Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Поддержка генерации тестов для Spring Security в UnitTestBot

ПАРФЁНОВ Данил Игоревич

Отчёт по учебной практике
в форме «Теоретическое исследование»

Научный руководитель:
к. ф.-м. н., доц. Е. К. Куликов

Санкт-Петербург
2023

Оглавление

Введение

С развитием программного обеспечения и ускорением процессов разработки значимость автоматизированного тестирования становится всё более очевидной. Эффективное тестирование обеспечивает не только надежность кода, но и повышает уверенность разработчиков в стабильности и безопасности их приложений. В этом контексте инструмент для автоматической генерации тестов UnitTestBot представляет собой ценное новшество. Этот инструмент упрощает процесс создания тестов для кода на различных языках программирования, в частности для Java [13].

Однако, с расширением возможностей и диапазона применения этого инструмента, возникает потребность в генерации тестов, специфических для фреймворков, таких как Spring [7]. Несмотря на то что поддержка значительной части Spring-специфичного кода уже реализована, интеграция Spring Security представляет собой отдельный вызов, учитывая сложность и специфичность систем безопасности.

«Spring Security — это фреймворк, ориентированный на обеспечение аутентификации и авторизации Java-приложений» [8]. На данный момент в UnitTestBot поддержка Spring Security реализована довольно примитивным образом. Расширение функциональности UnitTestBot в генерировании тестов для данного фреймворка позволит увеличить покрытие кода с аутентификацией и авторизацией.

Данная работа представляет собой детальный анализ тестирования Spring Security-специфичного кода, сравнение с аналогичными генераторами тестов, а также предложение о дальнейшей реализации тестирования выявленных особенностей.

1. Постановка задачи

Целью работы является реализация поддержки генерации тестов, совместимых со Spring Security в UnitTestBot. Для её выполнения были поставлены следующие задачи:

1. Провести обзор существующих аналогов автоматических генераторов тестов на предмет генерации тестов для Spring Security.
2. Проанализировать методики ручного создания тестовых сценариев.
3. Провести сравнение тестов генерируемых UnitTestBot с предложенными в этой работе.
4. Предложить возможные способы реализации частичной поддержки Spring Security.

На весенний семестр:

1. Имплементировать предложенную функциональность.

2. Обзор

В данной главе проводится обзор существующих аналогов UnitTestBot, а также используемых технологий для выполнения поставленных задач.

2.1. Обзор существующих автогенераторов

Существует множество программ, генерирующих тесты для кода на Java, такие как EvoSuite[3], Kex[1] и т.д. Несмотря на то, что Spring является одним из самых популярных фреймворков для Java [9], не так много вышеупомянутых программ должным образом генерируют тесты для Spring-специфичного кода. Для данного обзора ввиду поддержки Spring были выбраны три аналога UnitTestBot: Diffblue Cover [2], Parasoft Jtest [5], Symflower [10].

Для проведения сравнения было написано mock-приложение[1] с часто используемыми особенностями Spring Security для авторизации и аутентификации. Это базовая функциональность Spring Security, без тестирования которой не имеет смысла расматривать более сложные случаи. В данном приложении есть три конечных точки:

1. `/admin` — вовзращает имя аутентифицированного пользователя, при этом требуя роль `ADMIN`, указанной в `SecurityConfig` при построении `HttpSecurity`;

2. `/user` — вызывает метод сервиса, возвращающий строку, при этом требуя право «read», указанной в аннотации `@PreAuthorize`;

3. `/all` — возвращает строку, при этом не требуя аутентификации.

