**Аннотация**

Тема: Разработка модуля аутентификации Web-приложения с применением сертификатов X.509.

Объем бакалаврской работы 51 страница (41 без учета приложений), на которых размещено 14 иллюстраций. При написании работы использовались 19 источников.

Ключевые слова: информационная безопасность, сертификат X.509, Web, Аутентификация.

Цель работы: реализация аутентификации на основе сертификата X.509.

Для данной работы использовались открытые источники.

Результаты: было разработано Web-приложение, которое способно регистрировать пользователей, выдавать им сертификат и аутентифицировать с помощью него.

Работа состоит из введения, основной части, состоящей из четырех глав, и заключения.

В первой главе была разобрана теоретическая часть, где были рассмотрены протокол TLS и сертификат X.509 и подведены их достоинства и недостатки.

Во второй главе разработана архитектура модуля аутентификации, его алгоритм работы и представлен код на Python для настройки TLS соединения и создание сертификата на стороне удостоверяющего центра.

В третьей главе реализован интерфейс Web-приложения, где пользователь уже может взаимодействовать с сервером.

В четвертой главе проведено внедрение модуля аутентификации и проведено тестирование реализованного Web-приложения.

В заключении делается вывод о проделанной работе. Проведенная работа показала, что аутентификация по сертификату дает возможности для более безопасного и удобного входа в систему из-за того, что не нужно запоминать пароли. Однако данный метод имеет недостатки, из-за чего его сложно внедрить в приложения с очень большим количеством пользователей.

# **СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 6](#_Toc136764168)

[**ГЛАВА 1 ОБЗОР ПРОТОКОЛА TLS И СЕРТИФИКАТОВ X.509** 7](#_Toc136764169)

[**1.1** **Основы протокола TLS** 7](#_Toc136764170)

[**1.2** **Применение сертификатов X.509** 9](#_Toc136764171)

[**1.3** **Процедура установки сессии TLS и проверка подлинности   
сертификатов X.509** 12](#_Toc136764173)

[**1.4 Достоинства и недостатки применения протокола TLS и сертификатов X.509** 14](#_Toc136764173)

[**Выводы по первой главе** 16](#_Toc136764174)

[**Глава 2 Разработка модуля аутентификации** 17](#_Toc136764175)

[**2.1 Разработка архитектуры модуля аутентификации** 17](#_Toc136764176)

[**2.2 Проектирование алгоритма аутентификации** 18](#_Toc136764177)

[**2.3 Разработка программной реализации модуля аутентификации** 20](#_Toc136764177)

[**Выводы по второй главе** 24](#_Toc136764179)

[**Глава 3 Разработка Web-приложения** 25](#_Toc136764180)

[**3.1 Разработка интерфейса приложения и работа серверной части** 25](#_Toc136764181)

[**3.2 Работа клиентской части** 27](#_Toc136764182)

[**Выводы по третьей главе** 31](#_Toc136764183)

[**Глава 4 Внедрение и тестирование модуля аутентификации в Web-приложение** 32](#_Toc136764184)

[**4.1 Тестирование модуля на работоспособность** 32](#_Toc136764186)

[**4.2 Оценка использования модуля в практических условиях** 36](#_Toc136764187)

[**Выводы по четвертой главе** 38](#_Toc136764188)

[**зАКЛЮЧЕНИЕ** 39](#_Toc136764189)

[**сПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 40](#_Toc136764190)

[**Приложение а. КОД ПРОГРАММной реализации модуля аутентификации Web-приложения с применением   
сертификатов X.509** 42](#_Toc136764191)

# **ВВЕДЕНИЕ**

В современном информационном обществе безопасность и аутентификация пользователей в Web-приложениях имеют важное значение. Кибератаки, взломы аккаунтов и кража личных данных становятся все более распространенными и серьезными проблемами. Протокол TLS и сертификаты X.509 являются стандартами для обеспечения безопасности в сети, их применение в разработке модуля аутентификации позволит повысить уровень защиты Web-приложений и обеспечить безопасную аутентификацию пользователей. Разработка модуля аутентификации с применением сертификатов X.509 актуальна для обеспечения безопасности Web-приложений и защиты пользователей от киберугроз.

Помимо сертификатов X.509, при разработке модуля аутентификации Web-приложения следует учитывать использование протокола TLS (Transport Layer Security). Для работы TLS необходимы сертификаты X.509, которые используются для аутентификации сервера и клиента. При установке защищенного соединения клиент проверяет сертификат сервера на валидность, что позволяет убедиться в том, что он связывается с тем сервером, который действительно является владельцем домена или поддомена.

Цель работы: разработка модуля аутентификации Web-приложения с применением сертификатов X.509.

Задачами работы являются:

* изучение концепции аутентификации и сертификатов X.509;
* исследование протоколов аутентификации Web-приложений;
* разработка модуля аутентификации, использующего сертификаты X.509 для проверки подлинности пользователей;
* интеграция разработанного модуля с Web-приложением и его тестирование на различных сценариях;
* оценка безопасности и эффективности разработанного модуля с использованием сертификатов X.509.

# **ОБЗОР ПРОТОКОЛА TLS И СЕРТИФИКАТОВ X.509**

В этой главе будут решаться задачи изучения концепции аутентификации и сертификатов X.509 и исследование протоколов аутентификации Web-приложений. Таким образом будут рассмотрены основы протокола TLS (Протокол защиты транспортного уровня) и цифровых сертификатов X.509. Будет изучена история и развитие этих технологий, а также их ключевые функции и принципы работы, а также будут учтены достоинства и недостатки обоих технологий.

### **Основы протокола TLS**

Протокол TLS (Transport Layer Security) — это стандарт криптографической защиты связи в сетях Интернет. Он используется для обеспечения безопасной передачи данных между клиентским и серверным приложением [1] [2].

В 1994 году компанией Netscape Communications Corporation был разработан протокол безопасной передачи данных SSL, который обеспечивает безопасность в Интернете.

История развития SSL началась с появления в Интернете электронной коммерции. В то время онлайн-платежи были довольно рискованными, так как данные клиентов могли быть легко перехвачены. Разработчики Netscape создали SSL, чтобы решить эту проблему и обеспечить безопасность при передаче конфиденциальных данных.

SSL продолжал эволюционировать и получил две главные версии - SSL 2.0 и SSL 3.0. Однако последняя версия, SSL 3.0, стала уязвимой и была отключена во многих сетях Интернета.

В 1999 году протокол TLS был разработан как следующая версия SSL (Secure Sockets Layer), но в 2015 году наследовал SSL и сейчас широко используется во всем Интернете.

Протокол TLS выполняет функции [2]:

1. Аутентификации;
2. Шифрования;
3. Согласования ключей;
4. Контроля целостности;
5. Управления сессиями;
6. Поддержку различных протоколов.

Рассмотрим более подробно данные функции:

TLS использует цифровые сертификаты для проверки подлинности сервера и клиента, которые выдаются доверенными центрами сертификации (CA), проверяющие подлинность запроса и выдают сертификаты.

TLS использует различные алгоритмы шифрования, такие как AES (Advanced Encryption Standard), для защиты передаваемых данных от несанкционированного доступа или прослушивания.

TLS использует протокол обмена ключами, который позволяет клиенту и серверу согласовать ключи, используемые для шифрования и дешифрования данных, что позволяет обеспечить безопасность передачи данных при открытом канале связи.

TLS использует хэширование для контроля целостности передаваемых данных, что позволяет проверить, не были ли данные изменены или повреждены в процессе передачи.

TLS позволяет создавать и управлять сессиями между клиентом и сервером, что позволяет ускорить передачу данных и уменьшить нагрузку на сеть.

TLS поддерживает различные протоколы, такие как HTTP, SMTP, FTP и многие другие, что позволяет обеспечить безопасную передачу данных в различных приложениях и сетевых протоколах.

Сегодня TLS стал обязательным для защиты данных в Интернете, особенно при использовании онлайн-банкинга, электронной коммерции и других важных сетевых сервисов. Он обеспечивает шифрование данных, аутентификацию и целостность сообщений.

В целом, протокол TLS обеспечивает безопасный обмен данными в сети Интернет, снижая риски несанкционированного доступа и повышая уровень защиты конфиденциальной информации.

Однако, по историческим причинам на многих сайтах до сих пор можно увидеть использование обоих терминов - SSL/TLS. Это связано с тем, что термин SSL широко использовался в прошлом и до сих пор существует много серверов и устройств, которые используют устаревшие версии SSL вместо TLS. В этом случае термин SSL/TLS используется, чтобы охватить оба протокола и подчеркнуть их связь и совместимость.

### **1.2 Применение сертификатов X.509**

Сертификат X.509 — это стандарт криптографического сертификата для проверки подлинности участников взаимодействия в сети Интернет. Он используется для защиты конфиденциальной информации и обеспечения безопасности в области электронной коммерции, банковской деятельности, почтовой и телекоммуникационной связи и других областей, где требуется подлинность участников взаимодействия [1] [3].

В случае использования сертификата X.509, стороны получают документ, который подтверждает подлинность их личности, а также информацию о подписывающей стороне и срок действия сертификата.

Сертификат X.509 включает в себя [3]:

1. Идентификатор версии;
2. Серийный номер;
3. Имя владельца;
4. Имя издателя;
5. Срок действия;
6. Открытый ключ;
7. Алгоритм подписи.

Идентификатор версии указывает на версию стандарта X.509, используемую для создания сертификата.

Серийный номер содержит уникальный идентификационный номер, который присваивается каждому сертификату в момент его создания.

Имя владельца содержит информацию об имени и фамилии владельца сертификата, компании, которая выпустила сертификат, а также другую информацию об идентификации владельца.

Имя издателя содержит информацию о компании или организации, которая выпустила сертификат, т.е. о центре сертификации, который удостоверяет личность владельца.

Срок действия указывает на даты начала и окончания срока действия сертификата.

Открытый ключ содержит открытый ключ, который используется для проверки подписи сертификата владельца.

Алгоритм подписи содержит информацию об алгоритме, используемом для создания электронной подписи владельца сертификата.

Применение сертификата X.509 включает в себя [3]:

1. Проверку подлинности;
2. Шифрование;
3. Цифровую подпись;
4. Публичную часть ключа;

Сертификат X.509 позволяет проверить подлинность участников взаимодействия и обеспечить безопасную передачу данных.

Сертификат используется для шифрования данных в сети Интернет, что позволяет обеспечить конфиденциальность передаваемых данных.

Сертификат позволяет формировать цифровую подпись на документах и сообщениях, что обеспечивает подлинность и целостность передаваемых данных.

Сертификат содержит публичную часть ключа, которая позволяет использовать ключ для шифрования данных и проверке подписи владельца.

Прежде чем использовать сертификат X.509, необходимо установить доверительную связь с выдавшей стороной. В противном случае сертификат может быть недействительным или поддельным. Для этого необходимо иметь доступ к информации о выдавшей стороне, а также к открытым ключам для проверки подлинности сертификата.

Также стоит отметить, что, сертификаты X.509 могут быть созданы самостоятельно (самоподписанные сертификаты) или выданы Удостоверяющим Центром (УЦ).

Самоподписанный сертификат является сертификатом, который был создан и подписан самим владельцем без участия УЦ. Этот метод часто используется для собственных нужд или для ограниченного круга пользователей, так как его подлинность нельзя проверить автоматически. При попытке подключения к серверу, имеющему самоподписанный сертификат, пользователь может увидеть предупреждение в своем Web-браузере или почтовом клиенте о том, что подлинность сертификата не может быть подтверждена.

