Ветров С.Ю. Безопасность микросервисов с помощью SPRING BOOT, SPRING SECURITY и GATEWAY // Международный журнал информационных технологий и энергоэффективности. – 2024. – Т. 9 № 5(43) с. 33–39

Международный журнал информационных технологий и энергоэффективности

Сайт журнала: http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.056

БЕЗОПАСНОСТЬ МИКРОСЕРВИСОВ С ПОМОЩЬЮ SPRING BOOT, SPRING SECURITY И GATEWAY

Ветров С.Ю.
ФГБОУ ВО «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)», Москва, Россия, (125993, город Москва, Волоколамское ш., д. 4), e-mail: vetrov241201@yandex.ru

Данная статья посвящена анализу и рассмотрению методов обеспечения безопасности в микросервисной архитектуре с использованием инструментов Spring Boot, Spring Security и Spring Cloud Gateway. Мы рассмотрели ключевые аспекты аутентификации и авторизации пользователей, а также роль централизованного шлюза в управлении доступом к микросервисам. Подробно рассмотрены шаги по реализации данного подхода с использованием указанных инструментов и методов обеспечения безопасности передачи данных между клиентом и сервером. В результате статьи читатель получит понимание о том, как создать гибкую и безопасную систему, соответствующую современным требованиям безопасности при разработке микросервисных приложений.

Ключевые слова**:** Микросервисная архитектура, авторизация, аутентификация, API, разработка приложений, backend.

MICROSERVICES SECURITY WITH SPRING BOOT, SPRING SECURITY AND GATEWAY

Vetrov S.Y.
MOSCOW AVIATION INSTITUTE (NATIONAL RESEARCH UNIVERSITY), Moscow, Russia, (125993, Moscow, Volokolamskoye shosse, 4), e-mail: vetrov241201@yandex.ru

This article is devoted to analyzing and reviewing methods of providing security in microservice architecture using Spring Boot, Spring Security and Spring Cloud Gateway tools. We have considered key aspects of user authentication and authorization, as well as the role of a centralized gateway in managing access to microservices. The steps to implement this approach using the specified tools and techniques to secure data transfer between client and server are discussed in detail. As a result of the article, the reader will gain an understanding of how to create a flexible and secure system that meets modern security requirements when developing microservice applications.

Keywords: Microservice architecture, authorization, authentication, API, application development, backend.

Микросервисная архитектура — это некое развитие сервис-ориентированной архитектуры (SOA), направленное на взаимодействие небольших, слабо связанных и легко заменяемых модулей — микросервисов. Микросервис — это изолированная, слабосвязанная единица разработки, работающая над одной задачей [1].

Микросервисная архитектура стала одним из наиболее распространенных подходов к разработке современных приложений. Она позволяет создавать гибкие и масштабируемые системы, разбивая их на небольшие автономные сервисы. Однако при работе с

микросервисами встает вопрос об эффективном и безопасном доступе к самим микросервисам. В этой статье мы рассмотрим использование Spring Boot, Spring Security и Gateway для реализации авторизации посредством сессий в микросервисной архитектуре.

Рассмотрим пример: Проект, реализованный с использованием микросервисной архитектуры. Пользователи могут делиться информацией об интересных событиях и находить компанию для участия в них. Могут отправлять заявки на участие в событиях и оставлять к ним комментарии. Так же есть функционал администратора.

Выделяются 3 микросервиса (рисунок 1):

events – микросервис для работы с событиями;

requests – микросервис для работы с заявками;

comments – микросервис для работы с комментариями.

