МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Ярославский государственный университет им. П.Г. Демидова»
Кафедра компьютерной безопасности и математических методов
обработки информации

Курсовая работа

Некоторые вопросы обеспечения безопасности веб-приложений.

Научный руководитель

канд. физ.-мат. наук, доцент

_____ Д.М. Мурин

«__» _____ 2019 г.

Студент группы КБ 51–СО

_____А.А. Шарунцова

«__» _____ 2019 г.

Ярославль 2019 г.

Реферат

Объем 50 с., 4 гл., 7 рис., 1 табл., 7 источников, 8 прил.

Авторизация пользователей. Способы аутентификации. Парольная аутентификация. Двухфакторная аутентификация. Анализ безопасности.

Объектом исследования являются различные способы авторизации, выбор оптимального способа для конкретных задач.

Цель работы – реализация безопасной авторизации пользователей интернет-магазина.

Задачи:

- изучить различные виды аутентификации, их области применения, сильные и слабые стороны,
- выбрать способ аутентификации для дальнейшей реализации и рассмотреть его более детально,
- реализовать программу по авторизации пользователей с помощью выбранного способа аутентификации,
- защитить программу от потенциальных уязвимостей,
- проанализировать и оценить защищенность написанной программы.

Краткий теоретический обзор:

В данной работе содержатся основные сведения об авторизации пользователей. В главе 1 приведен подробный обзор наиболее известных способов аутентификации. В главе 2 детально рассматриваются возможные уязвимости парольной аутентификации и способ борьбы с ними. Глава 3 представляет собой описание реализованного в рамках данной курсовой работы программного продукта. В главе 4 дается анализ безопасности данного продукта.

Оглавление

Введение

Большую роль в веб-приложениях играют механизмы авторизации и аутентификации. Они позволяют разграничить доступ для различных групп пользователей, а также идентифицировать пользователей. Для ясности уточним понятия идентификации, аутентификации и авторизации и разницу между ними.

Идентификация — процедура распознавания пользователя по его идентификатору (имени). Эта функция выполняется, когда пользователь делает попытку войти в систему. Пользователь сообщает системе по ее запросу свой идентификатор, и система проверяет в своей базе данных его наличие.

Аутентификация — процедура проверки подлинности заявленного пользователя, процесса или устройства. Эта проверка позволяет достоверно убедиться, что пользователь (процесс или устройство) является именно тем, кем себя объявляет. При проведении аутентификации проверяющая сторона убеждается в подлинности проверяемой стороны, при этом проверяемая сторона тоже активно участвует в процессе обмена информацией. Обычно пользователь подтверждает свою идентификацию, вводя в систему уникальную, не известную другим пользователям информацию о себе (например, пароль или сертификат).

Идентификация и аутентификация являются взаимосвязанными процессами распознавания и проверки подлинности субъектов (пользователей). Именно от них зависит последующее решение системы: можно ли разрешить доступ к ресурсам системы конкретному пользователю или процессу. После идентификации и аутентификации субъекта выполняется его авторизация.

Авторизация — процедура предоставления субъекту определенных полномочий и ресурсов в данной системе. Иными словами, авторизация устанавливает сферу его действия и доступные ему ресурсы. Если система не может надежно отличить авторизованное лицо от неавторизованного, то конфиденциальность и целостность информации в этой системе могут быть нарушены. Организации необходимо четко определить свои требования к безопасности, чтобы принимать решения о соответствующих границах авторизации.

Получается, грамотно реализованная авторизация – один из важнейших этапов построения защищенного веб-сайта. В следующих главах мы рассмотрим, что понимается под грамотно реализованной авторизацией, какие бывают уязвимости и как с ними бороться.

1. Различные способы аутентификации

Существует несколько различных схем аутентификации и авторизации. В этой главе мы рассмотрим подробно некоторые из них.

1.1 Аутентификация по многоразовому паролю

Аутентификация по многоразовому паролю считается самым популярным видом аутентификации. Хотя современные технологии внедряют все более сложные и надежные способы аутентификации, этот способ имеет место быть в несложных приложениях.

Этот способ основывается на том, что пользователь должен предоставить логин и пароль для успешной идентификации и аутентификации в системе. Пара логин/пароль задается пользователем при его регистрации в системе, при этом в качестве логина, как правило, выступает адрес электронной почты или номер телефона пользователя.

Применительно к веб-приложениям, существует несколько стандартных протоколов для аутентификации по паролю, которые мы рассмотрим ниже.

1.1.1 HTTP-аутентификация

Этот протокол, описанный в стандартах HTTP 1.0/1.1, существует очень давно и до сих пор активно применяется в корпоративной среде. Применительно к веб-сайтам работает следующим образом:

1. Сервер, при обращении неавторизованного клиента к защищенному ресурсу, отсылает HTTP статус “401 Unauthorized” и добавляет заголовок “WWW-Authenticate” с указанием схемы и параметров аутентификации.
2. Браузер, при получении такого ответа, автоматически показывает диалог ввода логина и пароля. Пользователь вводит детали своей учетной записи.
3. Во всех последующих запросах к этому веб-сайту браузер автоматически добавляет HTTP заголовок “Authorization”, в котором передаются данные пользователя для аутентификации сервером.
4. Сервер аутентифицирует пользователя по данным из этого заголовка. Решение о предоставлении доступа (авторизация) производится отдельно на основании роли пользователя, ACL или других данных учетной записи.

Весь процесс стандартизирован и хорошо поддерживается всеми браузерами и веб-серверами. Существует несколько схем аутентификации, отличающихся по уровню безопасности:

1. **Basic** — наиболее простая схема, при которой логин и пароль пользователя передаются в заголовке Authorization в незашифрованном виде (закодированном base64). Однако при использовании HTTPS (HTTP over SSL) протокола, является относительно безопасной.

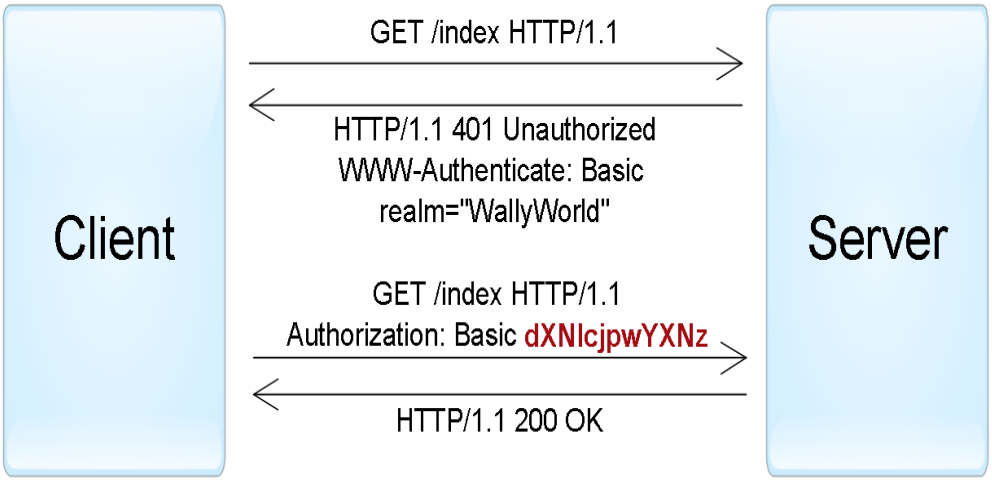

Рис. 1 Пример HTTP-аутентификации с использованием Basic-схемы

2. **Digest** — схема аутентификации «запрос-ответ», при которой сервер посылает уникальную произвольную информацию (например, текущее время), а браузер конкатенирует эту информацию с паролем и хеширует полученную строку, затем передает этот хеш серверу, который сравнивает это значение с вычисленным у него на основе тех же данных. Такая схема считается более безопасной альтернативой схемы Basic при незащищенных соединениях, но она подвержена атаке «человек посередине» (при которой происходит замена схемы на basic). Кроме того, использование этой схемы не позволяет применить современные хеш-функции для хранения паролей пользователей на сервере.
3. **NTLM** (известная как Windows authentication) — также основана на подходе «запрос-ответ», при котором пароль не передается в чистом виде. Эта схема не является стандартом HTTP, но поддерживается большинством браузеров и веб-серверов. Преимущественно используется для аутентификации пользователей Windows Active Directory в веб-приложениях. Уязвима к атаке повторного воспроизведения (pass-the-hash-attack).
4. **Negotiate** — еще одна схема из семейства Windows authentication, которая позволяет клиенту выбрать между NTLM и Kerberos

аутентификацией. Kerberos — более безопасный протокол, основанный на технологии единого входа (Single Sign-On), при использовании которой пользователь переходит из одного раздела портала в другой без повторной аутентификации.Однако он может функционировать, только если и клиент, и сервер находятся в зоне intranet и являются частью домена Windows.

Стоит отметить, что при использовании HTTP-аутентификации у пользователя нет стандартной возможности выйти из веб-приложения, кроме как закрыть все окна браузера.

1.1.2 Аутентификация с помощью форм

Для этого протокола нет определенного стандарта, поэтому все его реализации специфичны для конкретных систем, а точнее, для модулей аутентификации фреймворков разработки.

Работает это по следующему принципу: в веб-приложение включается HTML-форма, в которую пользователь должен ввести свои логин/пароль и отправить их на сервер запросом HTTP POST для аутентификации. В случае успеха веб-приложение создает сессию, которая, как правило, сохраняется в cookies браузера. При последующих веб-запросах данные о текущей сессии автоматически передаются на сервер, что позволяет приложению получить информацию о текущем пользователе для авторизации запроса.

Необходимо понимать, что перехват данных о текущей сессии зачастую дает аналогичный уровень доступа, что и знание пары логин/пароль. Поэтому все коммуникации между клиентом и сервером в случае аутентификации с помощью форм должны производиться только по защищенному соединению HTTPS.

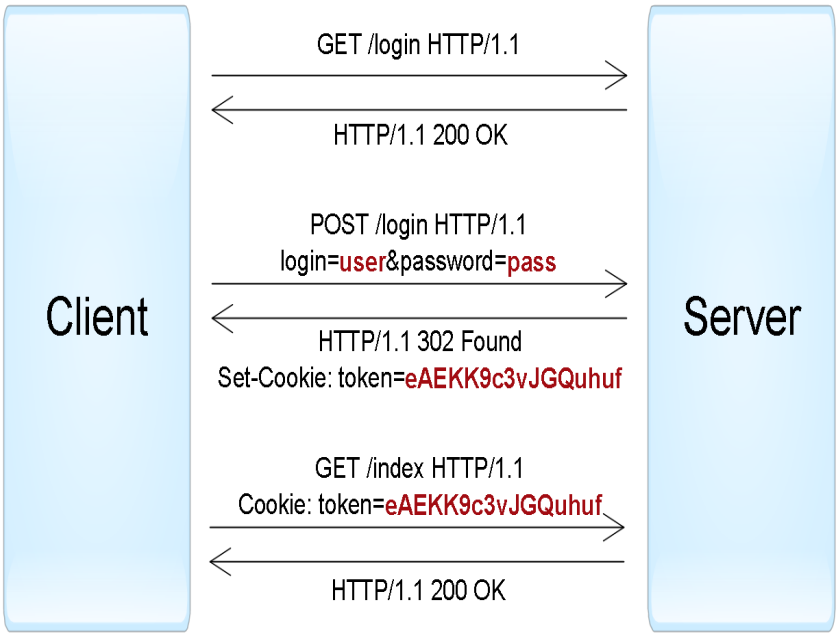

Рис. 2 Пример аутентификации с помощью форм

1.1.3 Другие протоколы аутентификации по многоразовому паролю

Протоколы, описанные выше, успешно используются для аутентификации пользователей на веб-сайтах. Но при разработке клиент-серверных приложений с использованием веб-сервисов (например, iOS или Android), наряду с HTTP аутентификацией, часто применяются нестандартные протоколы, в которых данные для аутентификации передаются в других частях запроса.