Тесты, сгенерированные Diffblue Cover, не брали в расчёт авторизацию и аутентификацию, вследствие чего сгенерированные тесты не тестировали авторизацию должным образом (рис. 1). Говоря конкретно про контроллеры, интеграционные тесты, генерируемые Diffblue Cover,

[1] https://github.com/ancavar/mock-repo.git

используют другой подход к имитированию отправки запросов. Они ограничивают контекст контроллера, что делает невозможным тестирование особенностей Spring Security.

```
@ContextConfiguration(classes = {Controller.class, Service.class})
@ExtendWith(SpringExtension.class)
class ControllerDiffblueTest {
    @Autowired
    private Controller controller;

    @Test
    void testUser2() throws Exception {
        MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders.get("/user");
        requestBuilder.contentType("https://example.org/example");
        MockMvcBuilders.standaloneSetup(controller)
                .build()
                .perform(requestBuilder)
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.content()
                        .contentType("text/plain;charset=ISO-8859-1"))
                .andExpect(MockMvcResultMatchers.content()
                        .string("Hello"));
    }
}
```

Рис. 1: Пример теста, сгенерированного Diffblue Cover.

В случае с Symflower удалось сгенерировать лишь один тест, использующий контекст Spring, но также не учитывающий авторизацию (рис. 2). Все остальные были модульными, то есть без контекста. Также по какой-то причине не была импортирована аннотация `@Test`.

Parasoft Jtest удалось сгенерировать тест для `/admin` таким образом, чтобы в запросе использовался `principal`, однако по какой-то причине он не был нормально инициализирован (рис. 3). Впрочем, в остальном тесты практически такие же, что и у Symflower, а поэтому и неработающие.

```java
@WebMvcTest(Controller.class)
public class ControllerSymflowerAdminTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private Service service;

    @Test
    public void admin() throws Exception {
        this.mockMvc.perform(get("/admin")).andExpect(status().isOk())
            .andExpect(view().name(""))
            .andExpect(content().string(""));
    }
}
```

Рис. 2: Пример теста, сгенерированного Symflower.

```java
@WebMvcTest(Controller.class)
@DirtiesContext
public class ControllerSpringTest {
    @Autowired
    MockMvc mockMvc;

    Authentication authentication;

    @MockBean
    Service service;

    @Test
    public void testAdmin() throws Throwable {
        // When
        ResultActions actions = mockMvc.perform(get("/admin").principal(authentication));

        // Then
        // actions.andExpect(status().isOk());
        // actions.andExpect(header().string("", ""));
        // actions.andExpect(view().name(""));
    }
}
```

Рис. 3: Пример теста, сгенерированного Parasoft Jtest.

2.2. Обзор используемых технологий

- Java — язык генерации тестов.
- Kotlin — язык разработки UnitTestBot.
- Spring Security [8] — то, для чего пишутся тесты. Фреймворк, отвечающий за авторизацию и аутентификацию, а также защищающий от разного рода атак, например, CSRF[2].
- JUnit4, JUnit5 [4] — фреймворки, предназначенные для написания и запуска тестов на языке программирования Java. Выбор обусловлен тем, что только данный фреймворк поддерживается UnitTestBot при генерации интеграционных тестов.
- spring-addons [15] — библиотека, упрощающая настройку OAuth2/OIDC и их тестирование. В данной работе будет использовано только последнее, позволяющее использовать аннотации вместо менее удобных способов создания контекста, а так же тестировать сервисы, репозитории.

[2]Cross-Site Request Forgery (CSRF) — это уязвимость веб-безопасности, которая позволяет злоумышленникам обманом заставить пользователей выполнить нежелательные действия в веб-приложении, где они прошли аутентификацию.

3. Методика написания ручных тестов и вопросы реализации

В данной главе описывается методика написания ручных тестов для Spring Security-специфичного кода и руководства по реализации в UnitTestBot. В этом исследовании рассматриваются только сервлеты, но все методики относятся и к реактивным приложениям, хоть и с небольшими изменениями в коде.

За основу для построения методики брались различные руководства и пособия [6][12][11][16].

3.1. Нынешняя реализацию тестирования Spring Security в UnitTestBot

На данном этапе в UnitTestBot тестирование реализовано как аннотация `@WithMockUser` поверх всего тест-класса, которая ставится в случае присутствия Spring Security. Это препятствует тестированию отдельных случаев аутентификации, речь о которых пойдет в дальнейших главах.

3.2. Основные особенности тестирования Spring Security

Основой тестирования Spring Security является mock-аутентификация. С помощью mock-пользователя полностью пропускается этап аутентификации и тестируется непосредственно авторизация и другая работа приложения с пользователем.

