Математические структуры и моделирование 2020. № 1(53). С. 108–138

УДК 004.77 : 004.424
DOI 10.24147/2222-8772.2020.1.108-138

РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ ПРОГРАММНОГО ПРИЛОЖЕНИЯ ДЛЯ УДАЛЕННОГО ХРАНЕНИЯ ДАННЫХ

Л.А. Володченкова
к.б.н., доцент, e-mail: volodchenkova2007@yandex.ru
Д.В. Козырев
студент, e-mail: kozyrev.danil@gmail.com

Омский государственный университет им. Ф.М. Достоевского, Омск, Россия

Аннотация. В этой статьи представлен разработанный программный интерфейс (API) серверной части облачного хранилища данных.

Ключевые слова: облачное хранилище, серверная часть, программный интерфейс.

Введение

За последние десятилетия Интернет стал неотъемлемой частью жизни любого современного человека. С развитием Всемирной паутины, появлением переносных компьютеров и мобильных телефонов возникла и потребность в доступе к файлам в любое время и из любой точки Земного шара. Облачные хранилища призваны решить эту проблему. Они представляют собой сервис для хранения данных, с помощью которого пользователь может загрузить свои файлы на сервер, а затем скачивать и просматривать их с телефона, компьютера или другого устройства, имеющего доступ к сети Интернет.

В отличие от классического способа хранения файлов на собственных компьютерах, переносных устройствах или выделенных серверах, которые арендуются или приобретаются специально для подобных целей, структура облачных хранилищ клиенту не видна. С его точки зрения, данные хранятся и обрабатываются в так называемом «облаке», которое представляет собой один большой виртуальный сервер, физически же таких серверов может быть несколько, они могут располагаться удалённо друг от друга, возможно, даже на разных континентах [1].

Несмотря на все описанные достоинства, современные сервисы хранения данных пока ещё не способны полностью заменить привычные нам жёсткие диски, поскольку для работы с ними необходимо подключение к сети Интернет, что существенно ограничивает скорость передачи данных. Тем не менее, они незаменимы для синхронизации файлов между компьютерами, организации совместной работы с данным и резервного копирования. А благодаря своей гибкости и масштабируемости они могут использоваться не только индивидуальным пользователями, но и крупными предприятиями.

Цель этой статьи — описание разработки программного интерфейса (API) серверной части такого хранилища данных. В качестве целевой аудитории итогового приложения рассматриваются индивидуальные пользователи, которые смогут использовать приложение для хранения личных файлов и обмена ими друг с другом. Сформулируем следующие основные задачи, поставленные в данной работе:

1) определить требования к системе и разработать архитектуру приложения в соответствии с этими требованиями;

2) выбрать и описать программные средства реализации разработанной архитектуры;

3) на основе вышеперечисленных пунктов реализовать программный интерфейс серверной части приложения, сопроводить его интерактивной документацией;

4) провести автоматизированное тестирование разработанного программного обеспечения.

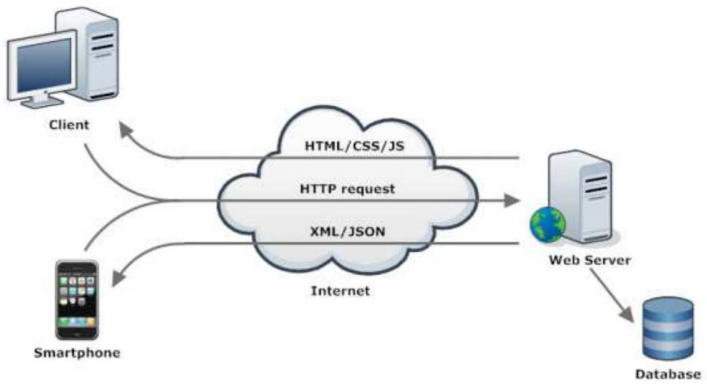

Рис. 1. Схема взаимодействия компонентов в архитектуре REST

1. Проектирование архитектуры

1.1. Анализ требований

Перед непосредственным проектированием архитектуры серверной части приложения должны быть проанализированы и сформулированы требования, которым эта архитектура должна отвечать. Требования к системе делятся на функциональные и нефункциональные. Функциональные описывают возможности программного интерфейса приложения, т. е. сценарии взаимодействия пользователя с системой, тогда как нефункциональные накладывают дополнительные ограничения на систему, не относящиеся к её поведению [3].

Изучив предметную область и существующие решения, мы сформулировали функциональные требования к API приложения (таблица 1).

К системе был также определён ряд нефункциональных требований. Их описание приведено в таблице 2.

Таблица 1. Функциональные требования к системе

№	Требование	Описание
1	Регистрация и авторизация пользователей	Для использования приложения необходимо наличие учётной записи. Это позволит разграничить дисковое пространство и обеспечит контроль доступа к хранимым данным
2	Группировка файлов в папки	Пользователь должен иметь возможность создавать папки и сохранять в них загруженные файлы или другие папки. Такая древовидная структура привычна и удобна
3	Просмотр и изменение метаданных	Метаданные о всех файлах и папках должны храниться на сервере, пользователь должен иметь возможность просматривать, изменять и добавлять собственные метаданные
4	Удаление файлов и папок	Пользователь должен иметь возможность удалять как отдельные файлы, так и папки целиком со всем их содержимым
5	Организация корзины	Корзина служит временным хранилищем для удалённых файлов и папок, обеспечивая возможность их последующего восстановления
6	Поиск данных	Пользователь должен иметь возможность искать файлы и папки по одному или нескольким параметрам
7	Обмен файлами	Пользователь должен иметь возможность поделиться своими файлами с другими пользователями

Таблица 2. Нефункциональные требования к системе

№	Требование	Описание
1	Удобство использования	API приложения должен отвечать современным требованиям к проектированию программного обеспечения, быть унифицированным и интуитивно понятным. Разработанный Программный интерфейс должен быть подробно задокументирован, что также упросит его использование
2	Разграничение доступа	Каждой учётной записи должна быть присвоена одна или несколько ролей в системе, с помощью которых разграничивается доступ к информации. Пользователь должен иметь доступ только к собственным данным
3	Ограничение на объем хранимых и передаваемых файлов	Система должна накладывать ограничения на размер загружаемых файлов и на максимальный объем дискового пространства, доступного пользователю
4	Шифрование данных при их передаче	Все взаимодействия клиента с сервером должны происходить по протоколу HTTPS, что обеспечивает конфиденциальность данных при их передаче
5	Надёжность	Система должна быть надёжной. Приложение должно продолжать функционировать при возникновении внештатных ситуаций
6	Простота тестирования	Архитектура должна обеспечивать простоту тестирования компонентов системы
7	Открытый программный код	Само приложение, а также используемое при его разработке программное обеспечение должно иметь открытый исходный код, что позволит убедиться в отсутствии уязвимостей и неприемлемого функционала

Определив требования к системе, можно приступать непосредственно к проектированию архитектуры.

1.2. Клиент-серверное взаимодействие в стиле REST

Взаимодействие клиента с программным интерфейсом приложения основано на использовании архитектурного стиля Representational State Transfer (сокращённо REST). Фактически REST API — это всего лишь набор конечных точек (веб-адресов), обращаясь к которым с помощью HTTP-запросов, клиент получает в ответ информацию с сервера в формате JSON, XML, HTML и т. д. Вся информация на сервере представлена в виде ресурсов и подресурсов: пользователи приложения, загруженные файлы, папки и т. д. У каждого ресурса есть свой уникальный идентификатор, ресурс имеет состояние, и клиент может получать или изменять состояние ресурса при помощи представлений (как уже упоминалось ранее, под представлением можно понимать JSON, XML, текст в определённом формате или что угодно, что позволяет нам понимать текущее состояние ресурса). При этом серверная часть никак не зависит от клиентской, клиентом веб-сервиса может выступать браузер пользователя, мобильное приложение, другой сервер и т. д.

Ниже представлено схематичное изображение взаимодействия компонентов в REST-архитектуре.

В программном интерфейсе REST для манипулирования ресурсами используются стандартные HTTP-методы в соответствии со спецификацией протокола [4]:

1) GET используется для получения текущего представления ресурсов;

2) POST используется для создания новых ресурсов;

3) PATCH используется для полного или частичного обновления существующих ресурсов;

4) PUT используется для создания нового ресурса или обновления существующего. Если ресурс с заданным идентификатором найден, информация о нём должна быть обновлена, в ином случае будет создан новый ресурс;

5) DELETE используется для удаления существующих ресурсов.

Запросы в такой архитектуре являются самодостаточными, серверу при их обработке нет необходимости извлекать контекст приложения, поскольку клиент включает в запрос все необходимые данные, использую для этого заголовки и тело запроса [4]. Такой подход повышает производительность, упрощает дизайн и реализацию серверных компонентов системы.

Обобщая вышесказанное, можно выделить несколько основных принципов, которым должна следовать разрабатываемая архитектура:

1) использование клиент-серверной модели;

2) использование стандартных методов протокола HTTP;

3) использование уникальных идентификаторов ресурсов;

4) манипуляция данными через представления в формате JSON;

5) отсутствие состояния на стороне сервера.

