4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

Данное вэб-приложение разработано по принципу REST, архитектурного стиля, который используется для создания веб-приложений и API. RESTful веб-приложения предоставляют доступ к ресурсам, представленным в формате URI, и выполняют операции с этими ресурсами с помощью стандартных методов HTTP, таких как GET, POST, PUT, DELETE.

Схема работы RESTful приложения, которую можно посмотреть на рисунке 4.1, обычно выглядит следующим образом:

1. Клиент отправляет запрос на сервер, используя HTTP методы: GET, POST, PUT, DELETE.
2. Сервер обрабатывает запрос и возвращает ответ в формате JSON, XML или другом формате.
3. Клиент обрабатывает ответ сервера и обновляет интерфейс приложения.

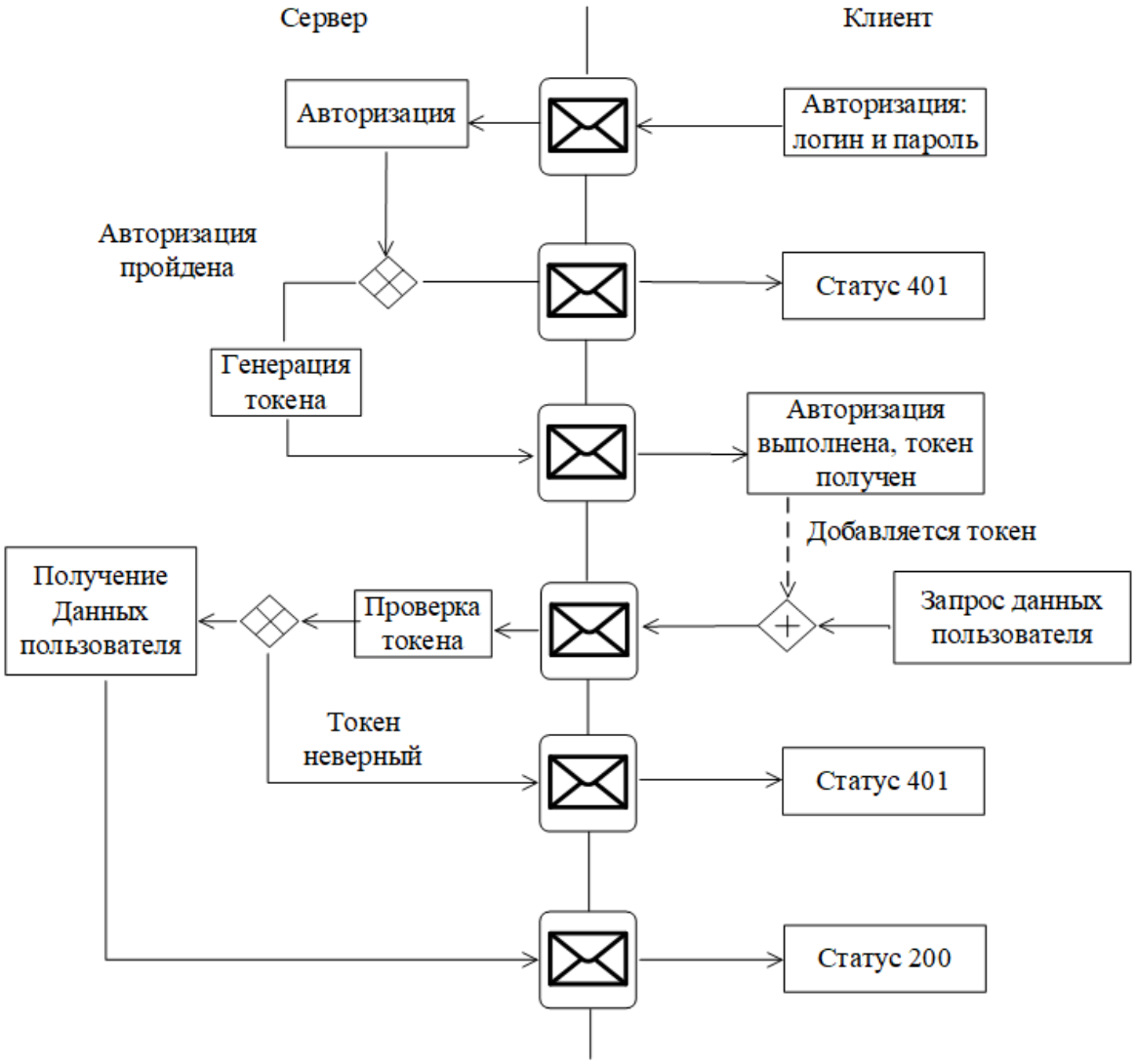

Рисунок 4.1 – Схема работы RESTful приложения

Диаграмма последовательностей для дипломного проекта на схеме ГУИР.400201.307 РР.3.

4.1 Алгоритм авторизации и регистрации

Для авторизации пользователей в данном приложении использовался JSON Web Token(JWT) – это стандарт создания токенов доступа, использующий формат JSON. Он открытый и определен в RFC 7519. Обычно, он используется для передачи данных, необходимых для аутентификации в клиент-серверных приложениях. Токены создаются на сервере, затем подписываются с помощью секретного ключа и передаются клиенту. После этого клиент использует токен для подтверждения своей личности.

Когда пользователь успешно проходит аутентификацию на сервере, сервер создает JWT и передает его обратно клиенту. JWT состоит из трех частей: заголовка, полезной нагрузки и подписи.

Весь процесс авторизации представляется следующим образом:

Пользователь отправляет запрос на аутентификацию, POST-запрос на по пути `/authenticate` с помощью класса `AuthenticationRequest`, который содержит следующие поля:

- `private String login;`
- `String password;`

Они представляют собой логин и пароль пользователя.