3.2.1. `@WithAnonymousUser`

Тем не менее, полезно практиковать отрицательное тестирование с использованием аннотации `@WithAnonymousUser`. Это позволяет моделировать ситуации, когда пользователь не прошел аутентификацию. В таких случаях система должна корректно обрабатывать запросы

анонимного пользователя, например, ограничивать доступ к определенным ресурсам или функциям приложения. По своей сути, такие тесты должны сопровождать те, которые тестируют «ожидаемое» поведение.

3.2.2. CSRF

При тестировании POST-конечных точек в большинстве случаев важно снабжать запрос CSRF токеном. В некоторых случаях это может быть излишним — об этом пойдет речь в главе о `SecurityFilterChain`.

```
MockHttpServletRequestBuilder mockHttpServletRequestBuilder =
post("/hello", uriVariables).with(csrf());
```

Рис. 4: Строка из теста, сгенерированного UnitTestBot для POST запроса, с добавленным вручную `with(csrf())`.

3.3. Безопаность методов

Spring Security позволяет проводить авторизацию на уровне методов. Например, со Spring Security можно указать, что какой-то метод (то есть конечная точка) требует право на запись (рис. 5).

```
@GetMapping("/endpoint")
@PreAuthorize("hasAuthority('write')")
public String endpoint()
```

Рис. 5: Пример осуществления безопасности методов с помощью `@PreAuthorize`.

Здесь потребуется `@WithMockUser(authorities = "write")`, чтобы настроить mock-пользователя. Для таких примитивных случаев достаточно посмотреть на саму аннотацию, а конкретно спарсить и интерпретировать SpEL[3] выражение на наличие ограничений, нужных для

[3]Spring Expression Language

построение тестового `Authentication`. Также можно использовать рефлексию, «достать» `value` аннотации, и интерпретировать абстрактное синтаксическое дерево полученного SpEL выражения.

`@PostAuthorize` используется для валидации возвращаемого методом объекта (рис. 6). Для тестирования потребуется не только mock-пользователь, но еще и некий объект, который «достают» из некого хранилища. Помимо парсинга выражения, реализация такого подхода в автогенерации тестов требует получение типа объекта, возвращаемого методом, а также тестового хранилища.

```
@Test
@DisplayName("findProductContaining: text = 'abc'")
@WithMockUser(username = "user")
public void testFindProductContainingWithNonEmptyString() throws Exception {
    Product product = new Product();
    product.setName("abcd");
    product.setOwner("user");
    entityManager.persist(product);
    entityManager.flush();
    UriComponentsBuilder uriComponentsBuilder = fromPath("/products/{text}");
    Map uriVariables = new HashMap();
    uriVariables.put("text", "abc");
    UriComponentsBuilder uriComponentsBuilder1 = uriComponentsBuilder.uriVariables(uriVariables);
    String urlTemplate = uriComponentsBuilder1.toUriString();
    Object[] uriVariables1 = {};
    MockHttpServletRequestBuilder mockHttpServletRequestBuilder = get(urlTemplate, uriVariables1);

    ResultActions actual = mockMvc.perform(mockHttpServletRequestBuilder);

    actual.andDo(print());
    actual.andExpect((status()).is(200));
    actual.andExpect((content()).string("[{\"id\":1,\"name\":\"abcd\",\"owner\":\"user\"}]"));
}
```

Рис. 6: Тест, сгенерированный UnitTestBot для метода с аннотацией `@PostAuthorize("returnObject.owner == authentication.principal.username")`. Аквамариновым выделено то, что было добавлено вручную для корректности теста.

`@PreFilter` и `@PostFilter` работают и тестируются по такому же принципу. Их различие между вышеупомянутыми аннотациями лишь в том, что они фильтруют некую коллекцию, а не один объект.

