Санкт-Петербургский государственный университет
Факультет прикладной информатики — процессов управления

Кафедра технологии программирования

Грибков Кирилл Владимирович
Магистерская диссертация

Разработка Data API для нейрофизиологических исследований

Направление 02.04.02
Фундаментальная информатика и информационные технологии

Научный руководитель:
д. ф.-м. н., профессор Богданов А. В.

Рецензент:
к. т. н., доцент Руковчук В. П.

Санкт-Петербург
2019

Оглавление

Введение

Современную научную деятельность невозможно представить без активного использования информационных технологий. В частности, это относится к исследованиям в области патологий мозга.

Проведение комплексных исследований затрудняет неоднородность данных, разнообразие форматов представления и ресурсоемкая предварительная обработка. Для исследователя процесс объединения данных для каждого отдельного случая весьма трудоемок, кроме времени, требуются также глубокие знания в области информационных технологий. Решить проблему совместного использования разнородных данных, можно информационной системой с единым доступом к разнородным данным. Для внедрения такой системы требуется создания модели объединения разнородных данных в единую информационную среду и адаптации методов предварительной обработки, применяемых индивидуально к каждому отдельному типу данных.

Для решения вышеизложенных задач, совместно с институтом Бехтерева разрабатывается проект информационной системы в области исследований человеческого мозга[16]. Данный проект включает в себя множество компонентов, каждый из которых решает отдельную задачу: система сбора и хранения информации о пациентах и анализах в рамках проекта; консолидация разнородных данных; система Data API — хранение и доставка данных на узлы вычислительного кластера; система единой авторизации для всех компонентов проекта; разнообразные вычислительные компоненты, которые используют полученную информацию о пациентах и анализах для исследований.

Целью настоящей работы является разработка системы Data API и реализация единой авторизации в рамках проекта по созданию информационной системы в области исследований человеческого мозга. Конкретные задачи для достижения указанной цели: разработать Data API; разработать клиентское приложение Data API; разработать структуру БД; разработать систему загрузки и выгрузки данных; реализовать единую авторизацию всех компонентов проекта в системе.

1. Обзор предметной области

В данной работе были поставлены 3 основные задачи: передача больших файлов на узлы вычислительного кластера, сбор и хранения информации о файлах на узлах кластера и организация единой авторизации для всех компонентов проекта. Также важным пунктом реализации является интеграция системы с проектом института Бехтерева. Существующие решения могут выполнять часть функционала описанного в постановке задачи. Так некоторые имеют возможность передавать большие объемы данных, другие имеют возможность собирать и хранить информацию о данных. Но ни одно решение не отвечает всем поставленным требованиям. Таким образом, существующей функциональности проектов не хватает для выполнения поставленной цели, в этой связи актуальной задачей является реализация своего сервиса. Но сначала рассмотрим более подробно существующие сервисы и инструменты и их отличие от реализуемой системы.

1.1. ckan

ckan[18] — это инструмент для создания сайтов с открытыми данными.

- Предназначен для публикации открытых данных.
- Имеет встроенный поиск и базу данных.
- Предоставляет API для взаимодействия со сторонними приложениями.

Данный инструмент больше предназначен для публикации открытых данных. Соответственно, для использования данного инструмента, необходимо написать систему ограничения доступа к данным.

1.2. Распределенные файловые системы

Существует большое множество разнообразных распределенных файловых систем. В контексте данной работы, распределенные файловой системы решают задачи передачи файлов и хранение метаинформации по файлам.

- Позволяют хранить метаинфомрацию о файлах
- Позволяют передавать файлы между узлами
- Распределенные файловые системы предполагают использование своей файловой системы на узлах кластера

1.3. Протоколы передачи данных

С задачей передачи файлов мог бы справиться любой из существующих протоколов передачи файлов. Но существующие популярные протоколы обычно работают через соответствующее окружение. Например: ftp, BitTorrent, SFTP, http, https и т.д. Хорошей практикой, если нет надобности использовать окружение протокола, является написание собственного протокола передачи файлов поверх TCP. Файл представляет из себя набор байтов, если файл имеет небольшой размер, то нормальной практикой является передача файла сплошным потоком байтов через TCP сокет, если же файл имеет большой размер и не целесообразно передавать его одним потоком байтов, то нормальной практикой является передача файла блоками, с подтверждением передачи каждого блока.

1.4. Система единой авторизации

1.4.1. CAS

CAS[5] — открытый протокол аутентификации.

- Позволяет авторизовываться в сторонних приложениях с помощью единой системы

- Сервер реализован только на Java

Серверная часть авторизации реализована на Java, означает что при выборе данного решение добавляется дополнительная задача по разворачиванию дополнительного сервера авторизации.

1.4.2. OIDC

OIDC[13] (OpenID Connect) — открытый стандарт единой системы аутентификации.

- Является стандартом, по этому имеет множество реализаций на разных плтформах
- Позволяет авторизовываться в сторонних приложениях, при этом передавая им только данные необходимые для верификации пользователя. Логин и пароль нужен только для первичной атворизации, в дальнейшем они не используются.
- Имеет множество разнообразных методов авторизации. В том числе авторизация с помощью только одного клиентского приложения
- Популярное решение единой авторизации, в том числе в крупных кампаниях, таких как: Google, Microsoft, Oracle.

1.5. Вывод

Все существующие решения не решают проблему интеграции системы со всем проектом Бехтерева. Т.е. даже если система решает задачу сбора информации о данных в кластере и задачу загрузки и выгрузки данных с кластера, поверх такой системы все равно нужно писать Data API, которое предоставляла бы информацию с учетом роли пользователя в системе и возможностей клиентского приложения. Таким образом, было принято решение, разработать Data API, который никак не зависит от файловой системы на узлах кластера, с возможностью передачи файлов реализованной поверх обычного TCP соединения.

Для единой системы авторизации было принято решение использовать OIDC стандарт. Т.к. данный стандарт имеет большое количество реализаций в разнообразных системах, в том числе node.js, .Net, django. Также данная система является проверенным решением.

2. Используемые технологии

2.1. MEAN стек

Data API и его окружение основано на т.н. MEAN стеке. MEAN — это аббревиатура от четырех программных компонентов: MongoDB [7], Express.js [6], Angular [2], Node.js [10]. Все 4 компонента данного стека хорошо взаимодействуют друг с другом, все они связаны с JavaScript, даже в MongoDB запросы пишутся при помощи типа данных JSON, который является одним из стандартных типов JavaScript.

2.2. MongoDB

В качестве базы данных используется MongoDB. Это документо-ориентированная база данных, не требующая описания схемы таблиц; классифицируется как NoSQL база данных. Вместо традиционной реляционной структуры базы данных MongoDB использует JSON-подобные документы с динамическими схемами, из-за этого интеграция в определенных видах приложениях происходит проще и быстрее. По моему мнению в случае работы с Node.js особенности MongoDB перерастают в преимущества перед SQL базами данных. Из-за того что MongoDB хранит всю БД в Json-подобных документах отпадает надобность использовать маппинг данных при выборке из БД, т.к. в JavaScript одним из стандартных типов является JSON-объект.

2.3. Node.js

Node.js — это кроссплатформенная среда выполнения для языка JavaScript с открытым исходным кодом. По сути Node.js превращает JavaScript в язык общего назначения для выполнения на серверной части. Также данная программная платформа позволяет подключать внешние библиотеки, которые могут быть написаны на других языках.

2.4. Angular 6

Angular — это открытая и свободная платформа для разработки веб-приложений, написанная на языке TypeScript. Angular нацелен на разработку SPA-решений (Single Page Application), то есть одностраничных приложений.

2.5. express.js

express-js — это web-фреймворк для платформы Node.js. Express.js позволяет организовать на Node.js http-сервер. Данный пакет основан на пакете connect, который и является http-сервером. Сам же express имеет минималистичную и гибкую структуру для создания API.

