Санкт-Петербургский Государственный Университет
Математическое обеспечение и администрирование информационных систем

Технология программирования

Грибков Кирилл Владимирович

Разработка распределенной информационной системы для управления сведениями о результатах интеллектуальной деятельности: серверная часть.

Бакалаврская работа

Научный руководитель:
профессор кафедры информатики, д.ф.-м.н., доц. Тулупьев А. Л.

Рецензент:
с.н.с. лаб. ТиМПИ СПИИРАН, к.ф.-м.н. Суворова А. В.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY
Software and Administration of Information Systems

Technology of Programming

Kirill Gribkov

Development of distributed information system for managing information of intellectual activity results: server side.

Bachelor's Thesis

Scientific supervisor:
Dc. Sc. in Math, Assoc. Prof., Prof. Computer Science Department Alexander Tulupyev

Reviewer:
Ph. D. in Math, Senior Researcher Alena Suvorova

St. Petersburg
2017

Оглавление

Введение

Российские научные и научно-педагогические работники, исследователи и иные творческие деятели — всем им нужно предъявлять сведения о результатах работы в самых разных видах, структурах, срезах для рейтингов, конкурсов и т.д. Структурирование списка при большом количестве интеллектуальных результатов становится сложной задачей. Также, стоит заметить, что для разных ситуаций требуется представить список о результатах интеллектуальной деятельности в разных форматах, таких как *.xml*, *.doc* и другие.

Основная задача рамочного проекта RADOMS [14] заключается в том, чтобы хранить список результатов научных достижений (список публикаций, регистраций, отчетов), и в дальнейшем обрабатывать и выдавать этот список в разных формах, которые требуются для разных рейтингов и конкурсов. Иными словами, должна быть реализована система, в которой имеется клиент, и у которого есть возможность хранить в базе данных список результатов своей работы и в дальнейшем осуществлять необходимую выборку из списка работ и выводить этот список в самых разных видах, представленных в сервисе.

Целью настоящей работы является реализация серверной части в рамках проекта по созданию распределенной информационной системы для управления сведениями о результатах интеллектуальной деятельности. Конкретные задачи для достижения указанной цели: разработать структуру серверной части и структуру БД; написать API для соответствующей структуры; написать импорт публикаций из сервиса elibrary; настроить планировщик задач на создание production-версии сервиса.

1. Обзор предметной области

Существующие решения могут выполнять часть функционала описанного в постановке задачи. Так некоторые имеют возможность хранить списки публикаций, другие имеют экспорт списка публикаций в некоторых форматах, третьи имеют возможность импортировать метаданные публикации из файла в систему. Также некоторые решения имеют такой функционал, как отслеживание цитируемости. 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 их отличие от реализуемой системы.

1.1. Mendeley

Mendeley[7] — кроссплатформенное условно-бесплатное программное обеспечение, также представленное как веб-сервис, позволяющее управлять библиографической информацией. Позволяет хранить и просматривать исследовательские труды. Также поддерживает импорт метаданных из PDF-файла. Имеет в себе элементы социальной сети.

Отличие от сервиса RADOMS:

- Импорт из файлов .xml, .ris, .bib. Экспорт осуществляется только в файл .xml.
- Поддерживает аутентификацию только через электронную почту и пароль. Авторизация через разнообразные социальные сети отсутствует.
- Нет возможности отфильтровать список по критериям, только через поисковую строку.

1.2. Scopus

Библиографическая и реферативная база данных и инструмент для отслеживания цитируемости статей, опубликованных в научных изданиях, но только в изданиях, включенных в данную базу. База данных Scopus[15] предоставляет широкие возможности получения наукометрии. Большая часть функционала платная.

Отличие от сервиса RADOMS:

- Автоматическое добавление публикаций в базу данных без возможности редактирования.
- Нет возможности экспорта списка публикаций.

1.3. Google Scholar

Поисковая система со свободным доступом, которая индексирует тексты и метаданные научной литературы различных форматов. Google Scholar[16] индексирует наиболее рецензируемые онлайновые академические журналы и книги, доклады конференций, тезисы, диссертации и другую научную литературу.

Отличие от сервиса RADOMS:

- Импорт из файла отсутствует. Но ввод публикации вручную присутствует.
- Нет возможности экспорта списка публикаций.

1.4. eLIBRARY.RU

eLIBRARY.RU[3] — научная электронная библиотека, частью функционала которой является корректировка списка своих публикаций и цитирований в РИНЦ, Web of Science и Scopus.

Отличие от сервиса RADOMS:

- Отсутствует возможность аутентификации через социальные сети.

- Экспорт происходит, в одном, единственном виде.
- Отсутствуют разнообразные фильтры для выборки публикаций.
- Невозможно импортировать публикации из файла.

1.5. Выводы

Таким образом не один из представленных сервисов не отвечает всем требованиям одновременно. Поэтому было решено разрабатывать собственный проект, в котором будут учтены все преимущества и недостатки существующих систем.

2. Используемые технологии

2.1. MEAN стек

Данный сервис основан на т.н. MEAN стеке. MEAN — это аббревиатура от четырех программных компонентов: MongoDB [8], Express.js [4], Angular.js [1], Node.js [11]. Все 4 компонента данного стека хорошо взаимодействуют друг с другом, все они связаны с JavaScript, даже в MongoDB запросы пишутся на данном языке.

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. 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.

2.5. npm

npm[12] — 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 онлайновой базой данных общедоступных пакетов, называемой реестром npm. Доступ к реестру осуществляется через клиент, также возможно просматривать и искать доступные пакеты через вебсайт[12] npm.

2.6. Mongoose

Для работы с MongoDB в Node.js используется пакет mongoose [9]. 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. Дабы 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 т.н. триггеры.

2.7. passport

passport[13] — это пакет для авторизации в Node.js. Одно из преимуществ passport перед другими аналогами — это наличие большого количества стратегий авторизации, включая стратегии авторизации через социальные сети. Также passport поддерживает функции сериализации и десериализации, которые при правильной настройке позволяют по запросу определять конкретного пользователя и удобно добавляют объект пользователя из БД в запрос для дальнейшей работы.

2.8. cheerio

Парсинг html страниц для импорта списка публикаций с сервиса elibrary.ru осществляется с помощью пакета cheerio [2]. Данный пакет имеет очень удобный синтаксис, он основан на зарекомендовавшем себя синтаксисе фронтенд фреемворка jQuery [6]. Также cheerio работает с простой, согласованной DOM моделью, в результате чего парсинг, манипуляция и рендеринг HTML кода выполняется быстро.

