Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Южный Федеральный Университет»

Факультет математики, механики и компьютерных наук

Направление подготовки 010400
«Прикладная математика и информатика»

А. А. Тактаров

Реактивный фреймворк для организации мультиагентных распределенных вычислений

Магистерская диссертация

Научный руководитель:
старший преподаватель
В. Н. Брагилевский

Рецензент:
доцент, к. ф.-м. н.
С. А. Гуда

Ростов-на-Дону
2014

Содержание

Введение

Стремительный рост возможностей технологий беспроводной передачи данных, а также широкое распространение мобильных и встраиваемых устройств способствовали появлению концепции так называемого «Интернета вещей» (англ. *«Internet of Things»*)[1], которая заключается в объединении всех окружающих людей вещей в огромную вычислительную сеть. Участниками (**агентами**) такой сети являются устройства, которые способны собирать информацию о физической среде, обрабатывать ее и реагировать на изменение состояния других агентов и всей системы в целом. Стабильное функционирование такой сети позволит с огромной скоростью внедрять и использовать такие технологии, как «умные» датчики[2], носимые устройства (англ. *wearable devices*), а также системы автоматизированного управления домом. Кроме того, становление «Интернета вещей» влечет за собой появление принципиально новых потоков информации, тщательный анализ которых позволит улучшать существующие системы здравоохранения, безопасности и контролировать состояние окружающей среды.

Однако, создание подобного рода сети невозможно без наличия функционирующей инфраструктуры, которая бы позволила быстро и эффективно интегрировать новые компоненты. Исходя из распределенной природы описываемой сети, сформулируем необходимые для этого требования:

1. Соблюдение принципа системности при разработке[3]. Продукт должен быть представлен в виде целой системы компонентов, каждый из которых обладает определенной функцией. Такие компоненты автоматически становятся автономными участниками сети.

2. Однородность среды. Компоненты сети должны иметь возможность взаимодействовать между собой, используя стандартизированные протоколы и схемы. Задачи идентификации, обеспечения целостности, конфиденциальности передаваемых данных должны по возможно-

сти быть решены этими протоколами.

3. Открытость используемых технологий. Применение как программных, так и аппаратных решений, которые имеют открытую документацию, лояльные условия использования и одновременно поддерживаются разными разработчиками (обычно целым сообществом), позволяет в определенных случаях решить проблему интеграции компонентов и сократить разрыв между разработкой и запуском в производство. Кроме того, открытые платформы предоставляют широкие возможности для начинающих команд разработчиков, что является благоприятным для формирования рынка.

В данной работе описан процесс реализации и интеграции мультиагентной системы на примере задачи распределенной печати фотографий. В рамках разработанной системы устройство, печатающее фотографию, рассматривается как автономный агент, который обладает состоянием и способен принимать и исполнять задания. Принципы, сформулированные выше, были использованы в качестве основополагающих на этапах проектирования и разработки данного продукта.

Постановка задачи

Целью работы является разработка и развертывание системы, позволяющей организовать моментальную печать фотографий пользователей социальной сети Instagram, распределяя задания печати среди подключенных к системе агентов — **печатных станций**, в состав которых входит печатное устройство — принтер.

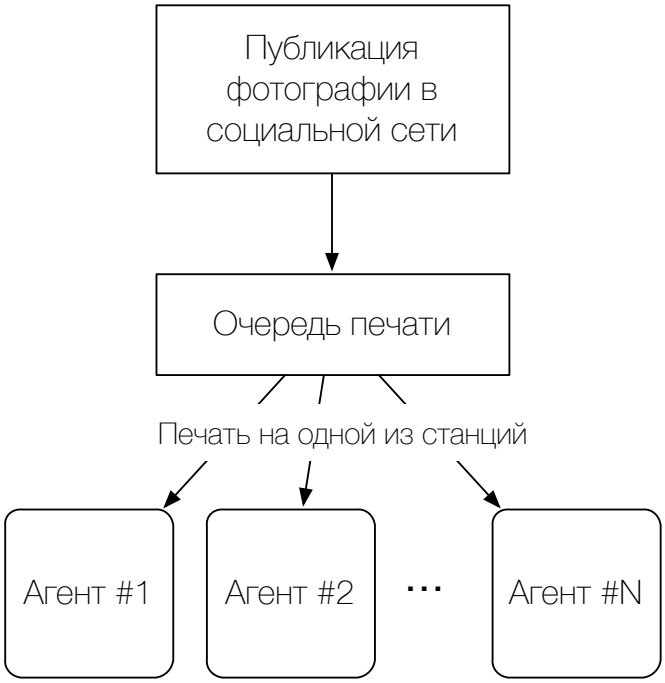

Рис. 1: схема исполнения заданий печати

Фотографии, которые публикуются пользователями социальной сети и удовлетворяют определенным условиям поиска (содержат заранее известную метку — *хештег*), должны автоматически поступать в очередь печати системы. Далее, исходная фотография, прошедшая определенную пост-обработку, печатается на одном из принтеров, входящими в состав печатных станций (рис. 1). Информация о напечатанной фотографии сохраняется в системе для отчетности. Функционирование такой системы позволяет организовать массовую печать фотографий во время проведения мероприятий или для организации отложенной печати.

Сформулируем основные требования, предъявляемые к системе:

1. Печатные станции могут быть физически отделены друг от друга, кроме того они могут находиться в разных сегментах сети. Необходим способ организации канала связи между агентами и контроль жизнеспособности этого канала.

2. Необходим интерфейс управления печатными станциями и заданиями печати.

3. Система должна иметь минимальный отклик и максимально быстро реагировать на изменение состояния компонентов. Изменение статуса задания печати (печать может завершиться успешно, а может закончиться неудачей в результате обрыва соединения) должно моментально отражаться в интерфейсе управления заданиями.

Выделим последовательные этапы решения поставленной задачи:

1. Проектирование архитектуры системы: разбиение системы на компоненты, выбор используемых при реализации каждого компонента технологий, построение схемы взаимодействия.

2. Реализация компонентов системы, покрытие отдельных частей функциональными тестами.

3. Решение задач интеграции и развертывании системы, настройка аппаратных средств.

4. Опытное тестирование работы продукта.

1. Архитектура системы

В состав разработанного продукта входят три основных компонента: центральный сервер, агент печатной станции и веб-приложение, предоставляющее интерфейс пользователя (рис. 2).