2.6. npm

npm[11] — это менеджер пакетов для языка JavaScript. Он является стандартным для програмной платформы Node.js. Данный менеджер пакетов состоит из клиентской программы с интерфейсом командной строки и онлайн базой данных общедоступных пакетов, называемой реестром npm. Доступ к реестру осуществляется через клиент, также возможно просматривать и искать доступные пакеты через веб-сайт[11] npm.

2.7. Mongoose

Для работы с MongoDB в Node.js используется пакет mongoose [8]. Данный пакет поддерживает стандартные для MongoDB функции для добавления, изменения и удаления объектов из БД. Также mongoose позволяет делать множество вспомогательных вещей для работы с БД. Например MongoDB само по себе не проверяет типы данных в полях каждого объекта. Т.е. в MongoDB в разных объектах одной и той же модели в одинаковом поле могут содержаться данные разных типов. Чтобы избежать такого поведения пакет mongoose позволяет наложить

на поля правила целостности. И касается это не только типа данных, но и так же, например, значения по умолчанию или проверки на то, что поле не может быть пустым. Также данный пакет позволяет добавить к схеме модели виртуальные поля, виртуальные методы и триггеры.

2.8. OpenID

OpenID [13] — открытый стандарт децентрализованной системы аутентификации, предоставляющей пользователю возможность создать единую учётную запись для аутентификации на множестве не связанных друг с другом интернет-ресурсов, используя услуги третьих лиц.

2.9. .Net Core

.Net Core[20] — модульная платформа для разработки программного обеспечения разрабатываемая Microsoft. Данная платформа основана на .Net Fraemwork и отличается от последней модульностью и кроссплатформенностью. Данная платформа используются в данном проекте для написания сервисов для узлов кластера.

3. Архитектура DataAPI

Сбор и анализ данных о нейрофизиологических исследований предполагает большие объемы данных, которые влекут за собой большие хранилища данных и большие объемы вычислительных ресурсов. Задачи Data API: хранить информацию о данных на серверах; осуществлять загрузку и выгрузку данных из серверов. Исходя из вышеперечисленных задач, была разработана архитектура, которую можно разделить на 4 основных компоненты:

- Data API
- Сервисы узла системы
- Клиентское приложение для загрузки и выгрузки данных
- Клиентские приложения Data API

3.1. Data API

Одна из основных задач непосредственно Data API — это хранения информации о данных на серверах и, соответственно, последующая работа с данной информацией. Загрузка и выгрузка данных с серверов ложится на плечи сервисов, которые расположены на узлах кластера. Но доступ к данным сервисам предоставляет также через Data API. Также Data API обрабатывает всю информацию, которую присылают данные сервисы, например такую как: информация о загружаемом файле, какой пользователь инициировал загрузку и т.д.

Сервер Data API написан на платформе node.js, реализован с помощью фреймворка express.js. Была реализована следующая структура сервера (Рис. 1):

server.js — точка входа сервера. В данном файле инициализируется сервер, подключается файл промежуточного слоя и также инициализируется конфигурация сервера

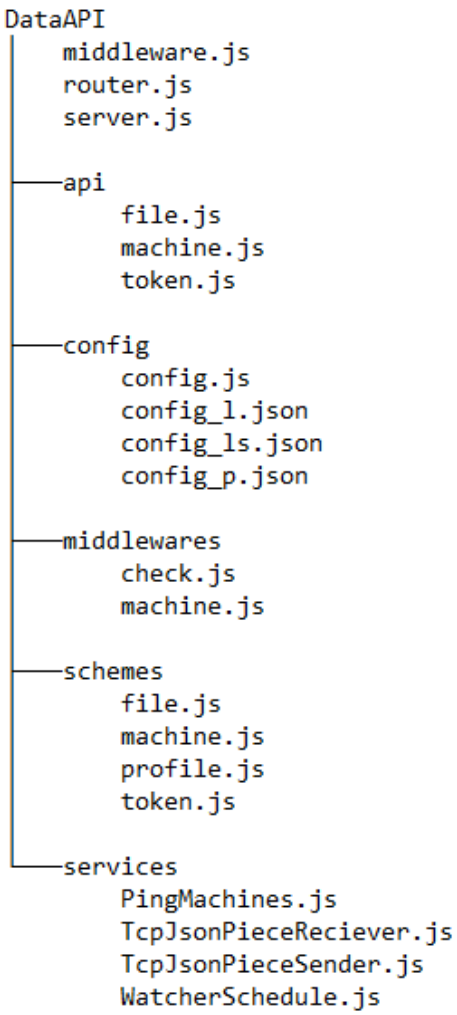

Рис. 1: Структура сервера Data API

middleware.js — промежуточный слой сервера. В данном файле подключаются все остальные компоненты сервера: инициализируется подключение к БД, подключаются роутеры обрабатывающие запросы к серверу, инициализируется настройка mongoose — пакета для работы с MongoDB, также в данном файле настраиваются статические файлы, в данном случае это файлы клиентского веб-приложения.

router.js — к данному файлу подключаются обработчики запросов. Сам router.js подключается к middleware.js.

./api — в данной директории расположены разнообразные обработчики запросов.

./middlewares — в данной папке расположены разнообразные промежуточные обработчики запросов. Данные обработчики используются перед непосредственной обработкой конкретного запроса. Промежуточный слой может пропустить запрос, а может отклонить запрос и отправить ответ с ошибкой. Некоторые из промежуточных обработчиков:

check.js — промежуточный обработчик, который может проверить авторизован ли пользователь и может проверить наличие определенных ролей у пользователя.

machine.js — проверяет валидность запроса от сервисов кластера. Он проверяет такие параметры как token и machineId.

./config — в данной папке хранится конфигурация сервера. Что бы использовать определенную конфигурацию, необходимо при запуске сервера указать соответствующий аргумент: -l — для локальной конфигурации, -p — для конфигурации на живом сервере, -ls — локальная конфигурация, которая запускает сервер под https.

./schemes — в данной папке расположены схемы моделей базы данных.

./services — в данной папке расположены разнообразные сервисы сервера Data API. Некоторые из них:

PingMachines.js — проверяет доступность сервисов на кластере. Проверка осуществляется по расписанию установленном на сервере.

WatcherSchedule.js — сервис который запускает расписания сверки всех файлов на узле кластера.

TcpJsonPieceSender.js/TcpJsonPieceReciever.js — сервисы с помощью которых можно отправить или получить json-объект через TCP-сокет. Используется для общения с сервисами кластера.

3.2. Сервисы узла системы

3.2.1. Сервис Sendfile

Данный сервис предназначен для загрузки и выгрузки файлов с узла системы. Сервис прослушивает заданный порт и принимает файлы, перед этим валидируя подключение с помощью заранее созданного пользователем токена в Data API. Валидация происходит также через данный API, только инициатором валидации является сервис Sendfile. Далее, перед непосредственной загрузкой, на Data API отправляются метаданные загружаемого файла: размер файла, оригинальное имя файла, контрольная сумма, время начала загрузки файла, с помощью какого токена была инициализирована загрузка файла. После загрузки файла, отправляется время окончания загрузки файла.

Данный сервис реализован с помощью платформы .net Core и представляет из себя TCP-сокет сервер, который принимает запросы от соответствующего приложения Sendfile.

3.2.2. Сервис Watcher

Данный сервис предназначен для сверки файлов. Он нужен для получения информации о файлах, попавших в систему не через сервис Sendfile или просто для получения свежей информации о файлах в системе. Реализуется данная задача с помощью двух функций сервиса: сверка определенного файла и сверка всех файлов узла находящихся в просматриваемых папках. Обе эти функции инициализируются через Data API. Сверка всех файлов узла проводится раз в день по расписанию. Сверка определенного файла нужна для прояснения и разрешения конфликтных ситуаций, которые обнаруживает либо Data API, либо пользователь Data API.

Данный сервис также реализован с помощью платформы .net Core и представляет из себя TCP-сокет сервер, который принимает запросы от Data API.

