ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет информатики, математики и компьютерных наук**

**Программа подготовки бакалавров по направлению**

**09.03.04 Программная инженерия**

*Бакаев Никита Александрович*

**КУРСОВАЯ РАБОТА**

Интернет-магазин (серверная часть, инфраструктура)

|  |  |
| --- | --- |
|  | Научный руководитель  Старший преподаватель  Сорокоумов Андрей Викторович |

Нижний Новгород, 2016

**Содержание**

1. Введение 2

2. Архитектура приложения 3 3. REST API 4

4.Swagger 6

5. MVC 7

6. Data Layer. Repository 9

7. Основные модели 11

8. MongoDB и sharding 13

9. Docker 15

10. Параметры для запуска 16

11. Запуск 18

12. Загрузка файлов. AWS S3 19

13. Тестирование 21

14. Continuous Integration 23

15. Доменная область 25

16. Защита информации 27

17. Дальнейшее развитие 28

18. Заключение 30

Список литературы 31

Приложения 32

**Введение**

В современное время развития IT технологий многие аспекты человеческой жизни перетекают в Интернет. Одним из таких проявлений может служить коммерция: продажа товаров и услуг. До развития Интернета, продажа осуществлялась оффлайн. Однако с появлением сети Интернет, как связующего звена множества людей, появился еще один канал коммуникации людей. В последние годы, серьёзное развитие получило направление **e-commerce,** электронная коммерция, продажа товаров через Интернет.

Площадками для таких продаж служат множественные интернет-магазины, как российские, так и зарубежные, глобальные. О разработки серверной части интернет-магазина и пойдет речь в данной работе. Разработка интернет-магазина включает множество составляющих, однако в данной работе вниманию будет уделен лишь технический аспект данного вопроса.

Основная часть

Архитектура приложения

Приложение написано на **Java** и использует Spring как Dependency Injection Container. Активно используется паттерн **MVC** для REST. **DI** через конструкторы, где это возможно и через сеттеры в остальных местах.

Для запуска приложения используется **Spring Boot**. Это современное решение, созданное для создания микросервисов и включает в себя embedded сервер приложений, автоконфигурация некоторых популярных решений, включая логирование и многое другое. В данном случае в качестве встроенного сервера используется embedded tomcat. Это позволяет запускать приложений без дополнительного уровня - сервер приложений и deploy на него. При этом возможен полный контроль над встроенным сервером.

Приложение рассчитано на работу по REST и не использует состояний на сервере, как сессии и прочее. Это позволит просто использовать горизонтальное масштабирование в будущем. Все изменений сбрасываются в persistence storage (MongoDB).

В качестве базы данных используется **MongoDB**. Именно эта NoSQL база данных выбрана в качестве основной из-за высокой скорости работы, возможности горизонтального масштабирования из коробки и отсутствия явной схемы на стороне базы данных.

REST API

Основой архитектуры служит клиент-серверная архитектура. Клиент (браузер, JavaScript) обращаться к серверу через публичный API (средствами асинхронных запросов). API представляет из себя REST сервис. Все endpoints версионируются по URL. Базовый URL для всех запросов на данный момент *протокол:адрес\_сервера:порт/api/v1/* . Возмодна работа только по HTTPS. Например, https://s2.nbakaev.ru/api/v1/ .Все запросы будут иметь ввиду именно этот адрес как относительный.

Для передачи данных по умолчанию используется распространённый формат сериализации JSON. Однако, возможно использование XML, в случае использования при запросе HTTP заголовка Accept: application/xml.

**Пользователи:**

- Получение всех пользователей ( GET /users ). Только админ

- Регистрация (Добавление нового пользователя). Требуется email и пароль для нового пользователя ( POST /users ). Добавляет пользователя и отправляет на указанный email приветственное письмо об успешной регистрации

- Обновление данных о пользователе ( PUT /users ). Только админ

- Удалить пользователя ( DELETE /users/{id} где {id} - ID пользователя ). Только админ

**Общее:**

GET / - Health-checking

**Покупки:**

* Получить список всех покупок (GET / purchase). Только админ
* Совершить покупку (POST / purchase).
* Редактировать данные о покупках (PUT / purchase). Только админ
* Удалить покупку ( DELETE /users/{id} где {id} - ID покупки). Только админ

**Товары:**

* Получить список всех товаров (GET /good).
* Добавить товар (POST /good). Только админ
* Редактировать данные о товаре (PUT /good). Только админ
* Удалить товар( DELETE /good/{id} где {id} - ID товара). Только админ
* Получить список товаров, которые нужно показать на главной странице (GET /good/category/index).
* Получить список товаров по фильтру (POST /good/filter). Можно использовать регулярные выражения по одному или более полям (например поиск по имени или описанию), фильтры по цене, дате создания, категориям и многое другое. Поддерживается постраничный вывод (pagination)
* выгрузка товров в excel (POST /good/excel/download) для выгрузки по определенному критерию (критерии аналогичны API POST /good/filter), либо (GET /good/excel/download.xlsx) для выгрузки всех товаров.

**Хранение данных:**

* POST /storage/upload. Загрузка любого файла в облачное хранилище AWS S3. Подробнее в секции AWS S3. Возвращается публичная ссылка на загруженный файл.