Существует всего несколько мест, где можно передать пару логин/пароль в HTTP запросах:

1. **URL query** — считается небезопасным вариантом, т. к. строки URL могут запоминаться браузерами, прокси и веб-серверами.
2. **Request body** — более безопасный вариант, но он применим только для запросов, содержащих тело сообщения (такие как POST, PUT, PATCH).
3. **HTTP header** — наиболее оптимальный вариант, при этом могут использоваться и стандартный заголовок Authorization (например, с Basic-схемой), и другие произвольные заголовки.

1.2 Аутентификация по сертификатам

Сертификат представляет собой набор атрибутов, идентифицирующих владельца, подписанный центром сертификации (ЦС). ЦС выступает в роли посредника, который гарантирует подлинность сертификатов. Также сертификат криптографически связан с секретным ключом, который хранится у владельца сертификата и позволяет однозначно подтвердить факт владения сертификатом.

На стороне клиента сертификат вместе с секретным ключом могут храниться в операционной системе, в браузере, в файле, на отдельном физическом устройстве (смарт-карта, USB-токен). Обычно секретный ключ дополнительно защищен паролем или PIN-кодом.

В веб-приложениях традиционно используют сертификаты стандарта X.509. Аутентификация с помощью X.509-сертификата происходит в момент соединения с сервером и является частью протокола SSL/TLS. Этот механизм также хорошо поддерживается браузерами, которые позволяют пользователю выбрать и применить сертификат, если веб-сайт допускает такой способ аутентификации.

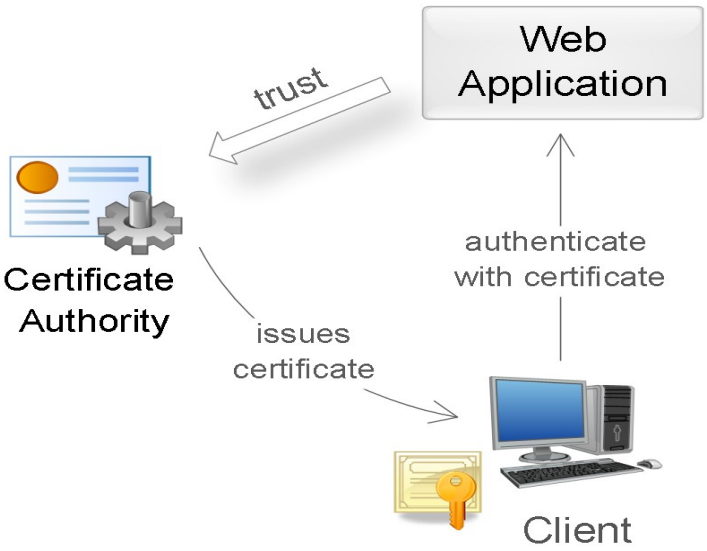

Рис. 3 Использование сертификата для аутентификации

Во время аутентификации сервер выполняет проверку сертификата на основании следующих правил:

1. Сертификат должен быть подписан доверенным центром сертификации (проверка цепочки сертификатов).
2. Сертификат должен быть действительным на текущую дату (проверка срока действия).
3. Сертификат не должен быть отозван соответствующим ЦС (проверка списков исключения).

После успешной аутентификации веб-приложение может выполнить авторизацию запроса на основании таких данных сертификата, как имя владельца, эмитент, серийный номер сертификата или отпечаток открытого ключа сертификата.

Использование сертификатов для аутентификации — куда более надежный способ, чем аутентификация посредством паролей. Это достигается созданием в процессе аутентификации цифровой подписи, наличие которой доказывает факт применения закрытого ключа в конкретной ситуации. Однако трудности с распространением и поддержкой сертификатов делает такой способ аутентификации малодоступным в широких кругах.

1.3. Аутентификация по одноразовым паролям

Аутентификация по одноразовым паролям обычно применяется дополнительно к аутентификации по паролям для реализации так называемой двухфакторной аутентификации. В этой концепции пользователю необходимо предоставить данные двух типов для входа в систему: что-то, что он знает (например, пароль), и что-то, чем он владеет (например, устройство для генерации одноразовых паролей). Наличие двух факторов позволяет в значительной степени увеличить уровень

безопасности, что может быть востребовано для определенных видов веб-приложений.

Другой популярный сценарий использования одноразовых паролей — дополнительная аутентификация пользователя во время выполнения важных действий: перевод денег, изменение настроек и т. п.

Существуют разные источники для создания одноразовых паролей. Наиболее популярные:

1. Аппаратные или программные токены, которые могут генерировать одноразовые пароли на основании секретного ключа, введенного в них, и текущего времени. Секретные ключи пользователей, являющиеся фактором владения, также хранятся на сервере, что позволяет выполнить проверку введенных одноразовых паролей. Пример аппаратной реализаций токенов — RSA SecurID; программной — приложение Google Authenticator.
2. Случайно генерируемые коды, передаваемые пользователю через SMS или другой канал связи. В этой ситуации фактор владения — телефон пользователя (точнее — SIM-карта, привязанная к определенному номеру).
3. Распечатка или скретч-карта (карта из картона или пластика с нанесённой на ней (под защитным непрозрачным и стирающимся слоем) некой секретной информацией) со списком заранее сформированных одноразовых паролей. Для каждого нового входа в систему требуется ввести новый одноразовый пароль с указанным номером.

В веб-приложениях такой механизм аутентификации часто реализуется посредством расширения аутентификации с помощью форм: после первичной аутентификации по паролю, создается сессия пользователя, однако в контексте этой сессии пользователь не имеет доступа к приложению до тех пор, пока он не выполнит дополнительную аутентификацию по одноразовому паролю.

1.4 Аутентификация по ключам доступа

Этот способ чаще всего используется для аутентификации устройств, сервисов или других приложений при обращении к веб-сервисам. Здесь в качестве секрета применяются ключи доступа (access key, API key) — длинные уникальные строки, содержащие произвольный набор символов, по сути заменяющие собой комбинацию логин/пароль.

В большинстве случаев, сервер генерирует ключи доступа по запросу пользователей, которые далее сохраняют эти ключи в клиентских приложениях. При создании ключа также возможно ограничить срок действия и уровень доступа, который получит клиентское приложение при аутентификации с помощью этого ключа.

Использование ключей позволяет избежать передачи пароля пользователя сторонним приложениям (в примере выше пользователь

сохранил в веб-приложении не свой пароль, а ключ доступа). Ключи обладают значительно большей энтропией по сравнению с паролями, поэтому их практически невозможно подобрать. Кроме того, если ключ был раскрыт, это не приводит к компрометации основной учетной записи пользователя — достаточно лишь аннулировать этот ключ и создать новый.

С технической точки зрения, здесь не существует единого протокола: ключи могут передаваться в разных частях HTTP-запроса: URL query, request body или HTTP header. Как и в случае аутентификации по паролю, наиболее оптимальный вариант — использование HTTP header. В некоторых случаях используют HTTP-схему Bearer для передачи токена в заголовке. Чтобы избежать перехвата ключей, соединение с сервером должно быть обязательно защищено протоколом SSL/TLS.

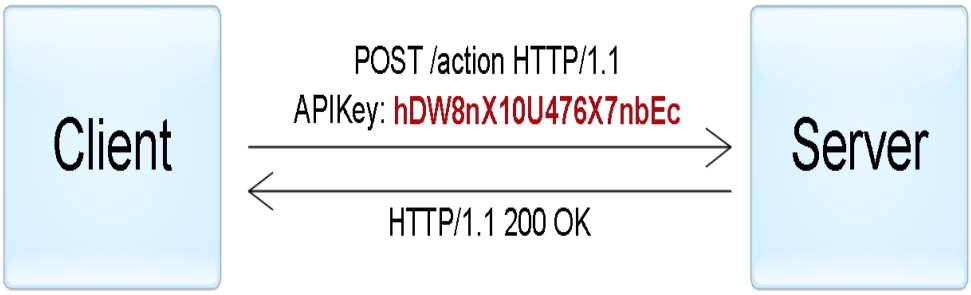

Рис. 4 Пример аутентификации по ключу доступа, переданного в HTTP заголовке

Кроме того, существуют более сложные схемы аутентификации по ключам для незащищенных соединений. В этом случае, ключ обычно состоит их двух частей: открытой и секретной. Публичная часть используется для идентификации клиента, а секретная часть позволяет сгенерировать подпись. Например, по аналогии с digest-аутентификацией, сервер может послать клиенту некоторое уникальное значение или временную отметку, а клиент — возвратить хеш или HMAC этого значения, вычисленный с использованием секретной части ключа. Это позволяет избежать передачи всего ключа в оригинальном виде и защищает от атак повторного воспроизведения.

1.5 Аутентификация по токенам

Аутентификация по токенам чаще всего применяется при построении распределенных систем единого входа (SSO), где одно приложение делегирует функцию аутентификации пользователей другому приложению. Типичный пример этого способа — вход в приложение через учетную запись в социальных сетях. Здесь социальные сети являются сервисами аутентификации, а приложение доверяет функцию аутентификации пользователей социальным сетям.

Реализация этого способа заключается в том, что сервер аутентификации (identity provider - IP) предоставляет достоверные сведения о пользователе в виде токена, а приложение использует этот токен для идентификации, аутентификации и авторизации пользователя.

На общем уровне, весь процесс выглядит следующим образом:

1. Клиент аутентифицируется на сервере аутентификации одним из способов, специфичным для него (пароль, ключ доступа, сертификат, Kerberos, и т.д.).
2. Клиент просит сервер аутентификации предоставить ему токен для конкретного приложения. Сервер аутентификации генерирует токен и отправляет его клиенту.
3. Клиент аутентифицируется в приложении при помощи этого токена.