3.3. Клиентское приложение для передачи файлов

Данное приложение является консольным и выполняет только две функции: загрузка и выгрузка файлов. Что бы начать загрузку или выгрузку файлов, нужно предварительно создать токен загрузки или токен выгрузки через Data API. Далее данное приложение использует введенный пользователем токен, что бы получить информацию о том, с какими узлами будет происходить последующая работа по загрузке или выугрзке данных.

Клиентское приложение Sendfile реализовано на платформе .Net Core. Представляет из себя TCP клиентское приложение, который отправляет указанные ему файлы на TCP сервер.

3.4. Клиентские приложения Data API

Data API имеет возможность предоставлять доступ к своему клиентскому API любому приложению зарегистрированному через единую систему авторизации. На текущий момент такое приложение только одно — это Angular web-приложение, которое предоставляет доступ ко всему клиентскому API, например: просмотр файлов на узлах, просмотр доступности сервисов на узлах, просмотр информации о узлах, инициализация конкретной или полной сверки файлов на узле и т.д. Клиентское приложение Data API — это одностраничное веб-приложение, реализованное с помощью фреймворка Angular. Структура клиентского приложения Data API (Рис. 2):

app.module — это главный модуль всего приложения, к нему подключаются остальные компоненты, модули, сервисы, провайдеры и т.п.

./files — в данной папке расположены компоненты, которые реализуют работу с файлами через Data API. Некоторые из компонентов:

file.service — компонент, через который непосредственно осуществляются файловые запросы к Data API.

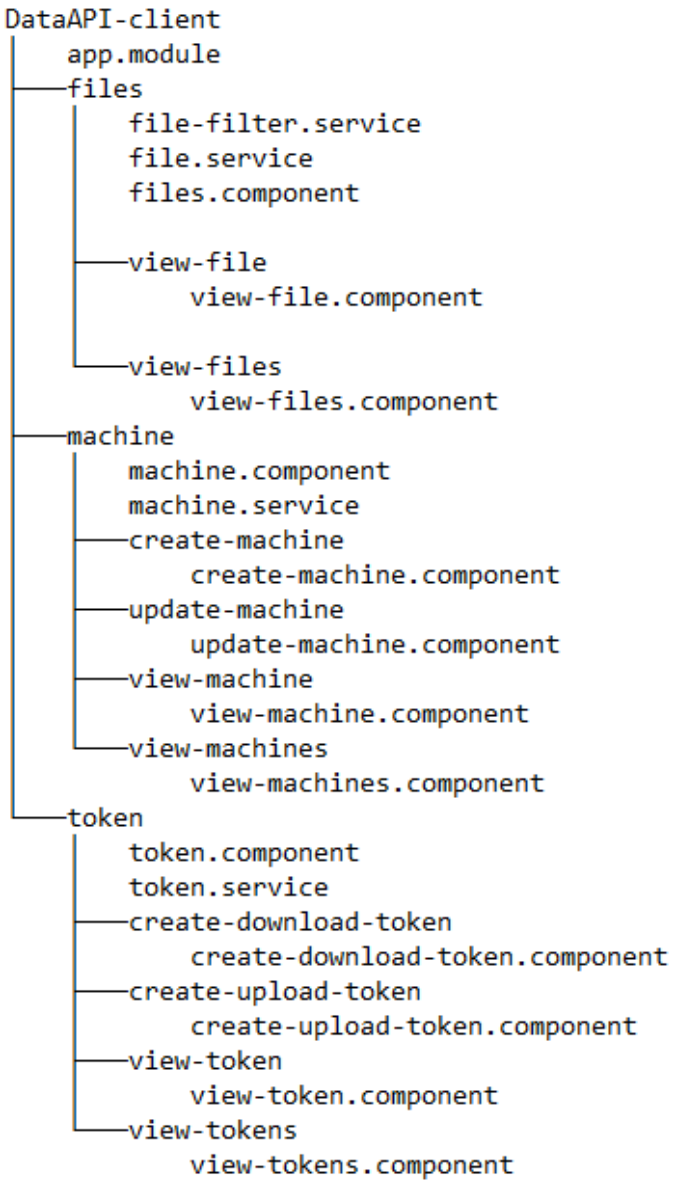

Рис. 2: Структура клиентского приложения Data API

file-filter.service — компонент, в котором реализована система фильтров, сортировки, группировки полученных файлов на клиентском приложении.

view-files — компонент отображения списка файлов. В данный компонент передается список файлов и он отображает только данный список. По этому с помощью этого компонента, можно отображать группировку файлов используя его несколько раз, для разных списков файлов.

view-file — компонент подробного отображения информации о фай-

ле.

machine — папка компонентов, реализующих работу с узлами кластера через Data API.

machine.service — реализует запросы к Data API.

view-machines — реализует отображение всех узлов кластера.

view-machine — реализует отображения подробной информации о узле кластера.

update-machine — реализует систему изменения информации о узле кластера.

create-machine — реализует систему создания узла кластера в системе.

token — папка компонентов, реализующих работу с токенами для загрузки и выгрузки файлов с узлов кластера.

token.service — реализует запросы к Data API.

view-tokens — реализует отображение всех токенов принадлежащих пользователю. Если пользователь администратор, в данном компоненте реализована возможность просмотра всех существующих токенов всех пользователей.

view-token — реализует отображения подробной информации о конкретном токене.

create-upload-token — реализует компонент для создания токена выгрузки файлов.

create-download-token — реализует компонент для создания токена загрузки файлов.

4. База данных: MongoDB

Как говорилось выше, в качестве базы данных используется документно-ориентированная MongoDB. В node.js для работы с этой базой используется пакет mongoose.

БД в MongoDB разделяется на коллекции. Коллекция — это смысловая часть базы данных, т.е. ее нужно воспринимать как отдельный класс для хранения. Непосредственно сами данные хранятся в виде json-подобных документов. MongoDB не требует строгого описания хранимых документов, т.е. в теории, в одной коллекции, могут храниться json-документы с совершенно разнообразной структурой. Тем не менее, в MongoDB существуют некоторые ограничения на хранимые документы, например, ограничение на вес хранимого документа, он не должен превышать 15 Мб или ограничение на глубину вложений в json-документе, не более 1000. Также существуют ограничения, которые может выставить сам пользователь, например, создание индекса в коллекции, тогда каждый документ в коллекции должен будет содержать информацию, удовлетворяющую проставленному индексу.

Для работы с MongoDB удобно использовать сторонние фреймворки, которые предоставляют привычные функции: описания хранимых данных, значения по умолчанию, тригеры, валидация данных. В данном проекте используется пакет mongoose для node.js. Данный пакет позволяет описать хранимые документы в коллекции с помощью json-подобной схемы. Внутри такой схемы можно указать какие поля будут храниться в определенной коллекции, каких они будут типов, значения по умолчанию и т.д. Валидацию этих условий mongoose осуществляет на стороне сервера, т.е. до отправления запроса к MongoDB.

4.1. Схема профиля пользователя

Данная схема используется для хранения данных о пользователе.

4.1.1. Схема

Листинг 1: Схема модели профиля пользователя

```
1 const Profile = new Schema({
2     username: { type: String, unique: true, required: true },
3     password: { type: String, required: true },
4     name: { type: String },
5     patronymic: { type: String },
6     surname: { type: String },
7     position: { type: String },
8     roles: [{ type: String }],
9 })
```

Как видно из схемы, данных хранится не так много. Имя пользователя и пароль используются для входа в систему. Из остальных полей, стоит обратить внимание на поле roles. Данное поле используется почти во всех API проекта, по данному полю разграничивается зона возможностей определенного пользователя. Например пользователь с ролью patient имеет возможность производить всевозможные действия с коллекцией пациентов в БД: добавлять, удалять, изменять. Стоит отметить, что разные роли, могут давать доступ к одинаковым API. Так например роль admin перекрывает все возможные API в проекте и таким образом, перекрывает все возможности роли patient.