Swagger

В идеи REST есть одна серьёзная проблема. В отличии от протоколов, где есть валидация данных и жесткая структура, в REST фактически нет никаких соглашений те клиент API фактически не знает, что ему вернет сервер в ответе. Остаётся лишь надеяться, что документация API актуальна. Кроме того, невозможно сформировать клиента(или SDK) под любой язык из-за проблем выше. Как способ решения этой проблемы есть swagger.

Swagger определяет структуру обмена сообщений, включая HTTP URL, объекты, что передаются и прочее. По сути, это JSON файл в котором определен актуальный API сервера.

Кроме того, этот JSON имеет определенную структуру и благодаря этому возможно автоматически сгенерировать клиенты[[1]](#footnote-0)(SDK) под более чем 40[[2]](#footnote-1) языков программирования и платформ.

MVC

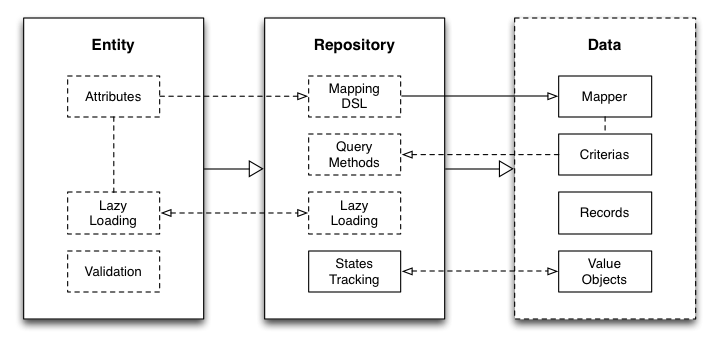
Model-view-controller (MVC, «модель-представление-контроллер», «модель-вид-контроллер») — шаблон проектирования, с помощью которых модель приложения, пользовательский интерфейс и взаимодействие с пользователем разделены на три отдельных компонента таким образом, чтобы модификация одного из компонентов оказывала минимальное воздействие на остальные.



Реализация паттерна MVC и пример HTTP контроллера для работы с объектом товар.

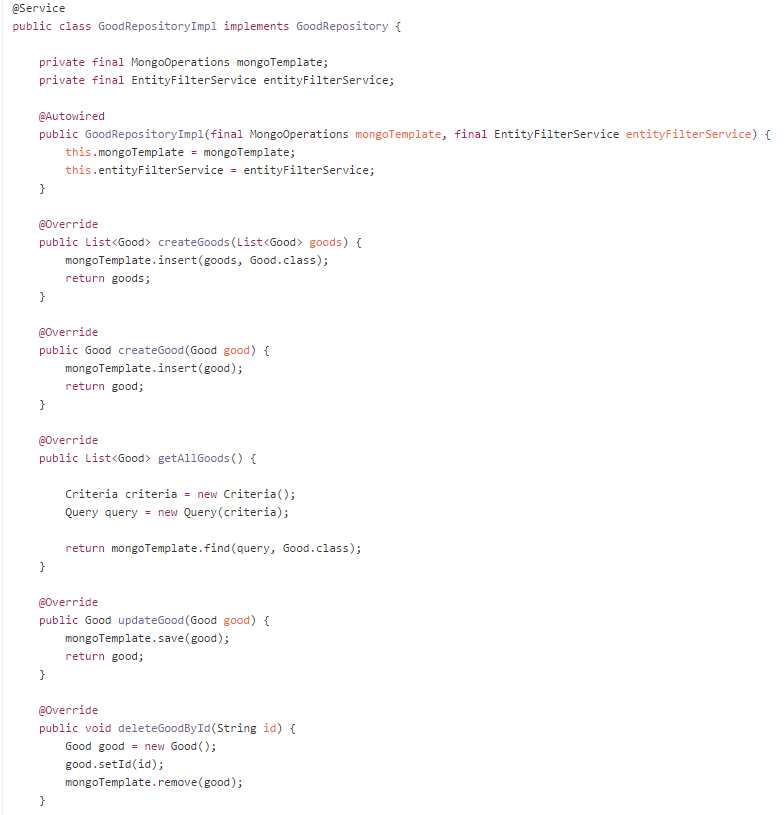


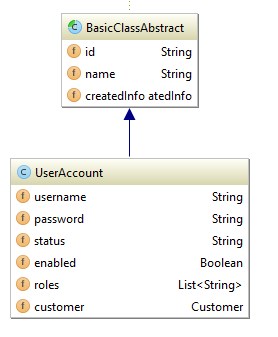
Data Layer. Repository

Репозиторий - это паттерн, представляющая собой идею хранения коллекции для сущностей определенного типа. Другими словами, для доступа к объектам определенного типа используется сервис посредник, который выполняет работу с базой данных и знает как выпонять различные операции, например, CRUD в базе данных и этим объектом.

Этот паттерн позволяет абстрагироваться от конкретной реализации системы хранения в базе данных и зачастую реализует Object-relational mapping (ORM) для создания объектов в конкретном языке программирования из данных на основе данных из иного хранилища.

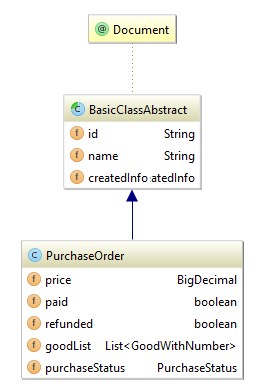
Пример сервиса по работе с объектом товар. CRUD операции.