Соблюдение перечисленных принципов повышает надёжность и производительность приложения. Разработанное программное обеспечение обладает такими преимуществами, как простота и единообразие интерфейсов, портативность программных компонентов, лёгкость внесения изменений, а также способность

эволюционировать и приспосабливаться к новым требованиям.

1.3. Схема авторизации

Авторизация в системе осуществляется с использованием протокола OAuth 2.0. Этот протокол даёт возможность сторонним приложениям получать доступ к защищённым ресурсам сервера от имени владельца этих ресурсов, при этом приложение не имеет доступа к регистрационным данным пользователя.

В протоколе OAuth 2.0 существуют четыре участника процесса авторизации [5]:

1) владелец ресурса. Под владельцем подразумевается пользователь, которому принадлежат данные на сервере. Он авторизует приложение, тем самым разрешая ему доступ к своим данным;

2) клиент. Клиент — это приложение, которое хочет получить доступ к данным пользователя;

3) сервер ресурсов. Все данные пользователя, имеющие отношение к функционалу приложения, хранятся на сервере ресурсов, а клиент может взаимодействовать с ним, используя API сервиса;

4) сервер авторизации. Используется для хранения и проверки регистрационных данных клиента, а также выдачи токенов (ключей доступа). Токен передаётся серверу ресурсов с каждым запросом и необходим для идентификации клиента.

Для использования протокола OAuth клиенту необходимо иметь собственную учётную запись на сервере. При регистрации каждому клиенту присваивается уникальный идентификатор и секрет, у него также могут быть указаны адрес перенаправления (redirect URI) и области видимости (scopes). Когда приложение отправляет запрос на получение нового токена, в параметрах запроса указываются области видимости, определяющие, какой тип доступа ему нужно получить (чтение, запись, полный доступ и т. д.). А на указанный адрес приложение будет перенаправлено сервером авторизации после того, как данные клиента были проверены, и пользователь разрешил доступ к запрошенным областям видимости.

На рис. 2 представлена абстрактная схема авторизации клиента в протоколе OAuth 2.0.

Для получения токенов в протоколе определены четыре возможных сценария, причём выбор конкретного способа зависит от типа приложения и поддержки этого сценария сервером авторизации [5]:

1) неявный доступ (implicit grant). Сначала клиент переходит на сервер авторизации, предоставляя при этом свой идентификатор и redirect URI (секрет при этом не используется). После проверки данных и получения от пользователя разрешения, сервер перенаправляет клиента по указанному адресу и передаёт ему токен в виде фрагмента адреса. Этот сценарий подходит для мобильных и веб-приложений, которые не могут хранить секрет в тайне;

2) код авторизации (authorization code grant). Используется преимущественно серверными приложениями, поскольку их исходный код недоступен третьим

Рис. 2. Абстрактная схема авторизации клиента в протоколе OAuth

лицам. Аналогичен сценарию неявного доступа, за исключением необходимости предоставления секрета, а также наличия дополнительного шага для получения кода авторизации и последующего запроса с его помощью токена доступа;

3) учётные данные владельца (resource owner password credentials grant). Заключается в передаче клиенту имени и пароля пользователя напрямую и в последующем получении с их помощью токена доступа. Подходит для надёжных приложений, которым разрешено иметь прямой доступ к регистрационным данным пользователя;

4) учётные данные клиента (client credentials grant). Необходимы в том случае, если клиент хочет получить доступ к собственной учётной записи, чтобы изменить адрес перенаправления или секрет.

Хотя протокол OAuth выглядит достаточно громоздким и сложным, он фактически стал стандартном в индустрии программного обеспечения. Все рассмотренные в первой главе сервисы хранения используют именно этот способ авторизации.

В разрабатываемой системе будет реализован как сервер ресурсов, так и сервер авторизации. Это даст возможность другим разработчикам использовать программный интерфейс сервиса для создания собственных клиентских приложений.

2. Трёхуровневая архитектура

В предыдущих параграфах мы рассмотрели взаимодействие приложения с внешним миром, а в этом будет описана его внутренняя структура.

Перечислим основные обязанности, которые должны выполнять компоненты программы:

1) обработка запросов и проверка введённых пользователем данных, а также

последующий возврат ему корректного ответа;

2) обработка исключений с предоставлением пользователю подробной информации об ошибке;

3) управление транзакциями базы данных;

4) авторизация пользователей;

5) реализация основной логики приложения;

6) взаимодействие с базой данных.

Реализация всех этих обязанностей возможна с использованием всего трёх архитектурных уровней, каждый из которых может взаимодействовать только с компонентами своего или нижележащего уровня:

1) уровень представления (web layer). Самый верхний уровень приложения. Он отвечает за обработку запросов и возврат корректного ответа. Уровень представления также должен обрабатывать все исключения, возникающие в нижележащих уровнях. Поскольку этот уровень — входная точка приложения, он должен отвечать за аутентификацию и выступать первой линией защиты от неавторизованных пользователей;

2) уровень логики (service layer). Находится на один уровень ниже слоя представления. Этот уровень определяет границы транзакций базы данных и содержит все логические операции на данными. Уровень логики также отвечает за контроль доступа на уровне вызова методов, определяя, есть ли у пользователей право читать или изменять данные;

3) уровень данных (repository layer). Самый нижний слой приложения. Отвечает за взаимодействие с базой данных.

В архитектуре используются два типа объектов, которые непосредственно хранят данные, но не содержат логики:

1) сущности базы данных (domain models). Сущности представляют собой проекции таблиц в базе данных на классы в программе. Каждый объект такого класса соответствует записи в таблице и хранит отношения с другими объектами;

2) объекты передачи данных (data transfer objects или DTO). Простой контейнер информации, которая передаётся по сети. Такие объекты используются, когда необходимо скрыть часть данных и/или изменить их структуру.

Рассмотрим схему движения данных между уровнями приложения. Сначала уровень представления получает от клиента данные в формате JSON и создаёт на их основе объекты передачи данных. Затем он вызывает уровень логики, передавая ему DTO. Логический слой преобразует полученные DTO в сущности базы данных, выполняет необходимые операции над сущностями и передаёт их уровню данных, который сохраняет значения полей этих объектов непосредственно в таблицы базы данных. Описанное взаимодействие изображено на рис. 3.

Такой подход к построению архитектуры изолирует компоненты системы друг от друга, упрощает повторное использование уже существующих модулей и их тестирование, обеспечивает целостность данных за счёт объединения нескольких логических действий в одну транзакцию и, наконец, повышает безопасность приложения, поскольку каждый компонент контролирует доступ к

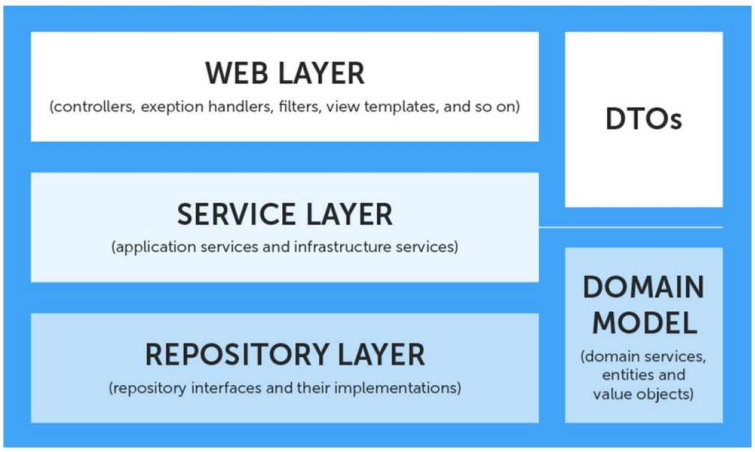

Рис. 3. Трёхуровневая архитектура

данным своего уровня.

2.1. Схема базы данных

В проектируемой системе используется реляционная модель данных. Для соответствия функционала приложения функциональным требованиям, определённым в начале этой главы, были выделены пять классов-сущностей базы данных:

1) User (пользователь). Хранит регистрационные данные всех пользователей приложения (адрес электронной почты, хеш пароля в формате bcrypt, имя и т. д.). У каждого пользователя есть набор принадлежащих ему файлов и папок;

2) Role (роль). Определяет роли, назначаемые пользователям, причём один пользователь может иметь несколько ролей;

3) Folder (папка). Хранит метаданные пользовательских папок. Каждая папка принадлежит одному владельцу, имеет одну родительскую папку и может содержать несколько вложенных папок и файлов, образуя тем самым древовидную структуру;

4) File (файл). Хранит метаданные пользовательских файлов и расположение файла на жёстком диске. Каждый файл принадлежит одному владельцу, имеет одну родительскую папку и может иметь несколько дополнительных свойств;

5) Property (свойство). Свойства файла — это дополнительные метаданные, которые были самостоятельно добавлены пользователем. Каждое свойство имеет поле key (ключ) и value (значение). У файла может быть только одно свойство с заданным ключом. Для отображения сущностей на таблицы базы данных используется технология объектно-реляционного отображения (Object-Relational Mapping), которая будет рассмотрена в следующих главах.