4.1.2. Виртуальные методы

Также вместе с данной схемой используется еще одна функция пакета mongoose, а именно виртуальные методы. В случае схемы профиля пользователя, используется два метода: метод для генерации хеша пароля пользователя, данный хеш будет храниться в БД и метод для валидации пароля.

Функция генерации хеша:

Листинг 2: Метод генерации хеша

```
1 Profile.methods.generateHash = function generateHash(password) {
2     return bcrypt.hashSync(password, bcrypt.genSaltSync(32), null)
3 }
```

Хеш генерируется с помощью функции bcrypt[17]. В её основе лежит криптографический алгоритм Blowfish[4].

Функция проверки пароля имеет схожую структуру, но использует другой метод bcrypt — compareSync. Который, как следует из названия, синхронно сверяет введенные в него данные и хеш.

4.2. Схема узла DataAPI

Данная схема предназначена для хранения всех основных настроек узла в DataAPI.

Листинг 3: Схема модели узла кластера

```
const Node = new Schema({
    node: {
        name: { type: String, required: true },
        host: { type: String, required: true },
        sendfile: {
            tcp: { type: Number, required: true },
            http: { type: Number },
        },
        watcher: {
            tcp: { type: Number, required: true }
        },
        secret: { type: String, required: true }
    },
    files: {
        count: { type: Number, default: 0 },
        size: { type: Number, default: 0 }
    },
    info: {
        avialibleTypes: [{ type: String }],
        avialibleExtentions: [{ type: String }],
    },
    creationDate: { type: Date, default: Date.now, required: true },
    createdBy: { type: String },
})
```

Данная схема разделена на блоки:

- В блоке node — хранится вся информация, которая непосредствен-

но описывает сам узел кластера: имя машины, хост машины, порт сервиса sendfile, порт сервиса watcher, секретный ключ узла. Почти все поля в данном блоке являются обязательными. Обязательность поля устанавливается с помощью ключа required.

- Блок files хранит информацию о количестве файлов на узле и их общий объем. Данная информация периодически обновляется Data API.

- Блок info — информацию о типах и расширениях файлов, которые рекомендуется загружать на данный узел. Данную рекомендацию можно нарушить.

4.3. Схема файла

Схема файла описывает информацию о файле на узле.

Листинг 4: Схема модели файла

```
1  const File = new Schema({
2      file: {
3          path: { type: String, required: true },
4          size: { type: Number, required: true },
5          checksum: { type: String, required: true },
6          type: { type: String },
7          realname: { type: String }
8      },
9      machine: {
10         _id: { type: Schema.Types.ObjectId, ref: 'Node', required: true }
11     },
12     recheck: [{
13         date: { type: Date },
14         path: { type: String },
15         size: { type: Number },
16         checksum: { type: String },
17         machineId: { type: Schema.Types.ObjectId, ref: 'Node'}
18     }],
19     lastcheck: { type: Date },
20     upload: {
21         tokenId: { type: Schema.Types.ObjectId, ref: 'Token' },
```

```
22          start: { type: Date },
23          end: { type: Date }
24      }
25  })
```

Данная схема также разделена на блоки:

file — основная информация о файле: его размер, путь к файлу на узле, его контрольная сумма, оригинальное имя

machine — хранит одно значение: id узла, на котором хранится файл. В данном поле установлено поле ref — означает, что данное поле нужно считать внешним ключем к коллекции Node

recheck — список сверок файла. Система DataAPI анализирует информацию присланную с watcher сервиса и заполняет данный список: дата сверки, полученный размер, путь файла, контрольная сумма и id узла. В дальнейшем данная информация используется для поиска проблемных файлов. Например файлов у которых сменилась контрольная сумма или размер.

upload — информация о загрузки файла: начало и конец загрузки файла на узел, с помощью какого токена была произведена загрузка файла. Данная информация заполняется при загрузке файла с помощью Sendfile сервиса.

4.4. Схема токена

Токен в системе нужен для ограниченного доступа к узлам. Например токен может выдаваться на загрузку или выгрузку файлов с узлов.

Листинг 5: Схема модели токена

```
1  const Token = new Schema({
2    token: { type: String, required: true },
3    machineId: [{ type: Schema.Types.ObjectId, ref: 'Node' }],
4    userId: { type: String },
5    creationDate: { type: Date, required: true, default: Date.now },
6    expiresIn: { type: Number, default: 60 * 60 * 1 },
```

```
7    type: { type: String, required: true },
8  });
```

Как видно из листинга, схема токена хранит: непосредственно сам токен; id узлов на которые токен распространяется; id пользователя который создал данный токен; дата создания; время действия токена в секундах; тип токен (загрузка, выгрузка).

5. Единая система аутентификации

Как и говорилось выше, проект в рамках которого выполняется данная работа подразумевает большое количество разнообразных сервисов, API и приложений которые должны взаимодействовать друг с другом. Реализовывать авторизационную систему в каждом приложении или API нецелесообразно. По этому было принято решение, реализовать децентрализованную систему авторизации, т.е. единую авторизацию с единым профилем пользователя для всех сервисов и приложений проекта.

Для реализации такого подхода авторизации, было решено использовать OpenID Connect. Это аутентификационная надстройка над протоколом авторизации OAuth 2.0[12]. Данное решение и данный протокол давно зарекомендовали себя как хорошее, надежное решение для децентрализованной авторизации. Его используют такие компании, как Google, Amazon, Steam и многие другие.

Основная идея данного протокола — это возможность предоставления третьим лицам доступ к защищённым ресурсам пользователя без передачи им логина и пароля. В данном проекте под третьими лицами понимаются сервисы, приложения и API.

Авторизационную систему было решено реализовывать на одном из серверов проекта, а именно в рамках системы учета пациентов. Данный сервер работает на базе node.js, по этому для реализации OIDC используется пакет oidc-provider, который реализует oidc-стандарт на платформе node.js.

Для работы oidc-provider[22] необходимо установить несколько компонентов:

- Общая конфигурация OIDC — всевозможные параметры системы, включение и выключения функция пакета oidc-provider, которые не требуют дополнительной реализации.

- Адаптер базы данных — он необходим для сохранения сессий авторизованных пользователей. Система учета пациентов использу-

ет базу данных MongoDB, по этому необходимо настроить адаптер для использования этой БД.

- Список клиентских приложений — это приложения, которым разрешено использовать данную систему авторизации. Если приложение не находится в данном списке, пользователь не сможет авторизоваться в данном приложении через данную систему авторизации.

- Ключи шифрования системы — необходимы для создания подписи для выдаваемых токенов и, соответственно, для верификации токенов по подписи. Данные третьим лицам предоставляются в том случае, если у них имеется соответствующий токен пользователя. Данный токен выдается пользователю, а точнее клиентскому приложению, при авторизации.

5.1. Общая конфигурация OIDC

Общая конфигурация OIDC делится на несколько блоков:

- Блок настройки доступа к данным.

- Настройка формата токенов пользователя. В данном случае используются jwt токены.

- Конфигурация доступных функций OIDC. В данном блоке включаются такие фнукции, как шифрование, управление сессиями, возможность аннулирования выданного токена и многие другие.

- Настройка времени действия. В данном блоке можно настроить время действия id-токена, токена доступа к данным, также можно настроить время, в течении которого можно обновить токен.

- Настройка куки — секретный ключ, время действия.

5.1.1. Блок настройки доступа к данным

В данном блоке, можно настроить т.н. клеймы. Клеймы — это те данные, к которым хотело бы иметь доступ клиентское приложение. Клиентское приложение при авторизации передает желаемые клеймы на сервер авторизации, далее сервер авторизации смотрит, можно ли выдавать данном клиентскому приложению данные клеймы. По этим клеймам, в дальнейшем, клиентское приложение может получать определенную информацию.