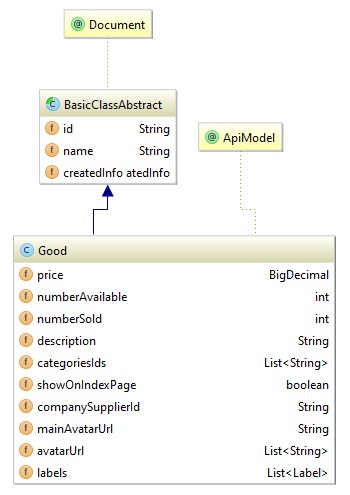
Основные модели

**Пользователь**

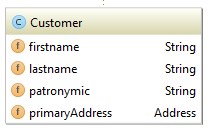
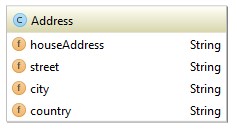
**Заказ**



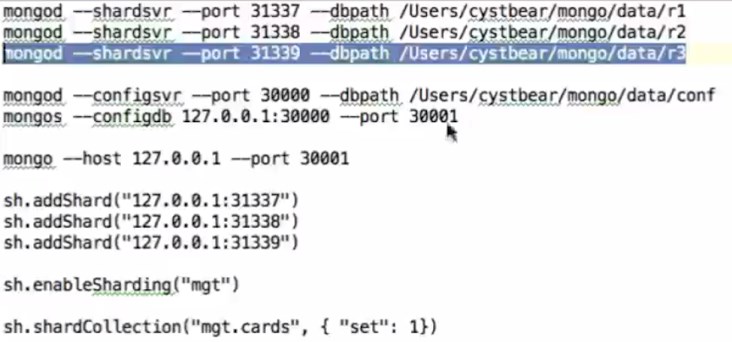
**Товар**



**Клиент и адрес клиента**



MongoDB и sharding

MongoDB — NoSQL документо-ориентированная система управления базами данных (СУБД) с открытым исходным кодом, не требующая описания схемы таблиц. Написана на языке C++.

Для проекта используется база данных MongoDB. Основными преимуществами которой является schemaless и шардирование из коробки.

Шардирование (горизонтальное партиционирование) — это принцип проектирования базы данных, при котором логически независимые строки таблицы базы данных хранятся раздельно, заранее сгруппированные в секции, которые, в свою очередь, размещаются на разных, физически и логически независимых серверах базы данных, при этом один физический узел кластера может содержать несколько серверов баз данных.[[3]](#footnote-2)

В примере выше запускается 3 ноды(инстанса mongod) базы данных на разных портах с разными папками для хранения данных. Создается configsrv - конфиг сервер. А затем на mongos - mongo shard master добавляются в шардинг 3 ноды с ip адресом и портом. И следующей строчкой включается шардирование между всеми участниками шарда на коллеции ‘mgt’.

Docker

Одной из важных задач, которая была решена и необходима для работы программы: простота запуска и установки (installation)

В работе используется механизм контейнеризации на основе Docker. Это позволяет:

* Независимо запускать несколько экземпляров приложения одновременно
* Не зависит от окружения (dependency hell). Приложения не требует ни установленной непосредственно Java на компьютере, ни базы данных. Кроме того, если они установлены, хоть и другой версии, например, то не будет никаких конфликтов.
* Переносимость. Образ(контейнер) легко использовать несколькими людьми, не пережимая о различиях в конфигурации операционной системы.

Веб-приложение может запускать отдельно через командную строку: *java -jar target/ru.nbakaev.hishop.jar*

Затем запустить базу данных. Например через команду: *mongod --dbpath "/data/db" --storageEngine wiredTiger --config "/etc/mongod.conf"*

Сам docker файл для построения выглядит следующим образом

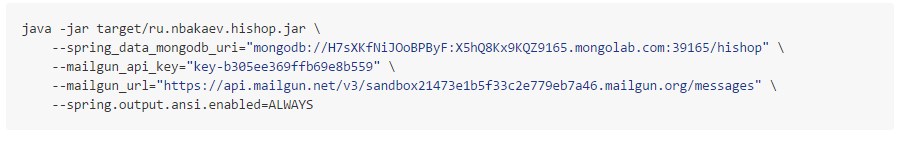


Параметры для запуска

Для запуска приложения необходимо передать программе конфигурационные параметры. Среди которых:

* MongoDB база данных
* Mailgun (сервис для отправки email сообщений с DKIM и спам фильтром)
* AWS credentials (API ключ для загрузки файлов на AWS S3)

Эти параметры могут быть переданы несколькими способами:

* Через переменное окружение
* Через аргументы командной строки для программы
* В случае запуска через docker через переменные окружения
* В конфигурационных файлах application.properties, application-development.properties или в application-production.properties в записимости от активного профиля в spring, конфигурируемого через spring.profiles.active