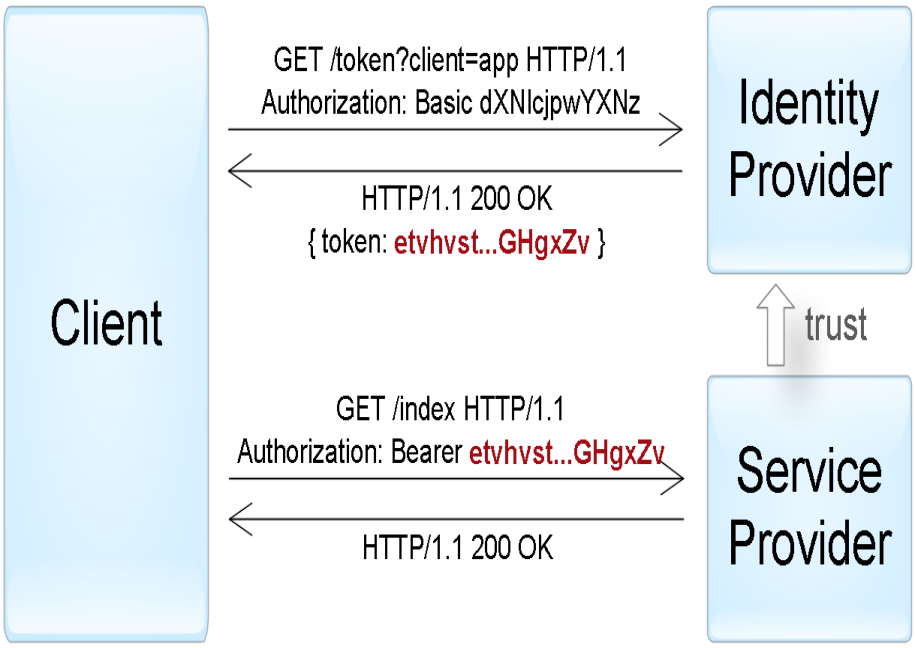

Рис. 5 Пример аутентификации по токенам

Сам токен обычно представляет собой структуру данных, которая содержит информацию, кто сгенерировал токен, кто может быть получателем токена, срок действия, набор сведений о самом пользователе (claims). Кроме того, токен дополнительно подписывается для предотвращения несанкционированных изменений и гарантий подлинности.

При аутентификации с помощью токена приложение должно выполнить следующие проверки:

1. Токен был выдан доверенным сервером (проверка поля issuer).
2. Токен предназначается текущему приложению (проверка поля audience).
3. Срок действия токена еще не истек (проверка поля expiration date).

4. Токен подлинный и не был изменен (проверка подписи).

В случае успешной проверки приложение выполняет авторизацию запроса на основании данных о пользователе, содержащихся в токене.

1.5.1 Форматы токенов

Существует несколько распространенных форматов токенов для веб-приложений:

1. Simple Web Token (SWT) — наиболее простой формат, представляющий собой набор произвольных пар имя/значение в формате кодирования HTML form. Стандарт определяет несколько зарезервированных имен: Issuer, Audience, ExpiresOn и HMACSHA256. Токен подписывается с помощью симметричного ключа, таким образом оба IP- и SP-приложения должны иметь этот ключ для возможности создания/проверки токена.

2. JSON Web Token (JWT) — содержит три блока, разделенных точками: заголовок, набор полей (claims) и подпись. Первые два блока представлены в JSON-формате и дополнительно закодированы в формат base64. Набор полей содержит произвольные пары имя/значения, притом стандарт JWT определяет несколько зарезервированных имен (iss, aud, exp и другие). Подпись может генерироваться при помощи и симметричных алгоритмов шифрования, и асимметричных. Кроме того, существует отдельный стандарт, отписывающий формат зашифрованного JWT-токена.

3. Security Assertion Markup Language (SAML) — определяет токены (SAML assertions) в XML-формате, включающем информацию об эмитенте, о субъекте, необходимые условия для проверки токена, набор дополнительных утверждений (statements) о пользователе. Подпись SAML-токенов осуществляется при помощи ассиметричной криптографии. Кроме того, в отличие от предыдущих форматов, SAML-токены содержат механизм для подтверждения владения токеном, что позволяет предотвратить перехват токенов через man-in-the-middle-атаки при использовании незащищенных соединений.

1.6 Резюме различных способов аутентификации

Выше были рассмотрены различные способы аутентификации в веб-приложениях. Ниже — таблица, которая резюмирует описанные способы и протоколы:

Таблица 1

Основные способы аутентификации в веб-приложениях

Способ	**Основное применение**	**Протоколы**
По паролю	Аутентификация пользователей	HTTP, Forms
По сертификатам	Аутентификация пользователей в безопасных приложениях; аутентификация сервисов	SSL/TLS
По одноразовым паролям	Дополнительная аутентификация пользователей (для достижения two-factor authentication)	Forms
По ключам доступа	Аутентификация сервисов и приложений	-
По токенам	Делегированная аутентификация пользователей; делегированная авторизация приложений	SAML, WS-Federation, OAuth, OpenID Connect

2. Основные уязвимости парольной аутентификации

В предыдущей главе были рассмотрены наиболее известные и применяемые способы аутентификации. Каждый из них имеет свою специфику применения.

Одной из задач данной курсовой работы была реализация авторизации пользователей в небольшом онлайн-магазине. Для этих целей лучше всего подходит аутентификация по многоразовым или одноразовым паролям. Это привычные для пользователей способы аутентификации, которые, к тому же, при правильной реализации обеспечивают достаточную безопасность для стандартного функционала интернет-магазина.

Рассмотрим возможные уязвимости выбранных способов аутентификации, возможные ошибки разработчика и как их предотвратить.

2.1 Распространенные приемы взлома парольной защиты и методы противодействия им.

2.1.1 Полный перебор (метод грубой силы, bruteforce).

Самая простая (с технической точки зрения) атака на пароль – перебор всех комбинаций допустимых символов (начиная от односимвольных паролей). Современные вычислительные мощности позволяют перебрать все пароли длиной до восьми символов за несколько секунд.

Некоторые системы не позволяют реализовать атаки, основанные на переборе, поскольку реагируют на несколько попыток неправильно набранного пароля подряд.

Однако существует множество систем, допускающих бесконечный перебор. Например, к защищенному паролем файлу (архив rar или zip, документ Microsoft Office и т.д.) можно пробовать разные пароли бесконечно. Существует множество программ, которые позволяют автоматизировать эту процедуру: Advanced RAR Password Recovery, Advanced PDF Password Recovery, Advanced Office XP Password Recovery.

И если получается похитить файл, содержащий хеши паролей доступа к операционной системе, то можно заниматься подбором паролей уже в обход системы, с помощью специальных программ.

Важной характеристикой пароля, затрудняющей полный перебор, является его длина. Современный пароль должен иметь длину не менее 12 символов.

Два лишних символа в пароле увеличивают время перебора в 40000 раз, а четыре символа — уже в 1.600.000.000 раз. Однако вычислительные мощности компьютеров постоянно растут (еще несколько лет назад безопасным считался пароль длиной 8 символов).

2.1.2 Перебор в ограниченном диапазоне.

Известно, что многие пользователи, составляя пароль, используют символы, находящиеся в определенном диапазоне. Например, пароль, состоящий только из русских букв или только из латинских букв или только из цифр. Такой пароль значительно легче запомнить, однако задача противника, осуществляющего перебор, неимоверно упрощается.

Пусть $n = 70$ — количество символов, из которых можно составить пароль, причем 10 из них — цифры, 30 — буквы одного языка и 30 — буквы другого языка. Пусть мы составляем пароль длиной $m = 4$ символа.

Если пароль составляется абсолютно случайно, то количество возможных комбинаций (которые необходимо перебрать) составляет $70^4 = 24010000$. Однако противник может сделать предположение, что пароль состоит из символов одного диапазона (пусть даже, неизвестно, какого). Всего таких паролей $10^4 + 30^4 + 30^4 = 10000 + 810000 + 810000 = 163000$. Если он оказался прав, то количество комбинаций (а следовательно, время, которое необходимо затратить на перебор) уменьшилось в 147 раз. Это число резко возрастает, когда увеличивается длина пароля и число диапазонов символов, из которых он может быть составлен.

Как следствие, надежный пароль должен содержать в себе символы из различных диапазонов. Рекомендуется использовать русские и английские, прописные и строчные буквы, цифры и служебные символы.

2.1.3 Атака по словарю

В качестве пароля очень часто выбирается какое-то слово или распространенная фраза. Программа автоматического перебора паролей проверяет слова, содержащиеся в заданном файле со словарем (существует огромное количество доступных словарей такого рода для разных языков). Словарь из двухсот тысяч слов проверяется такой программой за несколько секунд.

Многие пользователи считают, что если применить к задуманному слову некоторое простое преобразование, например, написать его задом наперед или русскими буквами в английской раскладке или намеренно сделать ошибку, то это обеспечит безопасность. На самом деле, по сравнению с подбором случайного пароля подбор пароля по словарю с применением различных преобразований (сделать первую букву заглавной, сделать все буквы заглавными, объединить два слова и т.д.) делает невыполнимую задачу вполне возможной.

Надежный пароль не должен строиться на основе слов естественного языка.

2.1.4 Атака по персональному словарю

Если атака по словарю и перебор паролей небольшой длины либо составленных из символов одной группы не помогает, злоумышленник может воспользоваться тем фактом, что для облегчения запоминания, многие пользователи выбирают в качестве пароля личные данные (номер сотового телефона, дату рождения, записанную наоборот, кличку собаки и т.д.).

В том случае, если цель злоумышленника — обойти парольную защиту именно этого пользователя, он может составить для него персональный словарь личных данных, после чего использовать программу автоматического перебора паролей, которая будет генерировать пароли на основе этого словаря.

Надежный пароль должен быть относительно «бессмысленным».

Существуют и другие атаки на парольную аутентификацию, такие как социальный инжиниринг или фишинг, но они напрямую связаны с грамотностью пользователя. Поскольку от технической реализации приложения в случае подобных атак ничего не зависит, то рассматривать их здесь не будем.

Резюмируем **рекомендации разработчикам** по предотвращению атак на парольную аутентификацию:

- программное наложение технических ограничений (пароль должен быть не слишком коротким, он должен содержать буквы, цифры, знаки пунктуации и т.п.);
- управление сроком действия паролей, их периодическая смена;
- ограничение доступа к файлу паролей;
- ограничение числа неудачных попыток входа в систему;
- обучение пользователей (хотя бы всплывающие подсказки с предупреждениями и рекомендациями);
- использование программных генераторов паролей (такая программа позволяет придумать «бессмысленный» пароль).

2.2 Рекомендации по разработке безопасной парольной аутентификации

Также не стоит забывать о технических уязвимостях приложения, связанных с ошибками разработчиков. Дадим рекомендации, о которых не стоит забывать при разработке приложения.

- Если веб-приложение само генерирует и распространяет пароли пользователям, то обязательно требуется смена пароля после первого входа (т.к. текущий пароль где-то записан).
- Не допускать передачу паролей по незащищенному HTTP-соединению либо в строке URL.
- Использовать только безопасные хеш-функции для хранения паролей пользователей.