Например можно создать клейм, для определенного API. В данном клейме пишется вся информация, которая необходима для работы данного API с пользователем. Например, для работы с data API и с bekhterev API необходимы роли пользователя и id пользователя:

Листинг 6: Клеймы

```
1 claims: {
2         ''bekhterev-api'': [''sub'', ''roles''],
3         ''data-api'': [''sub'', ''roles''],
4 }
```

Соответственно, если клиентское приложение хочет использовать данный API, оно должно будет передать клейм "data-api" при авторизации. Если данному приложению разрешено использовать данный клейм, то пользователь сможет авторизоваться. В листинге выше кроме "roles", разрешен еще доступ к "sub" — это id пользователя.

5.2. Адаптер базы данных

В данном адаптере реализовано взаимодействие oidc-provider с базой данных MongoDB. Необходимо что бы адаптер имел следующие функции:

- Создание или изменение экземпляра модели oidc-provider.
- Поиск экземпляра модели oidc-provider по id.

- Возможность пометить некоторую модель oidc-provider, как использованную.
- Удалением модели по id.
- Функция подключения к БД.

5.3. Список клиентских приложений

Данный список должен содержать те клиентские приложения, которым разрешена авторизация через данную систему. Каждый объект списка должен содержать следующую информацию:

- Уникальный идентификатор приложения
- Секретный ключ приложения
- Ссылки на страницы, на которые разрешается перенаправлять пользователя после авторизации. Если приложение запросит авторизацию с использованием другой ссылки перенаправления, в авторизации будет отказано.
- Разрешенные типы ответа. Можно указать что в ответе будет содержаться id-токен и токен доступа к данным пользователя, а можно указать что данному приложению разрешается передавать только id-токен.
- Страницу, на которую будет происходить перенаправление, после выхода пользователя из системы

5.4. Ключи шифрования

В OIDC можно использовать разные алгоритмы шифрования. Данные алгоритмы в данной системе используются для создания подписей токенов, для их дальнейшей верификации. В данном случае используется асимметричный алгоритм RS256 с размером ключа 2048. Данный

алгоритм использует алгоритм шифрования RSA[14] с алгоритмом хеширования SHA256[15]. Данный алгоритм использует приватный ключ для создания подписи, и публичный ключ, для того что бы сторонние приложения могли верифицировать полученные пользовательские токены по подписи.

Для создания ключей RS256 использовался пакет @panva/jose[23].

6. DataAPI

6.1. API для клиентских приложений

Data API нацелен предоставлять и взаимодействовать с информацией о хранимых файлах на узлах кластера. С помощью API для клиентских приложений можно выполнять следующие функции:

- Создавать, просматривать, изменять и удалять информацию о узлах кластера.
- Просматривать, изменять, удалять информацию о файлах на узле кластер.
- Создавать и просматривать токены загрузки.

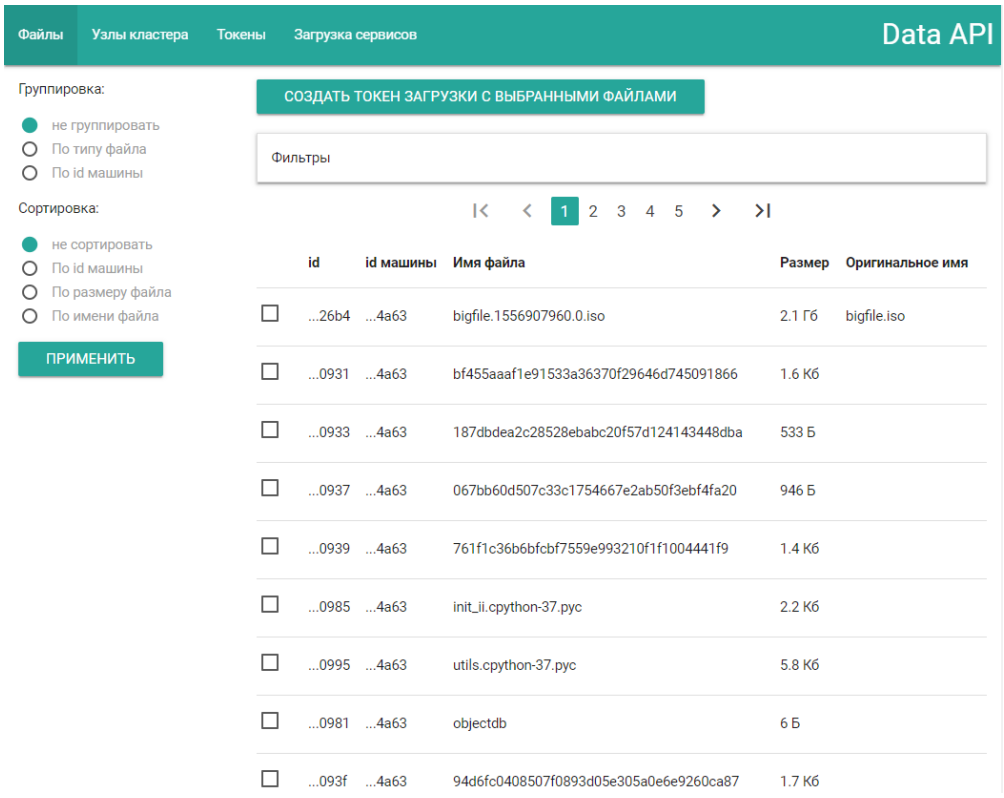

Рис. 3: Представление списка файлов на клиенте Data API

Каждый API из данного списка, имеет разный уровень доступа. Так например просматривать информацию о узлах кластера могут все авторизованные пользователи, в то время как удалять информацию может

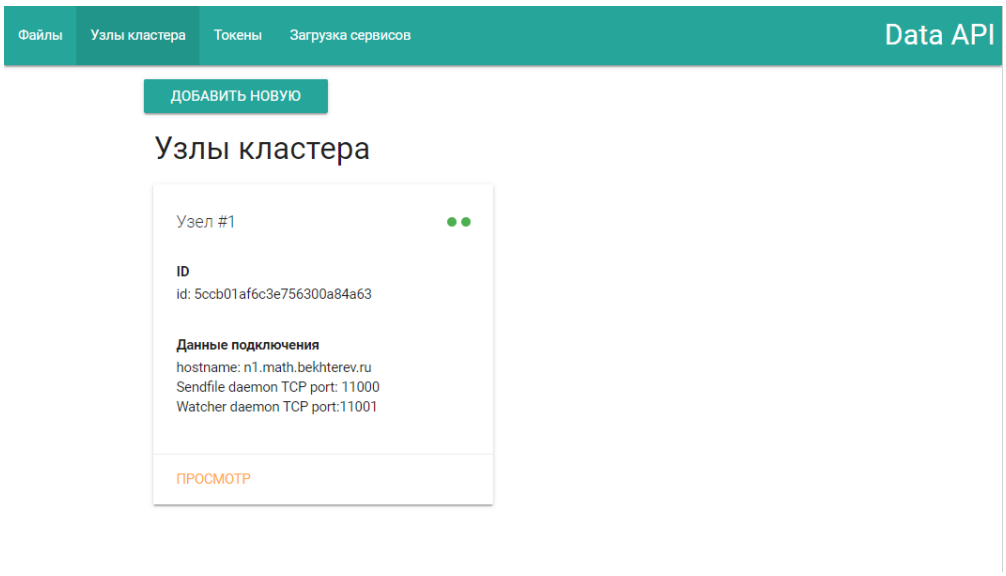

Рис. 4: Представление списка узлов кластера на клиенте Data API

только администратор системы или пользователь с ролью ”node:remove”. Для обеспечения разного уровня доступа на сервере используется т.н. предобработчики или промежуточный слой обработки запроса.

Пример использования промежуточного слоя обработки запроса:

Листинг 7: Пример использования предобработчика

```
1 router.route(''/machine/secret'')
2     .get(
3         check(),
4         check(['admin'', ''machine:secret''],
5         async (req, res) => {
6                 // ...
7                 }
8     )
```

router.route — функция фреймворка express.js, в аргумент записывается адрес, на котором будет расположен данный API. В данном случае показан обработчик для адреса /machine/secret.

get — после route(””) метода по цепочке записываются методы HTTP которые необходимо обрабатывать на данном адресе. В данном случае показан обработчик для метода GET. В аргуметах методов HTTP записываются через запятую обработчики запроса. В данном случае это check без аргументов, check с аргументом в виде

массива и последний аргумент метода GET — это лямбда функция. Первые два обработчика являются промежуточными обработчиками, последний — конечным.