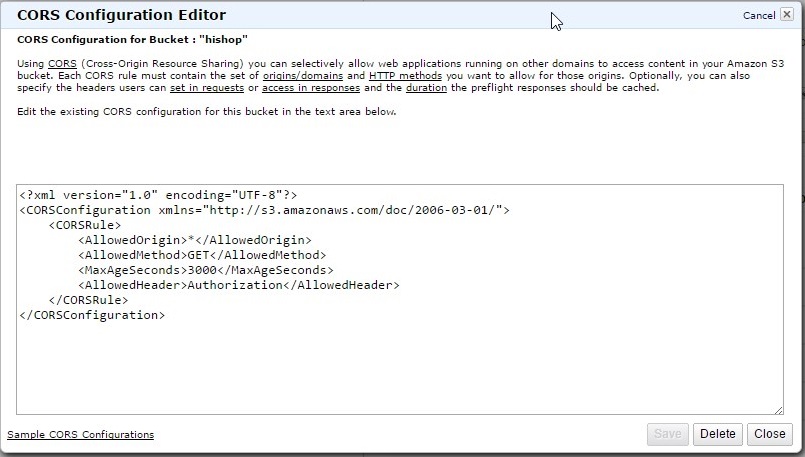
Запуск

Запуск поддерживается на всех операционных системах, на которых возможен запуск Java приложений, включая Windows, Linux, Mac OS. Платформозависимый код отсутствует. Возможно много вариантов запуска непосредственно самого приложения:

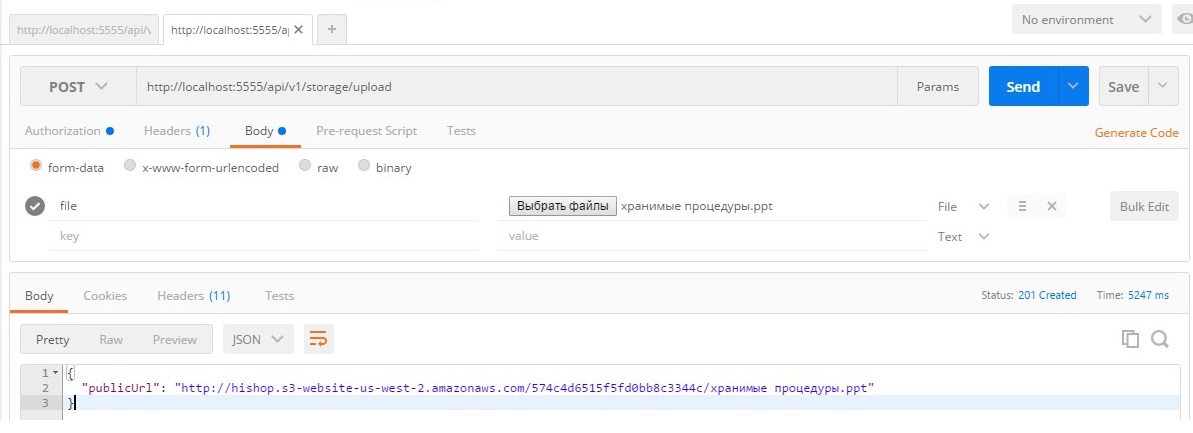
* запуск как обычый **jar файл** в операционной системе с jvm. Например java -jar hishop.jar
* Запуск **docker контейнера**. Например docker run -d nbakaev/hishop-api-backend
* Запуск в PaaS **Heroku**. В корне git репозитория есть файл Procfile для запуска.
* Запуск в облаке Amazon: **AWS Beanstalk**. Этим сервисом поддерживается как запуск jar файлов, так и docker контейнеров, как описано выше
* в git репозитории присутствуют файлы для запуска в системе оркестрации docker контейнерами kubernetes (pod и service)

Загрузка файлов. AWS S3

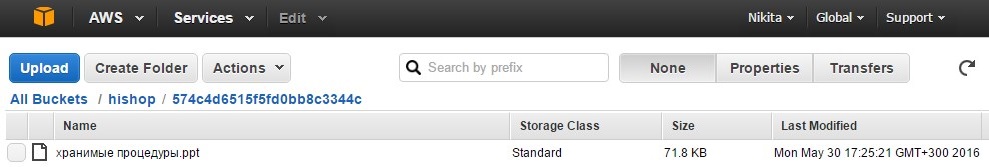
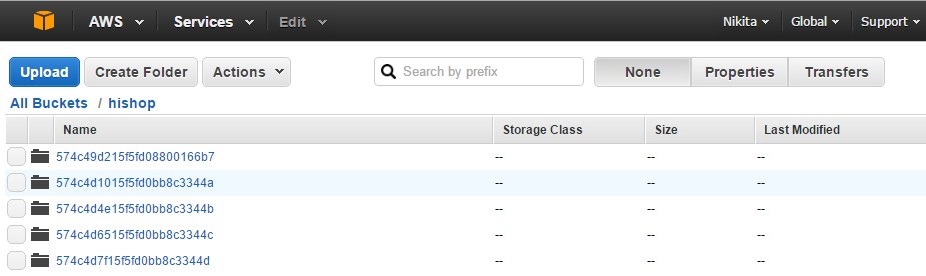
Для загрузки файлов используется публичный AWS bucket, названный hishop. Он поддерживает CORS и доступен публично. Загрузка происходит через backend API.

Конфигурация выглядит следующим образом:



Пример запроса и ответа от сервера. Сервер возвращает публичную ссылку на загруженный файл. Загрузка разрешена пользователем справами администратора.

Каждый файл загружается в отдельную “папку” (в архитектуре AWS S3 - это не папка а лишь путь). Список всех загруженных файлов.