3. Определения и обозначения

Определение 1 Объект запроса в MongoDB — JSON объект, по которому осуществляется поиск в БД. Такой объект содержит разнообразные критерии для поиска данных. Например можно указать конкретное значение какого-нибудь поля:

Листинг 1: Обычный объект запроса

```
1 {
2         name: 'Kirill',
3         surname: 'Gribkov'
4 }
```

Тогда после применения такого объекта запроса MongoDB выдаст документы из БД с полями которые содержат именно такое конкретное значение. Также можно, например, искать по регулярному выражению или использовать какое-нибудь логическое выражение, для применения таких операций в объекте запроса используются специально зарезервированные названия ключей:

Листинг 2: Объект запроса со специальными ключами

```
1 {
2         $or: [{
3                 link: {$regex: /[^[a-zA-Z]]*/}
4         },{
5                 link: ''someLink''
6         }]
7 }
```

Так для поиска через регулярное выражение используется ключ $\$regex$, а для дизъюнкции ключ $\$or$. Полный список поддерживаемых операций можно посмотреть в документации [10] MongoDB.

Определение 2 Объект обновления в MongoDB — JSON объект, по которому осуществляется редактирование документов в БД. Данный объект содержит информацию о том, какие поля необходимо редактировать и каким образом. Для обозначения специальных способов редактирования используются специальные зарезервированные ключи.

Так, например, для добавления некоторого значения в список, необходимо использовать ключ $\$push$:

Листинг 3: Объект обновления со специальным ключом push

```
{
        $push: {
                users: {
                        id: 1234567890,
                        role: 3
                }
        }
}
```

Полный список всех операций редактирования можно также посмотреть в документации [10] MongoDB.

Определение 3 Роли пользователя в модели группы:

- **-1** — объект пользователя с данной ролью, в списке пользователей группы, является обычным объектом запроса на вступление в группу. Данный пользователь не имеет никаких прав в данной группе.
- **0** — пользователь с данной ролью в группе может просматривать все публикации всех пользователей группы.
- **2** — данная роль в группе является административной. Администраторы могут изменять метаданные группы, также могут принимать заявки на вступление в группу и удалять пользователей из группы(всех, кроме администратора и создателя группы).
- **3** — пользователь с такой ролью в группе является создателем группы. Создатель может все что и администратор, только вдобавок еще может назначать или разжаловать администраторов. Также создатель группы может удалить группу полностью.

4. Организация структуры кода серверной части

Изначально структура проекта предполагала, что все обработчики запросов, которые поступают на сервер, а также создание соединения с БД и пересылка файлов приложения с веб-формами, хранятся в одном файле server.js. При масштабировании приложения и создания новых обработчиков, актуальной задачей представляется разделение файла server.js на несколько, а именно: файл для настройки самого сервера, файл для настройки и подключения разных обработчиков запросов, файл с запуском самого сервера. Такое разбиение позволяет независимо добавлять и изменять разные обработчики запросов, потому что разные обработчики запросов будут содержаться в разных файлах и подключаться исключительно через файл отвечающий за подключение обработчиков запросов. Также, т.к. настройка самого сервера будет содержаться в одном файле и не будет содержать ничего лишнего, то, соответственно, можно будет легко понять как сервер настроен в данный момент и легко добавить или удалить дополнительные модули. После разбиения и рефакторинга серверной части, была получена следующая структура (Рис. 1).

В файле server.js остался только запуск сервера, а все остальное было перенесено в новую папку server. Настройка сервера была перемещена в файл middleware.js. По сути, этот файл связующее звено между server.js и другими частями сервера, ибо непосредственно к этому файлу подключается основная часть сервера. Файл router.js содержит в себе настройки и подключение обработчиков запросов. Сами обработчики находятся в папке api. Папка config отвечает за конфигурацию отдельных частей сервера. А в папке models находятся схемы для базы данных.

В итоге получилась достаточно независимая структура: так, к примеру, добавление новых обработчиков запросов не составит труда, потому что они вынесены в отдельную папку, нужно будет просто добавить файл с обработчиками в папку api и вызвать его в файле router.js. То

Рис. 1: Структура серверной части

же самое касается и конфигурации отдельных частей сервера и новых схем базы данных.

5. Использование базы данных MongoDB

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

Данные в MongoDB хранятся в виде json-подобных документах, по этому для описания модели данных используется т.н. схема, представленная в виде JSON объекта. По сути в виде такой схемы все данные и будут храниться. Для того чтобы обозначить что некоторый ключ в данной схеме должен соответствовать некоторым настройкам, т.е. например быть строковым типом или обязательным к заполнению, значением этого ключа должен выступать т.н. объект-настройки. Под таким объектом подразумевается объект с определенными ключами и значениями, так, например, для обозначения строкового типа, такой объект должен содержать ключ *type* со значением *String*. Если никакие настройки для ключа в схеме не нужны, то значение такого ключа должно содержать пустой объект.

5.1. Схема профиля пользователя

Основная задача модели профиля пользователя — это хранение всех основных данных о пользователе: email, пароль в зашифрованном виде, имя, фамилия и прочее. В пакете mongoose для создания схемы используется соответствующий конструктор Scheme:

Листинг 4: Схема модели профиля пользователя

```
1  var Profile = new Schema({
2     local: {
3        email: {type: String, required: true},
4        username: {type: String},
5        password: {type: String, required: true},
6        verify: {type: Boolean, default: false},
7        token: {type: String}
8     },
9     name: {type: String, required: true},
10    surname: {type: String, required: true},
11    //...
```

```
12    role: {type: Number, default: 0},
13    registrationDate: {type: Date, default: Date.now},
14    commit: {type: String, default: ''}
15 });
```

Как видно из схемы, большинство полей имеют строковый тип. Чтобы обозначить, что поле должно быть обязательно заполнено, нужно установить настройку required в положение true. Для обозначения того, чтобы поле было уникальным используется настройка unique. Настройка default — это значение поля по умолчанию, в случае registrationDate — это функция Date.now, которая возвращает текущую дату.

5.2. Схема группы

Модель группы содержит все данные о группе: название, описание, уникальная ссылка и другие, также в модели находится список пользователей группы с ролями в ней:

Листинг 5: Схема модели группы

```
1 var Group = new Schema({
2    title: {type: String, unique: true, required: true},
3    description: {type: String, required: true},
4    link: {type: String, unique: true, required: true},
5    users: [{
6       id: {type: ObjectId},
7       role: {type: Number, default: -1, min: -1, max: 4}
8    }]
9 });
```

Как можно видеть из листинга выше, отличительной особенностью данной схемы является массив данных в качестве значения поля users. Т.е. данное поле в объекте группы является списком, в него можно добавлять, удалять объекты и взаимодействовать с ними.