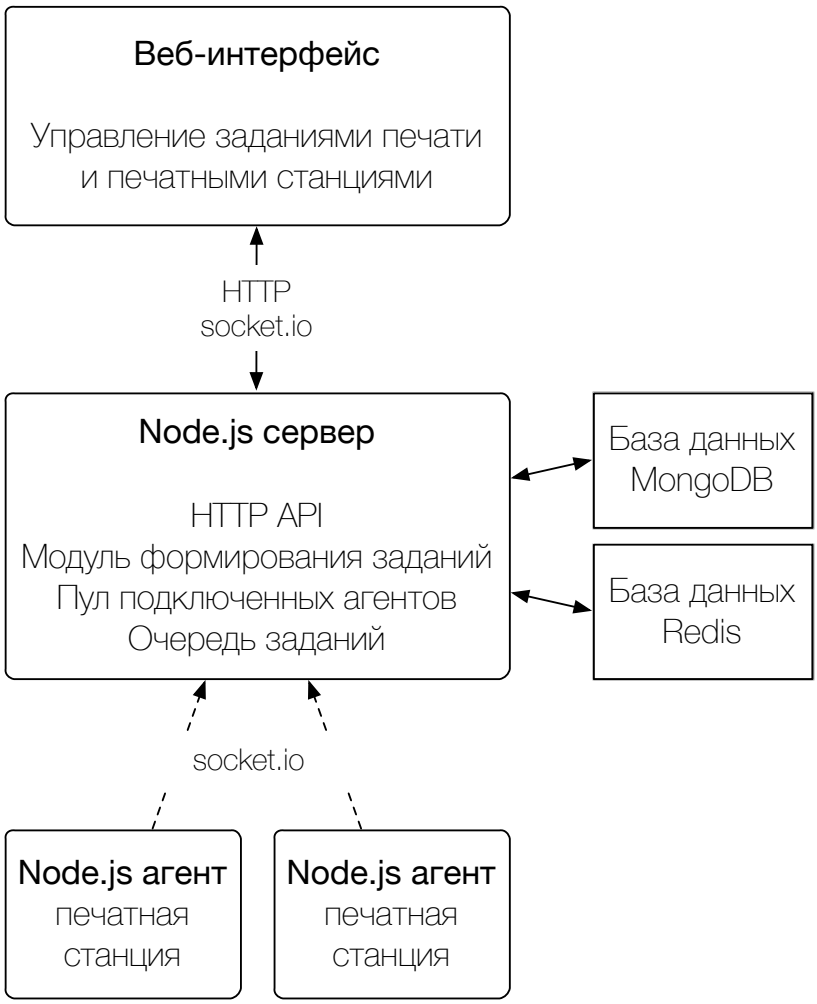

Рис. 2: архитектура системы

Для хранения данных используются базы данных MongoDB и Redis, обращение к которым происходит через центральный сервер. База данных MongoDB содержит информацию о зарегистрированных печатных станциях, администраторах системы, а также хранит историю всех завершенных заданий печати. Средствами MongoDB реализована возможность гибкого поиска и фильтрации данных[4].

База данных Redis, являющаяся быстрым хранилищем типа «ключ-значение», используется для организации очереди заданий печати. Кроме того, благодаря возможности хранения данных в оперативной памяти данная база данных выступает в роли хранилища сессий центрального веб-сервера.

1.1. Центральный сервер

Ядром системы является центральный сервер, в функции которого входит:

1. Управление очередью печати. Модуль формирования заданий используется для поиска в социальной сети новых отмеченных для печати фотографий, которые помещаются в очередь печати. Распечатанные фотографий извлекаются из очереди, а ненапечатанные дополняются сообщением об ошибке для отчетности.

2. Взаимодействие с подключенными по каналу связи агентами. Контроль качества канала связи и авторизация печатных станций.

3. Предоставление прикладного программного интерфейса (API) на основе протокола HTTP.

Данный компонент разработан на языке программирования CoffeeScript[5] и работает на основе асинхронного серверного фреймворка Node.js. Решение по использованию данного инструментария было принято, исходя из следующего:

1. Платформа Node.js предоставляет широкие возможности по использованию низкоуровневых средств (работа с процессами, сокетами, бинарными данными и потоками ввода-вывода), предоставляя для этого удобный интерфейс на языке программирования JavaScript. Кроме того, все операции ввода вывода в Node.js являются асинхронными (т.е. не блокируют исполнение программы), а контроль

завершения происходит с помощью функций обратного вызова и событий. Обработка завершения асинхронных действий реализована в так называемой *очереди обработки событий* (англ. *event loop*)[6], работающей на основе паттерна Проактор[7].

2. Node.js позволяет разработчику использовать сторонние модули благодаря мощному пакетному менеджеру NPM. Простота публикации модулей и открытое сообщество разработчиков по всему миру способствовали развитию огромной инфраструктуры пакетов[8]. Таким образом, проектирование приложений заключается в разбиении на мелкие подзадачи, которые решаются с использованием готовых пакетов, что позволяет оптимизировать процесс разработки.

3. Благодаря тому, что JavaScript является интерпретируемым языком, программы, написанные с использованием Node.js, являются кроссплатформенными. Существует поддержка операционных систем, совместимых с ARM-процессорами, что делает возможным запуск кода даже на встраиваемых устройствах.

4. Язык программирования CoffeeScript является компилируемым в JavaScript языком, расширяющим возможности последнего за счет полноценной поддержки объектно-ориентированной парадигмы и добавления «синтаксического сахара». Синтаксические особенности языка делают возможным реализацию сложных паттернов проектирования, что является важным при разработке больших приложений[9].

1.2. Агент печатной станции

В представленной системе компонентом, который исполняет задания печати, является агент печатной станции, подключенный к центральному серверу. В задачи данного модуля входит:

1. Принятие заданий печати от центрального сервера, представленных

в виде изображения и метаинформации, в которую входят параметры печати и другие вспомогательные данные.

2. Работа с локальной очередью печати подключенного принтера. Постановка на печать полученного изображения.

3. Отправка отчета о статусе завершенного задания.

Компонент разработан на языке программирования CoffeeScript и работает под управлением фреймворка Node.js. В отличие в центрального сервера, который функционирует в режиме демона, подразумевается, что агент может быть запущен по требованию. Более того, одновременно могут быть доступны несколько печатных станций, разнесенных физически и представленных в виде отдельных экземпляров данного компонента.

1.3. Организация канала связи между сервером и агентом

При разработке мультиагентных систем особенно остро встает проблема организации канала связи, который обеспечивает взаимодействие агентов с сервером, распределяющим задания. При проектировании таких систем становится очевидным, что невозможно решить данную проблему, используя только возможности протокола TCP. Во-первых, следует учитывать, что выход в сеть чаще всего всего происходит посредством NAT или межсетевого экрана, что ограничивает возможность подключения (в таких условиях необходимо выделять центральный узел, чаще всего расположенный на выделенном сервере). Во-вторых, должен быть способ поддержания длительных сессий между участниками (например, опция keepalive[10] протокола TCP не реализована в старых версиях ядра Linux). Наконец, традиционная схема передачи данных не является удобной при реализации реактивных систем, взаимодействие компонентов в которой обычно организовано в виде двунаправленной передачи сообщений.

В для организации канала связи между сервером и печатной станцией в описываемой системе выбор был сделан в пользу свободно распространяемой библиотеки socket.io[11]. Данная библиотека предоставляет следующие возможности:

1. Организация дуплексного канала связи, который абстрагирован от среды запуска и транспорта. Возможно использование socket.io как в серверных и клиентских приложениях (написанных на Node.js), так и в веб-браузере. Для передачи данных могут использоваться технологии WebSockets или JSON Polling (периодический опрос сервера о наличии новых сообщений); переключение между этими транспортами является прозрачным.

2. Автоматический контроль жизнеспособности канала. Это реализовано при помощи отправки серии служебных сообщений (heartbeats) от клиента к серверу и наоборот. В случае обрыва соединения происходит попытка повторного подключения, причем существует стратегия экспоненциального увеличения интервала между неудачными попытками. Такое поведение доступно по-умолчанию, и чаще всего процесс восстановления соединения скрыт от программиста.