Сертификаты, выданные УЦ, обеспечивают большую степень доверия и надежности, так как УЦ обеспечивает проверку личности владельца сертификата перед его выпуском. Этот процесс обеспечивает подтверждение подлинности сертификата в глазах пользователя. Такие сертификаты используются для защиты сайтов, электронной почты, VPN и других сетевых сервисов, где требуется безопасность.

В целях упрощения процесса выдачи и установки сертификата на сервере, некоторые программы и операционные системы предоставляют пользователю самоподписанный сертификат в качестве временной меры, пока не будет получен сертификат от УЦ.

Еще одной важной особенностью является то, что сертификаты безопасности можно передавать не только от сервера к клиенту, но и наоборот. Для этого необходимо внести изменения в настройках соединения по протоколу TLS. После настройки сервер будет отправлять запрос на клиентский сертификат, когда пользователь будет заходить на сайт. Через диалоговое окно будет необходимо выбрать сертификат и отправить его. Если этот сертификат подходит для сервера, то происходит аутентификация клиента, и доступ предоставляется. Если у пользователя не установлено клиентских сертификатов или отправленный сертификат не прошел проверку, то соединение будет разорвано.



При этом клиентские сертификаты для данных сайтов не являются самоподписанными. Обычно, для получения такого сертификата пользователь должен обращаться к своему провайдеру услуг и получать от него сертификат, который будет предъявляться на соответствующем сайте в качестве подтверждения личности пользователя. Кроме того, для получения клиентского сертификата, пользователь должен предоставить персональные данные. Таким образом, клиентские сертификаты для сайтов являются выданными в организациях сертификации и не могут быть самоподписанными.

После этого сертификат для сайта загружается на компьютер пользователя и устанавливается в хранилище сертификатов. Это может быть как собственное хранилище браузера, так и хранилище операционной системы. В системе Windows клиентские сертификаты относятся к категории "личные" (Private), что означает, что они используются для идентификации пользователя и не могут быть перенесены на другой компьютер без ущерба для безопасности соединения, что является одним из способов обеспечения безопасности в интернет-коммуникациях.

Таким образом, сертификат X.509 является важным инструментом для обеспечения безопасности в сети Интернет, который позволяет проверять подлинность участников взаимодействия и обеспечивает защиту конфиденциальной информации при передаче через сеть.

### **1.3 Процедура установки сессии TLS и проверка подлинности сертификатов X.509**

Установка сессии TLS (Transport Layer Security) обеспечивает безопасное соединение между клиентом и сервером в сети Интернет. Эта процедура включает в себя несколько этапов [5] [6]:

1. Начало соединения;
2. Обмен сертификатами;
3. Согласование параметров шифрования;
4. Обмен ключами;
5. Проверка целостности данных;
6. Завершение сессии.

Клиент отправляет запрос на установку соединения с сервером, а тот в свою очередь отправляет ответ, содержащий информацию о себе, как правило, это сообщение "Server Hello".

Сервер отправляет клиенту свой сертификат, который содержит открытый ключ сервера и запрос на клиентский сертификат и если он не самоподписанный, то клиент проверяет его подлинность, используя доверенный сертификат, который заранее был получен от Удостоверяющего центра.

На этапе установления соединения клиент и сервер согласовывают параметры шифрования и настройки безопасности, которые будут использоваться для защиты передаваемых данных - клиент генерирует случайный ключ сеансового шифрования, который шифруется с использованием открытого ключа сервера и отправляется ему, а сервер расшифровывает ключ и соглашается использовать его для шифрования и расшифровки данных, которые будут передаваться между клиентом и сервером.

Клиент и сервер используют функцию хэш-суммы и ключа шифрования, которые были установлены ранее, для проверки целостности передаваемых данных.

Когда сессия завершается, клиент и сервер отправляют друг другу сообщение об окончании сессии, что переводит соединение в состояние "закрыто".

Проверка подлинности происходит на этапе обмена сертификатами. Клиент использует доверенный сертификат и открытый ключ Удостоверяющего центра для проверки подлинности сертификата, который был отправлен сервером. Если сертификат действительный и проверка проходит успешно, клиент может установить безопасное соединение с сервером.

### **Достоинства и недостатки применения протокола TLS и сертификатов X.509**

Протокол TLS является одним из самых распространенных протоколов для защищенной передачи данных в Интернете. Однако, как и любая технология, у него есть свои достоинства и недостатки.

К достоинствам можно отнести [2] [7]:

1. Безопасность;
2. Распространенность;
3. Простоту использования;
4. Поддержка SSL.

Протокол TLS обеспечивает высокий уровень безопасности для передачи данных, что предотвращает несанкционированный доступ и обеспечивает конфиденциальность передаваемых данных, при условии правильной настройки.

TLS является самым распространенным протоколом для безопасной передачи данных в Интернете, что означает, что большинство браузеров, почтовых программ, серверов и сайтов поддерживают TLS как стандартный протокол передачи данных.

TLS достаточно прост в использовании и настройке, что делает его популярным среди разработчиков и сервисных компаний.

TLS совместим с SSL (Secure Sockets Layer) и несколько улучшает его функциональность.

К недостаткам можно отнести [2] [7]:

1. Небольшую задержку;
2. Сложность при настройке;
3. Отсутствие поддержки протокола на некоторых устройствах.

Использование TLS может привести к небольшой задержке при передаче данных, что может оказывать влияние на производительность приложений и сайтов.

Настройка TLS может оказаться сложной, особенно для технологически отсталых компаний, которые не занимаются безопасностью.

Некоторые программы и устройства могут не поддерживать TLS, что может создать проблемы при передаче данных.

Сертификаты X.509 широко используются в различных приложениях, таких как Web-браузеры, электронная почта, VPN и многих других. Рассмотрим некоторые преимущества и недостатки применения сертификатов X.509.

К преимуществам можно отнести [3] [7]:

1. Безопасность;
2. Проверка подлинности;
3. Широкое распространение.

Сертификаты X.509 обеспечивают безопасность передаваемых данных, так как позволяют проверить подлинность Web-сайта или сервера, а также защищают данные, передаваемые между клиентом и сервером.

Проверка подлинности является важной частью безопасности, и сертификаты X.509 позволяют гарантировать, что пользователь связался именно с ожидаемым сервером.

Сертификаты X.509 широко используются и поддерживаются многими платформами, программным обеспечением и устройствами.

К недостаткам можно отнести [3] [7]:

1. Платную сертификацию;
2. Сложность настройки;
3. Необходимость обновления.

Использование сертификатов X.509 от удостоверяющего центра требует регулярной оплаты.

Настройка сертификатов X.509 может быть сложной и требовать определенных знаний и опыта.

Сертификаты X.509 нужно периодически обновлять, чтобы обеспечить безопасную передачу данных, что может привести к дополнительным расходам на обслуживание.

### **Выводы по первой главе**

В данной главе были решены задачи изучения концепции аутентификации и сертификатов X.509 и исследование протоколов аутентификации Web-приложений. Для этого были рассмотрены основы протокола TLS и сертификатов X.509, а также процедура установки сессии TLS и проверка подлинности сертификатов. Были выделены как достоинства, так и недостатки применения протокола TLS и сертификатов X.509, такие как безопасность, проверка подлинности, широкое распространение, дороговизна, сложности в настройке и необходимость периодического обновления. В целом, протокол TLS и сертификаты X.509 являются важными инструментами для обеспечения безопасности передачи данных в Интернете.

На основе этого вывода можно сказать, что данные протоколы являются актуальными и по сей день и далее будет реализовано Web-приложение на их основе.

# [**Разработка модуля аутентификации**](#_Toc104675229)

В данной главе будет решена задача разработки модуля аутентификации, использующего TLS и сертификаты X.509. Будет рассмотрена архитектура модуля, его ключевые компоненты и особенности работы.

В главе подробно описаны процессы генерации ключевой пары, создания и подписи сертификатов, взаимной аутентификации и аутентификации на сервере с использованием сертификата.

### [**2.1 Разработка архитектуры модуля аутентификации**](#_Toc104675230)

Модуль аутентификации предназначен для проверки подлинности пользователей при попытке установить соединение с сайтом. Существует множество методов аутентификации, но одним из наиболее надежных и безопасных является использование протокола TLS и сертификатов X.509.

Архитектура модуля аутентификации с использованием протокола TLS и сертификатов X.509 будет включать компоненты [6] [9] [10]:

1. Клиентского модуля аутентификации;
2. Осинового сервера;
3. Сервера регистрации
4. Удостоверяющего центра
5. Базы данных пользователей.

Компонент клиентского модуля аутентификации будет встроен в пользовательский интерфейс приложения, где будет предоставлена возможность для взаимодействия с сайтом.

Компонент основного сервера будет проверять сертификат пользователя на основе подписи УЦ и аутентифицировать в случае успеха.

Так как сервер запрашивает сертификат сразу при установлении соединения, то необходим такой компонент, как сервер регистрации, на который пользователь будет заходить, чтобы иметь возможность зарегистрироваться и получить сертификат и после регистрации пользователь также будет добавлен в базу данных.

Компонент удостоверяющего центра будет создавать сертификаты и, в случае подлинности данных владельцев, ставить свою подпись для подтверждения [11].

Компонент базы данных будет хранить информацию о пользователях и управляться через язык SQL.

### [**2.2 Проектирование**](#_Toc104675230) **алгоритма аутентификации**

Реализация базовой аутентификации с помощью TLS и X.509 включает в себя следующие этапы [6] [9] [10]:

1. Генерацию ключевой пары на стороне сервера;
2. Отправку запроса на выпуск сертификата;
3. Создание и подпись сертификата у удостоверяющего центра;
4. Отправку подписанных сертификатов серверу и клиенту;
5. Взаимную аутентификацию;
6. Аутентификации на сервере.

На этапе генерации ключевой пары (приватного и публичного ключей) [12]:

Сервер отправляет запрос (CSR - Certificate Signing Request), который содержит публичный ключ и информацию о клиенте, такую как имя и электронная почта, а приватный ключ должен находиться у владельца и никогда никому не должен передаваться.

Удостоверяющий центр выпускает сертификаты на основе полученных данных и добавляет подпись с использованием своего закрытого ключа, что гарантирует, что сертификат не был подделан и его данные не были изменены.

УЦ отправляет подписанные сертификаты клиенту, которые затем могут использоваться для аутентификации.

Сервер отправляет свой сертификат клиенту при установке соединения. Опционально добавляем настройку, по которой клиент отправляет в ответ свой сертификат и на сторонах производится проверка подлинности, основанная на доверии к удостоверяющему центру, а именно на основе доступа к публичному ключу УЦ для проверки подписи сертификата и проверки его целостности.

Если проверка сертификата успешна, сервер аутентифицирует клиента и устанавливает сеанс соединения и когда клиент закрывает сайт, то соединение автоматически завершается.

Таким образом была создана диаграмма, на которой изображены шаги алгоритма аутентификации на рисунке 2.1. На нем изображены 4 субъекта:

* Клиент;
* Сервер регистрации;
* Основной сервер;
* Удостоверяющий центр (УЦ).

Клиентом является пользователь, который взаимодействует с Web-приложением посредством браузера.

Сервер регистрации представляет собой отдельный независимый сервер, с которым взаимодействует клиент для регистрации и получения сертификата или для перевыпуска сертификата. Сервер будет добавлять информацию о новых пользователях в свою базу данных.