Метод валидации ссылки

Каждая группа должна иметь свою уникальную ссылку, такая ссылка должна отвечать определенным требованиям, а точнее она должна

быть написана на латинице без пробелов и других посторонних символов. Для валидации такой ссылки на серверной части используется метод, который вызывается каждый раз, когда используется метод save, который сохраняет данные в БД, у модели group.

Листинг 6: Валидация ссылки модели группы

```
1 Group.pre('save', function (next) {
2    if (/[^[a-zA-Z]]*/.test(this.link)) {
3       var error = new ValidationError(this);
4       error.errors.link = new ValidatorError('link', 'Link is not valid', this
           ↪ .email);
5       return next(error);
6    }
7    next();
8 });
```

Проверка валидности ссылки осуществляется с помощью регулярного выражения (2 строка листинга 6). Данное регулярное выражение ищет в строке символы не из латинского алфавита. Соответственно если такие символы будут найдены, сохранение обрывается и вызывается ошибка. Если посторонних символов в ссылке найдено не было, происходит обычное сохранение нового объекта в БД.

6. Разработка API для групп

Одной из поставленных задач проекта является реализация формирования групповых списков результатов интеллектуальной деятельности. Один из примеров такого списка — годовой отчет некоторой лаборатории. Решить данную задачу можно без создания дополнительных структур. Например, в случае лаборатории, можно было создать новый обычный профиль пользователя в сервисе и добавить туда публикации всех сотрудников. Но такой подход является очень неудобным, такому профилю необходимо постоянная модерация, если у какого-либо сотрудника появилась новая публикация, её необходимо добавить не только в личный профиль, но и также в профиль лаборатории. Поэтому для решения данной задачи было принято создать новую модель в сервисе — модель группы. Данная модель позволяет объединить несколько пользователей в одну структуру и взаимодействовать со всеми публикациями всех пользователей в этой структуре. Такой подход позволяет удобно управлять групповыми списками публикаций. Если пользователь добавляет некоторую публикацию в свой личный профиль, данная публикация автоматически будет отображаться и в списке публикаций группы.

При постановки задачи, было описано, что должно делать API для групп:

1. Создание, удаление, изменение группы
2. Добавление и удаление пользователей из группы, а также изменение привилегий пользователей внутри группы
3. Возможность отправить запрос на вступление в группу посредством приглашения
4. Выдача списка публикаций всех участников группы

В процессе разработки API, была придумана следующая структура API для групп:

- API *groups*:
  - *POST* — создание группы. При создании группы, пользователь который создал группу автоматически становится её администратором.
  - *PUT* — обновление информации о группе. Т.е. обновление названия, описания группы и ссылки на группу. Обновлять группу может только администратор.
  - *DELETE* — удаление группы. Удалить группу может только создатель группы.
- API *groupUsers*:
  - *GET* — получение списка пользователей группы.
  - *POST* — добавление пользователя в группу. По сути этот обработчик высылает приглашение в группу.
  - *PUT* — изменение роли пользователя в группе.
  - *DELETE* — удаление пользователя из группы.
- API *groupsPublications*:
  - *GET* — выдача публикаций всех пользователей в группе.

Рассмотрим наиболее важные методы подробно.

6.1. API groups

6.1.1. POST

Создание группы происходит в несколько этапов:

1. Клиент передает на сервер название, описание и уникальную ссылку группы (см. пример в листинге 9).

2. Сервер проверяет полученные данные и создает новый объект *Group* с полученными данными. При ошибке, сохранение прерывается и на клиентскую часть отправляется соответствующее сообщение.

3. Далее объект *Group* сохраняется в базе данных.

Листинг 7: Пример запроса на создание группы

```
1 {
2         name: ''Title'',
3         description: ''Some description'',
4         link: ''someLink''
5 }
```

Листинг 8: Создание группы

```
1 var group = new Group({
2    title: req.body.title,
3    description: req.body.description,
4    link: req.body.link,
5    users: [{id: req.user._id, role: 4, commit: req.user.commit}]
6 });
7 group.save(function (err) {
8    // ...
9 })
```

Рассмотрим листинг 8 подробно. Весь процесс происходит в два шага: создание экземпляра группы и его сохранение. Поле *users* заполнено списком с одним единственным объектом пользователя, этому объекту присваивается *id* пользователя который создает группу и высшая роль в группе. Проверка корректности имени и описания группы производится встроенными средствами mongoose: на указанные поля налагаются правила целостности, указанные в схеме 5, в частности, проверяется, что данные поля являются строками и не пусты. Проверка корректности поля *link* осуществляется посредствам метода, который был написан при создании модели группы 6. Этот метод вызывается всякий раз, когда вызывается метод *save* у объекта группы.

6.1.2. PUT

Создатель группы должен иметь возможность изменять основные данные о группе. Изменение группы осуществляется следующим образом:

1. Клиент передает на сервер *id* группы для изменения и новые название, описание и ссылку (см. пример в листинге 9).

2. Сервер проверяет полученные данные, в случае некорректных данных редактирование прерывается, и соответствующее сообщение с ошибкой отправляется на клиент.

3. Сервер обновляет нужную группу.

Листинг 9: Пример запроса на изменение группы

```
{
        id: ''58c5aa083a53dd0f047fb86c'',
        name: ''New title'',
        description: ''New description'',
        link: ''newLink''
}
```

Обновление БД осуществляется с помощью метода *update* библиотеки *mongoose*:

Листинг 10: Изменение группы

```
Group.update({
   _id: req.body.groupId,
   users: {
      $elemMatch: {
         id: req.user._id,
         role: {$gt: 2}
      }
   }
}, {
   link: req.body.link,
   description: req.body.description,
   title: req.body.title
}, function (err, result) {
   if (err) {/*...*/}
   else if (result.n != 0) {/*...*/}
```

```
16     else {/*...*/}
17 });
```

Метод принимает три аргумента: объект запроса(Опр. 1), объект обновления(Опр. 2) и функцию обратного вызова.

В данном случае необходимо найти группу с определенным id. Также нужно убедится что пользователь, который отправил запрос на обновление, является администратором данной группы. Для этого в объект запроса необходимо добавить условие, что в списке пользователей присутствует пользователь с id равным id пользователя текущей сессии и также чтобы у такого пользователя роль в группе была административной, т.е. большей или равной 2.