- Предоставлять пользователям возможность изменения пароля и уведомлять их о произошедшей смене их пароля.
- Требовать повторной аутентификации пользователя для важных действий: смена пароля, изменения адреса доставки товаров и т. п.
- Не допускать передачу session tokens по незащищенному HTTP-соединению, либо в строке URL.
- Уничтожать сессии пользователя после короткого периода неактивности и обязательно предоставлять функцию выхода из аутентифицированной сессии.
- Обязательно проверять все введенные пользователем данные на предмет соответствия этих данных ожиданию сервера во избежание SQL-инъекций.

2.3 Уязвимости аутентификации по одноразовым паролям

Технология одноразовых паролей считается достаточно надежной. Однако стоит отметить, что и у нее есть недостатки, которым подвержены все системы, реализующие аутентификацию по одноразовым паролям в чистом виде. Подобные уязвимости можно разделить на две группы. К первой относятся потенциально опасные «дыры», присущие всем методам реализации. Наиболее серьезная из них - возможность подмены сервера аутентификации. При этом пользователь будет отправлять свои данные прямо злоумышленнику, который может тут же использовать их для доступа к настоящему серверу. В случае метода «запрос-ответ» алгоритм атаки немного усложняется (компьютер злоумышленника должен сыграть роль «посредника», пропуская через себя процесс обмена информацией между сервером и клиентом). Впрочем, стоит отметить, что на практике осуществить такую атаку зачастую совсем не просто.

Другая уязвимость присуща только синхронным методам и связана с тем, что существует риск рассинхронизации информации на сервере и в программном или аппаратном обеспечении пользователя. Допустим, в какой-то системе начальными данными служат показания внутренних таймеров, и по каким-то причинам они перестают совпадать друг с другом. В этом случае все попытки пользователей пройти аутентификацию будут неудачными (ошибка первого рода). К счастью, в подобных случаях ошибка второго рода (допуск «чужого») возникнуть не может. Впрочем, вероятность возникновения описанной ситуации также крайне мала.

Некоторые атаки применимы только к отдельным способам реализации технологии одноразовых паролей. Для примера опять возьмем метод синхронизации по таймеру. Как мы уже говорили, время в нем учитывается не с точностью до секунды, а в пределах какого-то установленного заранее интервала. Это делается с учетом возможности

рассинхронизации таймеров, а также появления задержек в передаче данных. И именно этим моментом теоретически может воспользоваться злоумышленник для получения несанкционированного доступа к удаленной системе. Для начала злоумышленник «прослушивает» сетевой трафик от пользователя к серверу аутентификации и перехватывает отправленные «жертвой» логин и одноразовый пароль. Затем он тут же блокирует его компьютер (перегружает его, обрывает связь и т. п.) и отправляет авторизационные данные уже от себя. И если злоумышленник успеет сделать это так быстро, чтобы интервал аутентификации не успел смениться, то сервер признает его как зарегистрированного пользователя.

Понятно, что для такой атаки злоумышленник должен иметь возможность прослушивания трафика, а также быстрого блокирования компьютера клиента, а это задача не из легких. Проще всего соблюсти эти условия тогда, когда атака задумывается заранее, причем для подключения к удаленной системе "жертва" будет использовать компьютер из чужой локальной сети. В этом случае злоумышленник может заранее «поработать» над одним из ПК, получив возможность управлять им с другой машины. Защититься от такой атаки можно только путем использования «доверенных» рабочих машин и «независимых» защищенных (например, с помощью SSL) каналов выхода в Интернет.

3. Программная реализация приложения

Мною была реализована программа, предназначенная для авторизации пользователей в личном кабинете интернет-магазина. Ее функционал включает в себя регистрацию по номеру телефона, вход в личный кабинет по многоразовому паролю, восстановление пароля по номеру телефона и одноразовому паролю, а также двухфакторная аутентификация для совершения важных с точки зрения информационной безопасности действий.

3.1 Технические средства, используемые в программе:

Программа представляет собой web-сервис на платформе Node.js, реализованная с помощью фреймворка Express.js. Авторизация пользователей происходит с помощью модуля Passport.

Для реализации программы были использованы следующие модули:

- "connect-redis" версии 3.1.0,
- "cookie-parser" версии 1.4.4,
- "express" версии 4.16.1,
- "express-handlebars" версии 3.1.0,
- "express-session" версии 1.14.1,
- "hbs" версии 4.0.4,
- "http-errors" версии 1.6.3,
- "mongodb" версии 3.2.4,
- "mongoose" версии 5.5.7,
- "morgan" версии 1.9.1,
- "mysql-promise" версии 5.0.0,
- "passport" версии 0.3.2,
- "passport-local" версии 1.0.0,
- "password-validator" версии 4.1.1,
- "promise-mysql" версии 3.3.1,
- "promise-redis" версии 0.0.5,
- "random-number" версии 0.0.9,
- "request" версии 2.88.0,
- "request-promise-native" версии 1.0.7,
- "uglify-js" версии 3.5.11.

Была использована библиотека jQuery 1.8.3 и плагин jQuery.mask.js.

Также были использованы три базы данных: реляционная БД MySQL для хранения учетных данных пользователей, нереляционная высокопроиводительная БД Redis для хранения данных о текущих сессиях и нереляционная документоориентированная БД MongoDB для хранения временных одноразовых паролей.

Рассмотрим подробнее назначение каждой из них.

3.1.1 Использование СУБД MySQL

MySQL – реляционная система управления базами данных. Данные хранятся в четко структурированных таблицах, связанных между собой. Это очень удобно, когда заранее известна структура данных и предполагается наличие множества однотипных данных, которые можно описать единым образом.

В частности, в данной программе она была использована для хранения учетных записей пользователей, их ролей и прав доступа. Система предполагает две роли, продавца и покупателя. Каждой роли можно прописать соответствующие права. Если покупатель может только просматривать товары, то продавец еще и редактировать, то есть у него есть права на запись в таблицу с товарами.

Также есть таблица пользователей, в которой хранятся телефоны, что идентифицируют пользователя. Пользователи связаны с ролями в отдельной таблице. В ней же есть входные данные для пользователя, хеш его пароля и соль. Эти данные хранятся отдельно от списка телефонов для того, чтобы один человек под одним номером телефона мог выступать как в роли продавца, так и в роли покупателя.

Описанная структура данных представлена на рисунке 6.

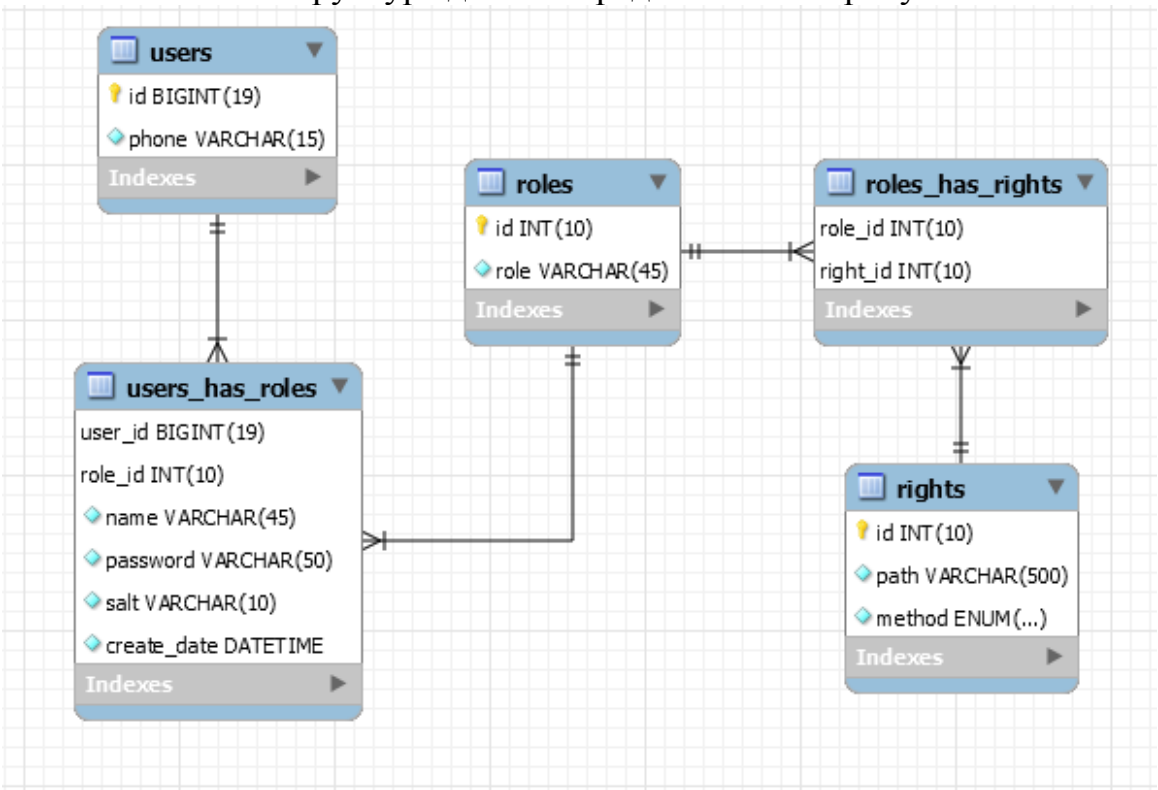

Рис. 6 Диаграмма таблиц и связей в БД MySQL

Обоснование выбора MySQL:

MySQL считается самой популярной из всех крупных серверных СУБД. Существует большое количество версий для множества операционных систем, что означает, что приложение не зависит от платформы и может быть расположено практически на любом сервере. Также в связи с популярностью этой СУБД существует множество документации по работе с ней и вариантов решения различных проблем. Система также довольно проста для понимания в отличие от некоторых похожих СУБД. MySQL поддерживает множество функций и имеет свой встроенный функционал по безопасности.

Хранимые процедуры в MySQL

Помимо хранения самих данных, в MySQL будут храниться функции и хранимые процедуры для взаимодействия с этими данными. Существуют средства, которые связывают базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Это так называемые ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение, или преобразование).

Однако есть ряд причин, по которым в данной программе были использованы именно хранимые процедуры для взаимодействия с данными БД. Приведем некоторые из них:

- Как правило, производительность приложения существенно возрастает. В рамках одного подключения при вызове процедуры она компилируется (выполняется синтаксический анализ и генерируется план доступа к данным) и сохраняется в кэш. MySQL заводит отдельный кэш на каждое подключение. Если приложение использует хранимую процедуру несколько раз, она вызывается из кэша, иначе хранимая процедура работает как обычный запрос к базе.
- Использование хранимой процедуры уменьшает трафик между приложением и базой данных. Всего одна команда на выполнение хранимой процедуры позволяет вызвать содержащийся в ней сложный сценарий, благодаря чему можно избежать пересылки через сеть сотен команд, и в особенности, необходимости передачи больших объемов данных для анализа, поскольку это можно реализовать в рамках сервера базы данных.
- Хранимая процедура может вызываться несколькими приложениями, использующими одну базу данных, позволяя не писать код повторно.
- Можно накладывать ограничение на вызов хранимой процедуры в зависимости от пользователя базы данных.

