**3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИС**

**3.1 Выбор программных средств реализации**

Разработка информационной системы требует тщательного подбора технологического стека, обеспечивающего высокую производительность, надежность и масштабируемость решения. Основной упор сделан на современные, проверенные технологии с открытым исходным кодом, обладающие широким сообществом разработчиков и хорошей документацией.

Для реализации серверной части информационной системы был выбран язык программирования Java. Этот выбор обусловлен рядом ключевых преимуществ, делающих Java оптимальным решением для разработки сложных enterprise-приложений. Java представляет собой зрелый, строго типизированный объектно-ориентированный язык с мощной экосистемой и проверенной временем архитектурой.

Одним из главных достоинств Java является ее кроссплатформенность. Благодаря виртуальной машине JVM (Java Virtual Machine), Java-приложения могут выполняться на любом оборудовании и операционной системе, где установлена соответствующая JVM. Это особенно важно для информационных систем, которые могут развертываться в различных инфраструктурных средах.

Безопасность Java обеспечивается несколькими механизмами. Строгая типизация предотвращает множество распространенных ошибок программирования. Управление памятью через сборщик мусора (Garbage Collector) минимизирует риски утечек памяти. Встроенные механизмы безопасности, такие как Security Manager и криптографические библиотеки, позволяют реализовывать надежные системы аутентификации и авторизации.

Для сравнения рассматривались альтернативные технологии. Python, несмотря на простоту синтаксиса, уступает в производительности из-за интерпретируемой природы. C# обладает схожими характеристиками, но сильнее привязан к экосистеме Microsoft. Go демонстрирует отличную производительность, но имеет менее развитую экосистему для enterprise-разработки.

Для разработки клиентской части информационной системы был выбран прогрессивный JavaScript-фреймворк Vue.js. Этот выбор обусловлен оптимальным сочетанием простоты интеграции, гибкости архитектуры и высокой производительности, что делает Vue.js идеальным решением для современных веб-приложений. Vue представляет собой реактивный фреймворк, специально разработанный для поэтапного внедрения и создания сложных пользовательских интерфейсов.

Vue.js отличается продуманной и интуитивно понятной архитектурой, основанной на компонентном подходе. Компоненты Vue инкапсулируют логику, шаблоны и стили, что способствует созданию поддерживаемого и переиспользуемого кода. Реактивная система данных автоматически отслеживает зависимости и эффективно обновляет DOM при изменениях состояния приложения, значительно упрощая разработку динамических интерфейсов.

Для сравнения рассматривались альтернативные фреймворки. React, хотя и обладает большим сообществом, требует дополнительных библиотек для полноценной разработки. Angular, несмотря на комплексность, имеет более высокий порог входа и менее гибкую архитектуру. Vue.js занял золотую середину, предлагая богатые возможности при относительно простом освоении.

Для хранения и обработки данных в информационной системе была выбрана реляционная СУБД PostgreSQL. Этот выбор обусловлен сочетанием надежности, производительности и расширенной функциональности, что делает PostgreSQL оптимальным решением для сложных информационных систем. PostgreSQL представляет собой объектно-реляционную систему управления базами данных с открытым исходным кодом, разрабатываемую более 30 лет.

Для сравнения рассматривались альтернативные СУБД. MySQL, хотя и проще в настройке, уступает в функциональности и надежности для сложных систем. MongoDB, как представитель NoSQL, не обеспечивает необходимого уровня целостности данных для транзакционных систем. Oracle Database обладает схожими возможностями, но является проприетарным и дорогостоящим решением.

Выбор PostgreSQL в качестве СУБД для информационной системы обеспечивает надежное хранение данных, высокую производительность запросов и гибкость при разработке. Расширенные функции, такие как оконные операции и поддержка JSON, позволяют эффективно решать сложные задачи обработки данных. Открытая лицензия и активное сообщество разработчиков гарантируют долгосрочную поддержку и развитие системы.

Выбранный технологический стек обеспечивает модульность архитектуры, что позволяет масштабировать систему и добавлять новый функционал. Сочетание Java и Vue.js создает оптимальный баланс между производительностью серверной части и отзывчивостью клиентского интерфейса. PostgreSQL гарантирует надежное хранение данных и возможность сложного анализа информации. Такой набор технологий полностью соответствует требованиям к современным информационным системам.

**3.2 Структура программного обеспечения**

**3.3 Разработка программного кода**

**3.3.1 Разработка серверной части кода**

Разработка back-end части для информационной системы является ключевым этапом в создании эффективного и надежного программного продукта. Серверная часть обеспечивает обработку данных, взаимодействие с базами данных и реализацию бизнес-логики, что позволяет системе выполнять свои функции на высоком уровне.