- Для поиска по id группы, достаточно в объекте запроса указать поле $\_id$ с нужным значением(2 строка листинга 10).
- Для поиска определенного объекта в списке используется специальный ключ $\$elemMatch$, который в объекте поиска должен являтся прямым подключом нужного списка. В данном случае поиск производится в списке $users$, поэтому $\$elemMatch$ является непосредственным подключом данного списка в объекте запроса. Значением ключа $\$elemMatch$ выступает объект, который также является объектом запроса.
  - Для поиска по id пользователя, достаточно в данном объекте запроса указать поле id с нужным значением(строка 5 листинга 10).
  - Для того чтобы указать, что роль должна быть больше либо равной 2 используется ключ $\$gte$, который должен являтся подключом поля $role$. Значением ключа $\$gte$ является цифра 2.

Для изменения только некоторых полей в найденных, после объекта запроса, документах, в объекте обновления используется ключ $\$set$. Значением такого ключа выступает объект с полями и их новыми значениями. В данном случае необходимо изменить только три поля: $name$,

description и *link* — поэтому именно они и указываются в объекте поля $*set* с соответствующими значениями для обновления.

Функция обратного вызова выполняется после обновления, в данном случае с помощью этой функции контролируется успешность обновления. Если переменная *err* пуста, значит процедура обновления прошла без ошибок. Также стоит проверить переменную *result*, если количество обновленных документов равно 0(за это отвечает поле *n* внутри объекта *result*), то значит мы не нашли подходящую группу для обновления, что может обозначать, что либо клиент прислал неправильные данные группы, либо пользователь не является модератором или администратором группы и не может её обновлять. В таком случае на клиент придет сообщение об ошибке.

6.2. API invite

Данное API нужно для отправки запроса на вступление в группу. Данный запрос отправляется в тот момент, когда клиент переходит по ссылке “radoms.ru/invite?link=someLink”. Именно для такой уникальной ссылки и нужно поле *link* в объекте группы.

Листинг 11: Приглашение в группу

```
1  Group.update({
2     link: req.query.link,
3     ''users.id'': {$ne: req.user._id}
4  }, {
5     $push: {
6        users: {
7           id: req.user._id,
8           commit: req.user.commit
9        }
10    }
11 }, function (err, result) {/*...*/})
```

Как видно из листинга выше, для отправки запроса в группу используется метод update. Запросом на вступление в модели группы считается пользователь с ролью равной -1(опр. 3). Соответственно цель данного обработчика — добавить в группу с определенным *link* пользователя

с ролью -1. Добавление элемента в список осуществляется с помощью оператора $\$push$. Также важно, чтобы пользователь с таким не состоял в данной группе на момент добавления, поэтому в операторе поиска указывается, что в объекте группы не должно быть пользователя с id равным id пользователя который отправил запрос. Отрицание выполняется с помощью ключа $\$ne$. Если же пользователь попытается зайти в группу в которой он состоит или уже послал запрос на вступление, то ему выводится соответствующая ошибка.

6.3. API groupUsers. GET

Данное API нужно для вывода группы и ее пользователей. Важно, чтобы информация о пользователях содержала также id, имя, фамилию, $commit$ и роль. Т.к. наличие имени, фамилии и $commit$'а пользователей в модели группы не предусмотрено, необходимо произвести соединение документов из разных моделей БД: модели группы и модели профиля пользователя. Для выполнения сложных операций над БД используется метод $aggregate$. Данный метод последовательно выполняет список заданных команд. Каждая последовательная команда помещается в свой объект.

Листинг 12: Выдача списка пользователей группы

```
1  Group.aggregate([
2  {
3     $match: {
4        _id: mongoose.Types.ObjectId(req.query.groupId)
5     }
6  }, {
7     $unwind: ''$users''
8  }, {
9     $lookup: {
10       from: ''profiles'',
11       localField: ''users.id'',
12       foreignField: ''_id'',
13       as: ''profile''
14    }
15 }, {
16    $unwind: ''$profile''
```

```
17 }, {
18    $group: {
19       ''_id'': {
20          ''id'': ''$_id'',
21          ''link'': ''$link'',
22          ''description'': ''$description'',
23          ''title'': ''$title''
24       },
25       ''users'': {
26          $push: {
27             ''_id'': ''$profile._id'',
28             ''role'': ''$users.role'',
29             ''commit'': ''$profile.commit'',
30             ''name'': ''$profile.name'',
31             ''surname'': ''$profile.surname''
32          }
33       }
34    }
35 }
36 ], function (err, result) {/*...*/});
```

Шаг 1. Первым шагом необходимо найти нужную группу. Поиск в методе aggregate осуществляется с помощью ключа $\$match$, значением данного ключа выступает обычный объект запроса(опр. 1).

Шаг 2. Далее необходимо произвести деконструкцию списка пользователей. Т.е. необходимо объект группы со списком пользователей превратить в список объектов группы, где каждый объект группы будет содержать информацию лишь об одном конкретном пользователе. В данном случае деконструируется поле $users$, которое содержит список пользователей. Такая деконструкция выполняется с помощью ключа $\$unwind$. Пример выполнения можно посмотреть листингах 13 и 14.

Листинг 13: Объект группы до деконструкции

```
{
        name: 'Example',
        ...,
        users: [{
                id: 1,
                role: 3
        },{
                id: 2,
                role: 2
        },{
                id: 3,
                role: -1
        }]
}
```

Листинг 14: Список объектов групп после деконструкции

```
[{
        name: 'Example',
        ...,
        users: {
                id: 1,
                role: 3
        }
},{
        name: 'Example',
        ...,
        users: {
                id: 2,
                role: 2
        }
},{
        name: 'Example',
        ...,
        users: {
                id: 3,
                role: -1
        }
}]
```

Шаг 3. Следующим шагом необходимо соединить каждый объект группы из списка с соответствующим профилем пользователя. Для соединения документов из разных моделей БД используется специальный ключ $\$lookup$. Значением данного оператора является объект с 4 полями: $from$ — название модели БД с которой будет происходить соединение; $localField$ — поле в локальной модели(в данном случае в модели группы) по которому будет происходить соединение; $foreignField$ — поле во внешней модели(в данном случае в модели профиля пользователя) по которому будет происходить соединение; as — новое поле в выходном объекте значением которого будет присоединенный объект внешней модели.