Конечно, есть и недостатки используемого подхода. Во-первых, требуется большее количество памяти, а если при этом используется много логических операторов, то еще и увеличивается нагрузка на центральный процессор. Также стоит заметить, что разработка хранимых процедур достаточно сложна, особенно учитывая тот факт, что в MySQL нет отладки кода.

Но, поскольку для данного приложения не требуется много различных операций с данными, и все они достаточно просты, то увеличение задействованной памяти не существенно, а логика реализации весьма прозрачна. При этом действительно важна производительность и размер трафика между приложением БД в случае большого количества пользователей.

Код создания таблиц, связей между ними и хранимых процедур доступен в приложении А.

3.1.2 Использование СУБД Redis

Redis — резидентная система управления базами данных класса NoSQL с открытым исходным кодом, работающая со структурами данных типа «ключ — значение». Используется как для баз данных, так и для реализации кэшей, брокеров сообщений. Ориентирована на достижение максимальной производительности на атомарных операциях.

Redis действительно обладает высокой производительностью, поскольку хранит данные в оперативной памяти. Это делает ее идеальным инструментов для хранения текущих сессий пользователей. Можно было бы хранить эти данные в памяти процесса, однако это делает приложения менее масштабируемыми.

3.1.3 Использование СУБД MongoDB

Для хранения одноразовых временных паролей используется СУБД MongoDB. MongoDB — нереляционная СУБД для хранения JSON объектов.

В данной программе используются достаточно однотипные данные, которые логично было бы хранить в реляционной БД, так как у них четкая структура. Однако есть ряд преимуществ, которые склоняют нас к выбору MongoDB.

Во-первых, используемая функциональность отправки смс-кодов может значительно поменяться при дальнейшей разработке приложения. На данном этапе неясно, как именно она может меняться. Нереляционная БД гораздо более масштабируемая и гибкая по сравнению с реляционной. Даже если данная структура изменится сильно, это не повлечет за собой глобальных изменений в коде.

Во-вторых, данная СУБД «легче» и быстрее. Смс-коды имеют срок действия. Учитывая время, необходимое сервису по отправке смс для обработки запроса, а также время доставки смс, необходимо

минимизировать время, используемое для взаимодействия с БД, хранящей данные коды.

3.2 Структура программного средства

Кратко описать структуру программного средства можно таким образом:

1. Пользователь получает страницу с возможностью зарегистрироваться или зайти в личный кабинет.
2. При выборе варианта «зарегистрироваться» пользователь
   2.1. Вводит свой номер телефона.
   2.2. Подтверждает, что телефон принадлежит ему, путем ввода смс-кода, отправленного ему на телефон.
   2.3. Вводит имя и пароль для окончания регистрации.
   2.4. Затем перенаправляется на страницу входа в личный кабинет.
3. При выборе варианта «зайти» пользователь
   3.1. Вводит номер телефона в качестве логина и пароль.
   3.2. Если он не помнит пароль, то может пройти процедуру сброса пароля, получив код по смс для подтверждения, что телефон принадлежит ему, и создав новый пароль.
      3.2.1. После чего отправляется на страницу входа в личный кабинет.
   3.3. После входа пользователь оказывается в личном кабинете, функционал которого представляет собой возможность выйти из личного кабинета.

Теперь рассмотрим более детально основные функциональные блоки.

3.2.1 Регистрация

Регистрация происходит по номеру телефона пользователя. С целью исключить возможность регистрации множества фальшивых пользователей, а также использование чужих номеров телефона, пользователю необходимо подтвердить свой номер. Для этого он должен ввести код, полученный в смс-сообщении.

Рассмотрим более детально, как это происходит.

При запросе пользователя на регистрацию сервер получает номер телефона, наличие которого проверяется в базе данных вызовом специальной хранимой процедуры (приложение А, строки 139-154). При отсутствии номера в базе, генерируется случайное четырехзначное число. Номер телефона, код и время его создания, а также время после которого, код перестанет быть валидным, сохраняется в базу данных. Схема описана в приложении Ж.

Затем код посылается абоненту с помощью сервиса sms.ru. Конфигурационные настройки отправки описаны в приложении Е в файле smsLib.js и в приложении Д, строки 15-20.

После чего пользователю отправляется страница с просьбой ввести код для подтверждения номера телефона. Также на странице есть счетчик времени, когда код считается действительным и возможность прислать новый по истечении срока годности.

После того, как пользователь отправляет код, сервер делает запрос к БД и сравнивает пару «телефон-код». В случае ошибки, запрашивает ввести код заново. В случае совпадения дает пользователю страницу с возможностью завершить регистрацию, введя имя и пароль. Пара «телефон-код» удаляется из БД.

На пароль наложены ограничения. Он должен содержать не менее 8 символов, строчные и заглавные латинские буквы и цифры. Данные ограничения необходимы для предотвращения слишком простых паролей, которые легко подбираются. Пока пользователь не выполнит всех условий, система не даст ему завершить регистрацию. После того, как пользователь ввел имя и валидный пароль, вызывается хранимая процедура по регистрации. В ней генерируется соль (приложение А, строки 120-135), хеш-функцией MD5хешируется строка, которая представляет собой конкатенацию пароля и соли (приложение А, строки 110-116), затем в базу сохраняется номер телефона пользователя и его роль вместе с именем, хешем пароля и солью.

После успешной регистрации пользователю предлагается войти в систему.

Аналогично происходит восстановление процедура пароля. В случае если пользователь забыл пароль, он вводит свой номер телефона, на который приходит код подтверждения. Перед отправкой кода проверяется наличие такого телефона в базе. В данном случае, наоборот, нужно, чтобы такой пользователь существовал, иначе программа предупредит пользователя, что такой номер не зарегистрирован и предложит пройти процедуру регистрации.

После подтверждения номера телефона с помощью кода из смс, пользователю предлагается ввести новый пароль, который также должен соответствовать требованиям безопасности.

3.2.2 Вход в систему

Для входа в систему пользователь должен ввести свой номер телефона и пароль. Аутентификация происходит с помощью модуля Passport, настройки которого прописаны в строках 40-67 приложения В. Вызывается хранимая процедура login (приложение А, строки 158-194). В ней производится поиск записи с указанным номером телефона. Если таких нет, то пользователю возвращается сообщение об ошибке с просьбой проверить номер телефона или зарегистрироваться. Если такая запись есть, то берется сохраненная в ней соль, складывается с паролем и хешируется хеш-функцией MD5. Затем полученных хеш сравнивается с записанных хешем в БД. В случае несовпадения, пользователь получает сообщение об

ошибке в пароле. Если хеши совпадают, то процедура возвращает true, модуль Passport создает сессию и сохраняет ее в БД Redis. Далее пользователь перенаправляется на домашнюю страницу, которая защищена проверкой на авторизованность пользователя. При попытке ввести адрес данной страницы в ручную, система проверит, авторизован ли пользователь, и в случае отрицательного ответа, автоматически перенаправит его на страницу с регистрацией или входом в систему.

Точно так же, при попытке авторизованного пользователя зайти на страницу для авторизации, он будет перенаправлен на свою домашнюю страницу, чтобы избежать повторной авторизации.

3.2.3 Выход из системы

При нажатии на кнопку «Log out», текущая сессия пользователя уничтожается и он перенаправляется на начальную страницу.

Используемые сессии не имеют ограничений по времени. Однако их можно установить, если считается, что приложение имеет важные с точки зрения защиты информации функции и несанкционированный доступ принесет значительный ущерб для кого-либо из сторон. Например, если в него будет встроена возможность оплаты покупок онлайн.

В таком случае, желательным будет использовать двухфакторную аутентификацию. Когда пользователь захочет совершить действие, которое в случае несанкционированного доступа сможет нанести некоторый ущерб, он будет обязан подтвердить свою личность с помощью одноразового пароля, полученного им в смс, несмотря на то, что он уже авторизован в системе.

3.2.4 Реализация программы

Мы рассмотрели функциональную структуру программы. Теперь рассмотрим архитектуру самого проекта и детали реализации.

На рисунке 7 представлено дерево каталогов и файлов проекта. Рассмотрим подробнее, что содержится, в каждом из них. Весь код будет приведен в соответствующих приложениях.

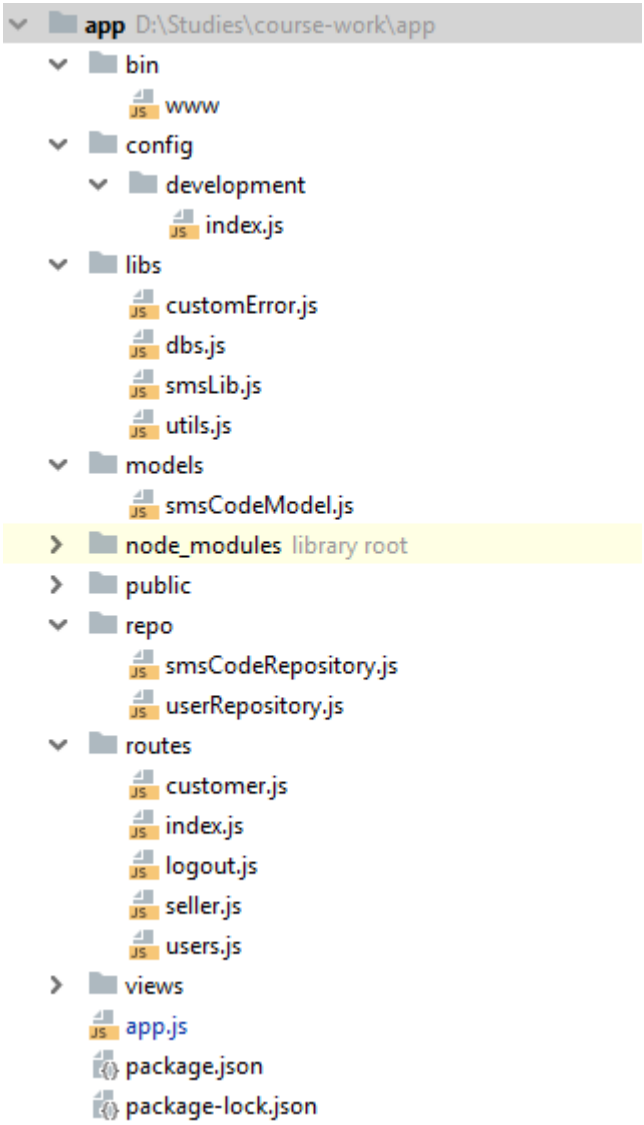

Рис. 7 Архитектура программы

Точкой входа в приложение является файл ./bin/www. Основная суть этого файла приведена в приложении Б. В нем подключается файл ./app.js и встроенный в Node модуль http. Далее создается и запускается сервер. Методу createServer() передается функция, которая вызывается каждый раз, когда возникает событие 'request', появляющееся, в свою очередь, при каждом обращении к серверу. В нашем случае функция описана в модуле ./app.js. Код этого модуля приведен в приложении В. В частности, в нем используется класс express.Router(), который позволяет создавать модульные монтируемые обработчики маршрутов. Все маршруты прописаны в папке ./routes. Они выполнены по аналогии, поэтому в приложении приведен только код файла ./routes/customer.js (приложение Г), в котором больше всего функционала.