Далее сервер проверяет данные с помощью класса `AuthenticationService`, который содержит следующий метод:

```
public AuthenticationResponse
     authenticate(AuthenticationRequest request) {
        authenticationManager.authenticate(new
     UsernamePasswordAuthenticationToken(request.getLogin(),
     request.getPassword()));
        var user =
userService.findUserByFirstName(request.getLogin());
        var jwtToken = jwtService.generateToken(user);
        return AuthenticationResponse.builder()
                .token(jwtToken)
                .build();
    }
```

В этом методе происходит проверка валидности данных в `authenticationManager.authenticate`. Далее в методе `.findUserByFirstName`, с помощью базы данных, метод сверяет логин и пароль.

Пароли хранятся в зашифрованном виде с помощью хэширования, поэтому для того, чтобы сверить пароль, его необходимо сначала

раскодировать, для этого в классе `ApplicationConfig` определено поле класса `passwordEncoder`, который возвращает объект `BCryptPasswordEncoder`, который используется для хэширования пароля пользователя

Если данные прошли проверку, то метод генерирует JWT токен на основе найденного пользователя с помощью `jwtService.generateToken()`. В конце метод создаёт и возвращает объект `AuthenticationResponse`, содержащий сгенерированный токен в поле `token`. Токен содержит информацию об аутентифицированном пользователе и его правах доступа к ресурсам. В классе `JwtService` метод `generateToken()` выглядит так:

```
public String generateToken(UserDetails userDetails){
        return generateToken(new HashMap<>(), userDetails);
}
```

Для генерации токена необходим класс `UserDetails`, который содержит данные об аутентификации пользователя. Сервер отправляет сгенерированный JWT токен обратно клиенту в ответе на запрос аутентификации с помощью класса `AuthenticationResponse`, который содержит одно поле `private String token`.

При регистрации пользователя отправляется POST-запрос на `/auth/register`.