Обработчик — это функция, которая принимает аргументы req, res и next (опционально). Внутри тела функции ожидается, либо отправка ответа на запрос, либо переход к следующему обработчику. Конечный обработчик по своей структуре ничем не отличается от промежуточного. Предполагается, что в конечном обработчике точно будет послан ответ на запрос или будет послана ошибка, если до данного обработчика дойдет очередь.

req — объект запроса, в данном объекте хранятся все данные о запросе. В том числе откуда и от кого он пришел, информация присланная с запросом, header запроса и т.д. Данный объект передается во все обработчики одного запроса. Т.е. можно, например, в данный объект записать в каком-нибудь промежуточном обработчике какую-нибудь информацию и в дальнейшем в последующих обработчиках использовать её.

res — объект ответа на запрос. Данный объект содержит методы для отправления ответа обратно на клиент. Также с помощью данного объекта можно менять статус-код ответа.

next — функция перехода к следующему обработчику. Можно не использовать, если переход не предполагается. Если функция вызывается без аргументов — то переход считается успешным и начинает выполняться следующий обработчик. В данный метод можно передать объект ошибки, если такой имеется, тогда переход будет считаться провальным и на клиент будет возвращена данная ошибка со статус кодом 500 (внутренняя ошибка сервера) и, соответственно, цепочка обработчиков прервется.

check — функция, которая возвращает функцию промежуточной обработки, данная функция будет валидировать либо авторизацию

пользователя, либо его роли, в зависимости от аргументов которые были переданы в функцию check. Подробнее данная функция будет рассмотрена в следующем разделе.

Последняя лямбда-функция является конечным обработчиком. В ней нет аргумента next, т.к. переход к следующему обработчику не предполагается, потому что следующего обработчика не существует.

6.1.1. Проверка авторизованности пользователя

Авторизованность пользователя который сделал запрос, проверяется через OIDC-сервер. Проверяется с помощью токена доступа, который пользователь должен отправить вместе с запросом. В данном токене должна содержаться валидная подпись и также должна содержаться информация, что у пользователя есть доступ к данному API.

В определениях OIDC такие сервисы как Data API, т.е. сервисы которые только предоставляют ресурсы клиентским приложениям, а сами не занимаются авторизацией пользователей, носят название resource server. Для обеспечения работы в эко-системе OIDC в data-api используется пакет для node.js @solid/oidc-rs[24]. Данный пакет позволяет валидировать токены доступа в приходящих запросах.

Листинг 8: Предобработчик авторизации

```
1  (req, res, next) => {
2    rs.authenticate({
3      scopes: [''data-api''],
4      handleErrors: false,
5    })(req, res, async (err) => {
6      if (!err || !req.auth) {
7        req.user = { _id: req.claims.sub, roles };
8        res.status(200);
9        next();
10     } else {
11       next(err);
12     }
13   });
```

```
14 };
```

В листинге выше представлена лямба-функция. Данную функцию возвращает метод check(), если в него не были переданы аргументы. rs.authenticate — это функция, которая предоставляется пакетом @solid/oidc-rs, данную функции можно сразу использовать как обработчик, но в данном случае вызов данной функции спрятан в другой обработчик. Это сделано для того, что бы подшить в объект req объект пользователя, который делал запрос. В данном объекте находится id пользователя и его список ролей в системе.

6.1.2. Проверка роли пользователя

Проверка роли пользователя может быть осуществлена, только если в запросе присутствует объект пользователя. Так что если нужно проверить роль пользователя в каком-либо запросе, перед проверкой роли необходимо поместить обработчик, который проверяет авторизованность пользователя, что бы тот подшил в запрос объект пользователя.

Листинг 9: Предобработчик проверки роли

```
1  return (req, res, next) => {
2    if (Array.isArray(roles)) {
3      for (const nr of roles) {
4        for (const r of req.user.roles) {
5          if (nr === r) {
6            next();
7            return;
8          }
9        }
10     }
11     res.sendStatus(403);
12     return;
13   }
14   // Проверкароли , если roles — этонемассив
15 };
```

Данную функцию-обработчик возвращается метод check(), если в

него был передан аргумент roles. Если данный аргумент является массивом, то запрос успешно пройдет данный обработчик, если хотя бы одна из ролей будет найдена у пользователя. В противном случае, пользователю будет возвращена ошибка с HTTP кодом 403 (ошибка доступа).

6.2. API для узлов кластера

Для валидации запросов к сервисам узла кластера, в Data API было реализовано API, которые обслуживает запросы только от узлов кластера. К таким запросом относятся:

- Запрос на валидации токена загрузки
- Запрос на сохранения времени начала загрузки файла
- Запрос на сохранения времени окончания загрузки файла
- Запрос на сохранения информации о сверке файла

6.2.1. Предобработка запроса от узла кластера

Обработчики данных запросов, отличаются от остальных обработчиков в Data API предобработкой. Данному API не нужно проверять наличия пользовательского токена доступа от OIDC системы, потому что запрос осуществляет сервис на узле кластера. Вместо этого, узел кластера в запросе должен передать свой ID, подпись к запросу созданную с помощью секретного ключа узла и токен загрузки, если этого требует запрос (для запроса о сверке файла, данный токен не требуется). Предобработчик должен по переданным данным проверить:

- Существование такого узла в базе с помощью переданного ID узла
- Проверить присланную подпись по серкретному ключу узла
- И, если требуется для запроса, проверить валидность присланного токена загрузки и проверить валидность доступа с помощью данного токена к данному узлу.

Только после прохождения данных пунктов предобработки, начинается обработка самих запросов.

6.2.2. Подпись с помощью секретного ключа узла

Сервисы узлов обращаются напрямую к Data API. Т.к. не предполагается что что-то, кроме Data API, должно иметь к ним доступ, то добавлять их в систему единой авторизации всех компонентов нет смысла. Не предполагается, потому что запросы к сервисам Data API делает только сам Data API, а не пользователи какого-либо компонента. Тогда, если сервисы не используют OIDC, возникает вопрос доверия к запросам, которые идут с сервисов на Data API.

Для решения задачи валидации запросов, было принято решение использовать верификационную подпись, которая создается средствами симметричного шифрования. Для осуществления данной логики верификации, сервисы и Data API должны иметь и знать серкретный ключ, который хранится только на Data API и только внутри сервисов и не передается в запросах.

Алгоритм создания подписи для верификации сервисов:

1. Создается json объект, который включает в себя такие поля как: дата и время создания данного объекта; время действия данного объекта; id узла кластера с которого будет осуществляться запрос.

2. Далее данный объект переводится в строчную форму.

3. Далее к данному объекту применяется симметричный алгоритм шифрования aes-256-cbc[1], с использованием уникального секретного ключа, который знает только Data API и узел кластера.

4. Далее полученный набор байтов переводится в строчную форму с помощью кодировки base64[3].

5. Полученная строка передается вместе с запросом в качестве подписи.

Подтверждение валидности подписи осуществляется по такому же алгоритму, только в обратном порядке.

6.3. API для клиентского приложения Sendfile

Данное API необходимо для получения информации о узле кластера по загрузочному токену. Sendfile клиент передает на Data API загрузочный токен и, если он валидный, отправляет ответ с информацией о узле кластера.

6.4. Передача файлов

Общий алгоритм передачи файлов:

1. Пользователь создает токен, для загрузки файлов на определенный узел кластера. Данное действие пользователь может сделать в клиентском приложении Data API.

2. Далее пользователь запускает клиентское приложение Sendfile и передает ему в качестве аргументов токен полученный на 1 шаге, и путь к файлу или к папке с файлами для загрузки на узел.