3.3.1. `@Query`

Так как для базы данных `@PostFilter` является довольным медленным за счёт фильтрации, грубо говоря, всей базы, часто используется `@Query`. Здесь также нужно спарсить выражение внутрии аннотации, однако оно не является SpEL выражением.

```
public interface ProductRepository extends JpaRepository<Product, Integer> {
    @Query("SELECT p FROM Product p WHERE p.name LIKE %:text% AND p.owner=?#{authentication.principal.username}")
    List<Product> findProductByNameContains(String text);
}
```

Рис. 7: Пример запроса к базе данных, используя элементы Spring Security.

3.3.2. `@WithUserDetails`, `@WithSecurityContext`

Аннотация `@WithUserDetails` позволяет настроить контекст с объектом `UserDetails`, возвращаемым из указанного `UserDetailsService`. Этот подход, в отличие от `@WithMockUser`, позволяет присвоить mock-пользователю атрибуты в том виде, в котором они существуют в системе управления пользователями приложения. С одной стороны, это увеличит покрытие за счет тестирования случаев, где метод зависит от атрибутов `UserDetails`, а также за счет взаимодействия с базой данных. С другой стороны, автогенерация таких тестов представляет некоторую сложность и некий «оверхед», поскольку для каждого возможного случая надо сохранять пользователя в базу данных и «вытаскивать» его при запуске каждого теста.

Альтернативой этому может служить аннотация `@WithSecurityContext`, с помощью которой можно создать любой `SecurityContext`, тем самым не прибегая к излишним запросам к базе данных. Например, аннотация `@WithMockAuthentication` из spring-addons позволяет нам сымитировать `UserDetails` (рис. 8).

```
@WithMockAuthentication(name = "demoUser", principalType = DemoUserDetails.class)
public void test() {
    assertEquals("demoUser", userDetails.getUsername());
}
```

Рис. 8: Пример использования `@WithMockAuthentication`.

3.3.3. Делегирование авторизации `@Bean`

Авторизация может выглядеть иным образом (рис. 9).

```
@Component("authz")
public class AuthorizationLogic {
    public boolean decide(MethodSecurityExpressionOperations operations) {
        // ... authorization logic
    }
}

//...

@PreAuthorize("@authz.decide(#root)")
@GetMapping("/endpoint")
public String endpoint()
```

Рис. 9: Использование отдельного класса для авторизационной логики [14].

Для тестирования такого случая авторизации надо понимать, как работает используемый `@Bean`. От него зависит сможем ли мы вызвать метод, так как авторизационная логика делегируется ему. Поэтому для построения mock-пользователя нам надо знать всевозможные «пути» `authz.decide`. Конечно, можно тестировать эту логику отдельно, но тогда генерируемые тесты на обезопасенные методы не будут являться корректными.

3.3.4. Пользовательский `PermissionEvaluator`

Данный класс позволяет пользователю настроить доступ к доменному объекту, например, разделяя роль чтения и записи к документу

(рис 10).

```
@PreAuthorize("hasPermission(#spreadsheet, 'READ')")
    public void read(Spreadsheet spreadsheet) {
    }

@PreAuthorize("hasPermission(#id, 'com.test.model.Spreadsheet', 'READ')")
    public void readById(Long id) {
    }
```

Рис. 10: Два семантически одинаковых метода, первому на вход дается сам объект, а второму — `id` объекта.

Для имплементации автогенерации тестов для таких аннотаций препятствуют две проблемы:

1. Для первой аннотации надо вычленять тип доменного объекта, так как метод интерфейса не обязует указывать его явно.

2. Mock-пользователя нужно наделить теми правами, которыми мы хотим протестировать метод. Эти права храняться чаще всего в кэше, являющимя приватным полем класса `PermissionEvaluator`.

Для решения первой проблемы можно посмотреть на тип аргумента функции, однако для «post-» аннотаций это способ не подойдет. В итоге реализация сводится к парсингу или символьному исполнению метода `hasPermission`, в котором должна быть проверка вида `if targetDomainObject instanceof SomeClass`, откуда можно взять класс доменного объекта.

Решение второй проблемы зависит от типа хранилища. Например, для базы данных можно использовать `EntityManager`. Сам `permission` всегда можно получить из аннотации.