И вот только что загруженный файл.

Тестирование

Для поддержания высокого качества кода и избежании регрессии в производительности и функционале используется тестирование кода. Установлен уровень **покрытия кода тестами 90%**. Используется следующие методы тестирования:

* **Unit тесты**. Тестируются отдельные фунции, Spring компоненты. В случае необходимости, используется мокирование методов объектов с помощью фреймворка **Mockito**.
* **Интеграционные тесты**. Используется **TestNG** и spring-tests фреймворки для тестов. Используется для тестирования корректности связанности компонент системы. В данном случае поднимается полностью Spring container со всеми бинами и DI. И тестируются как выполнение HTTP REST контроллеров, так и отдельных модулей.
* Создание процесса базы данных **MongoDB** полностью чистой (без каких либо коллекций или документов) при каждом запуске тестов. Это полноценная база данных, запускаемая автоматически. Это используется для полной проверки корректности работы с реальной базой данных. При этом - это просто отдельная тестовая база данных и она совершенно никак не связана на production.

Пример теста на загрузку файла в AWS S3. В этом примере мы делаем интеграционный тест. Считываем данные с локальной файловой системы в byte[]. Затем делаем запрос к нашему API контроллеру, как администратор. Получаем ссылку на загруженный в s3 файл. Затем делаем HTTP запрос и получаем этот файл и сравниваем загруженный файл с тем, что у нас на файловой системе.

Continuous Integration

Важный аспект при разработки ПО это постоянный мониторинг корректности работы актуального кода, включая контроль регрессии производительности и некорректность работы программы. С этой целью используется метод из технологий гибких методологий разработки ПО, а именно постоянный контроль работоспособности текущего программного кода.

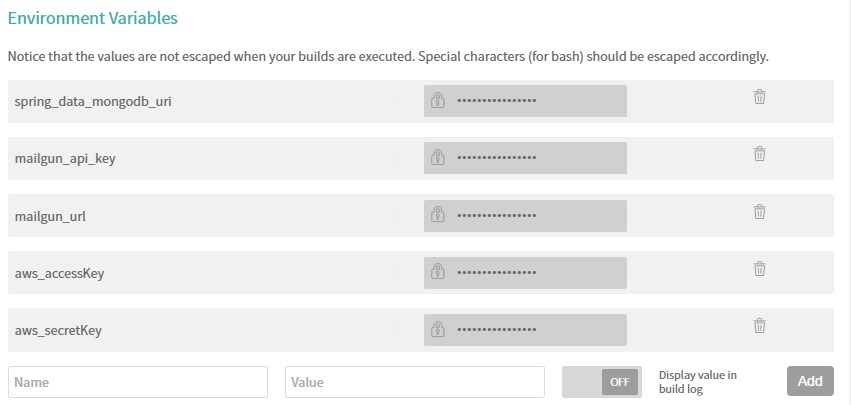
С этой целью, каждый коммит в публичный git репозитория на github вызывает webhook на тестирование билда на CI сервере. В данном случае, в качестве CI сервера выбран бесплатный для open source проектов сервис **travis ci**. При этом, каждый билд вызывает следующие действия:

* сборка проекта с помощью **maven** (команда mvn package). В результате создается полностью исполняемый jar файл.
* выполнение всех java тестов. Это выполняется просто с помощью команды mvn test
* в случае успешного прохождения теста - загрузка jar артефакта в публичный maven binary repository (**bintray**)
* построение docker контейнера и загрузка в публичный репозиторий **docker hub**.
* отправка уведомления о результате работы в slack: время сборки, успешно прошли тесты(или сборка билда в целом) или нет.

Для описанной выше работы разработан bash скрипт, доступный в репозитории.

Так как некоторые действия, как загрузка в maven репозиторий или загрузка в docker hub требует данных для аутентификации, а репозиторий публичен, то эти данные не могут безопасно храниться вместе с кодом и с этой целью, данные передаются через переменные окружения.