Он принимает в теле запроса объект `RegisterRequest`, содержащий данные, необходимые для регистрации нового пользователя, а именно следующие поля:

- `private String firstName;`
- `private String middleName;`
- `private String lastName;`
- `private String login;`
- `private String password;`
- `private String mail.`

Далее передает их в сервис `AuthenticationService` для выполнения регистрации с помощью метода `register`.

```
public AuthenticationResponse register(RegisterRequest
request) {
            Employee employee = employeeRepository
                    .findByFirstNameAndMiddleNameAndLastName(
                            request.getFirstName(),
                            request.getMiddleName(),
                            request.getLastName()).orElse(new
Employee());
            var user = User.builder()
```

```
                        .idLogin(employee.getId())
                        .firstName(request.getLogin())
                        .mail(request.getMail())

.password(passwordEncoder.encode(request.getPassword()))
                        .role(Role.USER)
                        .build();
                userService.saveUser(user);
                var jwtToken = jwtService.generateToken(user);
                return AuthenticationResponse.builder()
                        .token(jwtToken)
                        .build();
        }
```

В данном методе, как и при авторизации задействовано шифрование паролей. Это значит, что при регистрации и сохранении нового пароля для пользователя система вызывает метод `passwordEncoder.encode()`, чтобы пароли хранились сразу в зашифрованном виде.

Затем он возвращает объект `AuthenticationResponse`, содержащий информацию об успешности регистрации и авторизации нового пользователя, то есть содержит поле `private String token` с токеном.

При каждом запросе, требующем авторизации, клиент добавляет JWT токен в заголовок запроса `Authorization`. Например: `Authorization: Bearer <JWT_TOKEN>/`

Если токен истек или был поврежден, сервер возвращает ошибку аутентификации. Клиент в таком случае может запросить новый токен, повторив процесс аутентификации снова.

Для работы с токеном используется класс `JwtService`, содержит ряд методов:

1. Метод extractUsername извлекает имя пользователя из токена.
2. Метод extractClaim позволяет извлекать другие поля из токена, используя функцию-резольвер.
3. Метод generateToken генерирует токен, используя переданные дополнительные поля и данные пользователя. Токен подписывается с помощью ключа HMAC SHA-256.
4. Метод isTokenValid проверяет, действителен ли переданный токен и соответствует ли он данным пользователя.
5. Метод isTokenExpired проверяет, истек ли срок действия токена.
6. Метод extractExpiration извлекает дату истечения срока действия токена.
7. Метод extractAllClaims извлекает все поля токена.
8. Метод getSignInKey возвращает ключ для подписи токена. Ключ загружается из свойства SECRET_KEY, которое содержит закодированный ключ в формате Base64.

```
    private static final String SECRET_KEY =
"77217A25432A462D4A614E645267556B58703273357538782F413F4428472B4
B";

        public String extractUsername(String jwToken) {
            return extractClaim(jwToken, Claims::getSubject);
        }

        public <T> T extractClaim(String jwToken,
Function<Claims, T> claimsResolver){
            final Claims claims = extractAllClaims(jwToken);
            return claimsResolver.apply(claims);
        }

        public String generateToken(
                Map<String, Object> extraClaims,
                UserDetails userDetails
        ){
            return Jwts
                    .builder()
                    .setClaims(extraClaims)
                    .setSubject(userDetails.getUsername())
                    .setIssuedAt(new
Date(System.currentTimeMillis()))
                    .setExpiration(new
Date(System.currentTimeMillis() + 1200 * 60 * 24))
                    .signWith(getSignInKey(),
SignatureAlgorithm.HS256)
                    .compact();
        }

        public String generateToken(UserDetails userDetails){
            return generateToken(new HashMap<>(), userDetails);
        }

        public boolean isTokenValid (String jwToken, UserDetails
userDetails){
            final String username = extractUsername(jwToken);
            return (username.equals(userDetails.getUsername()))
&&
                    !isTokenExpired(jwToken);
        }

        private boolean isTokenExpired(String jwToken) {
            return extractExpiration(jwToken).before(new
Date());
        }

        private Date extractExpiration(String jwToken) {
            return extractClaim(jwToken, Claims::getExpiration);
        }
```

```
        private Claims extractAllClaims(String jwToken){
            return Jwts
                    .parserBuilder()
                    .setSigningKey(getSignInKey())
                    .build()
                    .parseClaimsJws(jwToken)
                    .getBody();
        }

        private Key getSignInKey() {
            byte[] keyBytes =
Decoders.BASE64.decode(SECRET_KEY);
            return Keys.hmacShaKeyFor(keyBytes);
        }
```

4.2 Работа Spring Security

Spring Security – это фреймворк для обеспечения безопасности в приложениях на платформе Spring. Он предоставляет широкий спектр функций для аутентификации, авторизации и защиты от атак, таких как CSRF, XSS и других.