Узлы, использующие socket.io, обмениваются друг с другом сообщениями, которые сериализуются для отправки выбранным транспортом. Сообщение можно представить в виде кортежа $(T, a_1, a_2, ..., a_N)$, где T — текстовый идентификатор типа сообщения, $a_1, a_2, ..., a_N$ — сериализованные параметры сообщения.

Библиотека socket.io предоставляет также возможность реализации удаленного вызова процедур (англ. RPC — Remote Procedure Call) благодаря механизму подтверждений (англ. *acknowledgement*). Использование удаленного вызова процедур на примере функции возведения в квадрат представлено в листинге 1. Вызов удаленной функции происходит как отправка сообщения, последним параметром которого является функция обратного вызова, которая будет исполнена, как только удаленная сторона отправит ответ. Реализация удаленной функции также

содержит параметр-функцию, которую необходимо вызвать для возврата значения. Заметим, что этот параметр не является исходной функцией, которая была передана при отправке сообщения. Ее вызов приводит к отправке подтверждающего сообщения, содержащего результат (рис. 3).

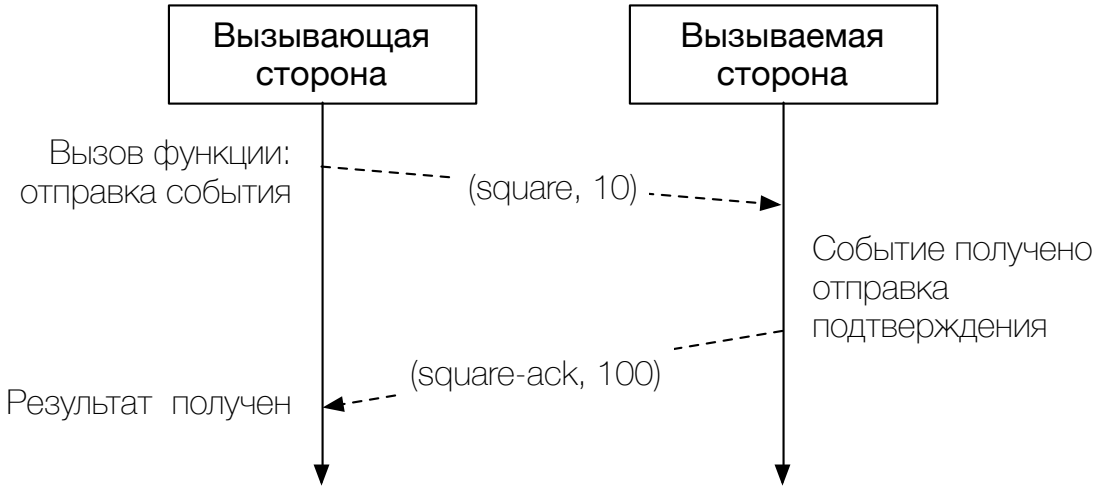

Рис. 3: схема организации удаленного вызова процедур

Листинг 1: Удаленный вызов процедур в socket.io

```
# RPC функция возведения числа в квадрат
socket.on "square", (n, callback) ->
  # Вызов переданной функции выглядит как вызов
  # обычной функции
  callback n * n


# Вызов RPC функции на клиенте выглядит как отправка сообщения.
# Первый аргумент: тип сообщения
# Второй аргумент: параметр сообщения
# Третий аргумент это функция реакции на acknowledgement
socket.emit "square", 10, (result) ->
  # Полученный результат: 100
  console.log result
```

При реализации агента печатной станции были использованы возможности удаленного вызова процедур для организации печати заданий, полученных от центрального сервера.

1.4. Веб-интерфейс

Для удобного управления очередью печати и мониторинга состояния подключенных станций разработан интерфейс, представляющий собой одностраничное веб-приложение, написанное на языке CoffeeScript. Взаимодействие интерфейса происходит через HTTP API центрального сервера с использованием парадигмы REST.

Веб-интерфейс построен с применением паттерна MVC («Model-View-Controller» — «Модель-Вид-Контроллер»)[12], реализованного в клиентском веб-фреймворке Chaplin[13]. Особенность реализации заключается в том, что подгрузка данных с сервера, их обработка, генерация представления этих данных и управление переходами между состояниями веб-приложения полностью происходит на стороне веб-браузера, что позволяет снизить нагрузку с центрального сервера. Кроме того, применение такого подхода означает, что собранное веб-приложение представимо в виде набора статичных файлов, что делает возможным использование кеширования для ускорения загрузки страницы.

2. Особенности реализации

В данной главе затронуты детали реализации отдельных компонентов системы, касающиеся организации печати в рамках печатной станции,очереди печати и описана модель хранимых в системе данных.

2.1. Организация печати

Возможность совершения печати агентом реализована благодаря низкоуровневой работе с системным спулером печати и классу-обертки, позволяющему получать уведомления в виде событий о статусе печати.

2.1.1. Работа с системной очередью печати

Взаимодействие с принтером происходит посредством постановки заданий в системную очередь печати (такую очередь называется *спулером* печати). В UNIX-подобных системах это осуществляется через CUPS («Common Unix Printing System» — система печати в UNIX). Система CUPS была изначально спроектирована так, чтобы обеспечить поддержку печати в условиях модели «клиент-сервер», то есть допускает использование одного принтера сразу несколькими компьютерами в локальной сети.

Ключевым понятием в CUPS является **получатель** (англ. *destination*) печати. Получателем называется доступный по сети принтер вместе с очередью печати. Фактически, любой локальный принтер в CUPS также воспринимается как сетевой. Печать файла представляет собой подключение к определенному получателю и постановку файла в очередь.

Взаимодействие с CUPS происходит через API в виде библиотеки на языке Си[14]. Для обеспечения возможности работы с CUPS из кода агента (реализованного на Node.js) был разработан модуль-обертка на C++. Node.js предоставляет разработчикам средства, позволяющие писать низкоуровневые модули, работа с которыми доступна в контексте

исполнения JavaSript[15].

Задача модуля — предоставление интерфейса для вызова функций CUPS, посредством функций, доступных из JavaScript. При этом следует учитывать, что переданные аргументы должны быть преобразованы из JavaScript-примитивов в типы данных, которые поддерживает CUPS (например, экземпляр JavaScript класса `String` должен быть преобразован к Си-строке). Подобные трансформации в данном модуле реализованы благодаря функция и классам библиотеки V8, которая является движком, используемым платформой Node.js.

Приведем список функций CUPS, поддержка которых была реализована:

`cupsGetDests` Возвращает список всех доступных получателей в локальной системе.

`cupsGetDefault` Возвращает идентификатор получателя по-умолчанию.

`cupsGetJobs` Возвращает список всех заданий печати в очереди заданного получателя.

`cupsPrintFile` Печать файла. Создает новое задание печати в очереди заданного получателя.

`cupsCancelJob` Отменяет заданное задание печати.

Разработанный модуль опубликован в виде пакета в официальном репозитории NPM под названием **cupsidity**. Исходный код модуля доступен в виде открытого репозитория и снабжен документацией[16].

2.1.2. Высокоуровневый класс-обертка

К сожалению, CUPS не предоставляет интерфейса, с помощью которого можно получать асинхронные события по изменению статуса заданий печати, что противоречит принципу реактивности, указанному среди требований к разрабатываемой системе. Единственным способом,

позволяющим узнать, было ли завершено конкретное задание, является периодический вызов функции `cupsGetJobs`.