Под основным сервером понимается отдельный независимый сервер, на котором будет находиться основной контент возможной организации. Для демонстрации будет лишь разработана приветствующая страница. При попытке установить соединение, сервер будет запрашивать клиентский сертификат.

Удостоверяющий центр будет заниматься выпуском сертификатов, как для клиентом, так и для серверов. Сервер регистрации будет отправлять CSR-запрос УЦ от имени клиента, в который будет добавляться электронная почта для последующей идентификации пользователей на серверах.

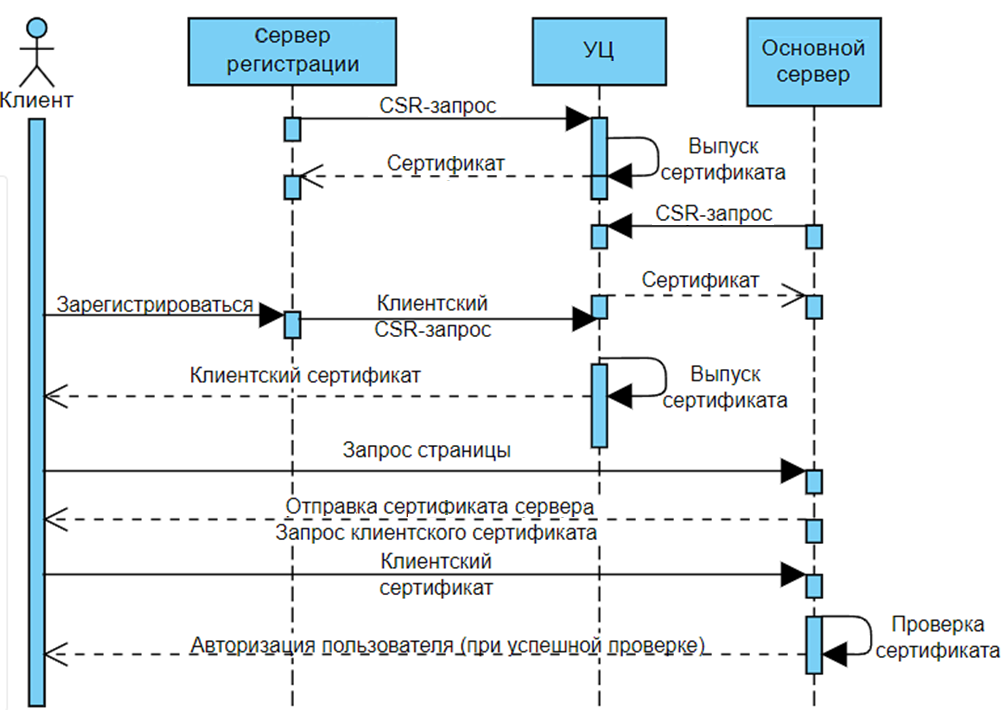


Рисунок 2.1 - UML-диаграмма работы ПО

### [**2.3 Разработка программной реализации модуля**](#_Toc104675230) **аутентификации**

Для разработки Web-приложения была выбрана библиотека Flask, которая способна не только реализовать графическое и функциональное взаимодействие для пользователя, но и организовать и настроить безопасное соединение HTTPS на основе протокола TLS версий 1.1, 1.2 и 1.3 [13] [14].

Web-приложения, которые создаются для широкого использования, применяют такие популярные серверные ПО как Apache или Nginx, библиотека Flask поставляется с встроенным сервером разработки. Для демонстрации работы аутентификации этого вполне достаточно, однако стоит отметить, что этот сервер не рекомендуется использовать в продуктивной среде, так как он не предназначен для обработки больших нагрузок, а также предлагает реализацию управления TLS на уровне приложения, а не на уровне сервера.

В работе будет развернуто 3 независимых сервера, каждый из которых будет отвечать за удостоверяющий центр, сервер регистрации пользователей и основной сервер. На всех будет настроен TLS для HTTPS соединения между каждым из них.

Для задания настроек используется так называется SSLContext [15] [16] [17] – класс, в котором содержаться необходимые функции для установки необходимых параметров безопасности на основе протокола TLS. Обязательной частью для настройки TLS является указание местоположения сертификата и приватного ключа, без которых соединение будет просто сбрасываться. Также необходимо указать местоположение корневого сертификата. Опционально можно отключить использование протокола TLS версий 1.1 и 1.2, тем самым принуждая использовать только версию 1.3, однако использование такой настройки будет зависеть от условий и целей развертывания сервера. Еще одной необходимой опциональной настройкой является запрос клиентского сертификата при установлении соединения пользователя с приложением. Это и будет ключевой настройкой, которая позволит аутентифицировать пользователей. Далее запускаем каждый сервер работать на разные порты можно делать запуск.

Следующей задачей будет реализация создания сертификата, его подписание УЦ и обработка, и ответ на CSR-запрос (Certificate Signing Request), который создается на стороне отправителя.

Для генерации сертификата и CSR-запроса воспользуемся библиотекой cryptography, в которой есть необходимые инструменты для работы. Первым делом клиент на своей стороне генерирует приватный и публичный ключи и создает CSR-запрос, который отправляет УЦ. Для УЦ клиентом будет любой субъект, делающий запрос, поэтому в данном случае это будет относиться как к пользователю, так и к серверу [18] [19].

УЦ будет получать на вход публичный ключ отправителя, его имя и почту. Все эти данные вкладываются в сертификат и будут отображаться при его открытии. Также сертификат подписывается публичным ключом УЦ для его проверки клиентом. Далее выставляется срок действия сертификата. Оптимальным сроком будет 365 дней. Одним из важных вложений является «альтернативное имя», которое содержит все используемые домены и поддомены приложением. Браузер проверяет соответствие домена со значением поля альтернативное имя и в случае их несоответствия считает сертификат недействительным. Так как опытным путем было выяснено, что при указании IP адреса в качестве альтернативного имени браузер не будет считать соединением безопасным, поэтому было принято решение установить домен «localhost».

Созданный сертификат сериализируется и кодируется в формат PEM, который удобно передавать, так как он представляет собой обычный текст. Также копия сертификата сохраняется на стороне УЦ в .pem формате. В качестве названия файла используется email адрес, так как он будет всегда уникальным и не позволит пользователям регистрироваться по нему более одного раза. Это необходимо для того, чтобы получать запросы на проверку существующего сертификата, его перевыпуск при истечении срока или утери пользователем.

Перед тем, как передать пользователю готовый сертификат он упаковывается в контейнер PKCS#12 совместно с его приватным ключом. Это необходимо, так как браузер будет работать только с такими контейнерами. После этого идет скачивание контейнера в формате .p12. Также пользователь скачивает корневой сертификат в формате .crt. При открытии файлов форматов .p12 и .crt . запускается окно мастера импорта сертификатов, который позволяет установить сертификаты в хранилище.

Клиентский контейнер помещается в хранилище сертификатов Windows в директорию «Личное», а корневой сертификат в «доверенные корневые центры сертификации». Если все было сделано верно, то браузер укажет, что соединение безопасное. По клиентскому сертификату теперь можно будет войти на основной сервер. При попытке установить соединение в браузере появится диалоговое окно для выбора клиентских сертификатов. После отправки и успешной установки произойдет аутентификация пользователя. Произойдет переход с сервера регистрации на основной сервер. Таким образом пользователь перейдет с сервера регистрации на основной сервер. Программная реализация настроек TLS и сессии представлена в листинге 2.1, создание клиентского и корневого сертификатов в листинге 2.2.

Листинг 2.1 - Настройка TLS и сессии с применением Flask для основного сервера

app = Flask(\_\_name\_\_)

app.secret\_key = os.urandom(24) # Генерация секретного ключа

app.config['SESSION\_TYPE'] = 'filesystem'

app.config['SESSION\_COOKIE\_SECURE'] = True

app.config['SESSION\_PERMANENT'] = True

Session(app)

context = ssl.SSLContext(ssl.PROTOCOL\_TLS) # Настраиваем TLS  
context.load\_cert\_chain('investCompany\_cert.pem', 'investCompany\_key.pem')  
context.load\_verify\_locations('CA\_cert.pem')  
context.options |= ssl.OP\_SINGLE\_ECDH\_USE  
context.options |= ssl.OP\_NO\_TLSv1 | ssl.OP\_NO\_TLSv1\_1  
context.verify\_mode = ssl.CERT\_REQUIRED  
app.run(host='localhost', threaded=True, debug=True, ssl\_context=context, port=443)

Листинг 2.2 - Выпуск клиентского и корневого сертификатов

def generate\_certificate(csr):

certificate = x509.CertificateBuilder()\

.subject\_name(csr.subject)\

.issuer\_name(issuer)\

.public\_key(csr.public\_key())\

.serial\_number(x509.random\_serial\_number())\

.not\_valid\_before(datetime.datetime.utcnow())\

.not\_valid\_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))\

.add\_extension(x509.BasicConstraints(ca=False, path\_length=None), critical=True,) \

.add\_extension(x509.AuthorityKeyIdentifier.from\_issuer\_public\_key(CA\_private\_key.public\_key()), critical=False)\

.add\_extension(x509.SubjectKeyIdentifier.from\_public\_key(csr.public\_key()), critical=False) \

.add\_extension(x509.SubjectAlternativeName([x509.DNSName('localhost')]), critical=False) \

.sign(CA\_private\_key, hashes.SHA256(), default\_backend())

# Поиск расширения с информацией о электронной почте

email = csr.subject.get\_attributes\_for\_oid(x509.NameOID.EMAIL\_ADDRESS)[0].value

# Сохранение сертификата в файл в формате PEM

cert\_pem = certificate.public\_bytes(serialization.Encoding.PEM)

with open(f'cert/{email}\_cert.pem', "wb") as file:

file.write(cert\_pem)

return cert\_pem

### [**Выводы по второй главе**](#_Toc104675237)

В данной главе была решена задача разработки модуля аутентификации, использующего TLS и сертификаты X.509, используемые для создания защищённого соединения между клиентом и сервером. Она включает в себя такие компоненты как клиентский модуль и основной сервер, аутентифицирующий по сертификату, сервер регистрации, удостоверяющий центр, протокол TLS, сертификаты X.509 и базу данных пользователей.

Аутентификация происходит по следующему принципу: генерация ключевой пары на сервере и клиенте, затем отправка запроса на выпуск сертификата. Удостоверяющий центр создаёт и подписывает сертификат, который затем отправляется серверу и клиенту. Это позволяет обеспечить взаимную аутентификацию. Далее на сервере происходит аутентификация с использованием сертификата. В итоге это обеспечивает уверенность в подлинности и безопасности соединения между сервером и клиентом.

# [**Разработка Web-приложения**](#_Toc104675238)

В данной главе будет рассмотрен процесс разработки Web-приложения на библиотеке Flask. Будут разобраны этапы проектирования и разработки приложения, которые будут состоять из реализации серверной и клиентской частей, а также развертывание удостоверяющего центра, который будет выпускать и проверять сертификаты. После этого необходимо будет настроить взаимную связь, чтобы каждый участник мог посылать друг другу запросы и получать их.

### **[3.1 Разработка интерфейса](#_Toc104675240) приложения и работа серверной части**

Для реализации графической части используется оператор @app.route(), который в качестве аргументов принимает относительный адрес страницы, где будет выполняться код, и методы отправки запроса пользователем. В данной работе используется метод для чтения данных GET и метод для отправки данных POST. Далее выполняется функция с кодом и обязательным условием является возвращение ответа return.