Ниже представлены таблицы 3–8 базы данных и схема отношений между

ними.

Таблица 3. Отношение «пользователи»

№	Наименование атрибута	Тип данных	Описание
1	id	bigint	Идентификатор пользователя
2	version	bigint	Версия
3	date_created	timestamp	Дата регистрации
4	date_modified	timestamp	Дата последнего изменения
5	email	varchar	Адрес электронной почты
6	password	varchar	Хеш пароля
7	first_name	varchar	Имя
8	last_name	varchar	Фамилия

Таблица 4. Отношение «роли»

№	Наименование атрибута	Тип данных	Описание
1	id	bigint	Идентификатор роли
2	name	varchar	Имя роли

Таблица 5. Отношение «пользователи — роли»

№	Наименование атрибута	Тип данных	Описание
1	user_id	bigint	Идентификатор пользователя
2	role_id	bigint	Идентификатор роли

Таблица 6. Отношение «папки»

№	Наименование атрибута	Тип данных	Описание
1	id	bigint	Идентификатор папки
2	version	bigint	Версия
3	date_created	timestamp	Дата создания
4	date_modified	timestamp	Дата последнего изменения
5	name	varchar	Имя
6	root	boolean	Флаг, указывающий на корневую папку
7	parent_id	bigint	Идентификатор родительской папки
8	owner_id	bigint	Идентификатор владельца

2.2. Организация корзины

При удалении файлы и папки не исчезают безвозвратно, а попадают в корзину. Корзина служит временным хранилищем удалённых объектов, позволяя восстановить или удалить их окончательно. В разрабатываемой архитектуре корзины как отдельной сущности не существует, она является всего лишь абстракцией. При создании учётной записи пользователя создаётся его корневая

Таблица 7. Отношение «файлы»

№	Наименование атрибута	Тип данных	Описание
1	id	bigint	Идентификатор файла
2	version	bigint	Версия
3	date_created	timestamp	Дата загрузки
4	date_modified	timestamp	Дата последнего изменения
5	name	varchar	Имя
6	size	bigint	Размер в байтах
7	mime_type	varchar	MIME-тип файла
8	location	varchar	Путь к файлу
9	parent_id	bigint	Идентификатор родительской папки
10	owner_id	bigint	Идентификатор владельца

Таблица 8. Отношение «свойства»

№	Наименование атрибута	Тип данных	Описание
1	id	bigint	Идентификатор свойства
2	version	bigint	Версия
3	date_created	timestamp	Дата загрузки
4	date_modified	timestamp	Дата последнего изменения
5	key	varchar	Ключ
6	value	varchar	Значение
7	file_id	bigint	Идентификатор файла

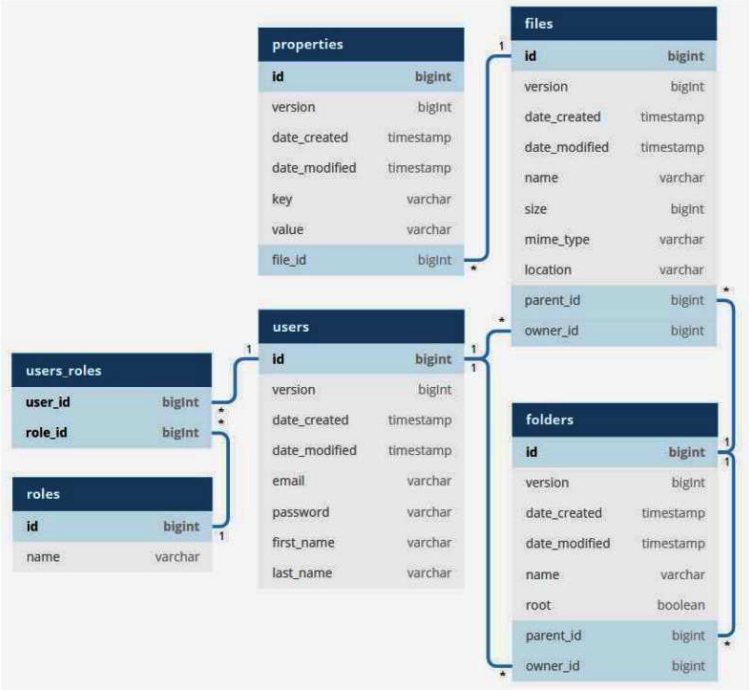

Рис. 4. Схема базы данных

папка с установленным флагом root. Корневые папки не имеют родителя, и

у каждого пользователя может быть только одна такая папка. Если у папки или файла нет родителя, они исключаются из основного дерева и считаются удалёнными в корзину (включая потомков). Таким образом, пользователь, перемещаясь по своему дереву папок, начиная от корня, видит только неудалённые объекты. Чтобы просмотреть содержимое корзины, необходимо всего лишь найти все некорневые папки и файлы, не имеющие родителя и принадлежащие данному пользователю. А чтобы восстановить удалённые данные, нужно вернуть их в основное дерево, присвоив нового родителя.

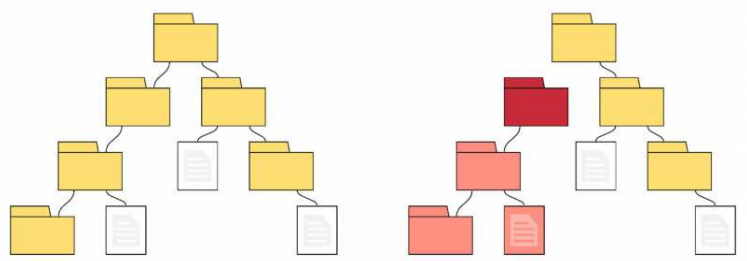

Рис. 5. Схема удаления папки и её потомков в корзину

2.3. Хранение файлов

Папки, создаваемые пользователями, существуют только в базе данных, это всего лишь строки в таблице. Файлы на сервере хранятся в собственной иерархии папок файловой системы.

Когда файл загружается на сервер, ему присваивается идентификатор UUID — это уникальный 128-битный номер, причём вероятность сгенерировать один и тот же UUID дважды так мала, что её можно считать нулевой. На основе этого идентификатора генерируется уникальный путь к файлу. Из UUID извлекаются три первые пары символов и используются в качестве имён папок, каждая последующая из которых вложена в предыдущую. Сам идентификатор становится именем файла. Содержимое файла копируются на жёсткий диск по этому пути, а все метаданные, включая его настоящее имя и путь к файлу, сохраняются в базу данных.

Рис. 6. Пример сгенерированного пути к файлу

Такой способ хранения помогает снизить нагрузку на файловую систему, поскольку большое количество файлов в одной папке сильно ухудшает производительность при работе с ними. Используемый алгоритм решает эту проблему, распределяя файлы равномерно. Кроме того, оригинальные имена файлов могут содержать запрещённые символы, и операционная система не позволит

сохранить файлы с такими именами. В UUID же используются только символ «-» и цифры шестнадцатеричной системы, любая комбинация из которых всегда является корректным именем файла.

2.4. Обмен файлами

Для обмена файлами между пользователями был использован механизм создания временных ссылок. Такая ссылка позволяет скачать файл любому пользователю и активна в течение нескольких часов после её создания.

Для генерации временных ссылок используются токены JSON Web Tokens. JWT — это стандарт для создания токенов доступа, которые состоят из трех частей в формате JSON: заголовка (header), полезной нагрузки (payload) и цифровой подписи (signature). Заголовок содержит метаданные, описывающие токен, в полезной нагрузке содержатся непосредственно данные, а подпись — это хеш-сумма, которая вычисляется на основании заголовка, полезной нагрузки и секретного ключа и необходима для подтверждения достоверности токена [8].

Стандартным способом подписи токена является использование алгоритма HMAC-SHA256. Для генерирования хеш-суммы этот алгоритм, помимо данных исходного сообщения, использует также секретный ключ. Любое изменение данных или значения хеша вызовет несовпадение подписи. А поскольку секрет для подписи известен только серверу, такой токен невозможно подделать.

После того, как все три элемента токена определены, его можно преобразовать в компактный формат. Для этого заголовок и полезная нагрузка кодируются с использованием схемы кодирования Base64, к ним добавляется хеш-сумма, и все составные части разделяются точками [8].

Рис. 7. Структура JWT

Когда пользователь запрашивает ссылку на файл, сервер генерирует новый токен, в качестве полезной нагрузки которого используются идентификатор файла и поле exp, обозначающее время, когда токен станет невалидным. К полезной нагрузке добавляется заголовок и подпись, после чего токен преобразуется в компактное представление. Клиенту возвращается ссылка вида

https://www.example.com/links/token, где переменная token — это сгенерированный компактный JWT. Если пользователь перейдёт по такой ссылке, сервер извлечёт токен, проверит его подпись, не истек ли его срок действия, а затем найдёт файл по идентификатору и вернёт содержимое.