Именно он позволяет шифровать пароли с помошью класса `UserDetails`, с использованием `BCrypt`. С его же помощью он позволяет созранять данные о авторизированном пользователе, что позволяет использовать это для работы с базой данных и получении данных о пользователе.

С помощью метода `securityFilterChain()` в классе `SecurityConfig` создается цепочка фильтров безопасности, которые позволяют управлять сессиями. Она определяет порядок, в котором фильтры будут применяться, и какие запросы будут обрабатываться каждым фильтром. Этот метод создает цепочку фильтров для обработки запросов HTTP. Метод использует объект `HttpSecurity`, который предоставляет DSL (Domain-Specific Language) для настройки конфигурации Spring Security.

```
    public SecurityFilterChain securityFilterChain(HttpSecurity
httpSecurity) throws Exception {
            httpSecurity
                    .cors(Customizer.withDefaults())
                    .and()
                    .addFilterBefore(new
CorsFilter(corsConfigurationSource()),
SessionManagementFilter.class)
                    .httpBasic().disable()
                    .csrf().disable()
                    .authorizeHttpRequests()
                    .requestMatchers("/**")
                    .permitAll()
```

```
                        .anyRequest()
                        .authenticated()
                        .and()
                        .sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                        .and()

.authenticationProvider(authenticationProvider)
                        .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class);
                return httpSecurity.build();
            }
```

Также в этом классе есть метод `CorsConfigurationSource`, который определяет, какие запросы от каких источников и с какими методами HTTP разрешены для взаимодействия с веб-приложением.

В методе создается объект `CorsConfiguration`, который содержит информацию о разрешенных `origins` и методах, а также другие настройки, такие как заголовки CORS (например, «`Access-Control-Allow-Origin`»). Затем определяется источник конфигурации на основе URL-адреса источника и конфигурации CORS, и возвращается созданный объект `CorsConfigurationSource`.

```
            CorsConfigurationSource corsConfigurationSource() {
                CorsConfiguration configuration = new
CorsConfiguration();

configuration.setAllowedOrigins(List.of("http://localhost:3000")
);

configuration.setAllowedMethods(Arrays.asList("GET","POST"));
                UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
                source.registerCorsConfiguration("/**",
configuration);
                return source;
            }
```

Эти же методы связаны с классом `JwtAuthenticationFilter`, который содержит один метод `doFilterInternal()`. Когда клиент отправляет запрос на сервер, он проходит через этот фильтр. В методе `doFilterInternal()`, он проверяет заголовок `Authorization` запроса, чтобы определить, есть ли токен в запросе. Если заголовок отсутствует или токен отсутствует, то фильтр пропускает запрос к следующему фильтру в цепочке.

```
    protected void doFilterInternal(@NonNull HttpServletRequest
request, @NonNull HttpServletResponse response, @NonNull
FilterChain filterChain) throws ServletException, IOException {
            final String authHeader =
request.getHeader("Authorization");
            final String jwToken;
            final String login;
            if (authHeader == null ||
!authHeader.startsWith("Bearer")) {
                filterChain.doFilter(request, response);
                return;
            }
            jwToken =  authHeader.substring(7);
            login = jwtService.extractUsername(jwToken);
            if(login != null &&

SecurityContextHolder.getContext().getAuthentication() == null){
                UserDetails userDetails =
this.userDetailsService.loadUserByUsername(login);
                if(jwtService.isTokenValid(jwToken,
userDetails)){
                    UsernamePasswordAuthenticationToken
authenticationToken =
                            new
UsernamePasswordAuthenticationToken(
                                    userDetails,
                                    null,

userDetails.getAuthorities());
                    authenticationToken.setDetails(
                            new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticat
ionToken);
                }
            }
            filterChain.doFilter(request, response);
        }
```