Работа с базой данных была сделана на сервере регистрации и при переходе на основной сервер происходит запрос к серверу регистрации для получения этих данных. Поиск в базе происходит на основании электронной почты, указанной в сертификате.

Для данной работы было разработаны 4 страницы, с которыми взаимодействует пользователь:

* Входная страница;
* Страница регистрации;
* Страница восстановления (перевыпуска) сертификата;
* Домашняя страница.

Реализация страниц представлена на Рисунок 3.1, Рисунок 3.2, Рисунок 3.3.



Рисунок 3.1 - страница входа

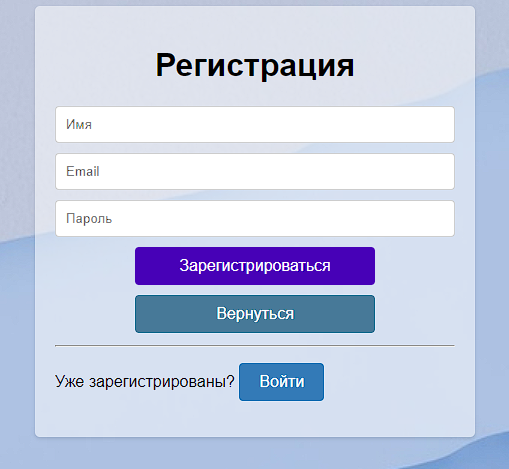


Рисунок 3.2 - страница регистрации

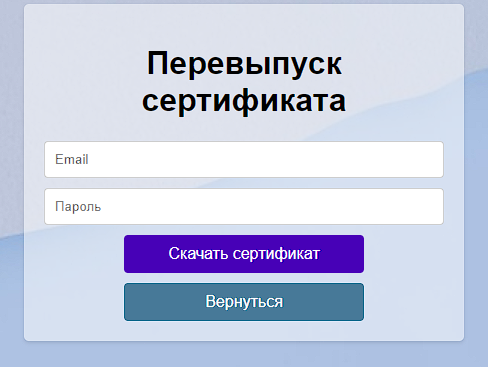


Рисунок 3.3 - Страница восстановления (перепуска) сертификата

### [**3.2 Работа клиент**](#_Toc104675240)**ской части**

Работа клиентской части в значительной степени связана с использованием HTML и CSS, которые формируют пользовательский интерфейс Web-приложения.

Когда пользователь проходит процедуру регистрации и решает получить сертификат, то соответствующий код выполняется на стороне сервера. Генерируется закрытый ключ. Затем открытый ключ отсылается в запросе к УЦ. УЦ возвращает серверу ответ в виде сертификата в формате PEM, после чего создается контейнер PKCS#12. Так как для контейнера необходимо задавать пароль, то будет использован тот, который пользователь вводит при регистрации. В этот контейнер помещается закрытый ключ пользователя и его сертификат и далее возвращается пользователю в виде ответа.

Следует отметить, что в этом процессе закрытый ключ создается на стороне сервера и после пересылается клиенту в виде PKCS#12, что является более безопасным способом, нежели реализовывать это на стороне клиента в коде HTML, так как приватный ключ таким образом можно скомпрометировать.

В целом, действия со стороны клиента включают в себя несколько ключевых шагов:

1. Скачивание корневого сертификата;
2. Регистрация на сайте;
3. Установка клиентского сертификата;
4. Аутентификация.

Рассмотрим более подробно данные шаги:

Пользователь скачивает корневой сертификат УЦ на странице сервера регистрации, который нужен для установления безопасного взаимодействия с серверами.

Далее пользователь регистрируется на сайте. При этом процессе генерируется ключевая пара, отправляется CSR-запрос в УЦ и после возвращается сертификат в виде файла контейнера PKCS#12;

После пользователь устанавливает полученный клиентский сертификат в хранилище сертификатов;

Последним шагом будет вход и аутентификация на основной сервере.

Эти действия формируют процесс работы клиента с Web-приложением и обеспечивают высокий уровень безопасности соединения и аутентификации пользователя. Листинг реализации клиентской части представлен в листинге 3.1 и листинге 3.2 и листинге 3.3.

Листинг 3.1 – HTML код страницы входа

<!DOCTYPE HTML>

<HTML>

<head>

<meta charset="UTF-8">

<title>Invest Company</title>

<link rel="icon" href="{{ url\_for('static', filename='favicon.ico') }}" type="image/x-icon">

</head>

…

<body>

<div class="container">

<h1>Добро пожаловать на Invest Company</h1>

<div class="button-container">

<a href="/login" class="button1">Войти</a>

<a href="/register" class="button2">Зарегистрироваться</a>

</div>

<form method="POST" action="/download\_CA\_cert">

<button type="submit" class="download" value=>Скачать<br>корневой сертификат</button>

</form>

</div>

</body>

</HTML>

Листинг 3.2 – HTML код страницы регистрации

<!DOCTYPE HTML>

<HTML>

<head>

<meta charset="UTF-8">

<title>Регистрация</title>

<link rel="icon" href="{{ url\_for('static', filename='favicon.ico') }}" type="image/x-icon">

</head>

…

<body>

<div class="container">

<h1>Регистрация</h1>

<form action="/register" method="POST">

<input type="text" name="name" id="nameInput" placeholder="Имя" required>

<input type="email" name="email" id="emailInput" placeholder="Email" required>

<input type="submit" value="Скачать сертификат">

<a href="/register\_end" class="register-button" id="registerLink">Зарегистрироваться</a>

</form>

<script>

document.getElementById("registerLink").addEventListener("click", function(event) {

var nameInput = document.getElementById("nameInput");

var emailInput = document.getElementById("emailInput");

if (nameInput.value.trim() === "" || emailInput.value.trim() === "")

{

event.preventDefault(); // Предотвращаем переход по ссылке

alert("Заполните все поля ввода");

}

});

</script>

<div class="message">

{% if message %}

{{ message }}

{% endif %}

</div>

<hr>

<p>Уже зарегистрированы? <a href="/login" class="login-button">Войти</a></p>

</div>

</body>

</HTML>

Листинг 3.3 - Страница восстановления (перевыпуска) сертификата

<!DOCTYPE HTML>  
<HTML lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Восстановление</title>  
 <link rel="icon" href="{{ url\_for('static', filename='favicon.ico') }}" type="image/x-icon">  
…  
</head>  
<body>  
 <div class="container">  
 <h1>Восстановление сертификата</h1>  
 <form action="/recovery" method="POST">  
 <input type="email" name="email" id="emailInput" placeholder="Email" required>  
 <input type="password" name="password" id="PasswordInput" placeholder="Пароль" required>  
 <input type="submit" value="Скачать сертификат">  
 <a href="/register\_end" class="register-button" id="registerLink">Вернуться</a>  
 </form>  
 </div>  
</body>  
</HTML>

### [**Выводы по третьей главе**](#_Toc104675244)

В данной главе был разработан интерфейс и реализовано поэтапное добавление функционала для Web-приложения. Разработка была разделена на основной сервер и сервер регистрации.

Было реализовано взаимодействие между всеми участниками, что позволило серверу и клиенту обмениваться запросами на получение сертификата с УЦ посредством протокола HTTPS. УЦ успешно создавал, подписывал и отправлял обратно необходимые сертификаты, которые обеспечат взаимодействие в безопасном режиме после их установки

Клиентская часть была разработана с использованием HTML страниц, включая ссылки на функции, которые позволили пользователям взаимодействовать с интерфейсом Web-приложения.

В результате, успешно разработано Web-приложение с учетом необходимых мер безопасности и удобства пользователей. Дальнейший план включает интеграцию модуля аутентификации и его тестирование.

# [**Внедрение и тестирование модуля аутентификации в Web-приложение**](#_Toc104675238)

Завершающая глава посвящена внедрению и тестированию модуля аутентификации в разработанном Web-приложении. Будут рассмотрены основные этапы проверки функциональности модуля, а также возможные проблемы и пути их решения от входа на страницы регистрации до успешной аутентификации и получения состояния, что соединение безопасное.

### [**4.1 Внедрение и тестирование модуля**](#_Toc104675239) **аутентификации на работоспособность**

Теперь попробуем внедрить модуль аутентификации в наше Web-приложение. Этот модуль работает на основе настроек ssl\_context, который отвечает за установление и поддержание безопасного TLS соединения.

Строка кода context.verify\_mode = ssl.CERT\_REQUIRED, которая отвечает за то, что сервер при запросе на соединение требует от пользователя отправить клиентский сертификат. Этот момент может вызвать некоторое неудобство, поскольку при первом посещении сайта у пользователя еще не будет этого сертификата, и он не сможет его предоставить.

Поэтому было принято решение создать отдельный модуль - сервер регистрации. При первом посещении сайта пользователь перенаправляется на сервер регистрации, где он может зарегистрироваться и скачать необходимые сертификаты. После их установки пользователь может перейти на домашнюю страницу Web-приложения.

При разработке возникает важный момент в вопросе безопасности. При первом заходе на сайт соединение не будет безопасным (рисунок 4.1), так как пользователь еще не установил корневой сертификат нашего УЦ. Только после его установки соединение становится безопасным. Это связано с тем, что УЦ не является доверенным, поэтому его сертификаты не установлены по умолчанию в хранилище сертификатов.

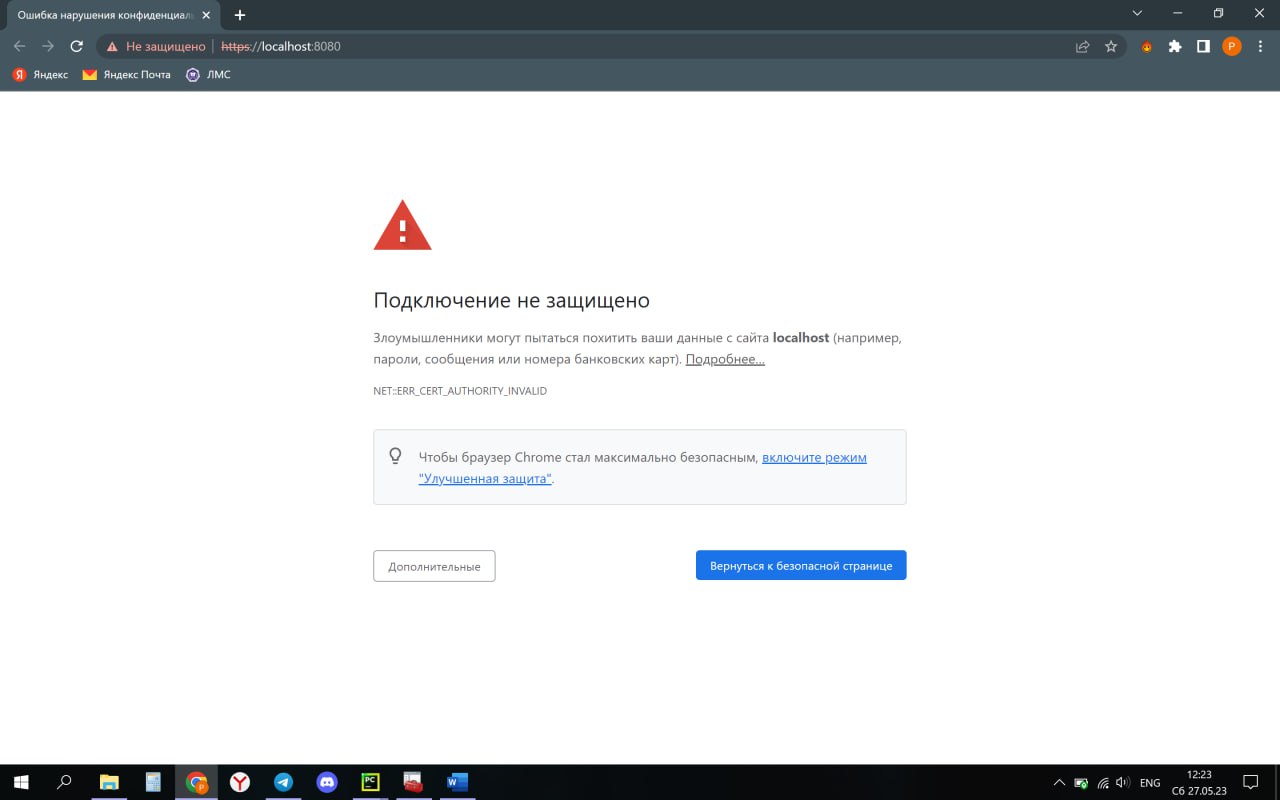


Рисунок 4.1 - первый вход клиента на сайт

Клиенту отсылается его сертификат в формате PKCS#12, который включает в себя как сам сертификат, так и закрытый ключ. Этот формат необходим для успешного взаимодействия с браузером. Реализация представлена на рисунке 4.2 и рисунке 4.3.