Первым шагом создания серверной части, является создание сущностей (моделей). Сущности представляют данные, которые сохраняются в базе данных. Каждая сущность соответствует таблице в реляционной базе данных, а поля сущности соответствуют столбцам таблицы. Сущности аннотируются с помощью @Entity, и для них обычно создаются классы с полями и методами для работы с данными. Они определяют, какие данные должны храниться, какие операции могут быть выполнены с этими данными, и как они соотносятся друг с другом. Они являются частью архитектурного паттерна MVC (Model-View-Controller), где отвечают за работу с информацией: валидацию, хранение, обработку и передачу между слоями приложения. Сущности абстрагируют взаимодействие с базой данных, предоставляя удобный интерфейс для операций.

В Java каждая модель является отдельным классом, в котором описаны поля, которые являются представлением полей в базе данных. Для более удобного создания моделей используется библиотека Lombok. Она позволяет избегать написания большого количества шаблонного кода, автоматически генерируя необходимые методы, такие как геттеры, сеттеры и конструкторы. Например, при создании модели Document, которая представляет собой таможенный документ, можно использовать аннотации @Data, @NoArgsConstructor и @AllArgsConstructor. Аннотация @Data автоматически генерирует методы для доступа к полям класса, а также методы equals(), hashCode() и toString(). Аннотации @NoArgsConstructor и @AllArgsConstructor создают конструкторы без параметров и с параметрами соответственно. Это позволяет легко инициализировать объекты модели.

Исходя из классов моделей, создаются классы DTO. DTO, или Data Transfer Object, представляет собой паттерн проектирования, используемый для передачи данных между сервисами и компонентами. Основная цель DTO заключается в том, чтобы упрощать и оптимизировать процесс передачи данных, минимизируя количество запросов и объем передаваемой информации. Оно позволяет передавать только те данные, которые необходимы, исключая лишнюю информацию. Вся получаемая или отправляемая информация в серверной части будет передаваться в виде DAO классов.

Для конвертации сущностей в DAO классы и обратно используются мапперы. Библиотека MapStruct генерирует реализацию мапперов на этапе компиляции, что обеспечивает высокую производительность и минимизирует вероятность ошибок, связанных с ручным написанием маппинга. Для этого необходимо только создать интерфейс и определить классы для конвертации.

Далее необходимо создать механизм взаимодействия с базой данных с помощью моделей. Для этого используется Java Persistence API (JPA). Это спецификация Java, предназначенная для управления персистентностью данных в Java-приложениях. JPA предоставляет стандартный способ работы с реляционными базами данных, позволяя разработчикам легко сохранять, извлекать, обновлять и удалять данные, представленные в виде объектов. С помощью неё возможно создание репозитория, который автоматически реализует все основные операции с базой данных. Для реализации более сложных запрос используется HQL. Он позволяет писать запросы, которые оперируют объектами и их свойствами, а не таблицами и столбцами реляционной базы данных. Например, вот этот запрос выдает список активных проверок ФЛК у которых совпадает тип документа:

@Query("SELECT f FROM FlkCheck f where f.documentType = :documentType and f.isActive = true")

List<FlkCheck> findByDocumentTypeAndActiveIsTrue(int documentType);

Следующим шагом разработки программного кода серверверной части является создание классов сервисов. Сервис является ключевым компонентом архитектуры приложений, представляя собой слой, который инкапсулирует бизнес-логику и предоставляет интерфейс для выполнения определённых операций. Основной характеристикой сервиса является его способность инкапсулировать бизнес-логику. Это позволяет организовать обработку данных, выполнение вычислений и управление взаимодействием с другими компонентами, такими как репозитории и внешние API. Кроме того, одним из значительных преимуществ использования сервисов является их тестируемость. Поскольку бизнес-логика изолирована от других компонентов, сервисы легко тестировать. Это позволяет писать юнит-тесты для проверки логики без необходимости взаимодействия с базой данных или пользовательским интерфейсом, что значительно ускоряет процесс разработки и повышает качество кода.

Класс DtDocumentService представляет собой сервисный компонент в архитектуре приложения, отвечающий за управление документами типа DT (документы таможенного оформления). Он инкапсулирует бизнес-логику, связанную с обработкой данных, и предоставляет методы для выполнения различных операций, таких как создание, получение и управление документами.