У такого подхода есть как свои плюсы, так и минусы. С одной стороны, нет необходимости хранить созданные ссылки в базе данных, поскольку JWT несут в себе всю необходимую информацию. С другой стороны, такие ссылки нельзя отозвать. Пока срок действия токена не истёк, и файл, на идентификатор которого ссылается токен, не удалён, ссылка на скачивание будет активна.

3. Описание используемых программных средств

3.1. Язык программирования Java

Java — это строго типизированный объектно-ориентированный язык программирования высокого уровня. Он был разработан компанией Sun, а позднее приобретён корпорацией Oracle, которой и принадлежит до сих пор. Несмотря на свой почётный возраст, Java продолжает развиваться, оставаясь одним из самых популярных языков программирования.

Одним из главных преимуществ этого языка является его кроссплатформенность. Программы на Java компилируются в байт-код, который затем выполняется виртуальной машиной JVM (Java Virtual Machine). Преимущество такого подхода в том, что байт-код не зависит от операционной системы, таким образом, программы могут выполняться на любой платформе, для которой существует виртуальная машина Java.

Другое важное преимущество Java — наличие огромного количества библиотек с открытым исходным кодом, которые подходят для решения задачи любой сложности. В частности, в работе будут использованы фреймворки и библиотеки для создания веб-сервисов REST, контроля доступа, авторизации пользователей по протоколу OAuth, взаимодействия с базой данных посредством технологии Object-Relational Mapping, внедрения зависимостей, аспектно-ориентированного программирования, генерации и валидации токенов JWT, работы с JSON и мн. др.

Кроме того, этот язык достаточно прост в использовании. В нем присутствует автоматическое управление памятью за счёт использования механизма сборки мусора, а также строгая типизация, которая делает Java удобным средство для написания крупных проектов. Нельзя не отметить и богатый инструментарий. Интегрированные среды разработки (IDE), написанные на Java, такие, как Eclipse, NetBeans и продукты компании JetBrains, давно вышли за пределы экосистемы языка, став стандартным средством разработки программ на многих современных языках.

3.2. Spring Framework

Разрабатываемый проект основан на использовании фреймворка с открытым исходным кодом Spring. Функционал фреймворка сгруппирован в несколько

модулей: Core Container, AOP and Instrumentation, Messaging, Data Access, Web, Test. В следующих параграфах будут кратко рассмотрены использованные в проекте модули.

3.2.1. Spring Container

Spring Container — это модуль, использующийся для внедрения зависимостей. Контейнер Spring управляет жизненным циклом объектов и связывает их друг с другом. Объекты, которые он контролирует, называются бинами (от слова Bean). Классы бинов могут зависеть друг от друга, и задача контейнера Spring состоит в создании экземпляров этих классов и внедрении их в другие зависимые от них бины. Такой механизм обеспечивает слабую связанность классов, что улучшает структуру кода, упрощает их взаимозаменяемость и тестирование;

3.2.2. Spring AOP

Aspect-oriented Programming (AOP) — это парадигма программирования, основанная на выделении «сквозной функциональности» (cross-cutting concerns) в отдельные объекты. Такая функциональность затрагивает многие компоненты системы, но при этом не относится непосредственно к логике приложения. Примерами сквозной функциональности можно назвать логирование информации и управление транзакциями базы данных.

3.2.3. Spring MVC

Spring MVC — это фреймворк, в основе которого лежит технология Java Servlets. Spring MVC предназначен для разработки веб-приложений, и именно с его помощью разрабатывается REST API сервиса. Этот модуль предоставляет возможность обработки пользовательских запросов, их валидации, генерации HTML-страниц и т. д. Иначе говоря, Spring MVC отвечает за всё, что связано с сетевой частью приложения.

3.2.4. Spring Security

Как нетрудно догадаться, главная задача Spring Security заключается в аутентификации и авторизации клиентов приложения. Контроль доступа осуществляется как на уровне HTTP-запросов, так и на уровне вызова отдельных методов. Spring Security также реализует поддержку протокола OAuth, позволяя развернуть и гибко настроить собственный сервер авторизации и сервер ресурсов.

3.2.5. Spring Data Access

Spring Data отвечает за взаимодействие приложения с базой данных. Этот модуль поддерживает интеграцию с другими технологиями работы с базами

данных на Java, такими как JDBC и JPA, а также даёт возможность работать с нереляционными хранилищами. Кроме того, Spring Data позволяет декларативно управлять транзакциями за счёт использования приёмов аспектно-ориентированного программирования. Но одним из главных преимуществ этого модуля является возможность динамически генерировать классы доступа к базе данных на основе интерфейсов, определённых разработчиком.

3.2.6. Spring Boot

Spring Boot — это проект, направленный на упрощение разработки приложений с помощью фреймворка Spring. Он позволяет в максимально короткие сроки создать полностью функциональное приложение, требуя при этом минимум усилий от разработчика. Особенности Spring Boot:

1) встроенный веб-сервер (Tomcat, Jetty или Undertow), который автоматически запускается вместе с приложением;

2) набор готовых пакетов зависимостей (starter dependencies);

3) автоматическая конфигурация как самого Spring, так и некоторых сторонних библиотек;

4) возможность следить за состоянием приложения в реальном времени с помощью Spring Boot Actuator;

5) полностью избавляет разработчика от необходимости использовать XML-конфигурацию.

3.3. Java Persistence API

Для работы с базой данных в проекте используется технология объектно-реляционного отображения (Object-Relational Mapping или ORM). ORM отображает таблицы базы данных на объектную модель языка программирования. Класс соответствует таблице, а объект этого класса — записи в таблице. Выделим основные плюсы использования данной технологии:

1) возможность описать и создать схему базы данных, используя классы и граф отношений между ними;

2) возможность прозрачно манипулировать данными как объектами, автоматически генерируя тем самым SQL-запросы;

3) возможность лёгкой замены системы управления базами данных (СУБД), поскольку ORM-фреймворк использует платформо-независимый язык запросов, напоминающий SQL;

4) кеширование уже загруженных данных;

5) возможность использования механизма оптимистичных блокировок, управляемых фреймворком автоматически.

В экосистеме Java существует спецификация Java Persistence API (JPA), цель которой — стандартизация технологии объектно-реляционного отображения. JPA, кроме самого программного интерфейса, описывает требования, предъявляемые к сущностям, платформо-независимый язык запросов JPQL и

много другое. В разрабатываемом проекте в качестве провайдера, реализующего эту спецификацию, используется фреймворк Hibernate, а в качестве СУБД — PostgreSQL.

3.4. MapStruct

Как уже объяснялось во второй главе, в трёхуровневой архитектуре есть две иерархии объектов: сущности базы данных и объекты передачи данных (DTO). DTO используются, когда передача непосредственно сущностей клиенту не подходит по тем или иным причинам, например, когда необходимо разорвать циклические связи между объектами или скрыть некоторые данные. Эти объекты могут иметь много схожих полей, и трансформация DTO в сущность и обратно требует написания большого объёма однообразного кода, подверженного ошибкам.

MapStruct — это генератор кода, который значительно упрощает написание классов отображения, трансформирующих один тип объектов в другой. Такие классы генерируются на основе интерфейсов и аннотаций, применяемых к ним. Генерация происходит при компиляции и использует обычные вызовы методов, практически не отличаясь от кода, который написал бы программист вручную, благодаря этому использование MapStruct не сказывается на производительности приложения, а сгенерированный код легко поддаётся отладке [9].

3.5. Swagger

REST — это один из самых популярных на сегодня способов взаимодействия между клиентом и сервером в веб-приложениях. Но проблема в том, что REST — это не стандарт или протокол, а всего лишь архитектурный стиль, набор требований к программному интерфейсу. Клиенту же необходимо точно знать, какие данные ожидает получить сервер, в каком формате должны быть эти данные, значение каждого возможного кода состояния и т. д. Таким образом, разрабатываемый API приложения должен быть подробно задокументирован.

Swagger — это проект с открытым исходным кодом, предназначенный для документирования программного интерфейса REST. Он даёт возможность сгенерировать документацию и визуализировать её, используя Swagger UI. Кроме того, с помощью Swagger UI также можно отправлять запросы для проверки и тестирования разработанного API.

Спецификация Swagger, также известная как OpenAPI, имеет несколько реализаций. Одной них является библиотека SpringFox, позволяющая интегрировать Swagger с приложениями, разработанными с помощью Spring Framework.