Ниже представлен интерфейс Travis CI



Доменная область

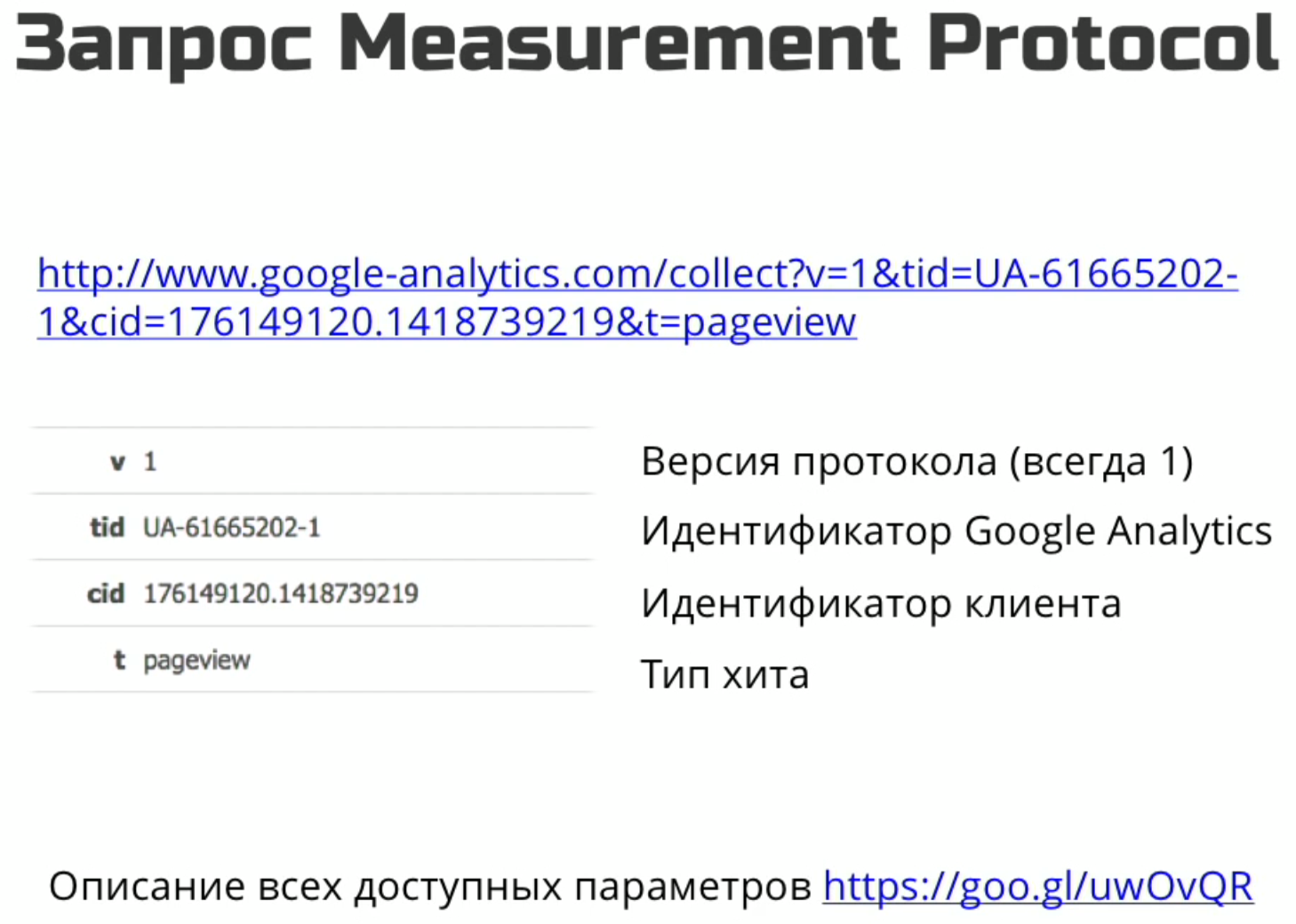
**Google Analytics**

**Google Analytics: Measurement Protocol**

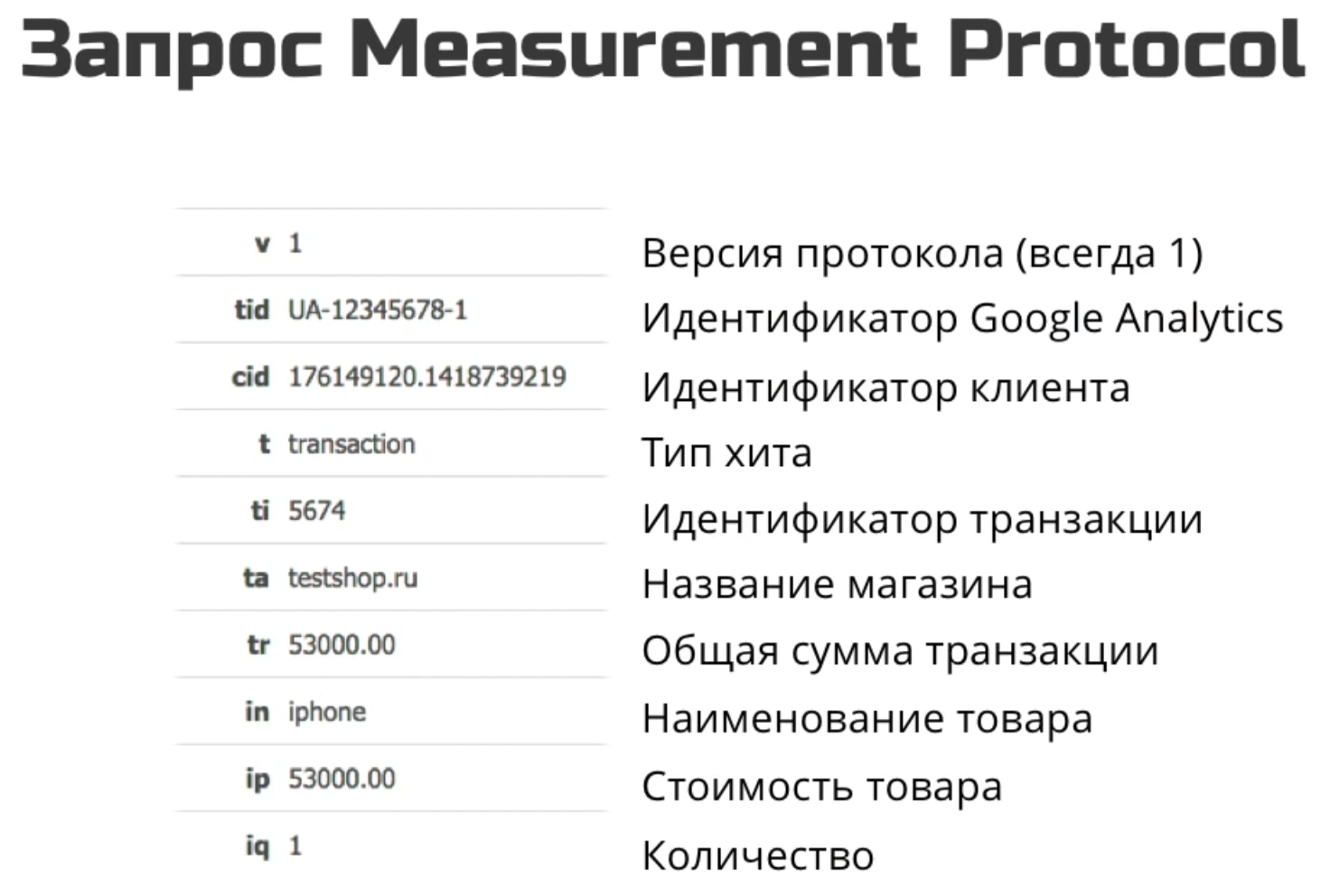
**Measurement Protocol** - Протокол передачи статистических данных Google Analytics позволяет отправлять необработанные данные напрямую на серверы Google Analytics посредством HTTP-запросов практически в любой среде.