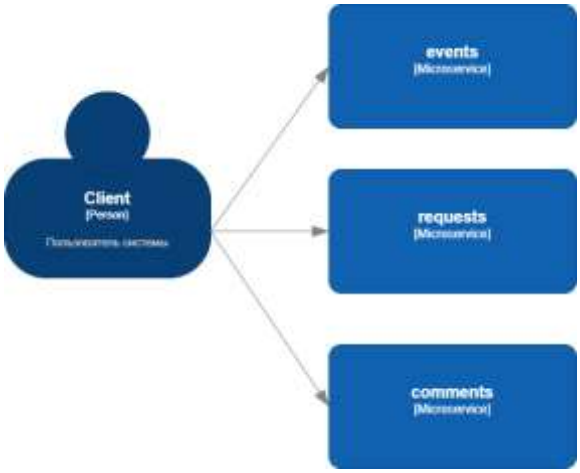

Рисунок 1 – Микросервисы

Как же нам организовать авторизацию пользователя, чтобы понимать какой пользователь обращается к микросервису.

Вариант 1. Отдельный микросервис для авторизации.

Добавляется новый микросервис Auth Service (Рисунок 2), который работает с таблицей users, а другие микросервисы, после получения запросов от пользователя, делают запрос на микросервис авторизации.

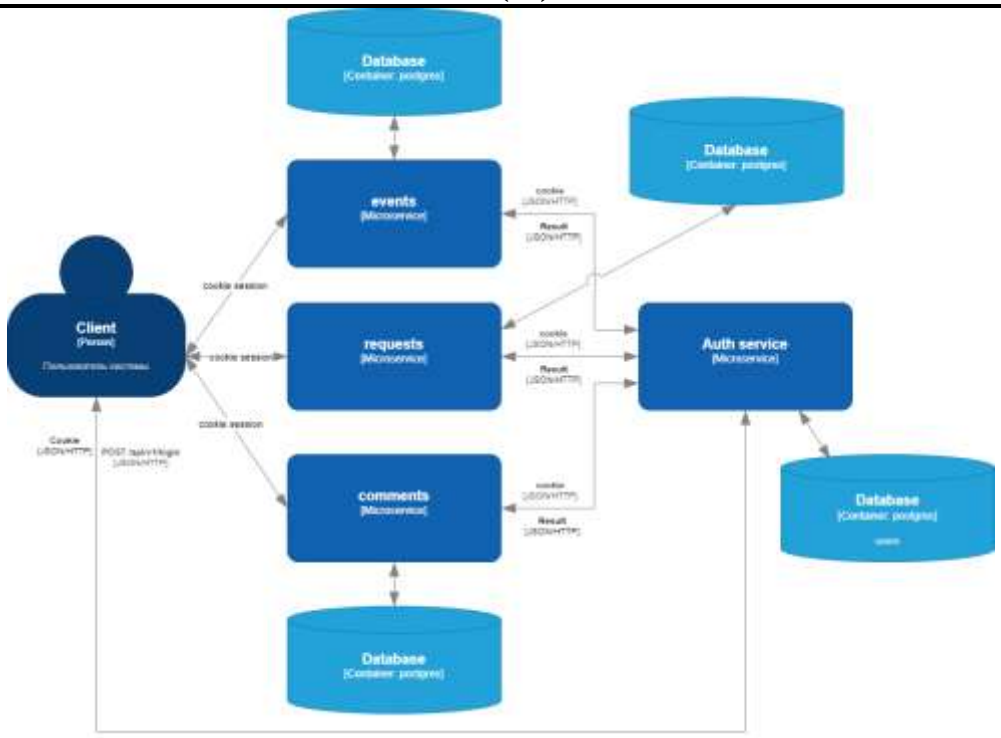

Рисунок 2 – Вариант 1

В данном сценарии, процесс начинается с того, что клиент отправляет запрос на микросервис аутентификации через метод POST (/api/v1/login). Если предоставленные логин и пароль верны, клиент получает в ответ ключ сессии, который сохраняется в куки. Затем клиент использует этот ключ для отправки запроса на целевой микросервис.