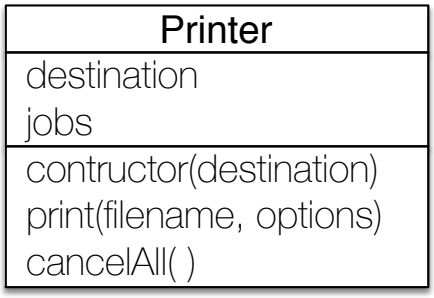

Рис. 4: класс-обертка Printer

Для предоставления асинхронного интерфейса для печати файлов был разработан класс-обертка `Printer` (рис. 4). Конструктор класса принимает параметр `destination`, являющийся названием принтера-получателя.

Листинг 2: Печать файла с помощью класса `Printer`

```
1  # Получаем доступ к принтеру
2  printer = new Printer config.get "printer:name"
3
4  # Печать файла
5  printer.print(filename, config.get "printer:options")
6  .then ->
7    log.info "Job successfully printed"
8    done null
9  .fail (err) ->
10   log.warn "Printing error: #{err}"
11   done "Printing error"
```

Печать заданного файла осуществляется посредством вызова функции `print`, принимающей путь к печатаемому файлу и необходимые параметры печати. Данная функция возвращает объект `Deferred`, который является одним из асинхронных примитивов, входящих в состав библиотеки Q[17]. `Deferred` — это представление завершенности отложенной асинхронной операции, которое может находиться в следующих состояниях: ожидается (англ. *pending*), завершено (англ. *resolved*), завершено

с ошибкой (англ. *rejected*). Пример вызова функции `print` и обработка состояний объекта `Deferred` представлены в листинге 2.

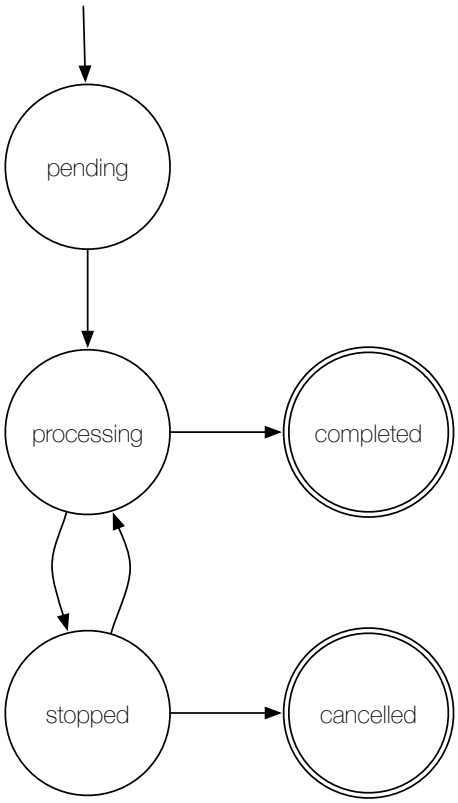

Рис. 5: диаграмма состояний задания печати

Вызов функции `print` приводит к созданию задания печати, которое может находиться в следующих состояниях:

`pending` Задание находится в очереди и ждет обработки. Это начальное состояние задания.

`processing` Задание исполняется (происходит печать).

`completed` Задание завершено (файл напечатан). Это терминальное состояние, которое свидетельствует об успешном завершении задания.

`stopped` Задание приостановлено. Причиной остановки могут быть проблемы, связанные с оборудованием и расходными материалами: закончилась бумаги или краска, связь с принтером прервалась и т.д.

`cancelled` Задание отменено. Терминальное состояние, свидетельствующее о том, что во время печати произошли ошибки.

Возможные переходы между описанными состояниями приведены на рис. 5. Класс `Printer` содержит приватную функцию `stateCheckRoutine`, которая вызывается с определенным интервалом времени и детектирует переход между состояниями заданий печати, которые были назначены на заданный принтер. Переход задания в одно из терминальных состояний означает завершение процесса печати и приводит к завершению соответствующего объекта `Deferred`.

2.2. Модель данных

Информация о зарегистрированных печатных станциях и фотографиях хранится в документо-ориентированной базе данных MongoDB. Для доступа к моделям данных используется библиотека Mongoose[18], предоставляющая объектно-документное отображение (англ. ODM — Object Document Mapping) документов базы данных в объекты на языке JavaScript. Несмотря на то, что данная база данных не требует задания схемы таблиц (документы MongoDB могут иметь совершенно произвольную структуру), использование Mongoose подразумевает определения схемы документов, которая включает список доступных полей вместе с типом поля. Такое требование открывает широкие возможности для проведения валидации объектов и контроля записываемых в базу значений.

Рассмотрим подробнее схему базы данных, используемой в системе (рис. 6).

Информация о конкретном агенте печатной станции хранится в модели `Station`. Каждая печатная станция имеет уникальное имя (поле `name`) и ключ (поле `secret`), используемый при подключении агента для авторизации. Кроме того, модель `Station` содержит описание (поле `description`) печатной станции для вывода в интерфейсе.

Фотография, которая попадает в очередь печати системы, представлена моделью `Shot`. Данная модель включает в себя: информацию о пользователе социальной сети, подпись к фотографии, ссылку на оригинальную запись, а также ссылки на сами изображения в разных размерах. Поле `hash` — это уникальный идентификатор фотографии в социальной сети, который используется модулем поиска новых фотографий для печати. Напечатанная фотография содержит поле `printedOn`, которая является ссылкой на печатную станцию, использовавшейся при печати. Модель также имеет текстовое поле `status`, характеризующее статус фотографии относительно очереди печати. Поле `status` может принимать следующие значения: `"initial"` (начальное состояние), `"queued"` (в очереди печати), `"printed"` (распечатана), `"failed"` (ошибка печати).

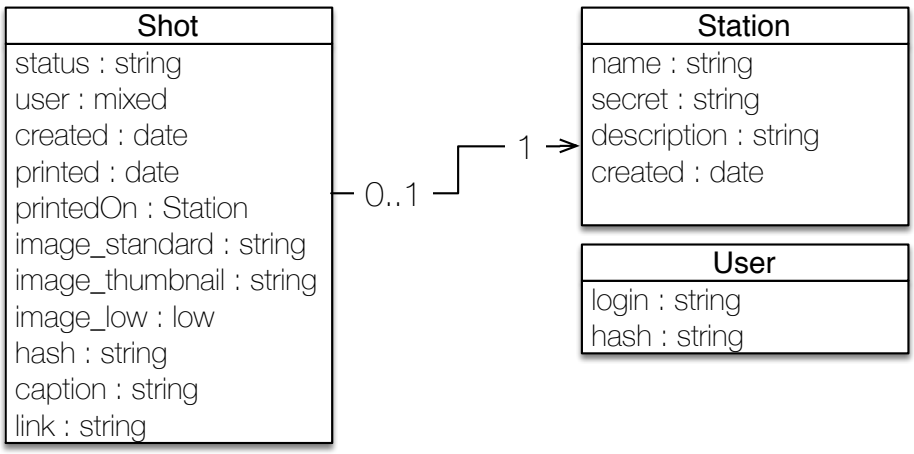

Рис. 6: схема базы данных

Модель `User` служит для хранения информации об администраторах системы, которые могут управлять очередью посредством веб-интерфейса.