3.6. JUnit и Mockito

Есть множество видов тестирования программного обеспечения: функциональное, интеграционное, нагрузочное и т. д. Одним из них является модульное, или юнит-тестирование, которое обычно выполняется программиста-

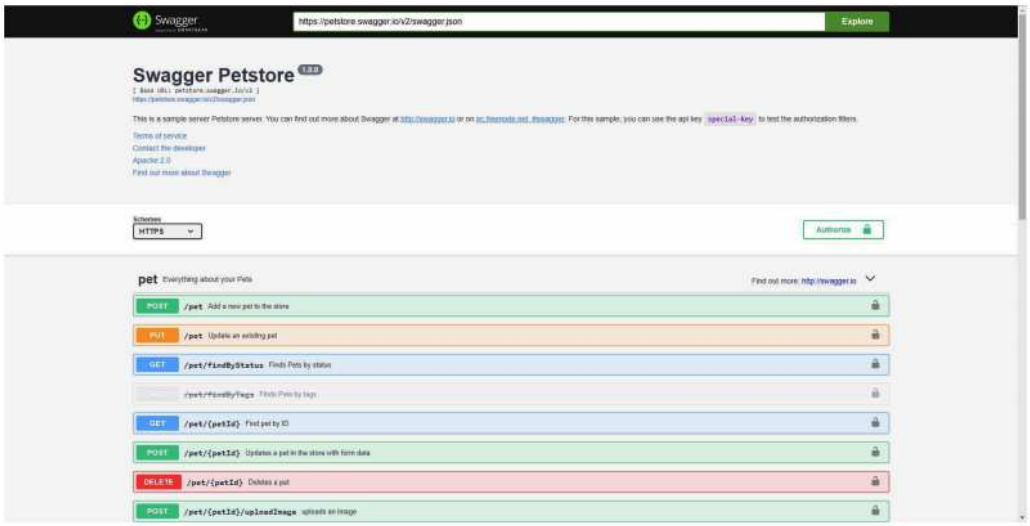

Рис. 8. Пример интерфейса Swagger UI

ми. Цель такого тестирования — проверка на корректность работы отдельных компонентов приложения (модулей) в изоляции друг от друга.

В проекте для написания юнит-тестов используются фреймворки JUnit и Mockito, которые чаще всего используются в связке друг с другом. JUnit является стандартным фреймворком для автоматизированного модульного тестирования приложений, написанных на Java. Он позволят запускать тестовые методы, группировать их для выполнения, проверять соответствие результатов теста ожидаемым и многое другое. Mockito же необходим для создания объектов-заглушек, которые позволяют тестировать модули изолированно. Они имитируют другие классы, и их поведение предопределено программистом. Кроме того, заглушки позволяют убедиться, что тестируемый объект выполнил требуемые от него действия.

3.7. Project Lombok

Программисты зачастую вынуждены писать большое количество однообразного кода. Описанные выше классы отображения являются одним из примеров такого кода. Но если необходимость писать подобные классы возникает не так часто, то писать стандартные методы доступа (getters и setters), сравнения (equals), вычисления хеш-кода (hashcode), преобразования объекта в строку (toString) и др. приходится в каждом Java-проекте без исключения. Project Lombok помогает упростить работу программиста, автоматизировав этот процесс.

Lombok — это проект для генерации исходного кода, добавляющий дополнительную функциональность в язык Java. Кроме описанных выше стандартных методов объектов, Lombok также позволяет:

1) использовать неявные типы переменных (var и val);

2) проверять аргументы методов на равенство null;
3) создавать конструкторы классов с разным набором параметров;
4) создавать классы-строители;
5) создавать неизменяемые классы;
6) автоматически освобождать ресурсы.

Всё, что нужно, чтобы использовать этот функционал, — это добавить аннотацию к классу или методу, и Lombok автоматически сгенерирует необходимый код при компиляции программы.

3.8. Maven

Maven — это фреймворк с открытым исходным кодом, который существенно упрощает и автоматизирует процесс сборки проектов. Maven позволяет добавлять внешние зависимости в проект, используя для этого центральный репозиторий, предоставляет богатый набор плагинов, расширяющих функционал, позволяет создавать дистрибутивы программ в виде архивов jar/war и многое другое. Кроме того, эта система сборки поддерживается всеми популярными средами разработки, которые позволяют импортировать проект Maven без необходимости его дополнительной конфигурации.

Центральным компонентом Maven является специальный XML-файл Project Object Model (pom.xml). POM используется для описания проекта и конфигурирования процесса сборки. Этот файл содержит информацию об имени и версии проекта, его авторе, используемой версии Java, внешних зависимостях, плагинах и т. д.

Все проекты, использующие Maven, имеют единую унифицированную структуру каталогов:

1) /src/main/java содержит исходный код программы;
2) /src/main/resources содержит дополнительные ресурсы приложения (SQL-скрипты, файлы конфигурации и т. д.);
3) /src/test/java содержит исходный код тестовых классов;
4) /src/test/resources содержит дополнительные ресурсы приложения, необходимые для тестирования;
5) /target содержит результаты сборки, которые включают в себя дистрибутив программы, скомпилированные классы, результаты тестирования и т. д.

4. Программная реализация

Файлы приложения распределены по каталогам согласно структуре проекта Maven: основные классы находятся в /src/main/java, тестовые классы –– в /src/test/java, а каталог /src/main/resources содержит файлы, необходимые для запуска и настройки приложения. Все классы принадлежат к базовому пакету com.github.danilkozyrev.filestorageapi и сгруппированы по функционалу. Для запуска приложения используется метод main из класса Application.

Далее будет рассмотрена реализация основных компонентов программы, описаны их структура и задачи.

4.1. Конфигурация

Для конфигурирования приложения используются классы из пакета config и файлы из директории /src/main/resources.

Директория resources содержит следующие файлы:

1) application.properties. Spring Boot позволяет настраивать большинство компонентов приложения, используя для этого свойства Java, которые хранятся в этом файле в виде пар ключ-значение;

2) import.sql. SQL-скрипт для загрузки исходных данных. Добавляет роли пользователя и администратора, учётные записи тестовых пользователей, их корневые папки;

3) keystore.p12. Хранилище ключей и сертификатов, необходимых для установки защищённого соединения по протоколу HTTPS.

Пакет config содержит конфигурационные классы, каждый из которых, за исключением ApplicationProperties, отмечен аннотацией @Configuration:

1) ApplicationProperties. Класс для чтения некоторых свойств из файла application.properties (максимальный объем хранилища, базовая папка, параметры генерации JWT). Используется сервисом FileService;

2) GlobalMethodSecurityConfig. Позволяет использовать аннотации @PreAuthorize и @PostAuthorize, контролирующие доступ к данным на уровне вызова методов;

3) OAuth2AuthorizationServerConfig. Необходим для включения и настройки сервера авторизации OAuth, также добавляет в приложение несколько тестовых клиентов;

4) OAuth2ResourceServerConfig. Необходим для включения сервера ресурсов OAuth и настройки контроля доступа к нему на уровне HTTP-запросов. К серверу ресурсов относятся все адреса, начинающиеся с префикса «api»;

5) SwaggerConfig. Необходим для включения и настройки Swagger. Позволяет авторизовать Swagger UI для доступа к защищенным ресурсам пользователя по протоколу OAuth;

6) WebConfig. Конфигурация веб-элементов. Содержит единственный бин, удаляющий имя класса исключения из сообщений об ошибках, передаваемых клиенту;

7) WebSecurityConfig. Позволяет настроить доступ к ресурсам, которые не относятся к серверу ресурсов, но должны быть защищены на уровне HTTP-запросов без использования протокола OAuth.

4.2. Сущности базы данных

Сущности базы данных находятся в пакете domain и отмечаются аннотацией @Entity. Базовым суперклассом для них является класс BaseEntity, который содержит единственное поле id, являющееся общим для всех сущностей. Значения идентификатора генерируются фреймворком последовательно, начиная с 1000.

У BaseEntity есть подкласс AuditableEntity, который содержит поля version,

Листинг 1. Класс BaseEntity

```
@MappedSuperclass
@Data
public abstract class BaseEntity implements Serializable {

    @Id
    @Column(updatable = false)
    @SequenceGenerator(name = "id_sequence_generator", initialValue = 1000)
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "id_sequence_generator")
    @ToString.Include
    private Long id;
```

dateCreated и dateModified. Значение поля version используется при оптимистичных блокировках и устанавливается фреймворком. Поля dateCreated и dateModified — это время создания и последнего изменения сущности, которые также устанавливаются автоматически в методах prePersist и preUpdate.

Листинг 2. Класс AuditableEntity

```
@MappedSuperclass
@Getter
@Setter
public abstract class AuditableEntity extends BaseEntity {

    @Version
    @Column(nullable = false)
    private Long version;

    @Column(nullable = false, columnDefinition = "TIMESTAMP WITH TIME ZONE")
    private Instant dateCreated;

    @Column(nullable = false, columnDefinition = "TIMESTAMP WITH TIME ZONE")
    private Instant dateModified;

    @PrePersist
    public void prePersist() {
        dateCreated = Instant.now().truncatedTo(ChronoUnit.MILLIS);
        dateModified = dateCreated;
    }

    @PreUpdate
    public void preUpdate() {
        dateModified = Instant.now().truncatedTo(ChronoUnit.MILLIS);
    }

}
```

Все классы сущностей также содержат аннотации для генерации схемы базы данных и конфигурирования фреймворка: имена таблиц, ограничения, индексы, каскадные операции, стратегии загрузки отношений и т. д.