После выполнением запроса к целевому микросервису (например, GET /api/v1/events), сам микросервис инициирует запрос к микросервису аутентификации для проверки сессии. Если пользователь успешно аутентифицирован и у него есть доступ к этому ресурсу, то происходит выполнение запроса на выборку событий и передача их клиенту. В случае, если аутентификация не подтверждена, возвращается ошибка с кодом 401 Unauthorized, уведомляя клиента о необходимости повторной аутентификации.

Проблемы этого варианта:

- Каждый микросервис должен выполнять свою роль, а в этом случае, ему необходимо делать дополнительные действия по валидации сессии;
- Микросервисы содержат разные пути, и придётся прописывать логику какие пути требуют авторизацию, а какие нет.

Вариант 2. Использование Api Gateway.

Здесь добавляется новый микросервис Api Gateway, далее шлюз [2]. У шлюза есть конфигурационный файл, в котором прописано, какой путь куда отправлять, например, GET **gateway**/api/v1/events, запрос на получение вообще всех событий. Шлюз должен направить этот запрос на **events**/api/v1/events, то есть на другой микросервис. Вернёмся к нашему случаю (рисунок 3). Шлюз получает запрос от клиента, например на получение всех своих событий. Запрос будет выглядеть так: GET /api/v1/user/events. В самом запросе идентификатор пользователя не указан. В шлюзе сначала идёт запрос на сервис авторизации, там проверяется,

если доступ у этого пользователя и далее подменяется запрос с GET /api/v1/**user/events** на GET /api/v1/**user/1/events**. Что изменилось. Добавился идентификатор пользователя. Сервер авторизации проверил сессию и вернул в шлюз этот идентификатор. И получается, что сам микросервис событий просто вернёт список событий именно этого пользователя, никакой проверки внутри микросервиса нет.

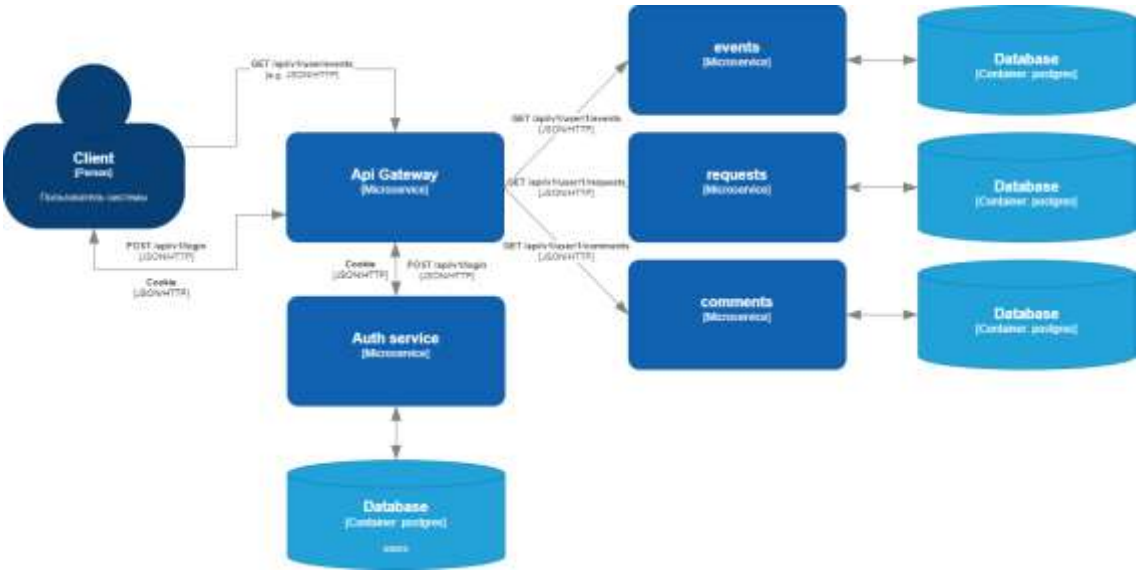

Рисунок 1 – Вариант 2

Проблемы этого варианта:

- Придётся писать конфигурации в шлюзе для каждого пути;
- На каждый запрос приходится ещё 2 дополнительных запроса:

клиент → шлюз → сервис авторизации → целевой микросервис.