Метод saveDtDocument принимает на вход DTO (Data Transfer Object) типа DtTypeDTO и выполняет несколько шагов для сохранения документа. Он сначала маппит DTO в сущность DtDocumentType и проверяет наличие декларанта в базе данных. Если декларант отсутствует, он сохраняется в репозитории. Затем метод обрабатывает список товаров, проверяя их наличие по коду и сохраняет новые товары, если они не найдены. Если документ с указанным кодом не существует, метод устанавливает дату создания, статус и уникальный код документа, после чего сохраняет его в репозитории. В противном случае метод возвращает null.

Метод decideByDtDocument реализует логику принятия решений по документам. Он проверяет, имеет ли пользователь соответствующую роль инспектора, и если это так, ищет документ по коду. Если документ найден, обновляется его статус и специальные отметки, после чего документ сохраняется обратно в репозиторий. В случае ошибок метод логирует информацию о проблемах, что помогает в отладке и мониторинге.

Метод generatorPDF отвечает за генерацию PDF-документов для документов типа DT. Он ищет документ по коду и, если документ найден, вызывает сервис генерации PDF для создания файла. Если документ не найден, метод выбрасывает исключение, что позволяет избежать ошибок при попытке генерации PDF для несуществующего документа.

Класс UserService является сервисным компонентом, предназначенным для управления пользователями в приложении. Он реализует интерфейс UserDetailsService, что позволяет интегрировать его с механизмом аутентификации Spring Security. Сервис инкапсулирует логику, связанную с обработкой данных пользователей, включая их создание, поиск и аутентификацию.

Метод loadUserByUsername реализует метод интерфейса UserDetailsService, обеспечивая механизм аутентификации. Он принимает имя пользователя в качестве параметра и вызывает метод getUserByUsername для получения соответствующего пользователя. Если пользователь найден, метод возвращает объект UserDetails, который содержит имя пользователя, хешированный пароль и роли.

Метод saveUser отвечает за создание нового пользователя. Он принимает DTO типа UserDTO, маппит его в сущность User и проверяет наличие пользователя с таким же именем в базе данных. Если пользователь уже существует, метод выбрасывает исключение, предотвращая дублирование. Перед сохранением нового пользователя его пароль хешируется с использованием BCryptPasswordEncoder, что обеспечивает безопасность хранения паролей. После успешной проверки и обработки данных сервис сохраняет пользователя в базе данных.

Класс PdfGeneratorService является сервисным компонентом, предназначенным для генерации PDF-документов на основе шаблонов HTML. Он использует библиотеку Thymeleaf для рендеринга HTML и библиотеку IText для создания PDF-файлов. Этот класс обеспечивает возможность динамического создания документов.

Метод generatePdf принимает на вход объект типа DtDocumentType, который содержит информацию, необходимую для создания документа. В начале метода создается контекст Context, в который добавляются переменные, представляющие декларанта, список товаров, отправителя, получателя и сам документ. Этот контекст будет использоваться при рендеринге HTML-шаблона. После подготовки данных метод вызывает TemplateEngine, чтобы сгенерировать HTML-код на основе шаблона и контекста. После получения HTML-кода метод использует ITextRenderer для создания PDF-документа. Он создает ByteArrayOutputStream, который будет содержать финальный PDF-файл. Для корректного отображения русского текста в документе используется шрифт, который загружается из ресурсов приложения. Установка базового URL для загрузки ресурсов также является важным шагом, чтобы все элементы, такие как изображения, корректно отображались в PDF. Наконец, PDF-документ создается с помощью метода createPDF, и результат в виде массива байтов возвращается как выходной результат метода.

Класс SignatureService является сервисным компонентом, предназначенным для обработки цифровых подписей документов. Он инкапсулирует логику, связанную с генерацией и верификацией подписей, а также управлением ключами пользователей. Этот компонент использует алгоритмы криптографии для обеспечения безопасности и целостности документов.

Метод getSignByDocument отвечает за создание цифровой подписи для документа. Он принимает код документа и идентификатор пользователя в качестве параметров. Сначала метод вызывает generatorPDF из DtDocumentService, чтобы получить данные документа в формате PDF. Затем он извлекает закрытый ключ пользователя, используя метод getUserById из UserService. Для создания подписи используется алгоритм SHA256withRSA. Метод инициализирует объект Signature с закрытым ключом и обновляет его данными документа. После этого метод обновляет подпись в соответствующем документе, извлекая его из базы данных по коду. Если документ не найден, выбрасывается исключение с соответствующим сообщением. В завершение, метод сохраняет документ с обновленной подписью в репозитории и возвращает сгенерированную подпись.