После применения данного оператора, каждый объект группы в списке будет содержать не только id каждого пользователя, но и также всю другую информацию о пользователе, которая содержалась в

профиле пользователя.

Шаг 4. После применения ключа $*lookup* значением нового поля всегда является список объектов, даже если количество объектов равно 1. Поэтому для дальнейшей работы необходимо деконструировать новое поле. Как и во 2 пункте применяется ключ $*unwind*.

Шаг 5. Далее необходимо сгруппировать все объекты и соединить всех пользователей в один список. Группирование делается с помощью оператора $*group*. Значением данного оператора выступает объект структуры, данный объект показывает каким образом необходимо сгруппировать объекты. В поле *_id* необходимо написать все поля по которым будет происходить группировка, в данном случае это поля id, title, link, description. Т.к. работа происходит с одной группой, данные поля во всех объектах являются одинаковыми. По этому после группировки должен получится один объект. Чтобы воспользоваться значениями полей из входного объекта достаточно взять название поля и дописать вначале знак доллара. Непосредственное соединение пользователей в список осуществляется с помощью оператора $*push*.

7. Общий обработчик для небольших запросов

При разработке серверной части возникла потребность в обработке небольших запросов. Т.е. таких запросов, которые используются малое количество ресурсов сервера и/или используются клиентской частью нечасто. Было принято решение все такие запросы на сервере обрабатывать в одном месте. Т.е. все такие запросы приходят на один URL адрес сервера, где находится один обработчик, который выступает т.н. распределителем. Этот обработчик каждый пришедший запрос распределяет посредством оператора switch-case на отдельные, более мелкие, специализированные обработчики. Далее такие обработчики, в зависимости от своей специализации, обрабатывают запрос и отправляют данные обратно клиенту.

Листинг 15: Общий обработчик

```
1  router.route('/suggest')
2        .get(function (req, res) {
3           switch (req.query.type) {
4              case ''multi_user'':
5                 multiUsers(req, res);
6                 break;
7              default:
8                 res.status(500).send({message: ''UNSUPPORTED OBJECT''});
9                 break;
10          }
11
12       });
```

Листинг 16: Пример мелкого ajax-запроса multi_users

```
1  $http({
2      url: 'api/suggest',
3      method: 'GET',
4      params: {
5          type: 'multi-user',
6          text: searchText
7      },
8      timeout: timeout,
```

```
 9   }).then(function (response) {
10       $scope.model.multiUsers = response.data;
11   })['finally'](function () {
12   });
```

В представленном выше коде запрос отправляется по ссылке api/suggest(именн там находится общий обработчик мелких запросов), по средствам метода *GET*, с двумя параметрами: *type* и *text*. Параметр *type* нужен для общего обработчика, чтобы он мог понять, с каким конкретно запросом он имеет дело и мог отправить данный запрос к нужному специализированному обработчику. Параметр *text* — это данные для конкретного специализированного обработчика.

Листинг 17: Пример специализированного обработчика multi_users

```
1  function multi_users(req, res) {
2     var str = req.query.text;
3     User.find({
4        $or: [
5           {name: new RegExp(str, 'i')},
6           {surname: new RegExp(str, 'i')}
7        ]
8     }, {
9        name: 1,
10       surname: 1,
11       _id: 0
12    }).find(function (err, users) {
13       if (err) {
14          res.status(500).send({message: 'Error on read user list. Refresh the
                ↪ page, please'});
15       } else {
16          res.json(users);
17       }
18    })
19 }
```

Данный специализированный обработчик возвращает список пользователей, у которых в имени или фамилии встречается строка переданная ранее в запросе.

8. Импорт списка публикаций из сервиса elibrary

В сервисе реализовано несколько способов ввода списка публикаций: ввод поштучно с заполнением формы вручную и импорт с помощью таблиц csv, xls. Добавление большого количества публикаций с помощью данных методов оказывается достаточно затруднительным. Поэтому необходимо было разработать новый метод импорта публикаций, который бы работал, буквально, в несколько кликов. Т.к. большое количество целевой аудитории нашего проекта пользуются сервисом elibrary, то решением данной задачи стала разработка импорта публикаций из сервиса elibrary.

Elibrary не имеет внешнего API, поэтому выгрузку метаданных публикаций из этого сервиса необходимо провести “вручную”. Т.е. необходимо скачать нужные страницы профиля, распарсить их, выделить нужные данные и загрузить в систему. Общая стратегия импорта метаданных публикаций выглядит следующим образом:

1. На первом шаге происходит проверка введенной пользователем ссылки. Проверяется корректность ссылки и если ссылка корректна, возвращается id пользователя в системе elibrary.

2. На следующем шаге происходит закачка первой страницы списка публикаций. После закачки первая страница разбирается и устанавливается точное количество страниц списка публикаций, также подготавливается объект, в котором будут содержаться все полученные данные.

3. После этого происходит закачка и разбор остальных страниц списка публикаций в параллельном режиме.

4. После первоначального парсинга, у каждой публикации есть три поля — это название публикации, список авторов и все остальные данные по публикации.

5. Все три поля представляют из себя обычную строку, поэтому следующим шагом необходимо из строки списка авторов получить список отдельных авторов. Также, по возможности, разбирается третье поле с остальными данными по публикации. В большинстве случаев, удачно можно получить год публикации, страницы и номер издания, где находится публикация. Также, довольно часто, разбирается название журнала или сборника публикаций.

6. Далее все полученные данные возвращаются на клиентскую часть, для дальнейшей работы с ними.

Рассмотрим наиболее значимые пункты стратегии подробнее.

8.1. Инициализация переменных

Первым делом инициализируются три переменные:

Листинг 18: Инициализация трех переменных

```
1 var mainObj = {};
2 var mainList = [];
3 var pageCount = -1;
```

mainObj — данная переменная является общим объектом куда будет помещена вся информация полученная при разборе страниц. *mainList* — это список всех метаданных по публикациям, данный список формируется в последнюю очередь и вызывается вместе с функцией обратного вызова. *pageCount* — количество страниц списка публикаций.