2.3. Обработчики запросов от веб-интерфейса

Предоставление прикладного интерфейса по протоколу HTTP реализовано с помощью библиотек Express.js[19] и Passport.js[20]. Express.js

представляет собой легковесный серверный веб-фреймворк, а Passport.js служит для организации возможности авторизации.

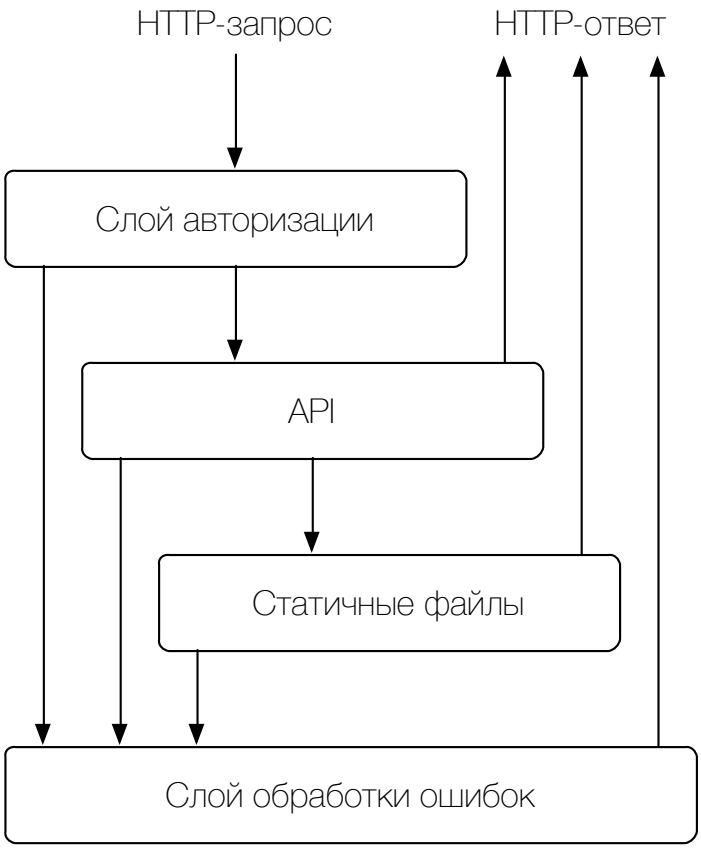

Рис. 7: схема обработки HTTP-запросов в приложении

Библиотека Express.js позволяет строить сложные веб-приложения на основе маршрутизации HTTP-запросов через так называемые промежуточные слои (англ. *middleware*). Каждый слой может на основе определенных условий послать ответ на запрос, а может пропустить запрос к следующему слою. Существуют слои, которые предоставляются самой библиотекой или сторонними пакетами: слой авторизации, слой статических файлов, слой обработки параметров форм и т.д. Кроме того, библиотека позволяет программисту определять собственные слои, которые работают на основе соответствия URL-адреса определенному шаблону (шаблоном может быть строка или регулярное выражение). Таким образом, веб-приложение представляет собой граф промежуточных слоев,

благодаря чему возможно построение гибких веб-приложений с понятной архитектурой.

Рассмотрим архитектуру промежуточных слоев, используемую в приложении (рис. 7). Слой авторизации необходим для определения пользователя, который в данный момент работает с системой (информация о пользователе хранится в модели `User`). Данный слой предоставляется библиотекой Passport.js и отвечает за обработку хранимых в Cookie сессий. Далее следует слой API, предоставляющий прикладные функции для работы с моделями. Наконец, слой статичных файлов служит для отдачи файлов клиентского веб-интерфейса. Любые ошибки, возникающие в слоях, перенаправляются в слой обработки ошибок (под ошибками подразумеваются ошибки времени выполнения, например, необработанные исключения).

Слой API реализует обработчики для управления моделями `Station` и `Shot` в соответствии с парадигмой REST. Это означает, что модели интерпретируются как ресурсы веб-сервера, над которыми можно выполнять действия (читать, создавать, удалять и обновлять), используя определенные соглашения о формировании запросов.

Рассмотрим методы, предоставляемые API для работы с ресурсом `Stations` (печатные станции):

`POST /stations` Регистрация новой печатной станции в системе. Данный метод требует права администратора.

`GET /stations` Получение списка всех печатных станций, зарегистрированных в системе.

`GET /stations/:id` Получение информации о конкретной печатной станции по идентификатору.

`PUT /stations/:id` Изменение полей печатной станции: описания, названия и ключа. Данный метод требует права администратора.

`DELETE /stations/:id` Удаление печатной станции по идентификатору. Данный метод требует права администратора.

Рассмотрим методы, предоставляемые API для работы с ресурсом `Shots` (фотография):

`GET /shots` Получение списка всех фотографий. Данный метод поддерживает: сортировку, фильтрацию по статусу и печатной станции и ограничение количества фотографий для показа (для организации постраничной загрузки). Параметры фильтров являются частью адреса, например запрос `GET /shots?limit=5&status=failed` вернет пять последних фотографий, которые не были напечатаны в результате ошибки.

`GET /shots/:id` Получение информации о конкретной фотографии по идентификатору.

`DELETE /shots/:id` Удаление фотографии по идентификатору. Данный метод требует права администратора.

`GET /shots/:id/queue` Помещение заданной фотографии в очередь печати. Данный метод требует права администратора.

Кроме того, API предоставляет следующие методы для авторизации:

`POST /auth/login` Метод, используемый для авторизации в системе. В случае успешной авторизации создается сессия, которая сохраняется между запросами с помощью Cookie.

`POST /auth/logout` Метод, используемый для очистки текущей сессии (выхода из системы).

`GET /auth` Данный метод возвращает информацию о текущей сессии. Он используется веб-приложением для проверки прав доступа.

Большая часть описанных методов использует формат JSON для формирования ответа. Ответ может содержать информацию о запрашиваемой сущности, а также дополнительную мета-информацию. Например, метод `GET /shots`, который может возвращать часть списка всех

фотографий для экономии трафика и организации постраничного вывода, также включает в результат мета-информацию о количестве всех фотографий с системе.


```
{
  meta: {
    query: {
      limit: "1",
      status: "failed"
    },
    limit: 1,
    total: 6,
    count: 1
  },
  shots: [{
    _id: "5394718ebf82f8205cbca783",

    image_low: "https://s3.amazonaws.com/4526gerg.jpg",
    image_standard: "https://s3.amazonaws.com/uws2a1.jpg",
    image_thumbnail: "https://s3.amazonaws.com/a7yfxe.jpg",

    user: {
      id: "67349823",
      name: "heisenberg",
      fullname: "Walter White",
      avatar: "https://s3.amazonaws.com/tfx9p0q43w.jpg"
    },

    caption: "I am the one who knocks!",
    status: "failed",
    created: "2014-06-08T14:22:06.527Z"
  }]
}
```

Рис. 8: пример ответа от сервера при вызове метода `GET /shots`

2.4. Веб-интерфейс и синхронизация моделей

Веб-интерфейс для управления заданиями печати представляет собой одностраничное веб-приложение, которое использует клиентский веб-фреймворк Chaplin[13].