В данной работе протокол используется для передачи данных о транзакциях в систему веб-аналитики. Это полезно для дальнейшего построения отчетов, графиков, KPIs в различных временных и других разрезах.

Пример построения запроса, включая список параметров изображен ниже.



Ниже демонстрируется пример отправки транзакции - данных о продаже товара в систему веб-аналитики. Информация включает общую цену, цену каждого товара, число товаров и некоторую другую служебную информацию.

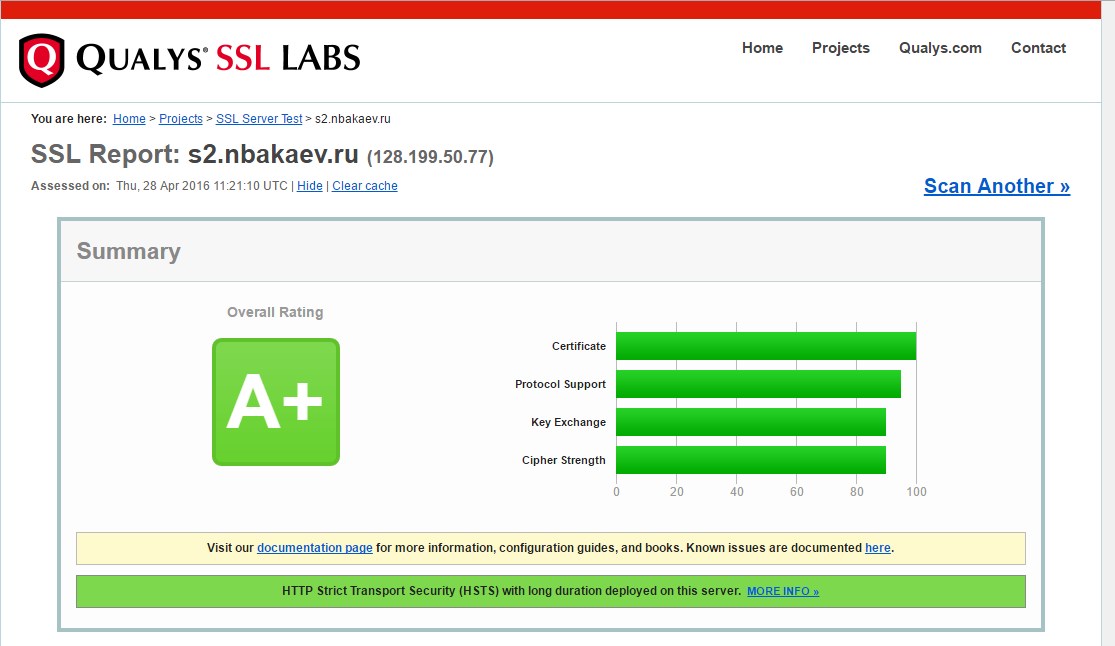


Защита информации

Для хранения паролей используется хеш функция bcrypt 2. Она превосходит md5 и другие в плане надедности.

Для всего API используется SSL. Оценка конфигурации SSL - высшая (A+) известным сервисом. В случае обращения по HTTP будет автоматическая переадресация на защищенный протокол HTTPS. Это сделано с целью, безопасной передачи паролей и иной чувствительной информации по сети Интернет.

Аутентификация пользователей происходит по Basic access authentication.



Дальнейшее развитие

Уже в данной работе продемонстрировано полностью рабочее приложение интернет-магазина.