В файле ./config/development/index.js содержаться все конфигурационные настройки для всех библиотек. Код этого файла приведен в приложении Д.

В папке ./libs находятся библиотеки для работы со сторонними ресурсами, например, с сервисом отправки смс sms.ru. Код этих файлов можно посмотреть в приложении Е.

Файл ./models/smsCodeModel.js содерит описание схемы MongoDB для хранения временных кодов (приложение Ж).

Папка ./node_modules содержит сторонние модули, используемые в приложении. Папка ./public содержит статические файлы, такие как клиентский код, стили, статические HTML-файлы. В папке ./views располагаются шаблоны для рендеринга страниц на клиентской части приложения. Код в этих папках не представляет ценности, поэтому приводить его мы не будем.

В папке ./repo находятся методы-оболочки для работы с базами данных. Код приведен в приложении З.

Файл ./package.json хранит список пакетов, необходимых для проекта, с нужными версиями.

4. Анализ безопасности приложения

Во второй главе были рассмотрены основные уязвимости парольной аутентификации, а также были даны рекомендации по разработке безопасных веб-приложений. На основе этого был проведен анализ безопасности реализованного приложения. Были отмечены сильные стороны приложения, выявлены его слабые стороны и определен план по усовершенствованию программы в более стойкую к основным уязвимостям.

Рассмотрим сначала сильные стороны приложения.

4.1 Сильные стороны приложения

Сложные пароли.

Программа реализована таким образом, что пользователь не может создать быстро-взламываемый пароль. На него наложены следующие ограничения: пароль должен содержать минимум 8 символов, латинские буквы разных регистров и цифры.

Если накладывать на пароль более жесткие ограничение (например, увеличить минимальную длину до 12 символов или потребовать наличия букв из разных языков), то это вызывает трудности у потенциального пользователя программы, то есть покупателя, которому в данном случае важно удобство, иначе он просто не станет пользоваться данной программой.

Использовать пароли, сгенерированные специальной программой тоже нецелесообразно.

Надежная хеш-функция для хеширования паролей

Пароли пользователей хешируются встроенной в хранимую процедуру MySQL хеш-функцией MD5.В настоящий момент времени она считается достаточно надежной для хранения паролей. Единственным недостатком ее считается быстрое время работы. В случае атаки перебором, время на хеширование практически не тратится, поэтому зачастую рекомендуют использовать хеш-функции, время работы которых искусственно замедленно. Однако ограничение по длине возможного пароля решает эту проблему, потому что для подбора длинных паролей требуется гораздо больше времени.

Безопасное восстановление пароля

Восстановление пароля требует аутентификации по одноразовому паролю, который пользователь получает на свой номер телефона. Это означает, что злоумышленник не может изменить чужой пароль, пока не завладеет чужим телефоном или не сможет перехватывать приходящие смс сообщения, что сделать достаточно трудно, особенно учитывая, что для

подобных сложных манипуляций у злоумышленника недостаточно мотивации.

Двухфакторная аутентификация для важных действий

В рекомендациях разработчику в главе 2 было указано, что для совершения важных действий необходимо требовать от пользователя повторной аутентификации. В данной программе в таких ситуациях пользователю не приходится повторно вводить свой пароль для совершения важных действий. Ему нужно подтвердить свой телефон введением одноразового пароля, полученного им по смс. Такой способ считается даже надежней, нежели повторная аутентификация.

Отдельно от защищенности самой аутентификации, хочется отметить, что в приложении все страницы защищены от несанкционированного входа путем ввода их адреса в адресную строку. Перед тем, как вернуть страницу пользователю, сервер обязательно проверяет авторизованность данного пользователя в системе.

Также сервер проверяет и обрабатывает все входящие данные от клиента по некоторым параметрам, соответствие которым мы ожидаем от полученных данных. Это снижает риск SQL-инъекций, потому что на введенные данные накладываются ограничения.

4.2 Слабые стороны

1. Данное приложение никак не ограничивает число попыток входа, что подвергает его риску атаки полного перебора паролей.
2. Также нет функционала, который заставлял бы пользователей менять пароль через определенное время. Однако, как правило, пользователи редко запоминают пароли от приложений, которыми они пользуются не каждый день. Это значит, что периодическая смена паролей в какой-то степени будет достигаться за счет восстановления забытого пароля.
3. Приложение автоматически не закрывает сессию после продолжительной неактивности клиента, что дает возможность несанкционированного доступа злоумышленнику, который каким-то образом завладел компьютером пользователя. Однако для всех важных действий, пользователю необходимо проходить повторную аутентификацию, что означает, что ничего существенного злоумышленник совершить не сможет.
4. Приложение осуществляет связь с сервером по незащищенному протоколу HTTP. Переход на работу по защищенному каналу планируется в дальнейшей работе над приложением.

4.3 Вывод

Приложение достаточно хорошо защищено для реализуемого функционала. Обнаруженные недостатки не считаются критичными для обеспечения безопасности интернет-магазина. В дальнейшем планируется

ввести дополнительную защиту от перебора паролей, а также перейти на клиент-серверную связь по защищенному HTTPS протоколу.

А также, при расширении используемого функционала планируется увеличение используемых средств защиты информации.

Заключение

В процессе работы были изучены различные способы аутентификации и сферы их применения. Было выбрано два способа для реализации их в логике авторизации на веб-сайте интернет-магазина. Оба способа были исследованы на возможные уязвимости и пути их устранения. На основе данного теоретического материала была реализована программа авторизации на веб-сайте интернет-магазина. Затем программа была проанализирована с точки зрения безопасности, были отмечены ее сильные стороны и выявлены слабые.

В дальнейшем планируется расширить функционал веб-сайта и усилить его защиту.

Список литературы

1. Основы криптографии: Учебное пособие. 3-е изд, испр.и доп. – М.: Гелиос АРВ, 2005. – 480 с., ил. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В.
2. Веб-сайт https://habr.com/ru/company/dataart/blog/262817/
3. Безопасность информационных систем. Учебное пособие, 2015. Ерохин В.В., Погонышева Д.А., Степченко И.Г.
4. Веб-сайт http://yztm.ru/lekc2/l18/
5. Веб-сайт https://ru.wikipedia.org/wiki/Хранимая_процедура
6. Веб-сайт https://habr.com/ru/post/322532/
7. Веб-сайт https://metanit.com/web/nodejs/4.1.php