Если в запросе есть токен, то фильтр извлекает логин пользователя из токена, затем проверяет, не была ли аутентификация пользователя уже выполнена в текущем контексте безопасности. Если пользователя еще не было аутентифицировано, фильтр получает детали пользователя из `userDetailsService` и проверяет токен на его действительность с помощью `jwtService.isTokenValid()`. Если токен действителен, то создается `UsernamePasswordAuthenticationToken`, содержащий `UserDetails`, и устанавливается в текущий контекст безопасности через `SecurityContextHolder.getContext().setAuthentication(authenticat ionToken)`.

Фильтр затем передает запрос следующему фильтру в цепочке с помощью `filterChain.doFilter(request, response)`. Если пользователь был успешно аутентифицирован, то его контекст безопасности будет доступен для последующих запросов в рамках этой сессии.

4.3 Алгоритм запросов

После того, как пользователь зарегистрируется, получит токен на время сессии он может осуществлять запросы к серверу, чтобы взаимодействовать с сервером.

При каждом запросе будет срабатывать `SecurityFilter`, рассмотренный в разделе.

Для того, чтобы пользователь получил данные, с клиента отправляется GET или POST-запрос.

Например, чтобы получить данные н главную страницу о авторизированном сотруднике в классе `MainPageController` вызывается метод `getEmployeeInfo(Authentication authentication)`

```
    @GetMapping(path = "/mainUserInfo", produces =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<List<EmployeeFullView>>
getEmployeeInfo(Authentication authentication) {
    return
ResponseEntity.ok(employeeFullViewService.findAllByLoginUser
    (authentication.getName()));
    }
```

Этот метод является обработчиком GET-запроса на путь «`/mainUserInfo`» и возвращает список информации о сотрудниках в формате JSON. В качестве параметра метод принимает объект `Authentication`, который содержит информацию об аутентифицированном пользователе.

Метод использует эту информацию для получения имени пользователя из объекта `Authentication` и передачи его в качестве аргумента в метод `employeeFullViewService.findAllByLoginUser()`, который возвращает список `EmployeeFullView`, содержащих информацию о сотрудниках.

Метод `findAllByLoginUser()` работает через класс `EmployeeFullViewService`, который связан с интерфейсом `EmployeeFullViewRepository` с расширением `JpaRepository<EmployeeFullView, Long>`.

В результате успешного выполнения метод возвращает HTTP-ответ со статусом 200 (OK) и телом ответа, содержащим список информации о сотрудниках в формате JSON.

Кроме информации о сотруднике на главной странице содержится информация о событиях, заявлениях и заданиях, все эти методы являются GET-запросами. Их можно рассмотреть далее.

Запрос о просмотре заявлений выглядит так:

```
     @GetMapping(path = "/ls", produces =
MediaType.APPLICATION_JSON_VALUE)
         public ResponseEntity<List<LogStatementsView>>
getLsRequests(Authentication authentication) {
             return
ResponseEntity.ok(this.logStatementsViewService.findAllByIdAppro
verAndStatus(

userService.findUserByFirstName(authentication.getName()).getIdL
ogin(), 3));
         }
```

Этот запрос о событиях авторизированного пользователя:

```
     @GetMapping(path = "/events", produces =
MediaType.APPLICATION_JSON_VALUE)
         public @ResponseBody List<EventsView>
getEvents(Authentication authentication) {
             return this.eventsViewService.findAllByIdRecipient(

userService.findUserByFirstName(authentication.getName()).getIdL
ogin());
         }
```

Этот запрос о задачах авторизированного пользователя:

```
     @GetMapping(path = "/tasks", produces =
MediaType.APPLICATION_JSON_VALUE)
         public @ResponseBody List<TasksView>
getTasks(Authentication authentication) {
                 System.out.println(authentication);
             return this.tasksViewService.findAllByIdExecutor(

userService.findUserByFirstName(authentication.getName()).getIdL
ogin());
         }
```

Все эти три запроса основаны на извлечении данных о пользователе с помощью объекта `Authentication`, переданного в качестве параметра.