8.2. Получение общих данных публикаций с первой страницы

Далее, для получения DOM-объекта первой страницы списка, используется функция *downloadElibraryPageByAuthoridAndPagenum* — данная функция представляет из себя простой запрос, который скачивает страницу, разбирает её на DOM-дерево, с помощью пакета cheerio,

и передает готовый объект в функцию обратного вызова. Далее из данного объекта необходимо достать данные публикаций, делается это с помощью методов пакета cheerio:

Листинг 19: Выборка нужных строк из страницы

```
1 var tr_it = [];
2 var tr_list = $('tr[id^=arw]');
3 tr_list.each(function () {
4         tr_it.push(objFromTr($(this)));
5 });
```

Каждая публикация из списка находится в своем строковом теге с идентификатором начинающимся с подстроки arw. Чтобы выбрать такое подмножество из DOM-объекта, необходимо воспользоваться соответствующим css-селектором: $tr[id\hat{} = arw]$.

Далее необходимо разобрать каждую строку с публикацией на отдельные фрагменты данных: название, авторов и описание.

Листинг 20: Парсинг общих данных

```
1 function objFromTr(tr) {
2    var td = tr.find('td[align=left]');
3    var name = td.find('a b').text();
4    var authors = td.find('font i').text();
5    var desc = td.find('font').last().text();
6    var authorsAP = authorsParser(authors);
7    var prDesc = descParser(desc);
8    return {name: name, authors: authors, desc: desc, authorsList: authorsAP,
        ↪ possibleDesc: prDesc}
9 }
```

В самом начале из строкового тега tr выбирается столбцовый тег td с выравниванием по левому краю, именно там находятся все данные по публикации. Название публикации находится с помощью css-селектора "ab", что обозначает тег b внутри тега a. Авторы — с помощью css-селектора "$fonti$" — тег i внутри тега font. Описание находится в самом последнем теге $font$, его выборка осуществляется с помощью css-селектора "$font$" и метода $last()$, который выдает только последний найденный элемент.

8.3. Подробный разбор объекта публикации

На данном этапе все описание конкретной публикации представляет из себя единую строку, где информация разделена точкой. Поэтому на этом шаге происходит парсинг этой информации на более подробные часть. Чтобы выделить из этой строки конкретные метаданные публикации используется свойства этих метаданных. Т.е. например год публикации всегда представлен в виде четырехразрядного числа:

Листинг 21: Парсинг года

```
1 if (tempDesc.length == 4 && tempDesc.search(/[1-3]\d\d\d/) != -1) {
2     prDesc.year = parseInt(tempDesc);
3 }
```

А, например, страницы на которых находится публикация всегда начинаются с буквы “С”:

Листинг 22: Парсинг страниц

```
1 else if (tempDesc[0] == C'') {
2     var fromToNum = parseFromToNum(dotSplit[i+1]);
3     prDesc.firstPage = fromToNum.from;
4     prDesc.lastPage = fromToNum.to;
5 }
```

Т.к. страницы могут быть представлены в виде диапазона, используется отдельный метод для парсинга диапазона страниц $parseFromToNum$.

8.4. Выяснение количества страниц списка публикаций

Следующий шаг, который необходимо сделать, это выяснить количество страниц списка:

Делается это с помощью css селектора: “$\#pagestrtda$”. Данный селектор возвращает ссылки на все объекты для навигации. Если список состоит всего из одной страницы, то активных ссылок не будет вовсе, т.е. селектор возвратит нулевой объект. В противном случае, необходимо воспользоваться самой последней навигационной ссылкой, которая

ведет на последнюю страницу, достаточно посмотреть её номер. Делается это с помощью небольшого регулярного выражения: .

8.5. Разбор всех остальных страниц

Все остальные страницы разбираются в параллельном режиме по алгоритму представленному в секциях 8.2 и 8.3. По завершении разбора очередной страницы в объекте mainObj выставляется флаг завершенности разбора текущей страницы. Далее проверяется, завершился ли разбор во всех остальных страницах, если да, то готовый объект возвращается в функцию обратного вызова.

9. Получение production-версии сервиса с помощью планировщика задач Grunt

Т.к. сервис представлен в производстве (production-версии), его нужно обеспечить соответствующей версией, которая будет отличаться от версии для разработчика. Для разработчика важно чтобы все было на своих местах, чтобы код был удобочитаем и было удобно работать с проектом. В случае версии для производства, все это не важно, важно чтобы продукт работал быстро и надежно. Для этого применяются многие методы оптимизации, например: сжатие файлов кода, сжатие картинок без потери качества, объединение файлов и т.п. Все эти действия требуют немалого времени по отдельности, поэтому для удобства и экономии времени для получения версии для производства используются т.н. *планировщики задач*. В случае нашего сервиса используется планировщик Grunt[5].

9.1. Общая настройка

Общая настройка Grunt производится в файле Gruntfile.js в папке с проектом. Этот файл служит для настройки задач, которые будут выполняться над проектом. В случае проекта RADOMS будут выполнены следующие задачи:

- Компиляция scss в css.
- Сжатие js, css файлов.
- Объединение js, css файлов.

Рассмотрим подключение пакета подробнее:

Листинг 23: Файл общей настройки

```
1 module.exports = function(grunt) {
2
3    require('time-grunt')(grunt);
4
5    require('load-grunt-config')(grunt, {
6       jitGrunt: true
```

```
7     });
8 };
```

Чтобы подключить пакет к Grunt, нужно импортировать его в файл Gruntfile.js. Импорт в node.js осуществляется с помощью функции require.

Для более удобной настройки используется пакет load-grunt-config, он позволяет помещать настройки для отдельных пакетов в отдельные файлы. Все такие файлы с настройками должны лежать в папке grunt в корне проекта. Так же для удобства используется пакет time-grunt, он, в процессе выполнения пакета задач, показывает сколько времени ушло на ту или иную задачу.

Настройка отдельных пакетов в Grunt осуществляется в едином стиле, но т.к. задачи бывают разной сложности, объем таких настроек и структура может разниться. Вся настройка происходит в JSON-объекте, ключом первого уровня должно выступать название отдельной настройки, т.е. можно задать несколько настроек для одного пакета и использовать их по отдельности по названию. Значением же такого ключа является объект с уже конкретными настройками для конкретного пакета. Такой объект обычно отвечает на вопросы: “где взять файлы?”, “какие файлы брать?”, “что с ними делать?”, “куда файлы положить?”.

9.2. Настройка копирования

Самая простая задача в случае создания версии для производства — это копирование файлов. Копировать нужно те файлы, которые не требуют отдельной обработки. Таких файлов в проекте не мало: файлы шрифтов, файлы сервера, большинство html файлов. За копирование файлов в Grunt отвечает пакет grunt-contrib-copy. Файл с настройками данного пакета должен называться copy.js и находится в папке grunt в корне проекта.