3.4. Безопасность на уровне `SecurityFilterChain`

Помимо авторизации на уровне методов, возможно также авторизовывать непосредственно в самой конфигурации, то есть настроить `SecurityFilterChain` (рис. 11).

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/api/**")
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers(HttpMethod.GET).hasAuthority("read")
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(withDefaults());
    return http.build();
}
```

Рис. 11: Типовой пример `SecurityFilterChain`.

В данном примере показаны ключевые настройки:

- `securityMatcher("/api/**")` — применение данной цепочки фильтров только для конечных точек вида `/api/**`.
- `requestMatchers(HttpMethod.GET).hasAuthority("read")` — нужно право «read» для любого GET запроса.
- `requestMatchers("/admin/**").hasRole("ADMIN")` — нужна роль `ADMIN`; семантически тоже самое, что и `hasAuthority("ROLE_ADMIN")`.
- `requestMatchers("/public/**").permitAll` — не нужна аутентификация.

Как можно видеть, тестирование авторизации, прописанной в `SecurityFilterChain`, сводится к другим описанным методам.

Тем не менее, в `SecurityFilterChain` прописаны не только правила авторизации, но и другие важные для тестирования вещи. Например, может быть явно отключена защита от CSRF и само приложение настроено в роли ресурсного сервиса по стандарту OAuth2 (рис. 12).

```
.csrf(csrf -> csrf.disable())
.oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))
```

Рис. 12: Настройка resource server с использованием JWT в `SecurityFilterChain` и отключенным `CsrfFilter`.

Извлечь полезные свойства и ограничения можно, спарсив `SecurityFilterChain`. Также во время выполнения можно «достать» этот `@Bean` из контекста Spring и рассмотреть фильтры, которые он в себе содержит. Например, наличие `BearerTokenAuthenticationFilter` означает, что приложение настроено в роли ресурсного сервиса. Это изменит подход к тестированию, об этом речь пойдет ниже.

Для просмотра авторизационный логики можно «вытащить» из `AuthorizationFilter` список инстанцированных `RequestMatcherEntry`, откуда потом можно получить паттерн конечных точек и необходимое условие.

Стоит также обратить внимание на то, что в конфигурации может быть несколько `SecurityFilterChain`. Порядок их применения зависит от порядка, заданного `@Order`, а что именно применится — зависит от `securityMatcher`.

3.5. OAuth2/OIDC

Тестирование OAuth2/OIDC в чём-то является производной от вышеперечисленных методик. Тем не менее, у этой технологии есть свои особенности, например JWT[4].

Для тестирования в случае роли приложения как resource server предлагается использовать `@WithJwt` из библиотеки spring-addons. Данная аннотация создает из предоставленного JSON контекст с `Authentication`, инстанцированного `JwtAuthenticationConverter` (обычно `JwtAuthenticationToken`). Это полезно по нескольким причинам:

[4]JSON Web Token

- Косвенно тестируется `JwtAuthenticationConverter`, отвечающий за конвертацию JWT в `Authentication`.
- Иногда логика приложения может зависить от заявок (англ. claims) токена, что нельзя сымитировать обычным `@WithMockUser` (рис. 13).
- Сгенерированные тесты более читаемы.

```
@PreAuthorize("authentication.principal.claims['name'] == 'io'")
public String demo()

//...