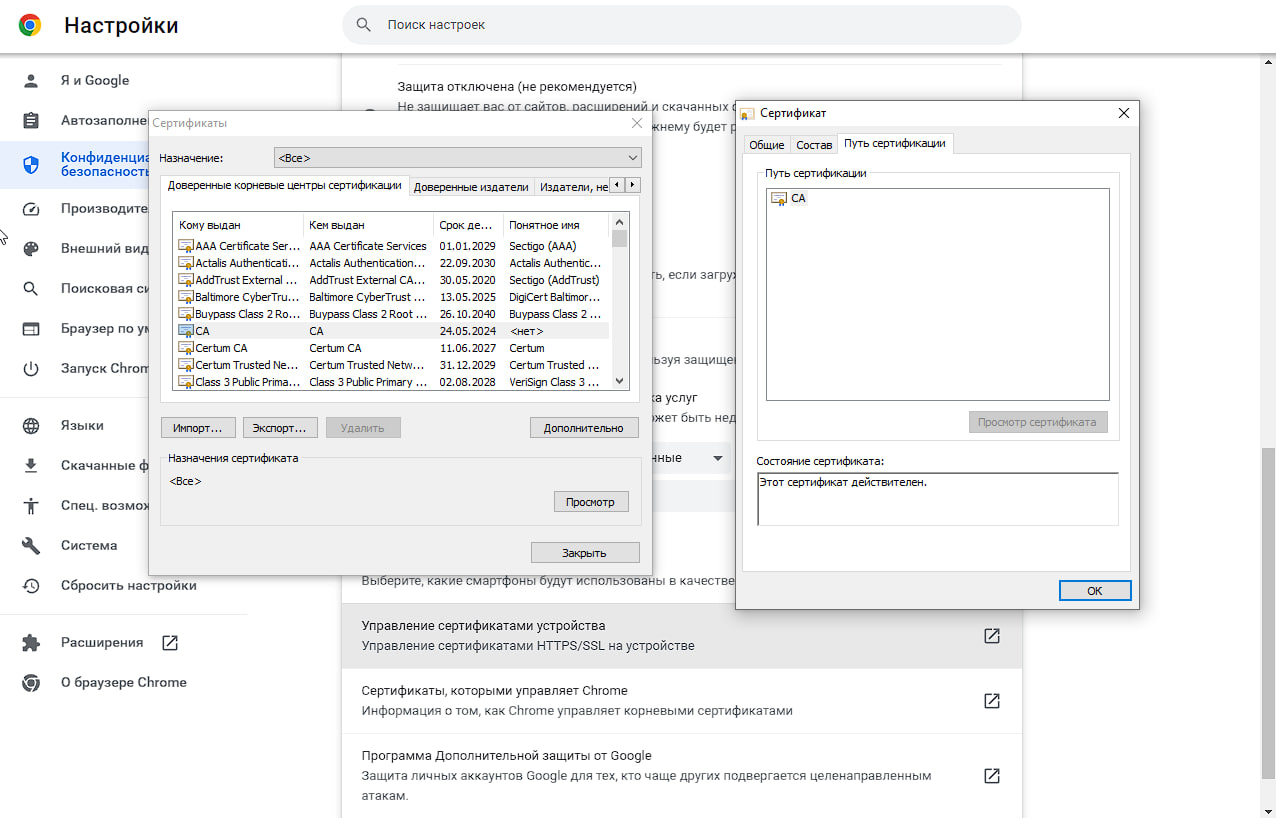


Рисунок 4.2 - установка корневого сертификата

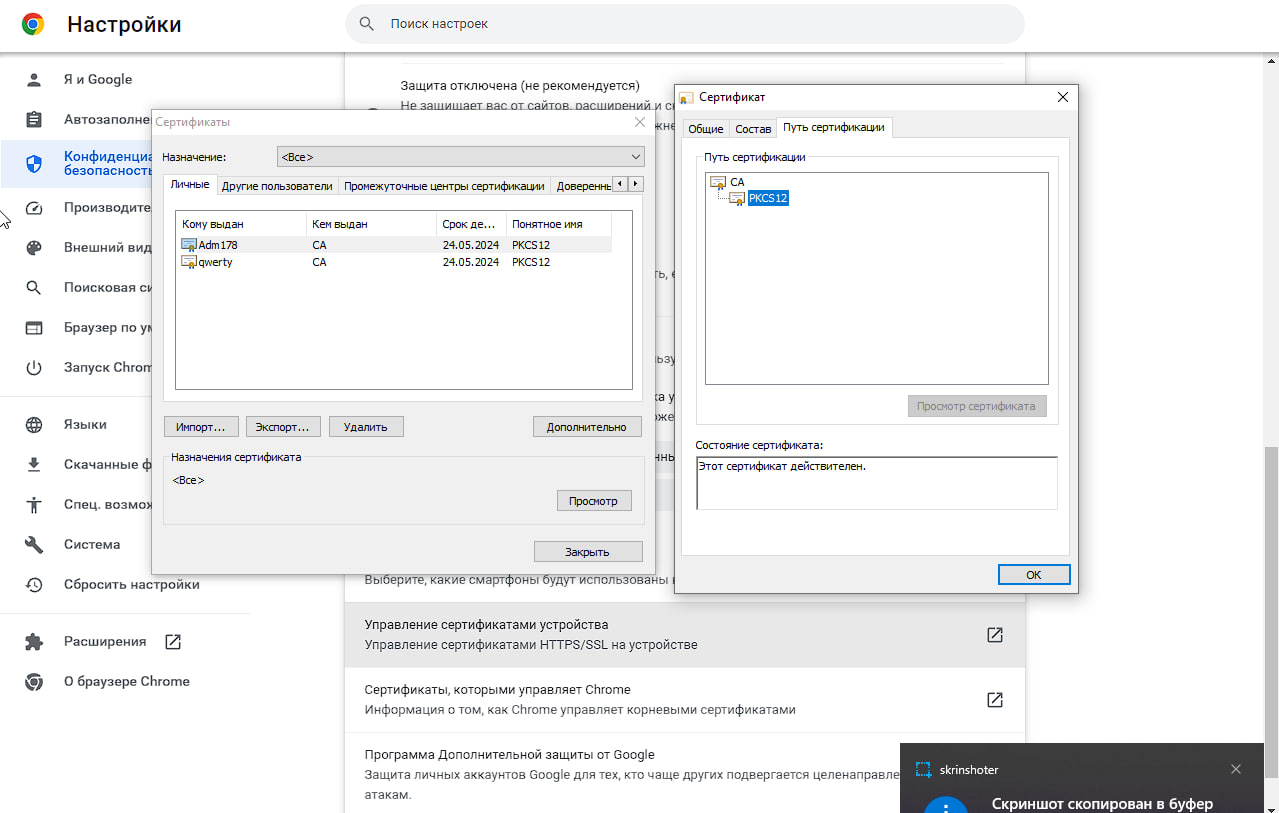


Рисунок 4.3 - установка клиентского сертификата

Как только клиент установит сертификат, он может попробовать процесс аутентификации, нажав кнопку входа. Появляется диалоговое окно от Windows 10 (рисунок 4.4) или Google Chrome (рисунок 4.5) (появление того или иного окна зависело от того был ли активирован сторонний антивирус), в котором клиент выбирает ранее полученный и установленный сертификат.

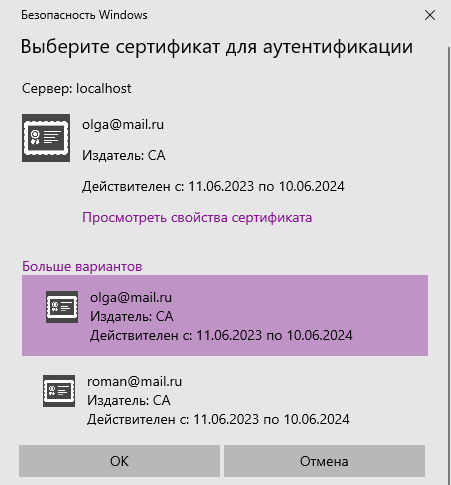


Рисунок 4.4 - диалоговое окно с выбором сертификата от Windows 10

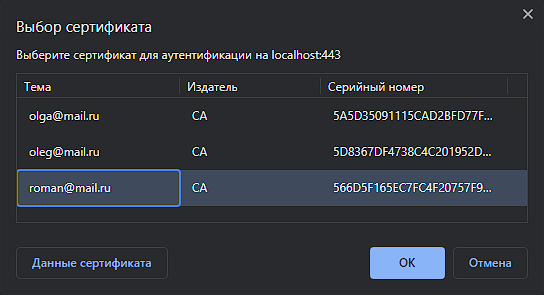


Рисунок 4.5 - Диалоговое окно с выбором сертификатов от Google Chrome (при выключенном антивирусе)

При успешной проверке сертификата на сервере клиент аутентифицируется и получает доступ к домашней странице сайта (рисунок 4.6). Выбранный сертификат будет использоваться для последующих соединений до полного закрытия браузера.

Таким образом, был успешно внедрён модуль аутентификации в Web-приложение, обеспечив простой и безопасный процесс входа для пользователей.