4.3. Объекты передачи данных

Объекты передачи данных разделены на два пакета: в dto.forms находятся объекты, передаваемые от клиента к серверу, а в dto.projections — передаваемые от сервера к клиенту. Все эти классы можно условно объединить в четыре группы:

1) MetadataForm и Metadata. Используются для изменения и получения метаданных файла или папки;

2) UserForm и UserInfo. Используются для изменения и получения информации о пользователе;

3) PropertyForm и PropertyInfo. Используются для изменения и получения информации о свойствах файла;

4) SearchParameters. Контейнер для параметров поиска.

Все DTO-классы представляют из себя простой набор значений, но в объектах-формах также содержатся аннотации, использующиеся для проверки синтаксиса полученного запроса. Каждая подобная аннотация относится к одной из трех групп: Create, Update или Default, причём Create и Update являются подгруппами Default. Аннотации без параметра groups относятся к группе Default. Контроллер при получении запроса должен указать желаемую группу, тем самым определяя, какие аннотации будут использоваться при проверке полей объекта. Аннотации валидации и их группы определены в пакете validation.

4.4. Исключения

Рассматриваемые здесь исключения определены в пакете exception и генерируются исключительно классами логического уровня (сервисами). В пакете представлены шесть классов исключений:

1) CircularFolderStructureException. Информирует о возникновении циклической зависимости в дереве папок;

2) EmailAlreadyExistsException. Информирует о том, что предоставленный клиентом адрес электронной почты уже используется;

3) ExpiredFileTokenException. Информирует о том, что у предоставленного токена для скачивания файла истёк срок действия;

4) FileSystemException. Информирует о возникновении непредвиденной ошибки при чтении или записи файла;

5) RecordNotFoundException. Информирует о том, что запрашиваемый ресурс не был найден;

6) StorageLimitExceededException. Информирует о том, что клиент превысил ограничение на объём хранимых файлов.

Каждый из этих классов отмечен аннотацией @ResponseStatus, содержащей код состояния, и имеет поле message, содержащее сообщение об ошибке. При возникновении одного из этих исключений Spring автоматически сгенерирует и отправит клиенту ответ с заданным кодом и сообщением.

4.5. Интерфейсы классов отображения

Классы отображения трансформируют сущности в объекты передачи данных и используются сервисами. Сами классы генерируются библиотекой MapStruct, а в пакете mapper находятся только их интерфейсы.

На основе сигнатур методов MapStruct сгенерирует код, копирующий значения полей, причем все поля с одинаковыми именами распознаются автоматически. Если имена не совпадают, они указываются в параметрах source и target аннотации @Mapping. Если имена полей совпадают, а типы нет, то необходимо создать в интерфейсе метод, трансформирующий один тип в другой, MapStruct автоматически найдет и использует этот метод при генерации кода. Примером служит метод roleToString из интерфейса UserInfoMapper, трансформирующий роль в строку.

4.6. Репозитории

Репозитории относятся к уровню данных приложения и представляют из себя набор интерфейсов, с помощью которых сервисы манипулируют данными, абстрагируясь от механизма их хранения. В проекте реализовано по одному репозиторию для каждой используемой сущности.

При запуске приложения Spring Data создает прокси-объекты для всех интерфейсов, которые являются наследниками org.springframework.data.repository.Repository или его производных, таких как JpaRepository [10]. Кроме стандартных методов из JpaRepository в интерфейсы были добавлены и собственные методы, причём Spring автоматически преобразует имена методов в запросы к БД. Альтернативой именам служит аннотация @Query, которую удобно использовать, если имя получается слишком длинным. @Query также позволяет написать запрос на чистом SQL.

В качестве примера в следующем листинге приведены три способа создать запрос для поиска файлов больше заданного размера: с помощью имени метода, с помощью языка JPQL и с помощью SQL.

Листинг 3. Способы создания запросов в Spring Data

```
List<File> findFilesBySizeGreaterThan(Long size);

@Query("SELECT f FROM File f WHERE f.size > :size")
List<File> findFilesJPQL(@Param("size") Long size);

@Query(value = "SELECT * FROM files WHERE size > :size", nativeQuery = true)
List<File> findFilesSQL(@Param("size") Long size);
```

Кроме того, Spring Data-репозитории позволяют создавать динамические запросы «по примеру», которые используется в приложении для поиска файлов и папок. Такие методы используют объект-образец с любой комбинацией полей и выполняют поиск по этому образцу, игнорируя при этом null-значения.

4.7. Сервисы

Сервисы можно назвать центральными компонентами приложения, поскольку именно эти классы содержат логику управления данными. Каждый сервис представлен интерфейсом из пакета service и снабжён подробными комментариями JavaDoc, описывающими его назначение, функционал методов, параметры и возвращаемые значения (см. [11]). Исключением является класс

DefaultUserDetailsService, который возвращает объект UserPrincipal из пакета security, представляющий текущего пользователя, и используется фреймворком Spring при аутентификации учётной записи.

В проекте реализованы следующие сервисы:

1) FileService. Предоставляет методы для работы с файлами, включая создание токенов JWT для генерации временных ссылок;

2) FolderService. Предоставляет методы для работы с папками;

3) PropertyService. Позволяет добавлять, просматривать и удалять свойства файлов;

4) SearchService. Позволяет производить поиск файлов и папок по определённым параметрам, а также просматривать недавно изменённые файлы и папки;

5) TrashService. Позволяет просматривать и очищать содержимое корзины;

6) UserService. Предоставляет методы для работы с учётными записями пользователей приложения.

В каждом интерфейсе сервиса используется аннотация @PreAuthorize, которая служит механизмом контроля доступа на уровне вызываемых методов. Проверка выполняется перед вызовом метода на основании значения аннотации. Если значение выражения равно true, то метод выполняется, в противном случае Spring возбудит исключение AccessDeniedException.

Листинг 4. Способы контроля доступа на уровне вызываемых методов

```
List<File> findFilesBySizeGreaterThan(Long size);

@Query("SELECT f FROM File f WHERE f.size > :size")
List<File> findFilesJPQL(@Param("size") Long size);

@Query(value = "SELECT * FROM files WHERE size > :size", nativeQuery = true)
List<File> findFilesSQL(@Param("size") Long size);
```

Рассмотрим пример из листинга 4, в котором представлены два метода: получение содержимого корневой папки по id владельца и получение содержимого папки по id папки. В первом случае проверка доступа тривиальна. Необходимо лишь проверить, что id текущего пользователя совпадает с параметром, переданным методу. Но для аналогичной проверки во втором примере нам необходимо знать владельца папки с заданным id. В таких случаях в аннотации @PreAuthorize используется выражение hasPermission, которое делегирует проверку классу StorageItemPermissionEvaluator из пакета security. Этот класс находит объекты с заданным id и проверяет условия доступа к ним.

Каждый метод в реализации сервиса отмечен аннотацией @Transactional. Эта аннотация означает, что при вызове метода будет автоматически создана новая транзакция, а при успешном возврате из него эта транзакция будет зафиксирована. В случае возникновения в методе непроверяемого исключения произойдёт откат транзакции.

Классы сервисов также используют механизм публикации событий, что даёт возможность выполнить некоторые действия, такие как удаление файла с жёсткого диска, только после успешного завершения или отмены транзакции,

обеспечивая тем самым целостность данных. События из пакета event являются наследниками класса ApplicationEvent и содержат информацию о созданных или удалённых сущностях. Для публикации событий используется класс ApplicationEventPublisher, а методы из сервиса DefaultFileService, отмеченные аннотацией @TransactionalEventListener, обрабатываю эти события.

5. Контроллеры

Контроллеры относятся к уровню представления и отмечаются аннотациями @Controller или @RestController. Каждый из них привязан к определённому веб-адресу и HTTP-методу, используя аннотацию @RequestMapping или её производные (@GetMapping, @PostMapping и т. д.).

Задача контроллеров заключается в преобразовании объектов передачи данных в формат JSON и обратно с помощью аннотаций @RequestBody и @ResponseBody, валидации синтаксиса запроса с помощью аннотации @Validated, извлечении параметров запроса с помощью аннотаций @RequestParam и @PathParam, работе с заголовками и установке статуса ответа. Кроме того, в методах контроллеров используется аннотация @AuthenticationPrincipal, позволяющая получить объект UserPrincipal, который представляет текущего пользователя.

Контроллеры также могут обрабатывать исключения, возникающие на нижележащих уровнях, и возвращать клиенту сообщения об ошибках. Примером такого контроллера служит класс ExceptionHandlerController. Он отмечен аннотацией @RestControllerAdvice, а его методы — аннотацией @ExceptionHandler, описывающей, какой тип исключения этот метод обрабатывает. Этот способ генерации сообщений об ошибках аналогичен использованию аннотации @ResponseStatus, которая была рассмотрена ранее при описании исключений.