Метод generatorSignatureKeys отвечает за создание пары ключей (закрытого и открытого) для пользователя. Он принимает идентификатор пользователя и использует KeyPairGenerator для генерации ключей с размером 2048 бит. После создания ключей метод выводит их в консоль в формате Base64 и сохраняет закрытый и открытый ключ в объекте пользователя. Затем обновленные данные пользователя сохраняются в репозитории.

Для получения запросов и отправки данных используются REST контроллеры. В архитектуре современных веб-приложений REST (Representational State Transfer) является популярным стилем взаимодействия, который основывается на принципах HTTP и предоставляет стандартизированный способ обмена данными между клиентом и сервером. REST-контроллеры играют ключевую роль в реализации этого подхода, обеспечивая обработку HTTP-запросов и управление бизнес-логикой приложения. Контроллеры принимают HTTP-запросы от клиентов (например, браузеров или мобильных приложений) и обрабатывают их в соответствии с заданной логикой приложения.

Каждый метод контроллера соответствует определенной операции. Например, метод для регистрации документа выглядеть так:

@PostMapping(ADD\_DT\_URL)

public ResponseEntity<Response> registrationDT(@RequestBody DtTypeDTO dt) throws Exception {

registrationService.registrationDtDocumentMain(dt);

return new ResponseEntity<>(Response.builder().message("Документ отправлен на рассмотрение")

.timestamp(System.currentTimeMillis()).build(), HttpStatus.CREATED);

}

Аннотация @PostMapping(ADD\_DT\_URL) связывает метод с определенным URL-адресом, который будет обрабатывать POST-запросы. Аннотация @RequestBody указывает, что данные для создания документа будут переданы в теле запроса в формате JSON. Объект DtTypeDTO представляет собой Data Transfer Object, который содержит все необходимые поля для регистрации документа. Это обеспечивает удобный способ передачи данных между клиентом и сервером. Далее происходит вызов сервиса для сохранения документа. После успешного выполнения регистрации документа метод формирует ответ в виде объекта ResponseEntity, который позволяет управлять ответами HTTP, включая статус и тело ответа. Метод возвращает статус HttpStatus.CREATED, что указывает на успешное создание нового ресурса. Это соответствует стандартам REST, где успешное создание ресурса обычно сопровождается статусом 201.

Для разработки базы данных использована библиотека Liquibase. Liquibase представляет собой инструмент, позволяющий автоматизировать процесс создания и изменения структуры базы данных с помощью декларативного подхода, что обеспечивает удобство и надежность. В проекте была создана XML-конфигурация, которая описывает изменения в базе данных. Файл databaseChangeLog.xml содержит несколько наборов изменений (changeSet), каждый из которых описывает определенные операции, такие как создание таблиц, добавление данных и изменение структуры. В каждом наборе описаны SQL-скрипты, которые применяются к базе данных. Например, одним из изменений, применяемых к базе данных, является создание таблицы для хранения документов:

create table documents(

id int primary key GENERATED BY DEFAULT AS IDENTITY,

declarant\_id int references persons(id),

document\_code varchar(32) not null UNIQUE,

sender varchar(50),

recipient varchar(50),

reason varchar(100) not null,

status varchar(32) default null,

inspector\_id int default null,

special\_mark text,

date\_created timestamp,

date\_updated timestamp default null,

user\_id int references users(id),

signature\_inspector text default null);

Кроме этого существуют специальные классы (Utils-классы) в программировании, которые содержат статические методы, предназначенные для выполнения общих задач или операций, которые могут быть полезны в различных частях приложения. Эти классы не предназначены для создания экземпляров и обычно не имеют состояния (нет полей экземпляра). Одним из таких классов является RandomStringGenerator. Он предназначен для генерации случайных строк, состоящих из букв нижнего регистра и цифр. Он может быть полезен в различных сценариях, таких как создание уникальных идентификаторов, паролей или токенов. Метод для генерации выглядит так:

public static String generateRandomString(int length) {

if (length < 1) throw new IllegalArgumentException();

StringBuilder sb = new StringBuilder(length);

for (int i = 0; i < length; i++) {

int rndCharAt = random.nextInt(DATA\_FOR\_RANDOM\_STRING.length());

char rndChar = DATA\_FOR\_RANDOM\_STRING.charAt(rndCharAt);

sb.append(rndChar);

}

return sb.toString();

}

Метод принимает один параметр length, который указывает длину генерируемой строки. Если length меньше 1, выбрасывается исключение IllegalArgumentException, что предотвращает создание строк нулевой или отрицательной длины. В цикле, который выполняется length раз, для каждой итерации. Генерируется случайный индекс, который используется для выбора символа из DATA\_FOR\_RANDOM\_STRING. Метод возвращает сгенерированную строку в виде строки.