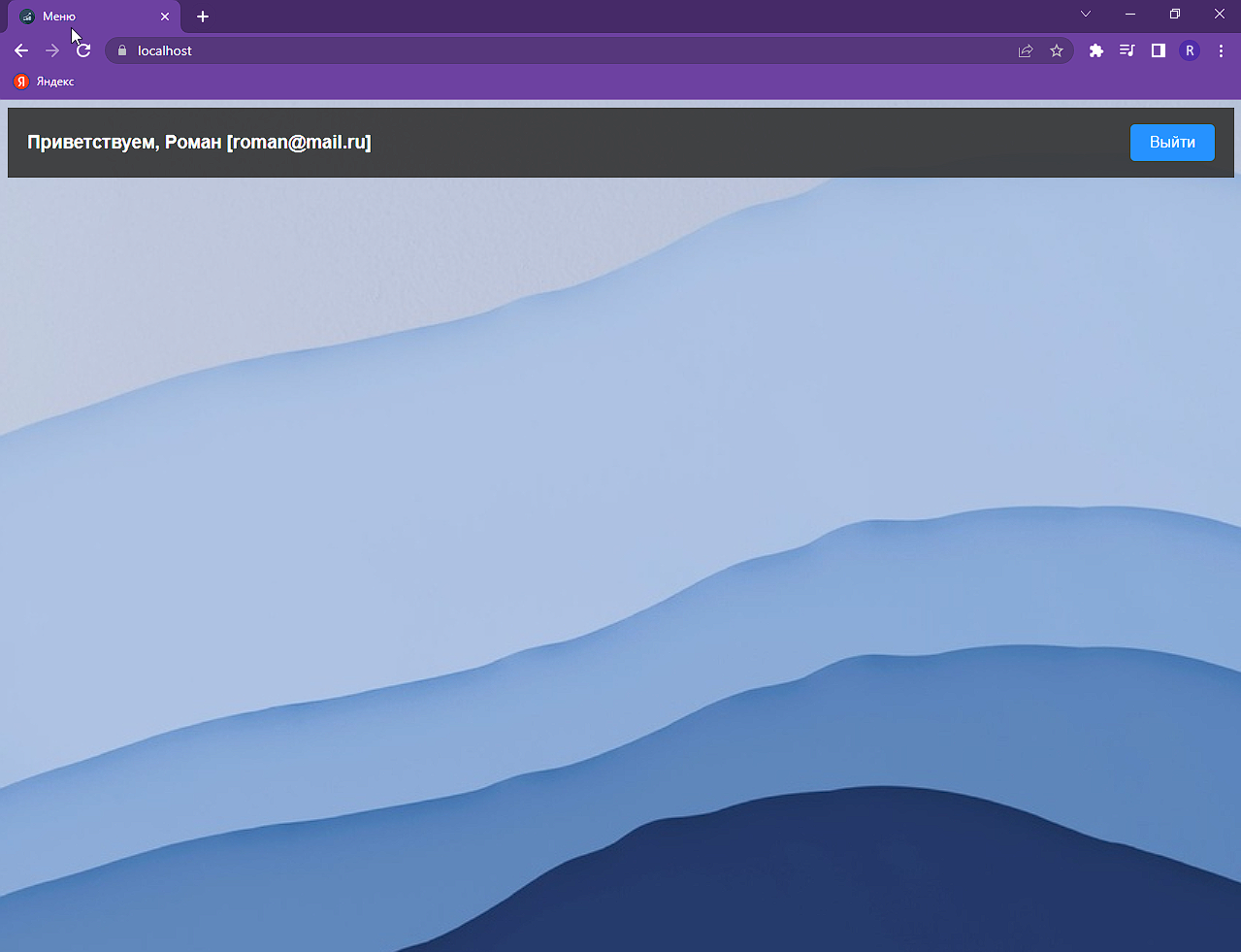


Рисунок 4.6 - Домашняя страница

Также было реализовано предупреждение, которое появляется при заходе на домашнюю страницу основного сервера. Если сертификат истекает через 14 дней или меньше, то диалоговое окно порекомендует перевыпустить сертификат. Данная реализация представлена на рисунке 4.7.

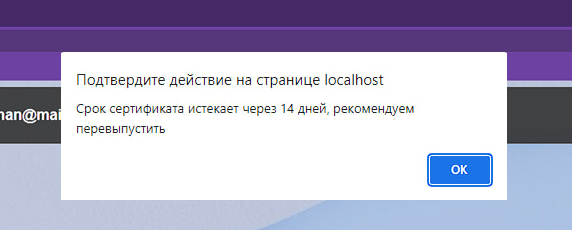


Рисунок 4.7 – Предупреждение о том, что у сертификата заканчивается   
срок действия

### [**4.2 Оценка использования модуля в практических условиях**](#_Toc104675239)

После успешной установки корневого сертификата, при установке соединения с сервером браузер будет показывать, что соединение является безопасным, что показано на рисунке 4.8, рисунке 4.9 и рисунке 4.10.

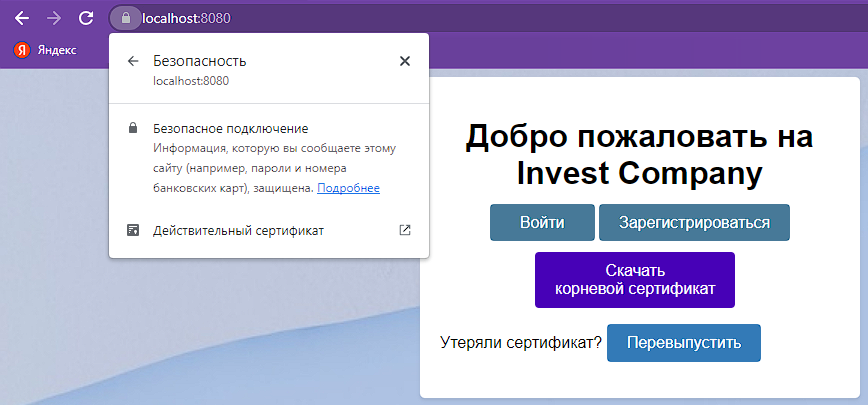


Рисунок 4.8 - Безопасное соединение на странице входа и регистрации

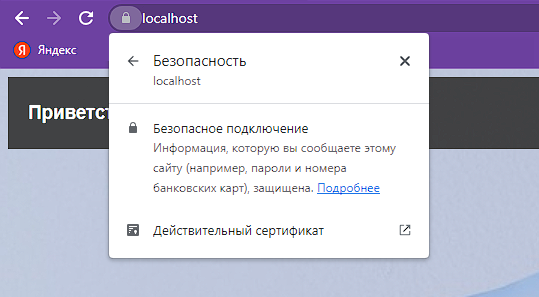


Рисунок 4.9 – безопасное соединение на домашней странице

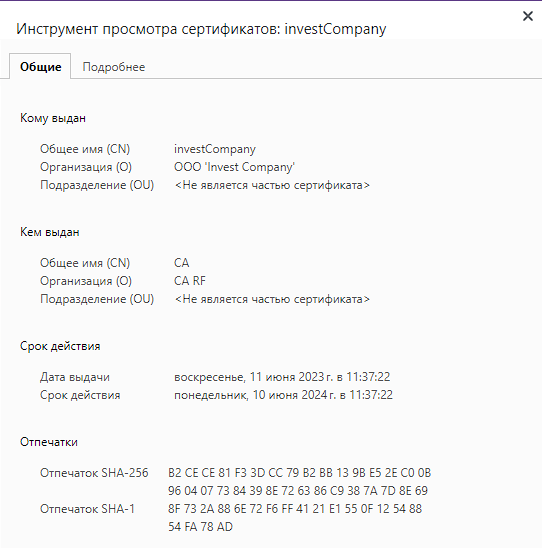


Рисунок 4.10 - Сертификат сервера

После успешной аутентификации от пользователя больше ничего не потребуется, и он сможет посещать страницы сервера. Повторная аутентификация также потребует выбрать только уже созданный сертификат, что заметно облегчит вход пользователей в систему и сделает это более безопасным.

### [**Выводы по четвертой главе**](#_Toc104675244)

В последней главе был успешно внедрён и протестирован модуль аутентификации в разработанном Web-приложении. Модуль работает эффективно, обеспечивая усиленную безопасность на основе TLS соединения.

Однако необходимо обратить внимание, что при первом посещении страницы пользователем соединение не будет считаться безопасным до тех пор, пока он не установит корневой сертификат. Это необходимое условие для того, что пользователь мог войти на сайт до аутентификации и зарегистрироваться.

После установки сертификатов соединение переходит в статус безопасного, а пользователь может произвести аутентификацию, выбрав сертификат в появившемся диалоговом окне. Этот процесс перенаправляет пользователя на домашнюю страницу, после чего сессия будет продолжаться до тех пор, пока пользователь не закроет соединение с сайтом.

Следовательно, модуль аутентификации был успешно разработан и внедрён в Web-приложение. Однако, важно обратить внимание пользователей на необходимость установки корневого и клиентского сертификата для обеспечения безопасности их соединения.

# [**зАКЛЮЧЕНИЕ**](#_Toc104675245)

В данной работе был изучен, разработан и реализован модуль аутентификации для Web-приложения с использованием сертификатов X.509. Используя библиотеку Flask для разработки Web-приложений, была разработана серверная и клиентская часть, а также настроены параметры TLS для обеспечения безопасного соединения.

Процесс создания модуля аутентификации предполагал последовательность действий от генерации ключевой пары на стороне сервера и клиента до аутентификации на сервере с использованием сертификата. В процессе был создан удостоверяющий центр, отвечающий выпуск, подпись и проверку сертификатов.

Одной из проблем во время разработки, стало немедленный запрос на представление сертификата при подключении к TLS для клиента, что не позволяло новым пользователям зарегистрироваться. В качестве решения этой проблемы был разработан отдельный сервер для регистрации, что позволило обойти проблему и предложить пользователям скачать корневой и клиентский сертификат после регистрации.

Также от клиентов потребовалось установить оба сертификата в хранилище. После этого при посещении сайта соединение считалось безопасным, а диалоговое окно с выбором сертификата позволяло пользователям аутентифицироваться и входить на сайт.

Таким образом, был успешно разработан модуль аутентификации, который позволяет обеспечить безопасную аутентификацию пользователей с использованием сертификатов X.509. Этот подход может быть масштабирован и использован для больших приложений и систем, требующих надежной аутентификации.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Основы HTTPS, TLS, SSL. Создание собственных X.509 сертификатов. Пример настройки TLSv1.2 в Spring Boot // Хабр // URL: <https://habr.com/ru/articles/593507/> (дата обращения: 02.05.2023)
2. Что такое TLS // Хабр // URL: <https://habr.com/ru/articles/258285/> (дата обращения: 02.05.2023)
3. Разбираем x.509 сертификат // Хабр // URL: <https://habr.com/ru/articles/194664/> (дата обращения: 02.05.2023)
4. «Как это работает»: знакомство с SSL/TLS // Хабр // URL: <https://habr.com/ru/companies/1cloud/articles/326292/> (дата обращения: 03.05.2023)
5. В чем разница между SSL и TLS? // PowerMark // URL: <https://powerdmarc.com/ru/difference-between-ssl-and-tls/> (дата обращения: 03.05.2023)
6. X.509 Authentication Service // GeeksforGeeks // URL: <https://www.geeksforgeeks.org/x-509-authentication-service/> (дата обращения: 04.05.2023)
7. Руководство по выживанию — TLS/SSL и сертификаты SSL (X.509) // OpenNET // URL: [https://www.opennet.ru/docs/RUS/ldap\_apacheds/tech/ssl.HTML](https://www.opennet.ru/docs/RUS/ldap_apacheds/tech/ssl.html) (дата обращения: 04.05.2023)
8. Mutual authentication // Википедия // URL: <https://en.wikipedia.org/wiki/Mutual_authentication> (дата обращения: 05.05.2023)
9. Простая настройка взаимной проверки подлинности клиента и сервера с использованием TLS // Хабр // URL: <https://habr.com/ru/companies/ruvds/articles/539090/> (дата обращения: 05.05.2023)
10. Взаимная аутентификация - Mutual authentication // Zahn info portal // URL: <https://ru.zahn-info-portal.de/wiki/Mutual_authentication> (дата обращения: 06.05.2023)
11. Зачем нужен собственный удостоверяющий центр // Хабр // URL: <https://habr.com/ru/companies/globalsign/articles/462501/> (дата обращения: 07.05.2023)
12. Разворачиваем собственный PKI // EasyCoding Team // URL: [https://www.easycoding.org/2020/07/05/razvorachivaem-sobstvennyj-pki.HTML](https://www.easycoding.org/2020/07/05/razvorachivaem-sobstvennyj-pki.html) (дата обращения: 07.05.2023)
13. Создание Web-приложения с помощью Flask в Python 3 // DigitalOcean // URL: [https://www.digitalocean.com/community/tutorials/how-to-make-a-Web-application-using-flask-in-python-3-ru](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-ru) (дата обращения: 10.05.2023)
14. Мега-Учебник Flask, Часть 1: «Привет, Мир!» // Хабр // URL: <https://habr.com/ru/articles/193242/> (дата обращения: 10.05.2023)
15. Подходы к разработке приложений на Flask Python фреймворке // Python Flask Tutorial // URL: <https://flask.ivan-shamaev.ru/approaches-to-development-flask-app-in-python/> (дата обращения: 12.05.2023)
16. Running Your Flask Application Over HTTPS // miguelgrinberg.com // URL: <https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https> (дата обращения: 15.05.2023)
17. flask-ssl-authentication // GitHub // URL: <https://github.com/ContinuumIO/flask-ssl-authentication> (дата обращения: 16.05.2023)
18. Шифрование и криптография в Python // Python 3 // URL: <https://python-scripts.com/encryption-cryptography> (дата обращения: 17.05.2023)
19. Х.509 // cryptography.io // URL: <https://cryptography.io/en/latest/x509/tutorial/> (дата обращения: 17.05.2023)