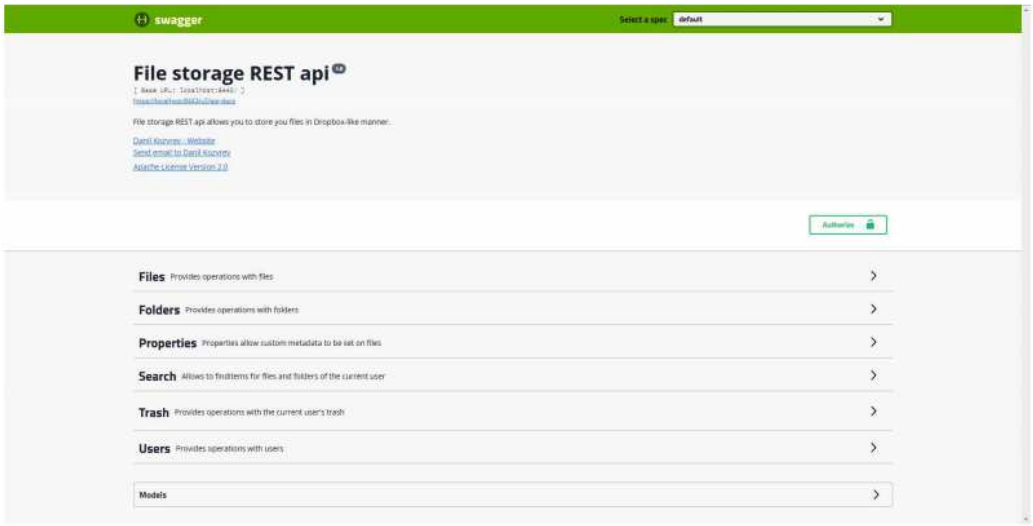

Рис. 9. Список ресурсов

5.1. Документация

Swagger автоматически генерирует документацию к программному интерфейсу на основе используемых контроллеров и аннотаций @Api, @ApiOperation, @ApiResponses. Документация используется в качестве домашней страницы сервера и доступна по адресу: https://localhost:8443.

Ресурсы в документации группируются согласно параметру tags аннотации @Api. Для каждой конечной точки приведены её адрес и HTTP-метод, параметры, которые необходимо передать серверу, а также пример успешного ответа и возможные сообщения об ошибках. Кнопка «try it out» позволяет отправить запрос на сервер, но для этого необходимо авторизоваться, нажав кнопку «Authorize» и введя данные учетной записи.

Секция Models содержит информацию об используемых в API структурах данных. Как можно заметить, эти модели соответствуют объектам передачи данных, поскольку именно их возвращают и принимают контроллеры.

6. Тестирование

В работе было проведено модульное тестирование классов логического уровня, поскольку именно они содержат основную логику приложения, а значит, и наиболее подвержены ошибкам.

6.1. Генерация тестовых данных

Для генерации тестовых данных приложения используются классы-строители (builders), которые генерируются с помощью библиотеки Lombok. Строители значительно улучшают читаемость кода, поскольку позволяют создавать множество объектов с заранее установленными значениями полей и перезаписывать необходимые поля в тестовых методах, используя «текучий интерфейс».

Строители в проекте — это внутренние классы, которые сгруппированы по типу генерируемых объектов в три класса-обёртки из пакета util: FormBuilders, ProjectionBuilders и EntityBuilders. Рассмотрим в качестве примера сгенерированный класс-строитель UserFormBuilder (см. Листинг 5).

Как можно заметить, этот класс имеет поля с заданными значениями по умолчанию, а также предоставляет возможность установить новые значения, вызвав соответствующие методы. Метод build создаёт экземпляр класса UserForm, а сам экземпляр строителя можно создать, вызвав метод defaultUserForm.

Все классы-строители однотипны, и использование библиотеки Lombok значительно упрощает написание кода.

6.2. Методика тестирования

Модульное тестирование предполагает тестирование каждого компонента системы в изоляции. Для достижения этой цели используются объекты-

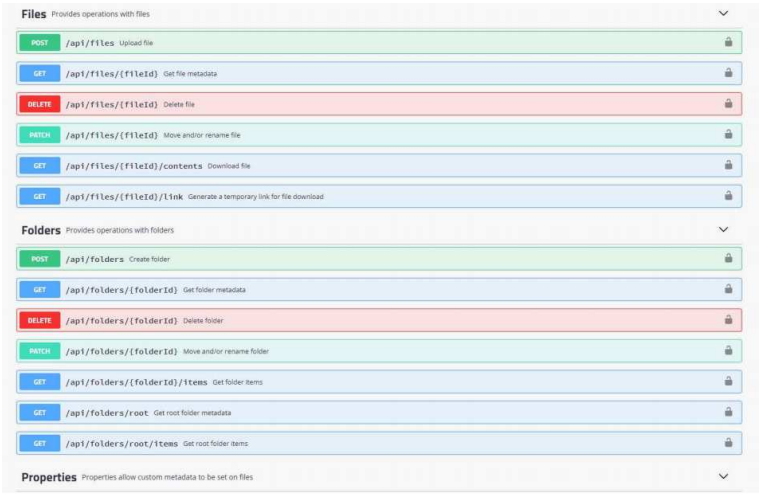

Рис. 10. Список конечных точек

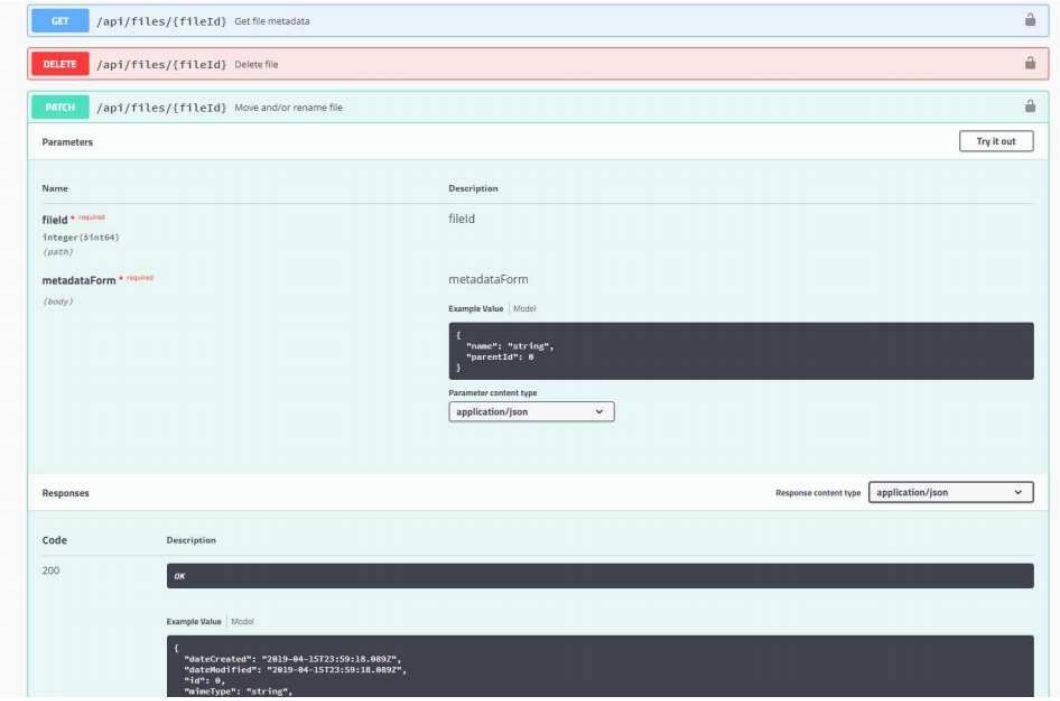

Рис. 11. Информация о конечной точке

заглушки, создаваемые библиотекой Mockito с помощью аннотации @Mock. Поля тестируемого класса заполняются этими объектами с помощью аннотации @InjectMocks. Кроме того, при необходимости можно «захватить» аргументы, переданные методам заглушек, используя объекты класса ArgumentCaptor.