На главной страничке предусмотрена возможность подтверждать или опровергать заявления, это происходит через POST-запрос `setLsApprove()`. Для него необходимо знать номер заявления. Он передается с помощью кода клиентской части, от пользователя будет необходимо только нажать кнопку.

```
     @PostMapping(path = "/ls/{id}", produces =
MediaType.APPLICATION_JSON_VALUE)
         public ResponseEntity<String> setLsApprove(
```

```
                @PathVariable Long id,
                Authentication authentication,
                @RequestBody LogStatementRequest request
        ) {
            LogStatement logStatement =
logStatementService.findByIdAndIdApprover(
                    id,

userService.findUserByFirstName(authentication.getName()).getIdL
ogin()
                    ).orElse(new LogStatement());
            logStatement.setStatus(request.getStatus());
            System.out.println(logStatement.getId() + "    " +
                    logStatement.getStatus());

if(logStatementService.saveLogStatement(logStatement))
                System.out.println("ok");
            return ResponseEntity.ok("Done ");
        }
```

Принцип создания заявлений, событий и заданий похож. Все они представляют собой POST-запросы, используя классы-сервиса, в которые с помощью классов-запросов с содержанием полей вносятся данные пользователя, потом проверяются на валидность и сохраняются в базу данных. Пример подобного запроса:

```
    @PostMapping(path = "/lscreate", produces =
MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<String> createLS( @RequestBody
LogStatementCreateRequest request,
                                                Authentication
authentication
                                                ){
            LogStatement logStatement = LogStatement.builder()
                    .id(0)
                    .idApprover(request.getIdApprover())
                    .status(3)

.idEmployee(userService.findUserByFirstName(authentication.getNa
me()).getIdLogin())
                    .commentLs(request.getCommentLs())
                    .typeLeave(request.getTypeLeave())
                    .dateLeave(request.getDateLeave())
                    .dateOfLs(request.getDateOfLs())
                    .daysSum(request.getDaysSum())
                    .build();

if(logStatementService.saveLogStatementAll(logStatement))
                System.out.println("ok");
            if(request.getBodyDoc()!= null) {
```

```
                Document document = Document.builder()
                        .bodyDoc(request.getBodyDoc())
                        .idLs(logStatement.getId())
                        .build();
                if(documentService.saveDocument(document))
                    System.out.println("ok");
            }
            return ResponseEntity.ok("Done ");
        }
```

В данном методе идет сохранение в две таблицы `LogStatement` и `Document` в силу того, что они связан друг с другом. К заявлению можно приложить документ, его физическую копию, если таковой имеется.

Запрос на смену представляет собой POST-запрос, который через класс `LoginRequest` , который также вызывает класс для шифрования `passwordEncoder` пароля.

```
    @PostMapping(path = "/password", produces =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<String> changePassword(@RequestBody
LoginRequest request,
                                                 Authentication
authentication) {
    User user =
userService.findUserByFirstName(authentication.getName());

user.setPassword(passwordEncoder.encode(request.getPassword()));
    if (userService.saveUserPassword(user))
    System.out.println("ok");
    return ResponseEntity.ok("Done");
    }
```

Метод обрабатывает GET-запрос на получение списка сотрудников в виде JSON. Он вызывает метод `findAll()` из `employeesViewService` для получения списка всех сотрудников, создает `ResponseEntity` с HTTP-статусом "200 OK" и списком сотрудников в теле ответа, и возвращает его в качестве ответа на запрос.

```
    @GetMapping(path     =     "/employees",     produces     =
MediaType.APPLICATION_JSON_VALUE)
    public @ResponseBody List<EmployeesView> getEmployees() {
    return this.employeesViewService.findAll();
    }
```