В основе приложения лежит паттерн проектирования MVC, подразумевающий использование следующих компонентов:

- **Роутер** — обработчик глобального состояния приложения, в задачи которого входит осуществление перехода между страницами.
- **Модели** — локальное (на стороне веб-браузера) представление моделей, хранящихся в базе данных.
- **Вид** — отображение моделей на странице, которое изменяется при изменении атрибутов моделей.
- **Контроллер** — компонент, отвечающий за подгрузку данных с сервера и обеспечение связи между моделью и видом.

При разработки данного интерфейса была использована система сборки Brunch[21], которая позволила:

1. Организовать компиляцию организованных в модули скриптов на CoffeeScript в один минифицированный JavaScript файл. Был использован модульный подход к организации файлов в соответствии со спецификацией CommonJS[22].
2. Использовать для описания элементов интерфейса препроцессор стилей Stylus[23] и шаблонный язык Jade[24]. Язык Jade предназначен для формирования шаблонов, на основе которых во время выполнения генерируются части страницы. Препроцессор Stylus расширяет возможности каскадных таблицей стилей (CSS), позволяя использовать так называемые «примеси» и обеспечивая поддержку нестандартизированных свойств во всех современных браузерах.
3. Обеспечить поддержку автоматической перезагрузки страницы (технология LiveReload) при сохранении исходных файлов. Применение такого подхода позволило сделать процесс разработки максимально прозрачным и гибким.

Описываемое приложение имеет следующие состояния:

- Форма авторизации.
- Список печатных станций.
  - Страница конкретной печатной станции с описанием.
  - Страница редактирования станции.
- Галерея фотографий, содержащая напечатанные и находящиеся в печати фотографии.

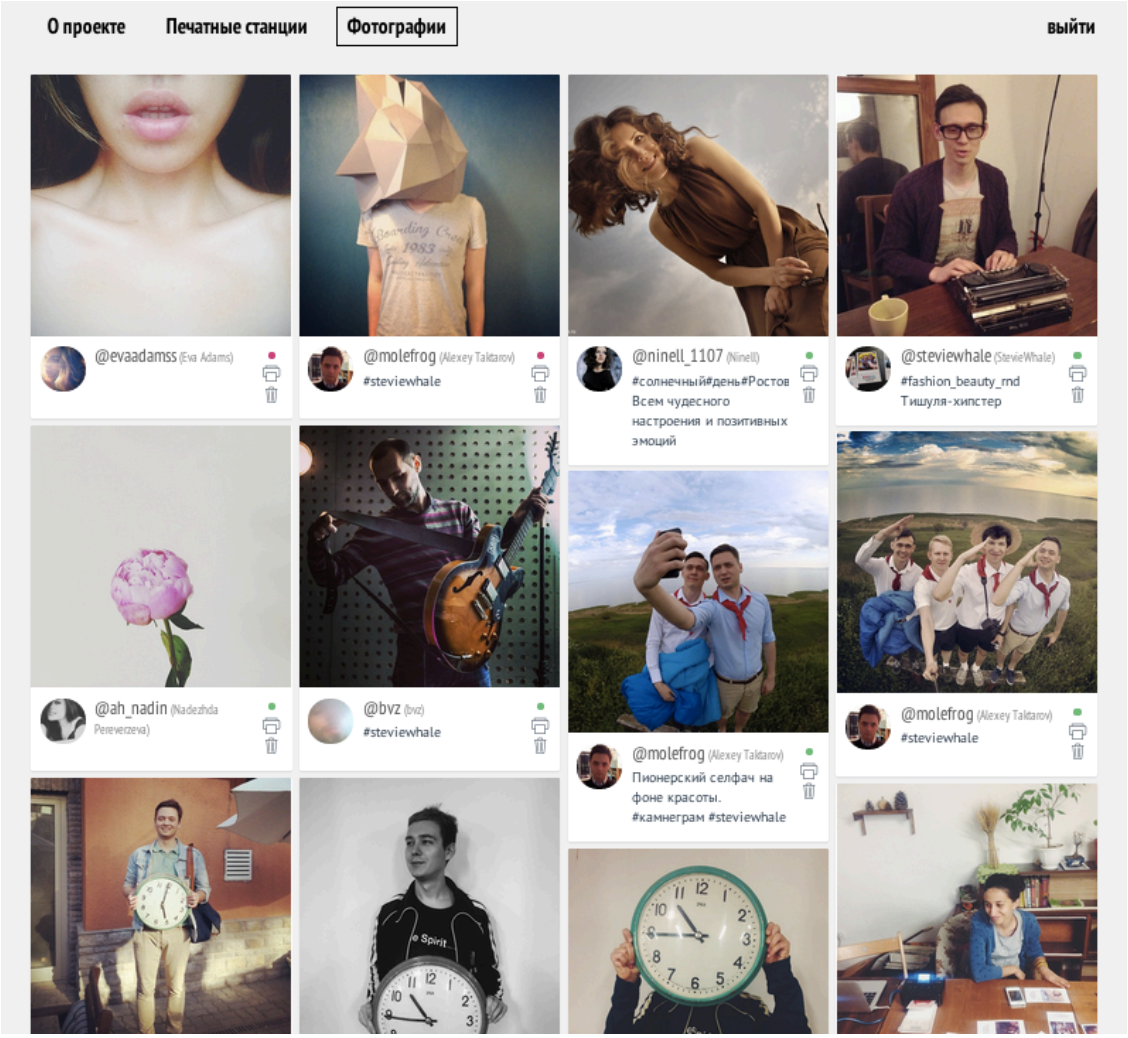

Рис. 9: интерфейс галереи фотографий

Центральным компонентом является галерея фотографий (рис. 9), которая одновременно служит архивов напечатанных фотографий и средством мониторинга процесса печати в режиме реального времени. Для компоновки фотографий в виде сетки была использована библиотека Masonry[25]. Одним из требований при реализации этого компонента была возможность моментального получения уведомлений об изменении фотографий на сервере. Подобная синхронизация моделей была достигнута благодаря использованию транспорта, который предоставляется библиотекой socket.io[11].

Изменение модели фотографии на сервере может привести к рассылке всем подключенным через веб-интерфейс клиентам следующих сообщений:

- `shot.updated`, когда статус фотографии был изменен (например, фотография перешла из статуса «в очереди» в статус «напечатана»)
- `shot.created`, когда новая фотография была добавлена в очередь печати.
- `shot.removed`, когда фотография была удалена.

Обработка серверных событий синхронизации происходит через глобальный объект **Медиатор**[26]. Данный компонент принимает уведомления, поступающие от сервера посредством канала socket.io. Отдельные части интерфейса (в данном случае вид галереи фотографий) подписываются на события, которые генерирует Медиатор, и производят соответствующие обновления отображения. Применение такого подхода позволило снизить зависимость частей приложения друг от друга и добиться максимальной гибкости при разработке.

2.5. Очередь печати фотографий

Распеделение задач печати между подключенными к системе агентами осуществляется посредством глобальной очереди печати, функци-

онирующей в рамках центрального сервера. Очередь является незаменимым средством при проектировании многокомпонетных систем, в которых происходит распределение вычислений среди нескольких узлов. Кроме того, использование очередей позволяет реализовать следующие схемы взаимодействия: очередь заданий (англ. *work queue*), отправка сообщений подписчикам[27] (англ. *publish/subscribe*) и удаленный вызов процедур[28].