При этом, есть основные векторы, которые будут реализованы и продемонстрированы в дальнейшем:

* **Шаблоны документов**. Пользователь полностью определяет шаблоны документов в HTML с возможностью использовать JavaScript для динамического создания документа, а затем выгружает их в PDF, Microsoft Word или иные форматы
* Дополнительные поля (**custom fields**) для объектов, включая товары. Отсутствие явной схемы в базе данных MongoDB позволит хранить любые поля. При этом каждая категория товаров может иметь свой уникальных список полей, доступный для выборок и полнотекстового поиска
* Создание заказа и некоторые другие ресурсоёмкие компоненты будут вынесены в отдельные **микросервисы**, таким образом будет микросервисная архитектура.
* Интернационализация и **локализация**: перевод на разные языки
* Товарные **рекомендации**
* Пример логирования, сбор метрик, **APM**
* горизонтальное масштабирование кластера **kubernetes**
* пример **шардирования** и replica set mongodb
* хранение конфигурации приложений и иных компонент во внешнем хранилище c strong consistency (**consul**) с возможностью динамического обновления конфигурации уже на запущенных приложения
* интеграция с одной из существующей облачной **CRM, ERP** для демонстрации работы интернет-магазины
* расширенные аналитические отчеты, дашборды

Заключение

В данной работе удалось реализовать полноценный интернет магазин с работой с товарами, выгрузкой, работой с пользовательской базой и многое другое. Система полностью построена на свободно распространяемых и кроссплатформенных технологиях. При разработки использовались современные методологии разработки и проверки качества разрабатываемого ПО для достижения высокого качества кода.

Все поставленные задачи - успешно выполнены.

Кроме того, поставлен ряд задач на улучшение и на разработку нового функционала и улучшения текущего в следующей итерации.

## 

## **Список литературы**

Интернет-ресурсы:

## [**https://spring.io/docs/reference**](https://spring.io/docs/reference) **-** Документация по Spring Framework & Spring Boot

* Документация по Docker <https://docs.docker.com/engine/reference/builder/>
* Resource Guide: Building an eCommerce Website https://www.drupal.org/resource-guides/building-ecommerce

## Книги:

## Spring in Action Fourth Edition Edition: http://www.amazon.com/Spring-Action-Craig-Walls/dp/161729120X/ref=sr\_1\_1?s=books&ie=UTF8&qid=1454910859&sr=1-1&keywords=spring

## Spring Boot in Action: <http://www.amazon.com/Spring-Boot-Action-Craig-Walls/dp/1617292540/ref=sr_1_3?s=books&ie=UTF8&qid=1454910859&sr=1-3&keywords=spring>

* Spring Recipes: A Problem-Solution Approach: <http://www.amazon.com/Spring-Recipes-Problem-Solution-Daniel-Rubio/dp/1430259086/ref=sr_1_4?s=books&ie=UTF8&qid=1454910859&sr=1-4&keywords=spring>
* Building eCommerce Applications - http://shop.oreilly.com/product/0636920023098.do

## 

**Приложения**

Ссылки на материалы:

* https://docs.google.com/document/d/1QhCjvqiGACP9OQohqe5BMcsHoedugAchMLozXtCoW64 - Текст курсовой работы в Google Docs
* [https://docs.google.com/presentation/d/1UFX-xWNX8DwdD1uIw4yO7Gc8Snupf74qfb5MSyjcgJU](https://docs.google.com/presentation/d/1UFX-xWNX8DwdD1uIw4yO7Gc8Snupf74qfb5MSyjcgJU/edit#slide=id.g1303f9e1ce_5_0) - Презентация курсовой работы в Google Docs

Что использовалось для работы:

* IDE: Intellij Idea 2016.1.2
* Ubuntu 16.04 server
* Charles - web proxy, sniffer. Для отладки запросов. (аналог wireshark)
* Postman - тесты API и создание/формирование запросов
* MongoChef - GUI для MongoDB
* Docker 1.11
* MongoDB 3.2 - NoSQL database
* библиотеки и фреймворки, описанные выше

## 

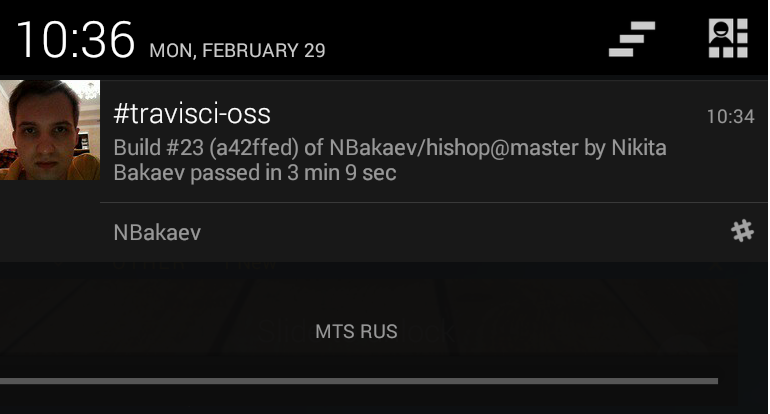
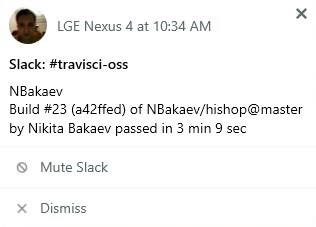
## 

## 

## 

## **Приложения Slack, уведомлени**

## 



## 

1. проект Swagger Code Generator https://github.com/swagger-api/swagger-codegen [↑](#footnote-ref-0)
2. https://github.com/swagger-api/swagger-codegen#customizing-the-generator [↑](#footnote-ref-1)
3. http://oleg.zorin.ru/doku.php?id=development:database:sharding [↑](#footnote-ref-2)