Создание БД MySQL

```
1.  CREATE DATABASE  IF
2.  -- Table structure for table `rights`
3.  --
4.  NOT EXISTS `online_store`;
5.  USE `online_store`;
6.
7.
8.  --
9.
10. DROP TABLE IF EXISTS `rights`;
11. CREATE TABLE `rights` (
12.   `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
13.   `path` varchar(500) NOT NULL,
14.   `method` enum('GET','POST','PUT','DELETE') NOT NULL,
15.   PRIMARY KEY (`id`)
16. ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
17.
18. --
19. -- Dumping data for table `rights`
20. --
21.
22. LOCK TABLES `rights` WRITE;
23. UNLOCK TABLES;
24.
25. --
26. -- Table structure for table `roles`
27. --
28.
29. DROP TABLE IF EXISTS `roles`;
30. CREATE TABLE `roles` (
31.   `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
32.   `role` varchar(45) NOT NULL,
```

```
33.   PRIMARY KEY (`id`)
34. ) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
35.
36. --
37. -- Dumping data for table `roles`
38. --
39.
40. LOCK TABLES `roles` WRITE;
41. INSERT INTO `roles` VALUES (1,'customer'),(2,'seller');
42. UNLOCK TABLES;
43.
44. --
45. -- Table structure for table `roles_has_rights`
46. --
47.
48. DROP TABLE IF EXISTS `roles_has_rights`;
49. CREATE TABLE `roles_has_rights` (
50.   `role_id` int(10) unsigned NOT NULL,
51.   `right_id` int(10) unsigned NOT NULL,
52.   PRIMARY KEY (`role_id`,`right_id`),
53.   KEY `fk_roles_has_rights_rights1_idx` (`right_id`),
54.   KEY `fk_roles_has_rights_roles1_idx` (`role_id`),
55.   CONSTRAINT `fk_roles_has_rights_rights1` FOREIGN KEY (`right_id`) REFERENCES `rights` (`id`) O
      N DELETE CASCADE ON UPDATE CASCADE,
56.   CONSTRAINT `fk_roles_has_rights_roles1` FOREIGN KEY (`role_id`) REFERENCES `roles` (`id`) ON
      DELETE CASCADE ON UPDATE CASCADE
57. ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
58.
59. --
60. -- Dumping data for table `roles_has_rights`
61. --
62.
63. LOCK TABLES `roles_has_rights` WRITE;
64. UNLOCK TABLES;
65.
66. --
```

```
-- Table structure for table `users`
--

DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `id` bigint(19) unsigned NOT NULL AUTO_INCREMENT,
  `phone` varchar(15) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `phone_UNIQUE` (`phone`)
) ENGINE=InnoDB AUTO_INCREMENT=17 DEFAULT CHARSET=utf8;

--
-- Dumping data for table `users`
--

LOCK TABLES `users` WRITE;
UNLOCK TABLES;

--
-- Table structure for table `users_has_roles`
--

DROP TABLE IF EXISTS `users_has_roles`;
CREATE TABLE `users_has_roles` (
  `user_id` bigint(19) unsigned NOT NULL,
  `role_id` int(10) unsigned NOT NULL,
  `name` varchar(45) NOT NULL,
  `password` varchar(50) NOT NULL,
  `salt` varchar(10) NOT NULL,
  `create_date` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`user_id`,`role_id`),
  KEY `fk_users_has_roles_roles1_idx` (`role_id`),
  KEY `fk_users_has_roles_users_idx` (`user_id`),
  CONSTRAINT `fk_users_has_roles_roles1` FOREIGN KEY (`role_id`) REFERENCES `roles` (`id`) ON
  DELETE CASCADE ON UPDATE CASCADE,
```

```
101.  CONSTRAINT `fk_users_has_roles_users` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`) O
      N DELETE CASCADE ON UPDATE CASCADE
102.) ENGINE=InnoDB DEFAULT CHARSET=utf8;
103.
104.--
105.-- Dumping data for table `users_has_roles`
106.--
107.
108.LOCK TABLES `users_has_roles` WRITE;
109.UNLOCK TABLES;
110.
111.DELIMITER ;;
112.CREATE DEFINER=`root`@`localhost` FUNCTION `generateHash`(
113.    in_password     VARCHAR(40),
114.    in_salt         VARCHAR(10)
115.) RETURNS varchar(40) CHARSET utf8
116.BEGIN
117.    RETURN SHA1(CONCAT(in_password, in_salt));
118.END ;;
119.DELIMITER ;
120.
121.DELIMITER ;;
122.CREATE DEFINER=`root`@`localhost` FUNCTION `generateSalt`() RETURNS varchar(10) CHARSET
      utf8
123.BEGIN
124.
125.    DECLARE  v_salt     VARCHAR(10)     DEFAULT '';
126.    DECLARE  i          INT             DEFAULT 0;
127.
128.    WHILE i < 10 DO
129.        SET v_salt = CONCAT(v_salt,
130.            SUBSTR('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890',
131.                FLOOR(1+RAND()*61), 1));
132.        SET i = i + 1;
133.    END WHILE;
134.
```

```
135.    RETURN v_salt;
136.
137.END ;;
138.DELIMITER ;
139.
140.DELIMITER ;;
141.CREATE DEFINER=`root`@`localhost` PROCEDURE `checkCustomerAbsenceByPhone`(
142.    IN  _phone      VARCHAR(15)
143.)
144.BEGIN
145.
146.    IF EXISTS
147.    (
148.        SELECT * FROM `users`
149.        JOIN `users_has_roles` ON `users_has_roles`.`user_id` = `users`.`id`
150.        WHERE `users`.`phone` = _phone AND `users_has_roles`.`role_id` = 1
151.    )
152.    THEN
153.        SIGNAL SQLSTATE '45001'
154.        SET message_text = 'Customer with this phone number already exist. Please, sign up or choose a
    different number.';
155.    END IF;
156.END ;;
157.DELIMITER ;
158.
159.DELIMITER ;;
160.CREATE DEFINER=`root`@`localhost` PROCEDURE `logIn`(
161.    IN  _role       VARCHAR(10),
162.    IN  _phone      VARCHAR(15),
163.    IN  _password   VARCHAR(20)
164.)
165.BEGIN
166.
167.    DECLARE  v_user_id         BIGINT UNSIGNED     DEFAULT 0;
168.    DECLARE  v_password        VARCHAR(50);
```

```
169.    DECLARE  v_salt           VARCHAR(10);
170.    DECLARE  v_user_role_id    INT UNSIGNED;
171.
172.    SELECT
173.        a.`id`,
174.        b.`id`,
175.        c.`password`,
176.        c.`salt`
177.    INTO
178.        v_user_id,
179.        v_user_role_id,
180.        v_password,
181.        v_salt
182.    FROM `users` a
183.    JOIN `roles` b ON b.`role` = _role
184.    JOIN `users_has_roles` c ON c.`user_id` = a.`id` AND c.`role_id` = b.`id`
185.    WHERE a.`phone` = _phone;
186.
187.    IF v_user_id = 0 THEN
188.        SIGNAL SQLSTATE '45004' SET message_text = 'Phone number unknown. Check your phone nu
    mber or sign up.';
189.    END IF;
190.
191.    IF generateHash(_password, v_salt) = v_password THEN
192.        SELECT v_user_id AS id, v_user_role_id AS roleId;
193.    ELSE
194.        SIGNAL SQLSTATE '45005' SET message_text = 'Password is incorrect.';
195.    END IF;
196.END ;;
197.DELIMITER ;
198.
199.DELIMITER ;;
200.CREATE DEFINER=`root`@`localhost` PROCEDURE `register`(
201.    IN  _phone      VARCHAR(15),
202.    IN  _name       VARCHAR(45),
```

```
    IN _password   VARCHAR(20),
    IN _role       VARCHAR(10)
)
BEGIN

    DECLARE exit handler for sqlexception
    BEGIN
        ROLLBACK;
        RESIGNAL;
    END;

    START TRANSACTION;

    SELECT `id` INTO @role_id FROM `roles` WHERE `role` = _role;
    SET @salt = generateSalt();
    SET @hash = generateHash(_password, @salt);

    SET @user_id = 0;
    SELECT `id` INTO @user_id FROM `users` WHERE `phone` = _phone;

    IF @user_id = 0 THEN
        INSERT INTO `users`(`phone`)
        VALUES(_phone);
        SET @user_id = LAST_INSERT_ID();
    END IF;

    INSERT INTO `users_has_roles`(`user_id`, `role_id`, `name`, `password`, `salt`)
    VALUES(@user_id, @role_id, _name, @hash, @salt);

    COMMIT;
END ;;
DELIMITER ;

DELIMITER ;
CREATE DEFINER=`root`@`localhost` PROCEDURE `checkPhoneExisting`(
```

```
238.    IN  _phone      VARCHAR(15)
239.)
240.BEGIN
241.
242.    IF NOT EXISTS
243.    (
244.        SELECT * FROM `users`
245.        JOIN `users_has_roles` ON `users_has_roles`.`user_id` = `users`.`id`
246.        WHERE `users`.`phone` = _phone AND `users_has_roles`.`role_id` = 1
247.    )
248.    THEN
249.        SIGNAL SQLSTATE '45001'
250.        SET message_text = 'Phone number unknown. Check your phone number or sign up.';
251.    END IF;
252.END;;
253.DELIMITER ;
254.
255.DELIMITER ;
256.CREATE DEFINER=`root`@`localhost` PROCEDURE `changePassword`(
257.    IN  _phone      VARCHAR(15),
258.    IN  _password   VARCHAR(20),
259.    IN  _role       VARCHAR(10)
260.)
261.BEGIN
262.
263.    DECLARE exit handler for sqlexception
264.    BEGIN
265.        ROLLBACK;
266.        RESIGNAL;
267.    END;
268.
269.    START TRANSACTION;
270.
271.    SELECT `id` INTO @role_id FROM `roles` WHERE `role` = _role;
272.    SET @salt = generateSalt();
```

```
273.    SET @hash = generateHash(_password, @salt);
274.
275.    SET @user_id = 0;
276.    SELECT `id` INTO @user_id FROM `users` WHERE `phone` = _phone;
277.
278.    IF @user_id = 0 THEN
279.        SIGNAL SQLSTATE '45001' SET message_text = 'Phone number unknown.';
280.    END IF;
281.
282.     UPDATE `users_has_roles`
283.    SET `password`=@hash,
284.    `salt`=@salt
285.    WHERE `role_id`=@role_id AND `user_id`=@user_id;
286.
287.    COMMIT;
288.END;;
289.DELIMITER ;
```

Код файла ./bin/www

```
1.  const app = require('../app');
2.  const debug = require('debug')('app:server');
3.  const http = require('http');
4.  /**
5.   * Get port from environment and store in Express.
6.   */
7.  const config = require('../config/' + (process.env.NODE_ENV || 'development')),
8.      dbs = require('../libs/dbs');
9.  //var port = normalizePort(process.env.PORT || '3000');
10. const port = normalizePort(process.env.PORT || config.port);
11. app.set('port', port);
12. /**
13.  * Create HTTP server.
14.  */
15. const server = http.createServer(app);
16. /**
17.  * Listen on provided port, on all network interfaces.
18.  */
19. dbs.init().then(() => {
20.     console.log('Соединения с базами данных установлены успешно');
21.     server.listen(port);
22. }).catch(err => {
23.     console.log(err);
24.     process.exit(1);
25. });
26. server.on('error', onError);
27. server.on('listening', onListening);
```

Код файла ./app.js

```
1.  const createError = require('http-errors');
2.  const express = require('express');
3.  const path = require('path');
4.  const logger = require('morgan');
5.  const bodyParser = require('body-parser');
6.  const session = require('express-session');
7.  const expressHbs = require('express-handlebars');
8.  const redisStore = require('connect-redis')(session);
9.  const dbs = require('./libs/dbs');
10. const config = require('./config/' + (process.env.NODE_ENV || 'development'));
11. const passport = require('passport');
12. const localStrategy = require('passport-local').Strategy;
13. const User = require('./repo/userRepository');
14.
15. const customer = require('./routes/customer');
16. const seller = require('./routes/seller');
17. const logout = require('./routes/logout');
18.
19. const indexRouter = require('./routes/index');
20. const usersRouter = require('./routes/users');
21.
22. const app = express();
23.
24. // view engine setup
25. app.set('views', path.join(__dirname, 'views'));
26. app.set('view engine', 'hbs');
27. app.engine("hbs", expressHbs(
28.     {
29.         layoutsDir: "views/layouts",
30.         defaultLayout: "layout",
31.         extname: "hbs"
32.     }
```

```
33. ));
34.
35. app.use(logger('dev'));
36. app.use(express.json());
37. app.use(express.urlencoded({ extended: false }));
38. app.use(express.static(path.join(__dirname, 'public')));
39.
40. app.use(session({
41.   secret: config.sessionSecret,
42.   store: new redisStore({client: dbs.redis}),
43.   saveUninitialized: false, resave: false }));
44.
45. app.use(passport.initialize());
46. app.use(passport.session());
47.
48. passport.serializeUser((userIdAndRoleId, done) => done(null, userIdAndRoleId));
49. passport.deserializeUser((userIdAndRoleId, done) =>
50. User.getByIdAndRole(userIdAndRoleId)
51.     .then(user => done(null, user))
52.     .catch(err => done(err)));
53.
54. passport.use('login', new localStrategy({
55.   usernameField: 'phone',
56.   passwordField: 'password',
57.   passReqToCallback: true },
58.     (req, phone, password, done) =>{
59.         User.login(req.body)
60.             .then(user => {
61.                 return done(null, {id: user.id, roleId: user.roleId})
62.             })
63.             .catch(err => {
64.                 return done(err)
65.             });
66.
67.     }));
```

```
68.
69.  app.use('/customer', customer);
70.  app.use('/seller', seller);
71.
72.  app.use('/logout', logout);
73.
74.  app.use('/', indexRouter);
75.  app.use('/users', usersRouter);
76.
77.  // catch 404 and forward to error handler
78.  app.use(function(req, res, next) {
79.      console.log('will create an error');
80.      next(createError(404));
81.  });
82.
83.  // error handler
84.  app.use(function(err, req, res, next) {
85.      // set locals, only providing error in development
86.      res.locals.message = err.message;
87.      res.locals.error = req.app.get('env') === 'development' ? err : {};
88.
89.      // render the error page
90.      res.status(err.status || 500);
91.      res.render('error');
92.  });
93.
94.  module.exports = app;
```