Практическая реализация.

Для реализации второго варианта будет использоваться фреймворк Spring Boot 3.2.2, Spring Security 6 и Spring Cloud Gateway 4.1.1.

Сервис авторизации.

Для начала нужно создать проект на сайте start.spring.io и выбрать две зависимости: Spring Web и Spring Security.

Далее создаём конфигурационный класс и в нём создаём бин, в котором определяются настройки авторизации.

Код SecurityConfig

```
    @Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http
        .csrf(AbstractHttpConfigurer::disable)
        .cors(AbstractHttpConfigurer::disable)
        .headers(h -> h.frameOptions(HeadersConfigurer.FrameOptionsConfig::disable))
        .anonymous(AbstractHttpConfigurer::disable)
        .requestCache(RequestCacheConfigurer::disable)
        .formLogin(form -> form.usernameParameter("usernameOrEmail").loginPage("/login").disable())
```

```
        .httpBasic(AbstractHttpConfigurer::disable)
        .logout(l -> l.logoutUrl("/logout").invalidateHttpSession(true).clearAuthentication(true).disable())
        .securityContext((securityContext) -> securityContext.requireExplicitSave(false))
        .sessionManagement(s -> s.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                .maximumSessions(1)
                .maxSessionsPreventsLogin(false)
                .sessionRegistry(sessionRegistry)
        )
        .authorizeHttpRequests((authorize) -> authorize
                .requestMatchers("/users/**").authenticated()
                .requestMatchers("/user/**").authenticated()
                .requestMatchers("/todos/**").authenticated()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/data/user/**").authenticated()
                .requestMatchers("/test").authenticated()
                .requestMatchers("/test2").authenticated()
                .anyRequest().permitAll()
        )
        .addFilterAfter(newLoginFilter, UsernamePasswordAuthenticationFilter.class)
        .exceptionHandling(c -> c.authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED)));
    return http.build();
}
```

Самое главное выделено жирным шрифтом, там определяются пути, которые будут доступны всем, только авторизованным пользователям или пользователям с определёнными ролями. Например, пути /users/** доступны только для авторизованных, а /admin/** только для пользователей с ролью администратора.

Api gateway.

Создаём проект на сайте start.spring.io и выбираем две зависимости: Spring Web и Gateway.

Далее создаём файл конфигурации application.yaml и прописываем правила, по которым будут распределяться маршруты.

Фрагмент файла application.yaml

```
        routes:
    - id: event-route
      uri: http://localhost:7989
      predicates:
        - Path=/user/events/**
      filters:
        - name: AccessSecurityFilter
        - name: AssignUserSecurityFilter
```

Тут назначается идентификатор маршрута, url, на который будет перенаправляться маршрут, и сам путь. Далее идёт фильтры в который как раз и будет подставляться идентификатор пользователя.

В AccessSecurityFilter происходит запрос на сервис авторизации и проверка сессии. Если пользователь прошёл проверку, то фильтр передаёт запрос дальше.

Код AccessSecurityFilter

```
    @Override
public GatewayFilter apply(Config config) {
    return (exchange, chain) -> {
        HttpCookie sessionCookieValue = exchange.getRequest().getCookies().getFirst("SESSION");
        if (sessionCookieValue == null) {
            return Mono.error(new ResponseStatusException(HttpStatus.UNAUTHORIZED, "Unauthorized"));
        } else {
            String requestPath = exchange.getRequest().getPath().toString();
            return webClientBuilder.build()
                    .get()
                    .uri(authServiceUrl + requestPath)
                    .header(HttpHeaders.COOKIE, sessionCookieValue.toString())
                    .exchange()
                    .flatMap(response -> {
                        if (response.statusCode().equals(HttpStatus.NOT_FOUND) ||
response.statusCode().equals(HttpStatus.OK)) {
                            return chain.filter(exchange);
                        } else {
                            return Mono.error(new ResponseStatusException(HttpStatus.UNAUTHORIZED, "Unauthorized)))"));
                        }
                    });
        }
    };
}
```

В AssignUserSecurityFilter происходит подстановка идентификатора пользователя, на красной строке.