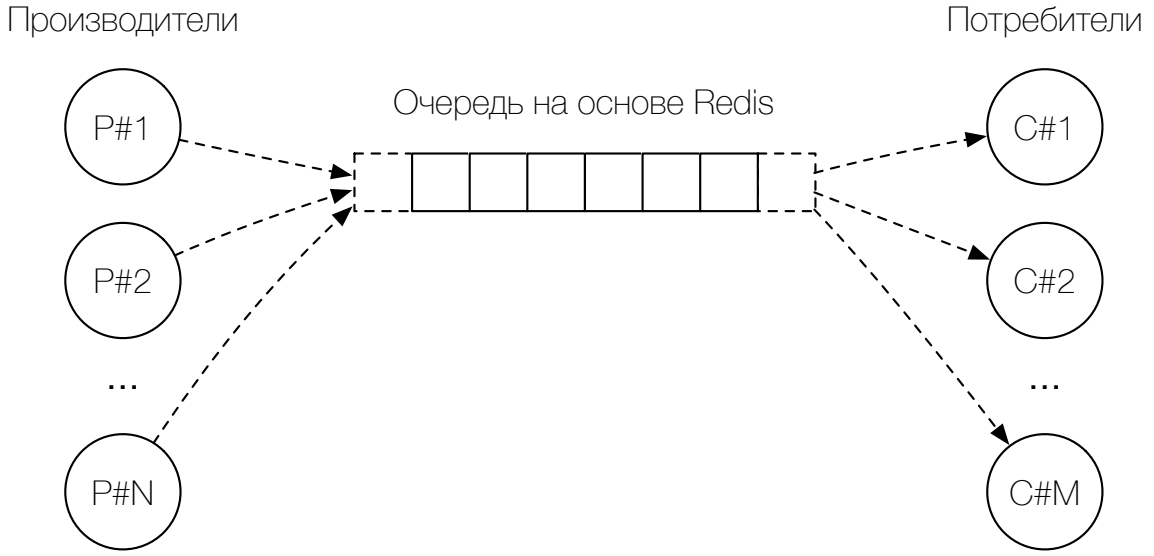

Рис. 10: схема работы очереди «производитель-потребитель»

В описываемой системе была использована очередь заданий, реализованная при помощи библиотеки Kue[29] и базы данных Redis. Данная очередь работает по принципу «производитель-потребитель». Компоненты приложения, называемые **производителями**, генерируют задания и помещают их в очередь, к которой имеют доступ **потребители**, извлекающие задания и исполняющие их (рис. 10). Использование хранилища Redis позволяет сохранять информацию о заданиях после перезапуска центрального сервера, кроме того примение данной схемы приводит к возможности разбиения приложения на несколько независимых процессов, каждый из которых выполняет заданную функцию (создает задания или исполняет их).

Опишем последовательно реализацию компонентов центрального сервера, взаимодействующих с очередью печати.

2.5.1. Модуль поиска фотографий в социальной сети

Новые фотографии для печати поступают в систему при помощи публикации в социальной сети Instagram. Однако, необходим способ, который бы позволил отделить фотографии для печати от общего потока фотографий в сервисе. Это может быть достигнуто при помощи добавлении специальной метки к описанию фотографии — *хештега*.

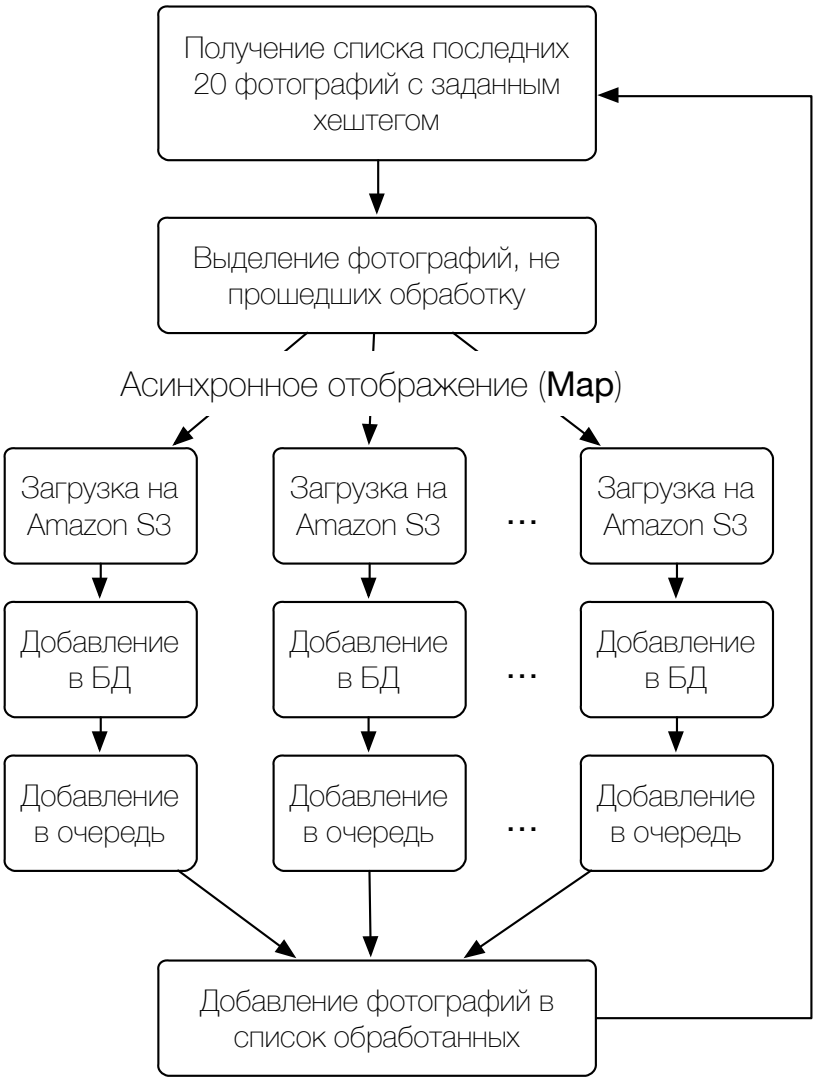

Рис. 11: схема работы модуля поиска фотографий

В рамках описываемой системы реализован модуль поиска новых фотографий в социальной сети, которые содержат заданный хештег. Для этого были использованы возможности публичного API сервиса Instagram. Принцип работы данного модуля заключается в периодическом опросе сервиса на предмет наличия новых фотографий с указанным хештегом.

Сервис Instagram предоставляет функцию `tags.recent`, возвращающую 20 последних записей, помеченных заданным хештегом. Полученная порция фотографий фильтруется модулем для выделения тех фотографий, которые еще не прошли обработку. Далее, список новых фотографий с помощью асинхронного паттерна Map (отображение), реализованного в библиотеке Async[30], проходит обработку (рис. 11). Прежде всего, происходит загрузка изображений в облачное хранилище файлов Amazon S3 (для того, чтобы избежать потери данных в результате удаления фотографии в социальной сети), затем создаются записи в базе данных (используется модель `Shot`). Наконец, идентификаторы созданных записей помещаются в очередь печати.

2.5.2. Обработка заданий печати

Опишем алгоритм обработки заданий печати, поступивших в очередь:

Шаг 1. Загрузить из базы данных соответствующую данному идентификатору модель `Shot`.

Шаг 2. Выбрать доступную печатную станцию из списка подключенных. При отсутствии станций пометить данную фотографию как ненапечатанную.