# [**Приложение а. КОД ПРОГРАММной реализации модуля аутентификации Web-приложения с применением сертификатов X.509**](#_Toc104675247)

Листинг А1.1 – server/main.py (сервер дом. страницы)

from flask import Flask, render\_template, redirect, request, flash  
from flask\_session import Session  
import os  
from func import server\_cert, parse\_client\_certificate  
import urllib3  
import ssl  
import requests  
  
app = Flask(\_\_name\_\_)  
  
urllib3.disable\_warnings(urllib3.exceptions.InsecureRequestWarning) # Отключение предупреждений о безопасности SSL  
ac\_url = 'https://localhost:5000' # URL УЦ  
register\_url = 'https://localhost:8080' # URL УЦ  
app.secret\_key = os.urandom(24) # Генерация секретного ключа  
app.config['SESSION\_TYPE'] = 'filesystem'  
app.config['SESSION\_COOKIE\_SECURE'] = True  
app.config['SESSION\_PERMANENT'] = True  
Session(app)  
  
@app.route("/") # Домашняя страница  
def home():  
 condition = False  
 client\_cert = request.environ.get('SSL\_CLIENT\_CERT')  
 # Парсим информацию из клиентского сертификата  
 cert\_info = parse\_client\_certificate(client\_cert)  
 date = cert\_info['date']  
 if date < 14:  
 condition = True  
  
 email = cert\_info['email']  
 response = requests.get(f'{register\_url}/data', verify=False, params=cert\_info)  
 if response.status\_code == 200:  
 user\_dict = {'name': response.content.decode('utf-8') + " [" + email + "]"}  
 return render\_template("home.HTML", user=user\_dict, condition=condition, variable=date), 200  
 else:  
 return render\_template("home.HTML", user={'name': "пользователь"}, condition=condition, variable=date), 200  
  
  
@app.route("/logout") # выход  
def logout():  
 return redirect(register\_url)  
  
server\_cert(ac\_url)  
if \_\_name\_\_ == '\_\_main\_\_': # Команды, которые будут исполняться при каждом запуске сервера  
 context = ssl.SSLContext(ssl.PROTOCOL\_TLS) # Настраиваем TLS  
 context.load\_cert\_chain('investCompany\_cert.pem', 'investCompany\_key.pem')  
 context.load\_verify\_locations('CA\_cert.pem')  
 context.options |= ssl.OP\_SINGLE\_ECDH\_USE  
 context.options |= ssl.OP\_NO\_TLSv1 | ssl.OP\_NO\_TLSv1\_1  
 context.verify\_mode = ssl.CERT\_REQUIRED  
 app.run(host='localhost', threaded=True, debug=True, ssl\_context=context, port=443) # Запускаем с учетом настроек

Листинг А1.2 – server/func.py (функции сервера дом. страницы)

from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import hashes  
from cryptography import x509  
from cryptography.x509.oid import NameOID  
from cryptography.hazmat.primitives.asymmetric import rsa  
from cryptography.hazmat.primitives import serialization  
import requests  
from datetime import datetime  
  
def generate\_key\_pair():  
 # Генерация приватного ключа  
 private\_key = rsa.generate\_private\_key(  
 public\_exponent=65537,  
 key\_size=2048,  
 backend=default\_backend()  
 )  
 return private\_key  
  
def CSR(username, email, private\_key):  
 csr\_builder = x509.CertificateSigningRequestBuilder()  
 csr\_builder = csr\_builder.subject\_name(x509.Name([  
 x509.NameAttribute(NameOID.COMMON\_NAME, username),  
 x509.NameAttribute(NameOID.EMAIL\_ADDRESS, email),  
 x509.NameAttribute(NameOID.COUNTRY\_NAME, u"RU"),  
 x509.NameAttribute(NameOID.STATE\_OR\_PROVINCE\_NAME, u"RF, Moscow"),  
 x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"OOO 'Invest Company'")  
 ]))  
  
 csr = csr\_builder.sign(private\_key, hashes.SHA256(), default\_backend())  
 csr\_pem = csr.public\_bytes(serialization.Encoding.PEM) # Сериализация CSR в формат PEM  
  
 return csr\_pem  
  
def server\_cert(uc\_url):  
 private\_key = generate\_key\_pair() # Генерация ключевой пары  
 key\_pem(private\_key)  
 csr = CSR('investCompany', 'investCompany@ic.ru', private\_key)  
 s\_responce = requests.post(f'{uc\_url}/issue\_certificate', verify=False,  
 headers={'Content-Type': 'application/x-pem-file'}, data=csr)  
 CA\_cent = requests.post(f'{uc\_url}/CA\_certificate', verify=False).content  
 if s\_responce.status\_code == 200:  
 with open(f'investCompany\_cert.pem', 'wb') as file:  
 file.write(s\_responce.content)  
 with open(f'CA\_cert.pem', 'wb') as file:  
 file.write(CA\_cent)  
  
def key\_pem(private\_key):  
 private\_key\_pem = private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.PKCS8,  
 encryption\_algorithm=serialization.NoEncryption()  
 )  
 with open(f'investCompany\_key.pem', "wb") as f:  
 f.write(private\_key\_pem)  
  
  
def parse\_client\_certificate(client\_cert):  
 cert\_data = client\_cert.encode('utf-8') # Предполагается, что клиентский сертификат представлен в виде строки  
 cert = x509.load\_pem\_x509\_certificate(cert\_data, default\_backend())  
  
 expiry\_date = cert.not\_valid\_after  
 expiry\_datetime = datetime.combine(expiry\_date.date(), datetime.min.time())  
  
 current\_datetime = datetime.now()  
 remaining\_days = (expiry\_datetime - current\_datetime).days  
  
 return {'email':cert.subject.get\_attributes\_for\_oid(x509.NameOID.EMAIL\_ADDRESS)[0].value,  
 'date': remaining\_days }

Листинг А2.1 – register/main.py (сервер регистрации)

from flask import Flask, request, render\_template, url\_for  
from flask import redirect, Response, send\_file, flash  
import requests  
import os  
from database import add\_user, get\_user  
from func import generate\_key\_pair, generate\_p12, register\_cert, CSR, hash\_string  
from cryptography import x509  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import serialization  
import tempfile  
import ssl  
import urllib3  
  
app = Flask(\_\_name\_\_)  
  
urllib3.disable\_warnings(urllib3.exceptions.InsecureRequestWarning) # Отключение предупреждений о безопасности SSL  
ac\_url = 'https://localhost:5000' # URL УЦ  
main\_url = 'https://localhost:5000' # URL УЦ  
app.secret\_key = os.urandom(24) # Генерация секретного ключа  
  
@app.route('/', methods=['GET'])  
def index():  
 return render\_template("index.HTML"), 200  
  
@app.route('/register', methods=['GET','POST']) # Регистрация  
def register():  
 if request.method == 'GET':  
 return render\_template("register.HTML"), 200  
 if request.method == 'POST':  
 email = str(request.form['email'])  
 if not get\_user(email, None):  
 username = str(request.form['name'])  
 password = str(request.form['password']) # Хэшируем полученный пароль  
 private\_key = generate\_key\_pair() # Генерация ключевой пары  
 csr = CSR(email, private\_key) # Создание запроса на выпуск сертификата  
 response = requests.post(f'{ac\_url}/issue\_certificate', verify=False,  
 headers={'Content-Type': 'application/x-pem-file'}, data=csr) # Отправка CSR запроса УЦ  
 if response.status\_code == 200:  
 add\_user(email, username, hash\_string(password), 3)  
 p12 = generate\_p12(response.content, private\_key, email, password)  
 certificate = Response(p12, mimetype='application/x-pkcs12')  
 certificate.headers.set('Content-Disposition', 'attachment', filename=f'{email}\_cert.p12')  
 return certificate, 200  
 else:  
 flash('Ошибка получения сертификата со стороны УЦ', 'error')  
 return redirect(url\_for("register"))  
 else:  
 flash('Данный email уже зарегистрирован', 'error')  
 return redirect(url\_for("register"))  
  
@app.route('/register\_end', methods=['GET'])  
def register\_end():  
 return redirect(url\_for("index"))  
  
@app.route('/download\_CA\_cert', methods=['POST'])  
def download\_ca\_cert():  
 CA\_cert = requests.post(f'{ac\_url}/CA\_certificate', verify=False)  
  
 if CA\_cert.status\_code == 200:  
 cert = x509.load\_pem\_x509\_certificate(CA\_cert.content, default\_backend())  
 crt\_data = cert.public\_bytes(encoding=serialization.Encoding.DER)  
  
 with tempfile.NamedTemporaryFile(delete=False) as temp\_file:  
 temp\_file.write(crt\_data)  
  
 return send\_file(temp\_file.name, mimetype='application/x-x509-ca-cert',  
 as\_attachment=True, download\_name='CA\_cert.crt')  
 else:  
 return 'Ошибка получения корневого сертификата', 401  
  
@app.route('/recovery', methods=['GET','POST']) # Вход  
def recovery():  
 if request.method == 'GET':  
 return render\_template("recovery.HTML"), 200  
 if request.method == 'POST':  
 email = str(request.form['email'])  
 password = str(request.form['password'])  
 username = get\_user(email, hash\_string(password))  
 if username:  
 private\_key = generate\_key\_pair() # Генерация ключевой пары  
 csr = CSR(email, private\_key) # Создание запроса на выпуск сертификата  
 response = requests.post(f'{ac\_url}/issue\_certificate', verify=False,  
 headers={'Content-Type': 'application/x-pem-file'}, data=csr)  
  
 if response.status\_code == 200:  
 p12 = generate\_p12(response.content, private\_key, email, password)  
 certificate = Response(p12, mimetype='application/x-pkcs12')  
 certificate.headers.set('Content-Disposition', 'attachment', filename=f'{email}\_cert.p12')  
 return certificate, 200  
 else:  
 flash('Ошибка получения сертификата со стороны УЦ', 'error')  
 return redirect(url\_for("recovery"))  
 else:  
 flash('Пользователь не найден или неверный пароль', 'error')  
 return redirect(url\_for("recovery"))  
  