Листинг 24: Файл настройки пакета копирования

```
1 module.exports = {
2     prod: {
```

```
3        files: [
4           {
5              expand: true,
6              cwd: ''app'',
7              src: ['**/*.html'', ''!index.html''],
8              dest: ''production/app''
9           },
10          {
11             expand: true,
12             cwd: ''server'',
13             src: ''**/*'',
14             dest: ''production/server''
15          }
16       ]
17    }
18 };
```

В листинге выше представлена лишь часть настроек, в данном случае копируются все html файлы, кроме index.html, из папки app, также копируются все серверные файлы из папки server. Как видно настройки из себя представляет объект из ключей и полей. Ключ первого уровня *prod* — это название конкретной настройки пакета copy. Чтобы использовать эту настройку в дальнейшем, в консоли надо будет набрать "grunt copy:prod". Значением ключа *prod* является объект настроек. Настройка $files$ — отвечает за обозначение тех файлов, которые будут использованы для данного пакета, т.е. которые будут скопированы. Значение ключа $files$ — это список объектов, каждый такой объект отвечает за отдельный набор файлов. В данном случае объект состоит из 4 ключей

- *expand* — если стоит *true*, значит текущий объект является расширенным и позволяет использовать динамическое сопоставление с образцом из списка — поле *src*.
- *src* — содержит массив строк. Строки — это т.н. маски для поиска файлов, они могут содержать конкретный путь к файлу, а могут содержать специальные символы, которые обобщают несколько файлов:
  - "*" — означает любой набор символов, любой длины.

– “**” — означает любой путь к файлу любой длины.

– “!” — ставится вначале строки и означает отрицание. Т.е. файлы попадающие под какую-либо обычную маску, но не попадающие под маску с “!” не будут обработаны. Важно заметить, что маска с отрицанием исключит файл, который попадает под какую-нибудь обычную маску, только в том случае, если маска с отрицанием стоит в списке после обычной маски.

- *cwd* — это корневая папка для массива масок, т.е. в этой папке будет происходить поиск по маскам из поля *src*
- *dest* — папка назначения, т.е. в эту папку будут скопированы файлы

9.3. Настройка css файлов

При разработке клиентской части использовался язык sass для описания стилей. Но браузер для описания стилей понимает только каскадный язык css, поэтому все файлы scss нужно скомпилировать в файлы .css. Также для производственной версии можно минимизировать все css файлы и соединить их в один, это увеличит скорость выкачивания css файлов с сервера.

В итоге, для конечного результата нужно:

1. Скомпилировать scss файлы в css файлы
2. Сжать все css файлы
3. Объединить все css файлы

Для компилирования файлов scss используется пакет “grunt-sass”:

Листинг 25: Файл настройки пакета компилирования

```
1 module.exports = {
2    prodCss: {
3       options: {
4          outputStyle: 'compressed',
```

```
5           sourceMap: false
6       },
7       files: [
8           {
9               expand: true,
10              flatten: true,
11              cwd: ''app'',
12              src: [''**/*.scss''],
13              dest: ''production/static/style/css'',
14              ext: ''.css''
15          }]
16   }
17 };
```

Как видно, структура настроек подобна настройкам пакета копирования “grunt-contrib-copy”. В данном случае есть дополнительное поле *options*, для дополнительных настроек:

- Поле *outputStyle* отвечает за стиль выходных файлов, т.е. в каком формате будет код внутри файлов. В данном случае выставлена опция *compressed*, что означает, что на выходе нужно весь css код сжать.

- Поле *sourceMap* отвечает за создание map-файлов. Такие файлы нужны для разработчика для просмотра и редактирования стилей прямо в браузере. В данном случае создание map-файлов отключено.

Также в объекте списка *files* можно заметить новое поле *flatten*. По умолчанию такое поле принимает значение *false*, но если его значение *true*, тогда все файлы после обработки(в случае пакета “grunt-sass” обработка — это компилирования в css) будут попадать прямиком в папку назначения, без сохранения структуры.

Далее все css файлы нужно соединить в один, для этого используется пакет “concat-css”:

Листинг 26: Файл настройки пакета объединения css файлов

```
1 module.exports = {
2    prodCss: {
```

```
3       files: {
4          'production/static/style/css/styles.css': [
5             'production/static/style/css/*.css'
6          ]
7       }
8    }
9 };
```

Как видно, в данном случае используется другая структура описания файлов для обработки. Поле *files* содержит объект, где ключом является файл назначения, а полем этого ключа является список масок для поиска файлов. Такая структура удобна, когда настроек мало и дополнительные настройки для поиска файлов не нужны.

После всех этих процедур, в папке “production/static/style/css”, лежит несколько скомпилированных css файлов и еще один общий css файл, который является объединением всех остальных файлов, который и нужен для версии производства. Соответственно из этой папки нужно удалить все лишнее, для этого используется пакет “grunt-contrib-clean”:

Листинг 27: Файл настройки пакета удаления

```
1 module.exports = {
2    prodCss: [
3       'production/static/**/*.css',
4       '!production/static/style/css/materialize/materialize.min.css',
5       '!production/static/**/styles.css']
6 };
```

В данном случае используется еще более простая структура описания файлов для обработки, т.к. все что требуется для настройки данного пакета — это указать какие файлы или папки нужно удалить. В данном случае удаляются все найденные css файлы в каталоге production/static и во всех его подкаталогах, кроме файла style.css и файла materialize.min.css.

9.4. Настройка js

С javascript файлами нужно поступить так же как с css. Их нужно сжать и объединить. Стоит отметить что в сервисе используется фреймворк angular.js, который имеет ряд особенностей, которые при сжимании файла могут быть утрачены и код в этом файле не будет выполняться корректно. Поэтому перед сжатием такие файлы нужно обработать, делается это с помощью пакета “grunt-ng-annotate”, данный пакет добавляет в angular-файлы аннотацию, если таковой нет.

Листинг 28: Файл настройки пакета аннотации

```
1  module.exports = {
2      angularUgly: {
3          options: {
4              singleQuotes: true
5          },
6          files: [
7              {
8                  expand: true,
9                  cwd: 'app',
10                 src: ['**/*.js', '!**/*.min.js'],
11                 dest: 'production/app',
12                 ext: '.js'
13             }
14         ]
15     }
16 };
```

Настройка имеет расширенную структуру. Есть дополнительная опция — *singleQuotes*, которая при значении *true* использует для аннотации одинарные кавычки. Так же в единственном объекте списка *files* используется поле ”ext”, оно служит для указания расширения выходных файлов, в данном случае расширение выходных файлов будет ”.js”.