Код AssignUserSecurityFilter

```
    @Override
public GatewayFilter apply(Config config) {
    return (exchange, chain) -> {
        HttpCookie sessionCookieValue = exchange.getRequest().getCookies().getFirst("SESSION");
        if (sessionCookieValue == null) {
            return Mono.error(new ResponseStatusException(HttpStatus.UNAUTHORIZED, "Unauthorized"));
        }

        return webClientBuilder.build()
                .get()
                .uri(authServiceUrl + "/user")
                .header(HttpHeaders.COOKIE, sessionCookieValue.toString())
                .retrieve()
                .bodyToMono(UserFullDto.class)
                .flatMap(user -> {
                    ServerHttpRequest request = exchange.getRequest();
                    String originalPath = request.getPath().value();
                    String modifiedPath = originalPath.replace("/user/", "/user/" + user.getId() + "/");
                    ServerHttpRequest modifiedRequest = request.mutate().path(modifiedPath).build();
                    return chain.filter(exchange.mutate().request(modifiedRequest).build());
                })
```

```
            .onErrorResume(error -> Mono.error(new ResponseStatusException(HttpStatus.UNAUTHORIZED,
"Unauthorized")));
    };
}
```

Заключение.

В данной статье мы рассмотрели два варианта организации авторизации пользователей в микросервисной архитектуре.

Первый вариант предполагает создание отдельного микросервиса для авторизации, который обрабатывает запросы от других микросервисов и осуществляет проверку сессий. Этот подход более простой, но требует каждому микросервису выполнять дополнительные действия по валидации сессии.

Второй вариант использует API Gateway для управления доступом к микросервисам. Здесь шлюз направляет запросы на сервис авторизации для проверки доступа пользователя. После успешной аутентификации шлюз подменяет запросы, добавляя идентификатор пользователя. Этот подход позволяет централизованно управлять авторизацией и уменьшает нагрузку на отдельные микросервисы, но требует дополнительных запросов и конфигураций в шлюзе.

В практической реализации второго варианта были использованы фреймворки Spring Boot, Spring Security и Spring Cloud Gateway.

Выбор конкретного варианта зависит от требований к безопасности, гибкости и производительности системы, а также от ее архитектурных особенностей. При правильной реализации оба варианта позволяют создать безопасную и удобную для использования систему, соответствующую современным стандартам разработки микросервисных приложений.

Список литературы

1. Аутентификация и авторизация в проекте с микросервисной архитектурой: стратегии, практический пример. — Текст : электронный//Harb: [сайт]. — URL: https://habr.com/ru/companies/spectr/articles/715290.
2. Pattern: API Gateway / Backends for Frontends. — Текст : электронный//microservices.io : [сайт]. — URL: https://microservices.io/patterns/apigateway.html.
3. Microservices with Spring. — Текст: электронный//Spring: [сайт]. — URL: https://spring.io/blog/2015/07/14/microservices-with-spring.

References

1. Authentication and authorization in a project with a micro-service architecture: strategies, a practical example. — Text : electronic//Harb : [website]. — URL: https://habr.com/ru/companies/spectr/articles/715290.
2. Pattern: API Gateway / Backends for Frontends. — Text : electronic // microservices.io : [website]. — URL: https://microservices.io/patterns/apigateway.html.
3. Microservices with Spring. — Text : electronic//Spring : [website]. — URL: https://spring.io/blog/2015/07/14/microservices-with-spring.