  
@app.route('/data', methods=['GET'])  
def data():  
 email = request.args.get('email')  
 username = get\_user(email, None)  
 if username:  
 return username  
 else:  
 return None  
  
@app.route('/login', methods=['GET']) # Вход  
def login():  
 return redirect("https://localhost:443/")  
  
register\_cert(ac\_url)  
if \_\_name\_\_ == '\_\_main\_\_':  
 context = ssl.SSLContext(ssl.PROTOCOL\_TLS)  
 context.load\_cert\_chain('register\_cert.pem', 'register\_key.pem')  
 context.load\_verify\_locations('CA\_cert.pem')  
 context.options |= ssl.OP\_SINGLE\_ECDH\_USE  
 context.options |= ssl.OP\_NO\_TLSv1 | ssl.OP\_NO\_TLSv1\_1  
 app.run(host='localhost', threaded=True, debug=True, ssl\_context=context, port=8080) # Запускаем с учетом настроек

Листинг А2.2 – register/func.py (функции сервера регистрации)

from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import hashes  
from cryptography import x509  
from cryptography.x509.oid import NameOID  
from cryptography.hazmat.primitives.asymmetric import rsa  
from cryptography.hazmat.primitives import serialization  
from cryptography.hazmat.primitives.serialization import BestAvailableEncryption, pkcs12  
from database import add\_user, create\_users\_table, get\_db  
import requests  
import sqlite3  
import hashlib  
  
def generate\_key\_pair():  
 # Генерация приватного ключа  
 private\_key = rsa.generate\_private\_key(  
 public\_exponent=65537,  
 key\_size=2048,  
 backend=default\_backend()  
 )  
 return private\_key  
  
def CSR(email, private\_key):  
 csr\_builder = x509.CertificateSigningRequestBuilder()  
 csr\_builder = csr\_builder.subject\_name(x509.Name([  
 x509.NameAttribute(NameOID.COMMON\_NAME, email),  
 x509.NameAttribute(NameOID.EMAIL\_ADDRESS, email),  
 x509.NameAttribute(NameOID.COUNTRY\_NAME, u"RU"),  
 x509.NameAttribute(NameOID.STATE\_OR\_PROVINCE\_NAME, u"RF, Moscow"),  
 x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"OOO 'Invest Company'")  
 ]))  
  
 csr = csr\_builder.sign(private\_key, hashes.SHA256(), default\_backend())  
 csr\_pem = csr.public\_bytes(serialization.Encoding.PEM) # Сериализация CSR в формат PEM  
  
 return csr\_pem  
  
def generate\_p12(certificate, private\_key, email, password):  
 # Создание файла PKCS#12  
 cert = x509.load\_pem\_x509\_certificate(certificate)  
 p12 = serialization.pkcs12.serialize\_key\_and\_certificates(  
 f"{email}".encode('utf-8'), private\_key, cert, None, BestAvailableEncryption(password.encode()))  
  
 return p12  
  
def key\_pem(private\_key):  
 private\_key\_pem = private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.PKCS8,  
 encryption\_algorithm=serialization.NoEncryption()  
 )  
 with open(f'register\_key.pem', "wb") as f:  
 f.write(private\_key\_pem)  
  
def register\_cert(uc\_url):  
 private\_key = generate\_key\_pair() # Генерация ключевой пары  
 key\_pem(private\_key)  
 csr = CSR('register@ic.ru', private\_key)  
 s\_responce = requests.post(f'{uc\_url}/issue\_certificate', verify=False,  
 headers={'Content-Type': 'application/x-pem-file'}, data=csr)  
 CA\_cent = requests.post(f'{uc\_url}/CA\_certificate', verify=False).content  
 if s\_responce.status\_code == 200:  
 with open(f'register\_cert.pem', 'wb') as file:  
 file.write(s\_responce.content)  
 with open(f'CA\_cert.pem', 'wb') as file:  
 file.write(CA\_cent)  
  
 try: # Чтобы БД не создавалась каждый раз, то просто поставим условия проверки  
 conn = get\_db() # Пытаемся получить соединение с базой данных  
 conn.execute('SELECT \* FROM users LIMIT 1') # Пытаемся создать таблицу users  
 except sqlite3.OperationalError:  
 create\_users\_table() # Если таблица не существует, создаем ее  
 add\_user('admin@ic.ru', 'admin', hash\_string("admin") , 1) # Добавляем администратора  
 finally:  
 if 'conn' in locals(): # Закрываем соединение с базой данных  
 conn.close()  
  
def hash\_string(string):  
 sha256\_hash = hashlib.sha256()  
 encoded\_string = string.encode('utf-8')  
 sha256\_hash.update(encoded\_string)  
 hashed\_string = sha256\_hash.hexdigest()  
 return hashed\_string

Листинг А2.3 – register/database.py (функции работы с БД сервера регистрации)

import sqlite3  
  
def get\_db(): # Подключение к базе данных SQLite через файл  
 conn = sqlite3.connect('database.db')  
 #conn.row\_factory = sqlite3.Row  
 return conn  
  
def create\_users\_table(): # Создание таблицы пользователей  
 with get\_db() as conn:  
 conn.execute('''  
 CREATE TABLE users (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 email TEXT UNIQUE NOT NULL,  
 username TEXT NOT NULL,  
 password TEXT NOT NULL,  
 access\_level INTEGER NOT NULL)  
 ''')  
  
def add\_user(email, username, password, access\_level): # Добавление нового пользователя в таблицу  
 with get\_db() as conn:  
 cursor = conn.cursor()  
 cursor.execute('''  
 INSERT INTO users (email, username, password, access\_level)  
 VALUES (?, ?, ?, ?)  
 ''', (email, username, password, access\_level))  
 conn.commit()  
  
def get\_user(email, password):  
 with get\_db() as conn:  
 cursor = conn.cursor()  
 if password is None:  
 cursor.execute("SELECT \* FROM users WHERE email=?", (email,))  
 else:  
 cursor.execute("SELECT \* FROM users WHERE email=? AND password=?", (email, password))  
 user = cursor.fetchone()  
 if user:  
 return user[2]  
 else:  
 return None

Листинг А3.1 – CA/main.py (удостоверяющий центр)

from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import serialization  
from flask import Flask, request  
from X509 import generate\_certificate, CA  
from cryptography import x509  
import ssl  
  
app = Flask(\_\_name\_\_)  
  
@app.route('/recovery\_certificate', methods=['POST'])  
def recovery():  
 email = request.data.decode('utf-8')  
 try:  
 with open(f'cert/{email}\_cert.pem', "rb") as cert\_file:  
 cert\_data = cert\_file.read()  
 certificate = x509.load\_pem\_x509\_certificate(cert\_data, default\_backend())  
 cert\_pem = certificate.public\_bytes(encoding=serialization.Encoding.PEM)  
 return cert\_pem  
 except FileNotFoundError:  
 return None  
  
@app.route('/issue\_certificate', methods=['POST'])  
def issue\_certificate():  
 # Получение CSR-запроса от клиента  
 csr\_data = request.data  
 csr\_request = x509.load\_pem\_x509\_csr(csr\_data, default\_backend())  
  
 cert\_pem = generate\_certificate(csr\_request) # Генерация сертификата  
 return cert\_pem # Возвращаем подписанный сертификат клиенту  
  
@app.route('/CA\_certificate', methods=['POST'])  
def CA\_certificate():  
 return CA\_cert  
  
CA\_cert = CA() # Генерация корневого сертификата  
if \_\_name\_\_ == '\_\_main\_\_':  
 context = ssl.SSLContext(ssl.PROTOCOL\_TLS)  
 context.load\_cert\_chain('CA\_cert.pem', 'CA\_key.pem')  
 app.run(host='localhost', debug=True, ssl\_context=context, port=5000)

Листинг А3.2 – CA/X509.py (функции по выпуску сертификатов УЦ)

from cryptography import x509  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import serialization  
from cryptography.x509.oid import NameOID  
from cryptography.hazmat.primitives.asymmetric import rsa  
from cryptography.hazmat.primitives import hashes  
import datetime  
  
# Генерация клиентского сертификата  
def generate\_certificate(csr):  
 certificate = x509.CertificateBuilder()\  
 .subject\_name(csr.subject)\  
 .issuer\_name(issuer)\  
 .public\_key(csr.public\_key())\  
 .serial\_number(x509.random\_serial\_number())\  
 .not\_valid\_before(datetime.datetime.utcnow())\  
 .not\_valid\_after(datetime.datetime.utcnow() + datetime.timedelta(days=365))\  
 .add\_extension(x509.BasicConstraints(ca=False, path\_length=None), critical=True,) \  
 .add\_extension(x509.AuthorityKeyIdentifier.from\_issuer\_public\_key(CA\_private\_key.public\_key()), critical=False)\  
 .add\_extension(x509.SubjectKeyIdentifier.from\_public\_key(csr.public\_key()), critical=False) \  
 .add\_extension(x509.SubjectAlternativeName([x509.DNSName('localhost')]), critical=False) \  
 .sign(CA\_private\_key, hashes.SHA256(), default\_backend())  
  
 # Поиск расширения с информацией о электронной почте  
 email = csr.subject.get\_attributes\_for\_oid(x509.NameOID.EMAIL\_ADDRESS)[0].value  
  
 # Сохранение сертификата в файл в формате PEM  
 cert\_pem = certificate.public\_bytes(serialization.Encoding.PEM)  
 with open(f'cert/{email}\_cert.pem', "wb") as file:  
 file.write(cert\_pem)  
  
 return cert\_pem  
  
# Генерация корневого сертификата  
def CA():  
 global CA\_private\_key  
 CA\_private\_key = rsa.generate\_private\_key(  
 public\_exponent=65537,  
 key\_size=2048,  
 backend=default\_backend())  
 public\_key = CA\_private\_key.public\_key()  
  
 global issuer  
 subject = issuer = x509.Name([  
 x509.NameAttribute(NameOID.COMMON\_NAME, "CA"),  
 x509.NameAttribute(NameOID.COUNTRY\_NAME, u"RU"),  
 x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"CA RF")])  
  
 cert = x509.CertificateBuilder().\  
 subject\_name(subject).\  
 issuer\_name(issuer).\  
 public\_key(public\_key).\  
 serial\_number(x509.random\_serial\_number()).\  
 not\_valid\_before(datetime.datetime.utcnow()).\  
 not\_valid\_after(datetime.datetime.utcnow() + datetime.timedelta(days=365)). \  
 add\_extension(x509.BasicConstraints(ca=True, path\_length=None), critical=True). \  
 add\_extension(x509.SubjectAlternativeName([x509.DNSName('localhost')]), critical=False). \  
 sign(CA\_private\_key, hashes.SHA256(), default\_backend())  
  
 CA\_cert = cert.public\_bytes(serialization.Encoding.PEM)  
 with open(f'CA\_cert.pem', "wb") as file:  
 file.write(CA\_cert)  
 private\_key\_pem = CA\_private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.PKCS8,  
 encryption\_algorithm=serialization.NoEncryption()  
 )  
 with open("CA\_key.pem", "wb") as f:  
 f.write(private\_key\_pem)  
  
 return CA\_cert