObjectWarehousingLocation destination = request.getDestinationLocation();
133.
134.             this
135.                 .blobStoreObjectWarehouserService
136.                     .copyObject( source.getProperties().isPubliclyAccessible() ?
137.                                     publicContainerName :
138.                                     privateContainerName,
139.                                 source.getFullPath() + source.getName(),
140.                                 destination.getProperties().isPubliclyAccessible() ?
141.                                     publicContainerName :
142.                                     privateContainerName,
143.                                 destination.getFullPath() + destination.getName() );
144.         }
145.
146.         public void moveObject(MoveObjectRequest request) {
147.             ObjectWarehousingLocation source = request.getSourceLocation();
148.             ObjectWarehousingLocation destination = request.getDestinationLocation();
149.
150.             this
151.                 .blobStoreObjectWarehouserService
152.                     .moveObject( source.getProperties().isPubliclyAccessible() ?
153.                                     publicContainerName :
154.                                     privateContainerName,
155.                                 source.getFullPath() + source.getName(),
156.                                 destination.getProperties().isPubliclyAccessible() ?
157.                                     publicContainerName :
158.                                     privateContainerName,
159.                                 destination.getFullPath() + destination.getName() );
160.         }
161.
162.     }
```