3. Первым делом Sendfile-клиент обращается к Data API за информацией о узле, на который будет происходить загрузка: хост адрес узла и порт Sendfile сервиса.

4. Далее начинается процесс пересылки файла:

   (a) Первым делом отсылается токен загрузки.

   (b) Узел кластера с помощью Data API проверяет валидность токена и то что данный токен предназначен для загрузки именно на этот узел. Без данной валидации, Sendfile-сервис не начнет прием остальных данных.

   (c) Далее клиент отсылает метаинформацию о файле: имя файла, размер файла и контрольную сумму файла. На данном

этапе Sendfile-сервис также не приступит к приему остальных данных, пока не будет получена вся метаифномрация о файле

(d) После приема метаинформации, Sendfile-клиент начинает передавать сам файл. Файл передается блоками, в каждом блоке содержится информации о его размере и о его смещении в файле и непосредственно сам блок данных.

5. 4 шаг повторяется для всех файлов в папке.

6.4.1. Протокол передачи файла

Данный протокол используется при передачи файлов через TCP соединение. Основная идея протокола — передача информация блоками. Каждая блок должен иметь: id блока, для того что бы один блок можно было отличать от другого; размер тела блока; само тело блока, информация в теле блока зависит от id блока.

id-часть в блоке занимает 1 байт, размер блока занимает 4 байта и, соответственно, тело блока занимает определенную величину в байтах.

Sendfile-сервис работает следующим образом:

1. Сначала принимает 5 байтов от клиента, в данных 5 байтах должен содержаться id блока и длинна блока.

2. Если данные 5 байтов валидны, т.е. сервис смог прочитать и id блока и его размер, то далее сервис принимает непосредственно передаваемые данные.

6.5. Перепроверка файлов

Перепроверка файлов нужна для обнаружения файлов, которые попали на узел не через Sendfile сервис, а также для обнаружения изменений в файлах. Data API инициирует перепроверку всех файлов на узле по расписанию раз в день. Также инициировать перепроверку файла может пользователь, который обладает определенными правами.

Алгоритм перепроверки файла:

1. DataAPI отправляет запрос на Watcher-сервис узла. Запрос либо на перепроверку определенного файла, либо на перепроверку всех файлов узла.
2. Watcher-сервис составляет список файлов для перепроверки
3. Watcher-сервис определяет основную информацию по каждому файлу
4. Отправляет по каждому файлу полученную информацию на Data API
5. Data API определяет к какому файлу в БД относится присланная информация

6.5.1. Data API

На стороне Data API реализованы два класса:

RecheckScheduler — класс расписания перепроверки файлов. Данный класс инициализируется один раз при запуске сервера в единственном экземпляре.

WatcherNode — класс сервиса Watcher на узле. Данный класс создается для каждого узла кластера. Внутри данного класса реализованы методы, которые позволяют отправить команду перепроверки файлов на узел.

RecheckScheduler. В конструктор данному классу передается список узлов кластера. В самом конструкторе создаются:

- Список nodes — это список WatcherNode объектов, в каждый из которых передается информация о определенном узле.
- График перепроверки файлов. Данный график вызывает метод RecheckAll(), который в свою очередь отправляет команды на узлы средствами списка nodes.

- График переинициализации списка узлов кластера. Данный график вызывается за 2 минуты до перепроверки файлов.

Листинг 10: График переинициализации узлов кластера

```
1 this.reinitSchedule = schedule.scheduleJob('0 0 * * * *', () => {
2   console.log(''reinit sch'');
3   this.Reinit();
4 })
```

Для создания расписаний и графиков используется пакет node-schedule[21]. Время вызова функции устанавливается в cron[19]-стиле. В данном случае установлено '0 0 2 * * *' — это означает, что метод перепроверки файлов будет вызываться тогда, когда на часах будет 0 секунд 0 минут и 2 часа, т.е. каждый день в 02:00:00.

WatcherNode. Данный класс предназначен для отправки команды на перепроверку всех файлов на определенный узел кластера. В конструктор данному классу передается информация о узле кластера. Для отправки команды на узел реализован метод RecheckCommand():

Листинг 11: Комманда перепроверки файлов

```
1  RecheckCommand() {
2    if (this.clientWatcher) {
3      this.clientWatcher.destroy();
4      delete this.clientWatcher;
5    }
6    this.clientWatcher = new net.Socket();
7    this.clientWatcher.connect(this.machine.machine.watcher.tcp, this.machine.
       ↪ machine.host, () => {
8      this.clientWatcher.write(new Buffer([0x2]));
9    });
10
11   this.clientWatcher.on('data', (data) => {
12     if (this.clientWatcher) {
13       this.clientWatcher.destroy();
14       delete this.clientWatcher;
15     }
16   });
```

```
17
18    this.clientWatcher.on('close', () => {
19      //...
20    });
21
22    this.clientWatcher.on('error', () => {
23      //...
24    });
25  }
```

Для общения с TCP сервером в node.js используется библиотека net[9]. В метод connect передается информация о подключении. Далее необходимо установить обработчики некоторых событий, а именно: data — событие получения данных, close — события закрытия соединения, error — событие ошибки. Т.к. данная команда перпроверки файлов не предполагает приема информации (вся информация о перепроверках идет прямиком в Data API).

6.5.2. Watcher-сервис

Как говорилось выше watcher-сервис имеет tcp-сервер который принимает команды. Для перепроверки всех файлов на watcher-сервисе реализовано два класса:

RecheckAll — класс который инициализирует перепроверку файлов в заданных папках.

FolderScanner — класс, который выдает в виде списка все файлы в заданной папке.

RecheckAll. В данном класе реализовано два метода: Start — метод инициализации перепроверки файлов и метод Process — данный метод запускает перепроверку конкретного файла.

Листинг 12: Метод start класса RecheckAll

```
1  public void Start() {
2      lock (lockObject) {
3          try {
```

```
4               Parallel.ForEach(FolderScaner.Scan(Folders),
5                    new ParallelOptions { MaxDegreeOfParallelism = 4 },
6                           x => Process(x)
7               );
8           } catch (Exception e) { }
9       }
10  }
```

Перепроверка списка файлов осуществляется параллельно, с ограничением на одновременное количество процессов — 4.

FolderScanner. Данный класс предназначен для сканирования файлов.

Листинг 13: Метод scan класса FolderScanner

```
1  public static IEnumerable<BFileInfo> Scan(string folder) {
2      string[] directoryArray = Directory.GetDirectories(folder);
3      string[] fileArray = Directory.GetFiles(folder);
4      foreach (var file in fileArray) {
5          yield return GetFileInfo(file);
6      }
7      foreach (var dir in directoryArray) {
8          foreach (var f in Scan(dir)) {
9              yield return f;
10         }
11     }
12 }
```

Главный метод Scan — рекурсивно сканирует заданные папки и выдает информацию о найденных файлах в виде списка. Притом выдача информации осуществляется ленивым образом, т.е. пока не произойдет вызов конкретного элемента списка, данный элемент не будет инициализорован.

6.6. Проверка доступности сервисов на вычислительном узле

Данная функция нужна для поддержания информации о доступности сервисов на вычислительном узле. Алгоритм проверки достаточно прост:

1. При запуске сервера Data API на нем создается график проверки доступности

2. В графике заложена информация о узлах кластера и о сервисах на данных узлах

3. Далее по расписанию Data API делает маленький запрос на каждый из сервисов, ожидая при этом получить в ответ определенный байт

4. Если байт был получен и он такой, который ожидается, то сервис помечается как доступный, иначе как не доступный

Данная информация, о доступности серверов, подшивается в некоторые запросы в Data API. Например при запросе всех узлов кластера, в ответ клиент получит не только информацию о узлах, а также их доступность.

Реализована данная функция аналогичным образом, как и функция перепроверки файлов. Т.е. на Data API заводится класс расписания проверки доступности и класс непосредственной проверки доступности.

Класс расписания проверки доступности — данный класс инициализируется при запуске сервера Data API. В нем инциализируется график проверки доступности и график, который запускает инициализацию списка узлов кластера.