Рассмотрим примерный алгоритм тестирования метода:

1) создаётся тестовый метод, отмеченный аннотацией @Test;
2) с помощью классов-строителей создаются тестовые объекты;

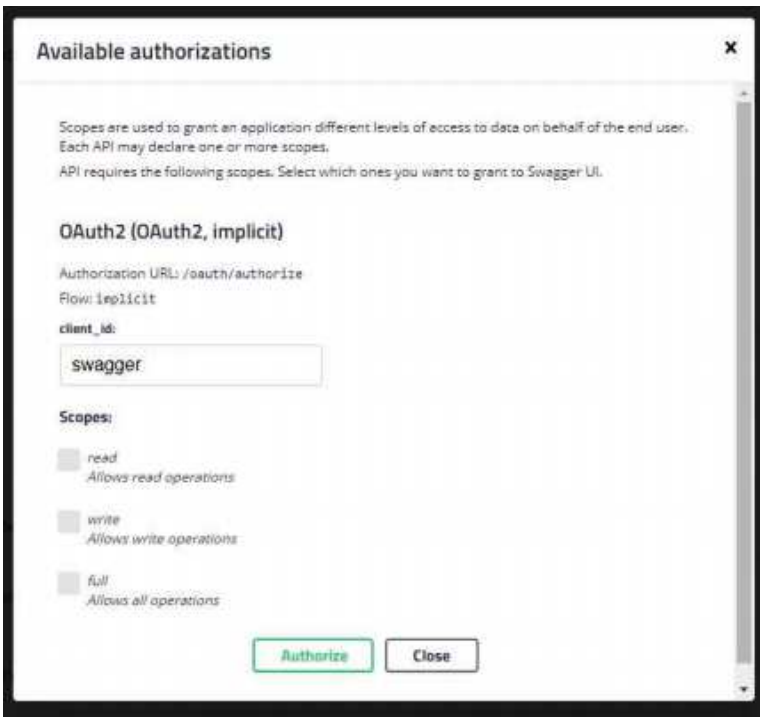

Рис. 12. Окно авторизации

Рис. 13. Секция Models

3) используя методы when и thenReturn, устанавливаются значения, которые должны возвращать заглушки при вызове их методов;

Листинг 5. Сгенерированный класс-строитель UserFormBuilder

```
public static class UserFormBuilder {

    private String firstName = "Fred";
    private String lastName = "Bloggs";
    private String email = "fred.bloggs@example.com";
    private String password = "fred.bloggs";

    UserFormBuilder() {
    }

    public FormBuilders.UserFormBuilder firstName(final String firstName) {
        this.firstName = firstName;
        return this;
    }

    public FormBuilders.UserFormBuilder lastName(final String lastName) {
        this.lastName = lastName;
        return this;
    }

    public FormBuilders.UserFormBuilder email(final String email) {
        this.email = email;
        return this;
    }

    public FormBuilders.UserFormBuilder password(final String password) {
        this.password = password;
        return this;
    }

    public UserForm build() {
        return FormBuilders.createUserForm(
                this.firstName, this.lastName, this.email, this.password);
    }

}

public static FormBuilders.UserFormBuilder defaultUserForm() {
    return new FormBuilders.UserFormBuilder();
}
```

4) вызывается тестируемый метод сервиса;

5) используя метод verify, проверяется, что ожидаемые методы заглушек были вызваны тестируемым классом;

6) используя метод assert, проверяется, что значения полей объектов соответствуют ожидаемым.

В следующем листинге приведен пример тестирования метода getUserInfo из класса UserService.

Тестирование разбито на две части. Сначала тестируется сценарий, когда пользователь найден, а затем проверяется обратный вариант. Названия тестовых методов состоят из трёх частей, разделённых подчёркиванием: имя тестируемого метода, условие, ожидаемый результат. Если в первом варианте тестируется успешный вызов метода, и порядок действий аналогичен алгоритму, то во втором аннотация @Test имеет параметр expected, который говорит о том, что тест будет считаться успешным только в случае возникновения исключения RecordNotFoundException. Для этого методы when/thenReturn устанавливают, что репозиторий при запросе к нему должен вернуть пустой объект.

Листинг 6. Тестирование метода getUserInfo

```
@Test
public void getUserInfo_whenUserFound_shouldReturnUserInfo() {
    var userId = 0L;
    var user = defaultUser().id(userId).build();
    var userInfo = defaultUserInfo().build();

    when(userRepository.findById(userId)).thenReturn(Optional.of(user));
    when(userInfoMapper.mapUser(user)).thenReturn(userInfo);

    var returnedUserInfo = userService.getUserInfo(userId);

    assertThat(returnedUserInfo, equalTo(userInfo));
}

@Test(expected = RecordNotFoundException.class)
public void  getUserInfo_whenUserNotFound_shouldThrowException() {
    var userId = 0L;
    when(userRepository.findById(userId)).thenReturn(Optional.empty());
    userService.getUserInfo(userId);
}
```

Тестирование всех остальных методов в приложении производится аналогично рассмотренному примеру.

Заключение

В результате нами разработан программный интерфейс серверной части приложения для удалённого хранения данных. В ходе работы был произведён анализ предметной области, рассмотрены существующие решения, изучены достоинства и недостатки систем облачного хранения. Разработана и детально описана архитектура системы, которая отвечает заявленным функциональным и нефункциональным требованиям. Выбраны и изучены программные средства реализации разработанной архитектуры. И, наконец, система реализована и протестирована. Кроме того, программный интерфейс приложения был снабжен подробной документацией в виде интерактивной веб-страницы.

Благодаря применению современных программных средств и подходов к разработке реализованная система легко расширяема и может быть использована с любым клиентом без внесения изменений в серверный код. Таким образом, дальнейшее приоритетное направление развития проекта — это разработка клиентской части приложения. Поскольку сервис ориентирован на индивидуальных пользователей, необходимо предоставить им удобный интерфейс для работы с данными.

Литература

1. Облачное хранилище данных // Википедия: свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Облачное$_{-}$хранилище$_{-}$данных (дата обращения: 01.04.2019).
2. Ковалык В., Томичева Т. Возможности облачных хранилищ для бизнеса и образования // Интерактивная наука. 2017. № 11. С. 185–187.

3. Анализ требований // Википедия: свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Анализ$_{-}$требований (дата обращения: 05.04.2019).
4. Веб-сервисы RESTful: основы // IBM Developer Россия. URL: https://www.ibm.com/developerworks/ru/library/ws-restfu/index.html (дата обращения: 08.04.2019).
5. Введение в OAuth 2 // DigitalOcean, Inc. URL: https://www.digitalocean.com/community/tutorials/oauth-2-ru (дата обращения: 10.04.2019).
6. Reinsel D., Gantz J., Rydning J. The Digitization of the World – From Edge to Core // An IDC White Paper. URL: https://www.seagate.com/files/wwwcontent/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата обращения: 03.04.2019).
7. What are public, private and hybrid clouds? // Microsoft Azure. URL: https://azure.microsoft.com/en-us/overview/what-are-private-public-hybridclouds/ (дата обращения: 04.04.2019).
8. Introduction to JSON Web Tokens // JWT.io by Auth0. URL: https://jwt.io/introduction/ (дата обращения: 12.04.2019).
9. MapStruct 1.3.0. Final Reference Guide // MapStruct. URL: http://mapstruct.org/documentation/dev/reference/html/ (дата обращения: 15.04.2019).
10. Spring Data JPA – Reference Documentation // Spring Framework 5. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (дата обращения: 17.04.2019).
11. Козырев Д.В. Разработка серверной части приложения для удалённого хранения данных. Выпускная квалификационная работа. Омск : ОмГУ, 2019. 57 с.

CREATION OF THE SERVER SIDE OF THE PROGRAMM FOR REMOTE DATA STORAGE

L.A. Volodchenkova
PhD. (Biology), Associate Professor, e-mail: volodchenkova2007@yandex.ru
D.V. Kozirev
Student, e-mail: kozyrev.danil@gmail.com

Dostoevsky Omsk State University, Omsk, Russia

Abstract. In this article the programm interface (API) of the server side of the cloud data storage is created.

Keywords: cloud storage, server side, software interface.

References

1. Oblachnoe khranilishche dannykh, Vikipediya: svobodnaya entsiklopediya. URL: https://ru.wikipedia.org/wiki/Oblachnoe$_{-}$khranilishche$_{-}$dannykh (01.04.2019). (in Russian)

2. Kovalyk V. and Tomicheva T. Vozmozhnosti oblachnykh khranilishch dlya biznesa i obrazovaniya. Interaktivnaya nauka, 2017, no. 11, pp. 185–187. (in Russian)
3. Analiz trebovanii, Vikipediya: svobodnaya entsiklopediya. URL: https://ru.wikipedia.org/wiki/Analiz$_{-}$trebovanii (05.04.2019). (in Russian)
4. Veb-servisy RESTful: osnovy. IBM Developer Rossiya. URL: https://www.ibm.com/developerworks/ru/library/ws-restfu/index.html (08.04.2019). (in Russian)
5. Vvedenie v OAuth 2. DigitalOcean, Inc. URL: https://www.digitalocean.com/community/tutorials/oauth-2-ru (10.04.2019). (in Russian)
6. Reinsel D., Gantz J., and Rydning J. The Digitization of the World – From Edge to Core. An IDC White Paper. URL: https://www.seagate.com/files/wwwcontent/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (03.04.2019).
7. What are public, private and hybrid clouds? Microsoft Azure. URL: https://azure.microsoft.com/en-us/overview/what-are-private-public-hybridclouds/ (04.04.2019).
8. Introduction to JSON Web Tokens. JWT.io by Auth0. URL: https://jwt.io/introduction/ (12.04.2019). (in Russian)
9. MapStruct 1.3.0. Final Reference Guide. MapStruct. URL: http://mapstruct.org/documentation/dev/reference/html/ (15.04.2019). (in Russian)
10. Spring Data JPA – Reference Documentation. Spring Framework 5. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (17.04.2019).
11. Kozyrev D.V. Razrabotka servernoi chasti prilozheniya dlya udalennogo khraneniya dannykh. Vypusknaya kvalifikatsionnaya rabota, Omsk, OmGU, 2019, 57 p. (in Russian)

Дата поступления в редакцию: 30.03.2020