Далее нужно сжать все js файлы, делается это с помощью пакета “grunt-contrib-uglify”:

Листинг 29: Файл настройки пакета сжатия js-файлов

```
1  module.exports = {
2      angularUgly: {
```

```
3         files: [
4             {
5                 expand: true,
6                 cwd: 'production/app',
7                 src: ['**/*.js', '!**/*.min.js'],
8                 dest: 'production/app',
9                 ext: '.js'
10
11            }
12        ]
13    }
14 };
```

Стандартная расширенная настройка пакета grunt. Берутся все .js файлы, кроме файлов .min.js(они уже сжаты), сжимаются и переносятся в папку production.

Далее нужно объединить файлы. В данном случае все .js файлы объединяются в три файла: libs.js — содержит все библиотеки; components.js — содержит все компоненты клиентской системы; administrator.js — отдельно содержит компоненты для административной страницы. Отдельный файл для административной страницы нужен для того, чтобы сервер мог блокировать доступ к нему у непривилегированных пользователей.

Листинг 30: Файл настройки пакета объединения

```
1  module.exports = {
2     prodJs: {
3        files: [
4           {
5              options: {
6                 separator: '';\n''
7              },
8              src: [
9                 'production/static/lib/jquery-2.1.4.min.js',
10                ''production/static/lib/angular.min.js'',
11                ''production/static/lib/angular-route.min.js'',
12                ''production/static/lib/angular-animate.min.js'',
13                ''production/static/lib/materialize.min.js'',
14                ''production/static/lib/*.js'',
15                ''!production/static/lib/vscroll.js'',
16                ''!production/static/lib/groupbox.js'',
17                ''production/static/lib/vscroll.js'',
```

```
18                 ''production/static/lib/groupbox.js''
19             ],
20             dest: ''production/static/lib/libs.js''
21         }
22     ]
23   }
24 };
```

Как видно из листинга выше, дополнительные опции применяются отдельно к группе файлов, а не ко всем используемым файлам. *separator* — указывает на разделитель в месте соединения файлов, в данном случае — это ;
n, т.е. точка с запятой и переход на новую строку. В данном случае важна последовательность соединения файлов, важно чтобы сначала шел файл jquery-2.1.4.min.js, затем файлы angular и также важно чтобы файлы vscroll.js и groupbox.js шли в конце, все остальные файлы могут располагаться между ними в произвольном порядке. Из списка *src* файлы берутся по порядку, по этому достаточно указать файлы jquery, angula" в начале списка. Затем должны идти все остальные js-файлы, кроме некоторых. Для "всех остальных" используется маска: $*.js$, но что бы в эту маску не попали vscroll.js и groupbox.js, нужно в списке *src* после данной маски, указать эти файлы с отрицанием, т.е. с "!" в начале пути к файлу. Далее после всех js-файлов нужно включить оставшиеся два файла, для этого после отрицательных масок просто указываем пути к оставшимся. Таким образом итоговый файл получился с нужной последовательностью.

Заключение

В рамках данной бакалаврской работы были достигнуты следующие результаты:

1. Создана структура серверной части.
2. Разработана структура БД. Созданы три модели: модель профиля пользователя, модель группы пользователей и модель публикации.
3. Разработано соответствующее API для работы с этими моделями. Также было разработано API для страницы администратора.
4. Реализован импорт списка публикаций из сервиса elibrary.
5. Настроен планировщик задач на получение production-версии сервиса.

Также в ходе работы над данным проектом:

- опубликована статья Грибков К.В., Хайдаршин А.М., Суворова А.В., Тулупьев А.Л. Проект RADOMS: программные компоненты серверной части // Материалы 6-й всероссийской научной конференции по проблемам информатики СПИСОК-2016. (26–29 апреля 2016 г. Санкт-Петербург). СПб.: ВВМ, 2016. С. 463–476;
- принята в печать статья Грибков К.В., Суворова А.В., Тулупьев А.Л. Проект RADOMS: программные компоненты серверной части — импорт метаданных из сервиса elibrary, группы пользователей и операции над ними. // Материалы 7-й всероссийской научной конференции по проблемам информатики СПИСОК-2017. (26-28 апреля 2017 г. Санкт-Петербург). СПб.: ВВМ, 2017.
- направлена заявка на регистрацию системы в РОСПАТЕНТ.

Список литературы

[1] AngularJS — Superheroic JavaScript MVW Framework. — 2017. — URL: https://angularjs.org/ (online; accessed: 23.05.2017).

[2] Cheerio. — 2017. — URL: https://cheerio.js.org/ (online; accessed: 23.05.2017).

[3] ELIBRARY.RU - НАУЧНАЯ ЭЛЕКТРОННАЯ БИБЛИОТЕКА. — 2017. — URL: https://elibrary.ru/defaultx.asp (online; accessed: 23.05.2017).

[4] Express - Node.js web application framework. — 2017. — URL: http://expressjs.com/ (online; accessed: 23.05.2017).

[5] Grunt: The JavaScript Task Runner. — 2017. — URL: https://gruntjs.com/ (online; accessed: 23.05.2017).

[6] JQuery. — 2017. — URL: https://jquery.com/ (online; accessed: 23.05.2017).

[7] Mendeley: Homepage. — 2017. — URL: https://www.mendeley.com/ (online; accessed: 23.05.2017).

[8] MongoDB for GIANT Ideas | MongoDB. — 2017. — URL: https://www.mongodb.org/ (online; accessed: 23.05.2017).

[9] Mongoose ODM v4.10.2. — 2017. — URL: http://mongoosejs.com/ (online; accessed: 23.05.2017).

[10] Mongoose Schemas v4.10.2. — 2017. — URL: http://mongoosejs.com/docs/guide.html (online; accessed: 23.05.2017).

[11] Node.js. — 2017. — URL: https://nodejs.org/en/ (online; accessed: 23.05.2017).

[12] Npm. — 2017. — URL: https://www.npmjs.com/ (online; accessed: 23.05.2017).

[13] Passport. — 2017. — URL: http://passportjs.org/ (online; accessed: 23.05.2017).

[14] Research and Development Outcomes Management System. — URL: http://radoms.ru/ (online; accessed: 23.05.2017).

[15] Scopus - Welcome to Scopus. — 2017. — URL: https://www.scopus.com/ (online; accessed: 23.05.2017).

[16] Академия Google. — 2017. — URL: https://scholar.google.ru/ (дата обращения: 23.05.2017).