@Test
@WithJwt(json = """
      {"name": "io"}
    """)
void testDemo() throws Exception
```

Рис. 13: Пример использования аннотации `@WithJwt`.

Все вышеперечисленное также относится и к opaque token — другой стандарт OAuth2/OIDC (отличие лишь в синтаксисе, например используется аннотация `@WithOpaqueToken`).

Для тестирования в случае роли приложения как client предлагается использовать `@WithOidcLogin` или `@WithOAuth2Login`. Это также можно вычленить из конфигурации путем нахождения фильтра `OAuth2LoginAuthenticationFilter`.

3.6. Ошибки, выявленные в ходе работы

Было выявлено четыре ошибки в генерированных тестах UnitTestBot:

1. Сгенерированный тест не смог пройти файрвол Spring Security.[5]

[5]https://github.com/UnitTestBot/UTBotJava/issues/2726

2. Сгенерировано огромное количество одинаковых несодержательных тестов для метода с аннотацией `@PreAuthorize`.[6]

3. Сгенерировано два одинаковых теста для метода с аннотацией `@PreAuthorize`.[7]

4. Сгенерирован несодержательный тест для POST-запроса при добавленном Spring Security.[8]

Также была выявлена ошибка при использовании `@WithJwt` без пользовательского `JwtAuthenticationConverter`, так как по умолчанию этот класс не отображается как `@Bean`.[9]

[6]https://github.com/UnitTestBot/UTBotJava/issues/2727
[7]https://github.com/UnitTestBot/UTBotJava/issues/2728
[8]https://github.com/UnitTestBot/UTBotJava/issues/2729
[9]https://github.com/ch4mpy/spring-addons/discussions/158

Заключение

1. Проведен обзор существующих аналогов автоматических генераторов тестов на предмет генерации тестов для Spring Security. На данный момент не существует автогенераторов тестов, тестирующих аутентификацию и авторизацию

2. Проанализированы методики ручного создания тестовых сценариев.

3. Проведено сравнение тестов генерируемых UnitTestBot с предложенными в этой работе. Так, например, первым шагом к увеличению поддержки Spring Security может стать имплементация `@WithAnonymousUser`.

4. Предложены возможные способы реализации частичной поддержки Spring Security.

В весеннем семестре планируется имплементировать в UnitTestBot некоторые из описанных методик тестирования.

Список литературы

[1] Abdullin A., Akhin M., Belyaev M. Kex at the 2022 SBST Tool Competition // 2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST). — Los Alamitos, CA, USA : IEEE Computer Society, 2022. — may. — P. 35–36. — URL: https://doi.ieeecomputersociety.org/10.1145/3526072.3527527 (дата обращения: 19 декабря 2023 г.).

[2] Diffblue Cover. — URL: https://www.diffblue.com/products/ (дата обращения: 6 декабря 2023 г.).

[3] Fraser Gordon, Arcuri Andrea. EvoSuite: Automatic test suite generation for object-oriented software. — 2011. — 09. — P. 416–419.

[4] JUnit 5. — URL: https://junit.org/junit5/ (дата обращения: 7 декабря 2023 г.).

[5] Parasoft Jtest. — URL: https://www.parasoft.com/products/parasoft-jtest/ (дата обращения: 6 декабря 2023 г.).

[6] Spilca Laurentiu. Spring Security in Action. — Manning, 2020. — ISBN: 9781617297731.

[7] Spring Framework. — URL: https://spring.io/ (дата обращения: 6 декабря 2023 г.).

[8] Spring Security Overview. — URL: https://spring.io/projects/spring-security#overview (дата обращения: 6 декабря 2023 г.).

[9] Stack Overflow Developer Survey 2023. — URL: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-frameworks-and-librarie (дата обращения: 6 декабря 2023 г.).

[10] Symflower. — URL: https://symflower.com/en/ (дата обращения: 6 декабря 2023 г.).

[11] Testing Spring OAuth2 Access-Control. — URL: https://www.baeldung.com/spring-oauth-testing-access-control (дата обращения: 16 декабря 2023 г.).

[12] Testing Spring Security. — URL: https://docs.spring.io/spring-security/reference/servlet/test/index.html (дата обращения: 16 декабря 2023 г.).

[13] UnitTestBot Java. — URL: https://github.com/UnitTestBot/UTBotJava.git (дата обращения: 6 декабря 2023 г.).

[14] Using a Custom Bean in SpEL. — URL: https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html#_using_a_custom_bean_in_spel (дата обращения: 15 декабря 2023 г.).

[15] Wacongne Jérôme. Ease spring OAuth2 resource-servers configuration and testing. — URL: https://github.com/ch4mpy/spring-addons/ (дата обращения: 7 декабря 2023 г.).

[16] te Beek Tim. Spring Security Samples. — URL: https://github.com/JDriven/spring-security-samples/ (дата обращения: 16 декабря 2023 г.).