Класс проверки доступности — данный класс проверяет доступность сервисов на определнном узле. Он отправляет короткую команду на каждый из сервисов и ожидает в ответ определенный набор байтов. Если ответ от сервиса пришел корректный, данный сервис помечается как доступный, если некорректный или ответ вообще не пришел или не получилось присоединиться к сервису, то сервис помечается как недоступный.

В классе расписания проверки доступности, также реализована функция подшивания информации о доступности сервиса:

Листинг 14: Информация о доступности

```
1  Addping(node) {
2    let lm = this.nodes.find((x) => x.node._id.toString() == node._id.toString()
       ↪ );
3    if (!lm) {
4      node.ping = {
5        sendfile: false,
6        watcher: false
7      }
8    }
9    else {
10     node.ping = lm.ping;
11   }
12 }
```

В данную функция передается объект узла, в нем содержится основная информация о узле. Далее проверяется есть ли такой узел в БД, если есть, то проверяется есть ли информация о таком узле в данном классе, если информация о доступности есть, то она добавляется в данный объект, если нет, то добавляется объект по умолчанию, который говорит что сервисы недоступны.

6.7. Обнаружение и устранение проблем у файлов

Под проблемой у файла подразумевается ситуация, когда у файла после перепроверки обнаружились изменения в контрольной сумме, размере, а также если файл поменял своё местоположение.

Данные проблемы обнаруживаются после перепроверок файлов, каждая информация о перепроверке записывается в БД к тем файлам, на которых присланная информация больше всего похожа. Если существует файл, такой что перепроверка полностью совпадает с уже известной информацией о файле, то перепроверка записывается только к этому файлу.

Обнаружением проблем занимается клиентская часть, а Data API предоставляет API, который позволяет устранить данную проблему.

Листинг 15: Функция обноружения проблем файлов

```
1  SomeProblemHere(file: BigFile): number {
```

```
2     let problem = 0;
3     if (!file.upload)
4       problem = (problem | FileProblem.NoUploadEnd | FileProblem.NoUploadStart |
          ↪  FileProblem.NpUploadTokenId);
5     if (file.upload && !file.upload.end)
6       problem |= FileProblem.NoUploadEnd;
7     if (file.upload && !file.upload.start)
8       problem |= FileProblem.NoUploadStart;
9     if (file.upload && !file.upload.tokenId)
10      problem |= FileProblem.NpUploadTokenId;
11    if (file.recheck.length > 0 && file.recheck.slice(-1)[0].checksum !== file.
        ↪ file.checksum)
12      problem |= FileProblem.DiffChecksum;
13    if (file.recheck.length > 0 && file.recheck.slice(-1)[0].size !== file.file.
        ↪ size)
14      problem |= FileProblem.DiffSize;
15    if (file.recheck.length > 0 && file.recheck.slice(-1)[0].path !== file.file.
        ↪ path)
16      problem |= FileProblem.DiffPath;
17    if (!file.file.realname)
18      problem |= FileProblem.NoRealName;
19    return problem;
20  }
```

В листинге выше представлен метод, который обнаруживает проблемы. Каждая проблема, помечается своим флагом в поле problems у файла. Некоторые проблемы, которые могут быть обнаружены:

- У файла нет объекта upload или некоторых полей внутри upload — т.е. скорее всего файл попал на узел системы в обход сервиса Sendfile.
- У файла не совпадает исходный размер с размером, который был получен в последней перепроверке.
- У файла не совпадает контрольная сумма с последней перепроверкой.
- У файла не совпадает путь с последней перепроверкой.
- У файла не установлено оригинальное имя.

При обнаружении проблем у файла на клиенте напротив данного файла будет высвечиваться сообщение, соответствующее проблеме. Далее клиентское приложение, может воспользоваться функцией обновления объекта файла в БД. Например если некто решил загрузить на узлы кластера данные в обход Sendfile, то необходимо будет вручную, т.е. через клиентское приложение, добавить к каждому файлу объект upload. Или если изменение файла было предвиденным и планировалось, тогда изменившаяся контрольная сумма — это нормально, тогда можно вручную поменять контрольную сумму на новую.

Также через клиентское приложение можно инициировать перепроверку конкретного файла, что бы убедиться что данные последней перепроверки верны.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- Разработан Data API
- Разработано клиентское приложение Data API
- Разработана структура БД
- Разработана система загрузки и выгрузки данных в виде клиентского приложения sendfile и сервисов узлов кластера: sendfile-сервис и watcher-сервис.
- Реализована единая система авторизации для компонентов проекта
- Опубликована статья V. Korkhov, V. Volosnikov, A. Vorontsov, K. Gribkov, N. Zalutskaya, A. Degtyarev, A. Bogdanov. Data storage, processing and analysis system to support brain research // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2018, vol. 10963, pp. 78–90, ISBN: 978-331962403-7.

Список литературы

[1] Advanced Encryption Standard - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard (online; accessed: 20.05.2019).

[2] Angular. — 2019. — URL: https://angular.io/ (online; accessed: 19.05.2019).

[3] Base64 - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/Base64 (online; accessed: 20.05.2019).

[4] Blowfish (cipher) - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/Blowfish_(cipher) (online; accessed: 20.05.2019).

[5] CAS - Home. — 2019. — URL: https://apereo.github.io/cas/6.0.x/index.html (online; accessed: 20.05.2019).

[6] Express - Node.js web application framework. — 2019. — URL: http://expressjs.com/ (online; accessed: 19.05.2019).

[7] MongoDB for GIANT Ideas | MongoDB. — 2019. — URL: https://www.mongodb.org/ (online; accessed: 19.05.2019).

[8] Mongoose ODM v4.10.2. — 2019. — URL: http://mongoosejs.com/ (online; accessed: 19.05.2019).

[9] Net | Node.js v12.2.0 Documentation. — 2019. — URL: https://nodejs.org/api/net.html (online; accessed: 20.05.2019).

[10] Node.js. — 2019. — URL: https://nodejs.org/en/ (online; accessed: 19.05.2019).

[11] Npm. — 2019. — URL: https://www.npmjs.com/ (online; accessed: 19.05.2019).

[12] OAuth 2.0 — OAuth. — 2019. — URL: https://oauth.net/2/ (online; accessed: 20.05.2019).

[13] OpenID Foundation website. — 2019. — URL: https://openid.net (online; accessed: 19.05.2019).

[14] RSA (cryptosystem) - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/RSA_(cryptosystem) (online; accessed: 20.05.2019).

[15] SHA-2 - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/SHA-2 (online; accessed: 20.05.2019).

[16] V. Korkhov V. Volosnikov A. Vorontsov K. Gribkov N. Zalutskaya A. Degtyarev A. Bogdanov. Data storage, processing and analysis system to support brain research // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — 2018. — Vol. 10963. — P. 78–90. — ISBN: 978-331962403-7.

[17] bcrypt - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/Bcrypt (online; accessed: 20.05.2019).

[18] ckan – The open source data portal software. — 2019. — URL: https://ckan.org/ (online; accessed: 20.05.2019).

[19] cron - Wikipedia. — 2019. — URL: https://en.wikipedia.org/wiki/Cron (online; accessed: 20.05.2019).

[20] dotnet/core: Home repository for .NET Core. — 2019. — URL: https://github.com/dotnet/core (online; accessed: 20.05.2019).

[21] node-schedule - npm. — 2019. — URL: https://www.npmjs.com/package/node-schedule (online; accessed: 20.05.2019).

[22] oidc-provider - npm. — 2019. — URL: https://www.npmjs.com/package/oidc-provider (online; accessed: 20.05.2019).

[23] @panva/jose - npm. — 2019. — URL: https://www.npmjs.com/package/@panva/jose (online; accessed: 20.05.2019).

[24] @solid/oidc-rs - npm. — 2019. — URL: https://www.npmjs.com/package/@solid/oidc-rs (online; accessed: 20.05.2019).