Код файла ./routes/customer.js

```
1.  const express = require('express');
2.  const router = express.Router();
3.  const smsCode = require('../repo/smsCodeRepository');
4.  const user = require('../repo/userRepository');
5.  const sms = require('../libs/smsLib');
6.  const customError = require('../libs/customError');
7.  const passport = require('passport');
8.  const passwValidator = require('password-validator');
9.  const utils = require('../libs/utils');
10.
11.
12. router.get('/', function (req, res){
13.     if (req.isAuthenticated()) {
14.         res.redirect('/customer/home');
15.     }
16.     else {
17.         res.render('authorization-page');
18.     }
19. });
20.
21. router.get('/signup', function (req, res) {
22.     res.render('signup.hbs');
23. });
24.
25. router.get('/signup/check-code', (req, res, next)=>{
26.     res.render('check-code', {
27.             phone: req.body.phone
28.         });
29. });
30.
31. router.post('/signup', (req, res, next) =>
32.     user.checkCustomerAbsence(utils(req.body.phone))
```

```
33.         .then(() => smsCode.remove(utils(req.body.phone)))
34.         .then(() => smsCode.add(utils(req.body.phone)))
35.         .then(code  => sms.send(utils(req.body.phone), code))
36.         .then(() => res.render('check-code', {
37.             phone: req.body.phone
38.         }))
39.         .catch(err => {
40.             res.render('signup', {errorMessage: err.sqlMessage});
41.         }));
42.
43.
44. router.post('/signup/checkcode', (req, res, next) => {
45.         smsCode.get(utils(req.body.phone), req.body.code)
46.             .then(result => {
47.                 if (!result) return Promise.reject(new customError('You put a wrong code. Please, try again.',
    400));
48.                 return res.render('registration', {
49.                     phone: req.body.phone,
50.                     code: req.body.code
51.                 });
52.             })
53.             .then(() => smsCode.remove(utils(req.body.phone)))
54.             .catch(err => res.render('check-code', {errorMessage: err.message, phone: req.body.phone})
55.             )
56. });
57.
58. router.post('/signup/registration', (req, res, next) =>{
59.     if (!schema.validate(req.body.password)){
60.         res.render('registration', {
61.             phone: req.body.phone,
62.             code: req.body.code,
63.             name: req.body.name,
64.             errorMessage: 'Password  is incorrect. Follow instructions about making passwords.'
65.         });
66.         return;
```

```
67.     }
68.     if (req.body.password===req.body.repeatPassword){
69.     user.register(req.body, 'customer')
70.         .then(() => smsCode.remove(utils(req.body.phone)))
71.         .then(() =>res.render('finish', {message:'You have been registrated!'}))
72.             .catch(err => next(err))
73.     }
74.     else res.render('registration', {
75.         phone: req.body.phone,
76.         code: req.body.code,
77.         name: req.body.name,
78.         errorMessage: 'Passwords do not match.'
79.     });
80. });
81.
82. router.post('/login',
83.     passport.authenticate('login'),
84.     function(req, res, next) {
85.         res.render('home');
86.     },
87.     function (err, req, res, next){
88.         let options = {errorMessage: err.sqlMessage};
89.         if (err.sqlState==='45005')
90.             options.phone = req.body.phone;
91.         res.render('login', options);
92. });
93.
94. router.get('/login', function (req, res) {
95.     if (req.isAuthenticated()) {
96.         res.redirect('/customer/home');
97.     }
98.     else res.render('login.hbs');
99. });
100.
101.router.get('/login/forgotpassword', (req,res)=>{
```

```
102.    res.render('forgot-password', {phone: req.body.phone});
103.});
104.
105.router.post('/login/confirmphone', (req, res)=>{
106.    user.checkPhoneExisting(req.body.phone)
107.        .then(() => smsCode.remove(utils(req.body.phone)))
108.        .then(() => smsCode.add(utils(req.body.phone)))
109.        .then(code  => sms.sendImitate(utils(req.body.phone), code))
110.        .then(() => res.render('confirm-phone', {
111.            phone: req.body.phone
112.        }))
113.        .catch(err => {
114.            res.render('forgot-password', {errorMessage: err.sqlMessage});
115.})});
116.
117.router.post('/login/checkcode', (req,res)=>{
118.    smsCode.get(utils(req.body.phone), req.body.code)
119.        .then(result => {
120.            if (!result) return Promise.reject(new customError('You put a wrong code. Please, try again.', 40
    0));
121.            return res.render('new-password', {
122.                phone: req.body.phone,
123.                code: req.body.code
124.            });
125.        })
126.        .then(() => smsCode.remove(utils(req.body.phone)))
127.        .catch(err => res.render('confirm-phone', {errorMessage: err.message, phone: req.body.phone})
128.        )
129.});
130.
131.router.post('/login/changepassword', (req, res)=>{
132.    if (!schema.validate(req.body.password)){
133.        res.render('new-password', {
134.            phone: req.body.phone,
135.            code: req.body.code,
```

```
136.            errorMessage: 'Password  is incorrect. Follow instructions about making passwords.'
137.        });
138.        return;
139.    }
140.    if (req.body.password===req.body.repeatPassword){
141.        user.changePassword(req.body, 'customer')
142.            .then(() => smsCode.remove(utils(req.body.phone)))
143.            .then(() => res.render('finish', {message:'Your password has been changed.'}))
144.            .catch(err => next(err));
145.    }
146.    else res.render('new-password', {
147.        phone: req.body.phone,
148.        code: req.body.code,
149.        errorMessage: 'Passwords do not match.'
150.    });
151.});
152.
153.router.get('/home', function (req, res, next) {
154.    if (req.isAuthenticated()) {
155.        return next();
156.    }
157.    res.redirect('/customer');
158.}, function(req, res){
159.    res.render('home');
160.});
161.
162.router.get('/account/changepassword', (req, res)=>{
163.    res.render('change-password', {phone:phone});
164.});
165.
166.router.get('/logout', (req, res) => {
167.    req.logout();                           // [1]
168.    req.session.destroy(() => res.redirect('/customer'));        // [2]
169.});
170.
```

```
171.router.get('/*', (req, res)=>{
172.    res.redirect('/customer');
173.});
174.
175.
176.const schema = new passwValidator();
177.schema
178.    .is().min(8)                                    // Minimum length 8
179.    .is().max(100)                                  // Maximum length 100
180.    .has().uppercase()                              // Must have uppercase letters
181.    .has().lowercase()                              // Must have lowercase letters
182.    .has().digits()                                 // Must have digits
183.    .has().not().spaces()                           // Should not have spaces
184.module.exports = router;
```

Код файла ./config/development/index.js

```
1.  const config = {
2.      db : {
3.          mysql : {
4.              host     : 'localhost',
5.              user     : 'root',
6.              database : 'online_store',
7.              password : 'yfghbvth?djn'
8.          },
9.          mongo : 'mongodb://localhost/ourProject'
10.     },
11.     redis : {
12.         port : 6379,
13.         host : '127.0.0.1'
14.     },
15.     port : 3000,
16.     sms : {
17.         api_id : '6E4C6315-99D0-5962-0CF7-4A664E97A0FC', //Выдается при регистрации на sms.ru
18.     },
19.     sessionSecret :'secret'
20. };
21.
22.
23.
24. module.exports =  config;
```

Код файлов в папке ./libs

customError.js

```
'use strict';

module.exports = function CustomError(message, status) {
    Error.captureStackTrace(this, this.constructor);
    this.name = this.constructor.name;
    this.message = message;
    this.status =  status || 500;
};

require('util').inherits(module.exports, Error);
```

dbs.js

```
const mysqlPromise = require('mysql-promise')();
const mongoose = require('mongoose');
const Redis = require('promise-redis')();
const config = require('../config/' + (process.env.NODE_ENV || 'development'));

mysqlPromise.configure(config.db.mysql);
const redis = Redis.createClient(config.redis.port, config.redis.host);

mongoose.Promise = Promise;

function checkMySQLConnection(){
    return mysqlPromise.query('SELECT 1');
}

function checkRedisReadyState() {
    return new Promise((resolve,reject) => {
        redis.once("ready", () => {redis.removeAllListeners('error'); resolve()});
        redis.once('error', e => reject(e));
    })
}
```

```
21.
22. function init() {
23.     return Promise.all([
24.         checkMySQLConnection(),
25.         new Promise((resolve,reject) => {mongoose.connect(config.db.mongo, err =>
26.             err ? reject(err):resolve())}),
27.         checkRedisReadyState()
28.     ]);
29. }
30.
31. module.exports = {
32.     mysql: mysqlPromise,
33.     redis: redis,
34.     init:  init
35. };
```

smsLib.js

```
1.  const doRequest = require('request-promise-native');
2.  const config = require('../config/' + (process.env.NODE_ENV || 'development'));
3.  const customError = require('./customError');
4.
5.  exports.send = (phone, text) => {
6.      const options = {
7.          url: 'http://sms.ru/sms/send',
8.          method: 'GET',
9.          qs: {
10.             to: phone,
11.             from: 'Customer Authorization',
12.             text: text,
13.             api_id: config.sms.api_id,
14.             test: config.sms.test
15.         }
16.     };
17.     return doRequest(options)
18.         .then(result => {
19.             let responseCode = result.split('\n')[0];
```

```
            if(responseCode != '100')
                return Promise.reject(new customError(
                    `Ошибка отправки смс, код ошибки: ${responseCode}`));
            return Promise.resolve();
        })
};
```

utils.js

```
function leaveDigitsOnly(text){
    let result = '';
    for (let i=0; i<text.length; i++){
        if (!isNaN(parseInt(text[i],10)))
            result+=text[i];
    }
    return result;
}

module.exports=leaveDigitsOnly;
```

Код файла ./models/smsCodeModel.js

```
1.  const mongoose = require('mongoose');
2.  const Schema = mongoose.Schema;
3.  const randomCode = require('random-number');
4.
5.  const smsCodeSchema = new Schema({
6.      code: {
7.          type: String,
8.          default: () => randomCode({min: 1000, max: 9999, integer: true})
9.      },
10.     phone: {
11.         type: String,
12.         unique: true
13.     },
14.     createdAt: {
15.         type: Date,
16.         default: Date.now,
17.         expires: 15
18.     }
19. });
20.
21. module.exports = mongoose.model('smsCode', smsCodeSchema);
```

Код файлов в папке ./repo

smsCodeRepository.js

```
1.  const MODEL_PATH = '../models/';
2.  const smsCode = require(MODEL_PATH+'smsCodeModel');
3.
4.
5.  exports.add = phone => smsCode.create({phone}).then(result => result.code);
6.  exports.get = (phone, code) => smsCode.findOne({phone, code});
7.  exports.remove = phone => smsCode.remove({phone});
```

userRepository.js

```
1.  const mysql = require('../libs/dbs').mysql;
2.  const utils = require('../libs/utils');
3.
4.  exports.checkCustomerAbsence = phone =>
5.      mysql.query('CALL checkCustomerAbsenceByPhone(?)', [utils(phone)]);
6.
7.  exports.checkPhoneExisting = phone =>
8.      mysql.query('CALL checkPhoneExisting(?)', [utils(phone)]);
9.
10. exports.checkSellerAbsence = phone =>
11.     mysql.query('CALL checkSellerAbsenceByPhone(?)', [utils(phone)]);
12.
13. exports.register = (body, role) =>
14.     mysql.query('CALL register(?,?,?,?)', [utils(body.phone), body.name, body.password, role]);
15.
16. exports.changePassword = (body, role) =>
17.     mysql.query('CALL changePassword(?,?,?)', [utils(body.phone), body.password, role]);
18.
19. exports.getByIdAndRole = user =>
20.     mysql.query(`SELECT a.phone, b.name, b.role_id FROM users a
21.                 JOIN users_has_roles b ON b.user_id = a.id AND b.role_id = ?
22.                 WHERE a.id = ?`, [user.roleId, user.id])
23.         .spread(result => result[0]);
```

```
24.
25.  exports.login = body =>
26.      mysql.query('CALL logIn(?,?,?)', [body.role, utils(body.phone), body.password])
27.          .spread(result => result[0][0]);
```