Шаг 3. Сформировать на основе данных о фотографии изображение для печати.

Шаг 4. Загрузить сформированное изображение в облачное хранилище файлов Amazon S3.

Шаг 5. Вызвать удаленную процедуру `print` на выбранной печатной станции, передав в качестве параметра ссылку на сформированное изображение.

Шаг 6. После завершения печати удалить изображение из хранилища и обновить статус фотографии в базе данных.

Рис. 12: пример сформированного для печати изображения

Печатаемое изображение содержит помимо самой фотографии также информацию о пользователе, опубликовавшем запись в социальной сети (рис. 12)). Для формирования данного изображения используется библиотека node-canvas, представляющая собой реализацию методов HTML5 Canvas для Node.js. Отличительной особенностью такого реше-

ния является возможность прямой загрузки результирующего изображения в облачное хранилище, минуя файловую систему (с использованием потоков ввода-вывода).

3. Интеграция

Разработка больших систем часто сопряжена с трудностями запуска работающей версии продукта в производство. Причинами могут являться: сложность развертывания компонентов, недостаточная стабильность работы системы и несоответствие окружения, в котором происходит разработка, и среды, предполагающей долгую работу приложения.

Данная глава посвящена описанию решений, с помощью которых были решены приведенные проблемы в рамках разработанного комплекса. За основу был взят принцип «12-факторного приложения»[31], представляющий собой методологию построения поддерживаемых и масштабируемых облачных сервисов.

3.1. Разделение окружений

В описываемой системе был применен подход, заключающийся в строгом разделении окружений запуска. Под окружением понимается совокупность конфигураций приложения и настроек, связанных с интеграцией сторонних сервисов. Таким образом, система может функционировать в рамках следующих окружений:

- `development` — окружение, используемое при разработке.
- `production` — окружение, используемое при развертывании приложения для запуска в производство.
- `testing` — окружение, используемое для запуска тестов.

Конкретное окружение может быть выбрано при запуске центрального сервера при помощи аргументов командной строки (параметр `env`)

или переменных среды (переменная `NODE_ENV`). Каждое окружение ассоциируется с конфигурационным файлом, который содержит следующие настройки:

- Адреса для подключения к базам данных MongoDB и Redis. При разработке и запуске используются разные экземпляры баз данных.
- Настройки, касающиеся веб-сервера (например, используемый порт).
- Секретные ключи доступа к API социальной сети Instagram.
- Настройки доступа к облачному хранилищу файлов Amazon S3.

3.2. Тестирование и непрерывная интеграция

Для контроля стабильности разработанной системы были написаны тесты с использованием фреймворка тестирования Mocha. Этим достигается покрытие кода, отвечающего за предоставление прикладного интерфейса для доступа к моделями `Shot` и `Station`. Рассмотрим этапы прохождения тестов:

Шаг 1. Запуск системы в рамках окружения `testing`. На данном этапе происходит подключение к тестовой базе данных (чаще всего это локальная база данных).

Шаг 2. Очистка базы данных и генерация произвольных моделей с использованием фабрик.

Шаг 3. Прохождение тестовых спецификаций, для каждого предоставляемого метода. На данном этапе происходит исполнение HTTP-запроса и сравнение результата с ожидаемым значением.

Во время разработки системы были использованы возможности облачного сервиса Travis CI для организации непрерывной интеграции. Любое изменение кодовой базы на удаленном репозитории приводило к автоматическому прохождению тестов и отправки отчетов в случае

ошибки. Использование подобной методологии позволило гарантировать стабильность системы на всех этапах разработки.

3.3. Процедура развертывания

В рамках разработанной системы была реализован процесс автоматического развертывания при сохранении кода в удаленный репозиторий. Это возможно благодаря событиям системы контроля версий Git и скрипту на языке Bash, выполняющему необходимую для запуска новой версии программы подготовку[32].

Опишем шаги, которые входят в процедуру развертывания:

Шаг 1. Остановить работающую версию приложения.

Шаг 2. Установить необходимые для веб-интерфейса и серверной части зависимости посредством пакетных менеджеров NPM и Bower.

Шаг 3. Выполнить сборку статичных файлов веб-интерфейса.

Шаг 4. Запустить обновленную версию приложения.

Заключение

Список литературы

1. B. Wasik. In the programmable world, all our objects will act as one. http://www.wired.com/2013/05/internet-of-things-2/, 2013.

2. Nest world's first learning thermostat. https://nest.com/.

3. М. Слюсаренко, И. Слюсаренко. Системологический подход к декомпозиции в объектно-ориентированном анализе и проектировании программного обеспечения. 2010.

4. K. Seguin. *The Little MongoDB Book*. 2012.

5. A. MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, 2012.

6. A. Hall. Understanding the node.js event loop. http://strongloop.com/strongblog/node-js-event-loop/, 2013.

7. C. Kohlhoff. The proactor design pattern: Concurrency without threads. http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/overview/core/async.html, 2011.

8. C. Robbins. Npm: innovation through modularity. http://blog.nodejitsu.com/npm-innovation-through-modularity/, 2013.

9. D. Brady et al. Coffeescript cookbook. http://coffeescriptcookbook.com/, 2014.

10. TCP keepalive overview. http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/overview.html.

11. Socket.io real-time bidirectional event-based communication. http://socket.io/.

12. C. Robbins. Scaling isomorphic javascript code. http://blog.nodejitsu.com/scaling-isomorphic-javascript-code/.

13. Chaplin HTML5 application architecture.
    http://chaplinjs.org/.

14. Introduction to CUPS programming.
    http://www.cups.org/documentation.php/api-overview.html.

15. Node.js documentation addons.
    http://nodejs.org/api/addons.html.

16. Cupsidity — CUPS (common unix printing system) native bindings for Node.js.
    https://github.com/molefrog/cupsidity.

17. Q — a tool for making and composing asynchronous promises in JavaScript.
    https://github.com/kriskowal/q.

18. Mongoose — elegant MongoDB object modeling for node.js.
    http://mongoosejs.com/.

19. Express — web application framework for Node.js.
    http://expressjs.com/.

20. Passport — simple, unobtrusive authentication for Node.js.
    http://passportjs.org/.

21. Brunch — is an ultra-fast HTML5 build tool.
    http://brunch.io/.

22. CommonJS notes.
    http://requirejs.org/docs/commonjs.html.

23. Stylus — expressive, dynamic, robust css.
    http://learnboost.github.io/stylus/.

24. Jade — node.js template engine.
    http://jade-lang.com/.

25. Masonry — cascading grid layout library.
http://masonry.desandro.com/.

26. A. Osmani. Patterns for large-scale javascript application architecture.
http://addyosmani.com/largescalejavascript/, 2011.

27. RabbitMQ tutorial.
http://www.rabbitmq.com/tutorials/tutorial-one-python.html.

28. М. Таненбаум, Э. Стеен. *Распределенные системы. Принципы и парадигмы.* Питер, 2003.

29. Kue — feature rich priority job queue for Node.js backed by Redis.
http://learnboost.github.io/kue/.

30. Async utilities for Node.js and the browser.
https://github.com/caolan/async.

31. A. Wiggins. The Twelve-Factor App.
http://12factor.net/.

32. K. Jordan. Setting up Push-to-Deploy with git.
http://krisjordan.com/essays/setting-up-push-to-deploy-with-git.