Разработка серверного ПО. Лекция 8. Введение в Spring Security

Spring Security является основным инструментом реализации безопасности на уровне приложений в Spring. Как правило, его цель – предложить вам настраиваемый способ реализации аутентификации, авторизации и защиты от распространенных атак. Spring Security – это программное обеспечение с открытым исходным кодом, выпущенное по лицензии Apache 2.0.

Вы можете удобно использовать аннотации, бины и в целом стиль конфигурации в стиле Spring при определении безопасности на уровне приложения. В приложении Spring поведение, которое вам нужно защитить, определяется методами.

- Spring Security – приоритетный способ защиты приложений Spring. Он очень гибок и может применяться к разным стилям и архитектурам.
- Безопасность системы должна быть организована послойно, и для каждого слоя необходимо применять разные технологии защиты.
- Безопасность – это сквозная задача, которую следует начинать решать с первых шагов и на всем протяжении проекта.

Наше первое приложение использует механизм HTTP Basic для аутентификации и авторизации пользователя при доступе к конечной точке. Оно предлагает конечную точку REST по указанному маршруту (/hello). При успешном запросе оно выдает сообщение о состоянии HTTP 200 вместе с телом ответа. Этот экземпляр иллюстрирует механизмы аутентификации и авторизации по умолчанию, добавленные с помощью Spring Security.

Единственные зависимости, которые нужно написать для первого проекта, — это spring-boot-starter-web и spring-boot-starter-security.

Если попробовать запросить эндпоинт /hello, то вернётся ответ 401 Unauthorized.

Если ввести логин и пароль, который сгенерировало приложение при запуске:

```
curl -u user:e8758d50-8663-4851-97a6-f48634374387 http://localhost:8080/hello
```

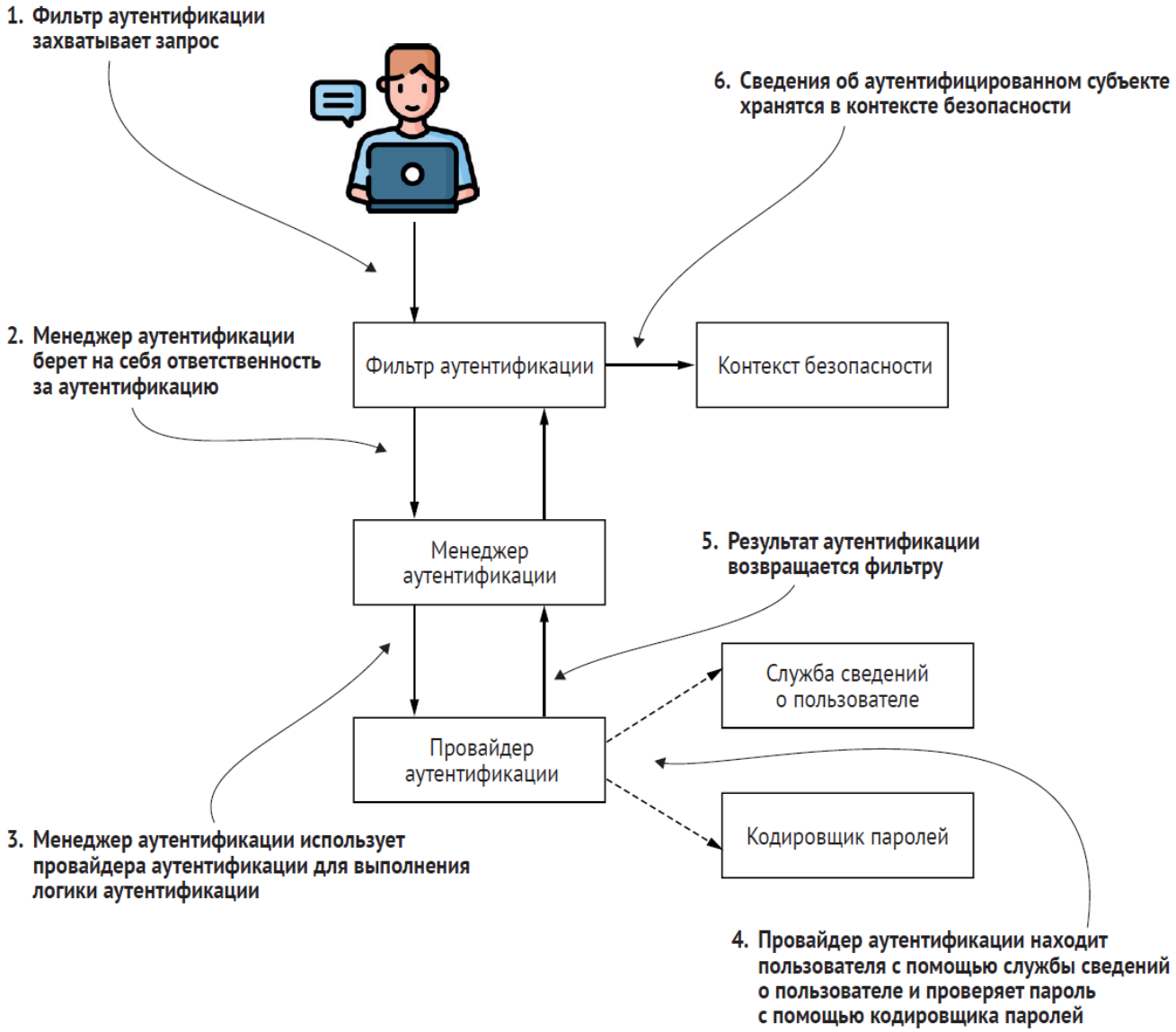
то в ответ получим строку `hello`.

Код состояния HTTP 401 Unauthorized не- много неоднозначен. Обычно он используется для обозначения неудачной аутентификации, а не авторизации. Разработчики используют его при проектировании приложения для таких случаев, как отсутствие или неверные учетные данные. Обычно HTTP 403 означает, что сервер идентифицировал вызывающую сторону запроса, но у нее нет необходимых привилегий для вызова, который она пытается сделать.

Благодаря этому примеру мы по крайней мере знаем, что Spring Security работает. Следующий шаг – изменить конфигурации так, чтобы они соответствовали требованиям нашего проекта.

Взаимосвязи между сущностями, которые являются частью представлен- ного потока аутентификации:

1. фильтр аутентификации (authentication filter) делегирует запрос менеджеру аутентификации и на основе ответа настраивает контекст безопасности;
2. менеджер аутентификации (authentication manager) использует провайдера аутентификации для обработки аутентификации;
3. провайдер аутентификации (authentication provider) реализует логику аутентификации;
4. служба сведений о пользователе (user details service) реализует ответственность за управление пользователями, которую провайдер аутентификации использует в логике аутентификации;
5. кодировщик паролей (password encoder) реализует управление паролями, которое провайдер аутентификации использует в логике аутентификации;
6. контекст безопасности (security context) сохраняет данные аутентификации после процесса аутентификации. Контекст безопасности будет хранить данные до тех пор, пока действие не завершится. Обычно в приложении типа «один поток на запрос» это означает, что приложение отправит ответ обратно клиенту.



Объект, реализующий интерфейс UserDetailsService с Spring Security, управляет сведениями о пользователях. До сих пор мы использовали реализацию по умолчанию, предоставляемую Spring Boot. Она регистрирует учетные данные по умолчанию только во внутренней памяти приложения.

PasswordEncoder делает две вещи:

- кодирует пароль (обычно с использованием алгоритма шифрования или хеширования);
- проверяет, совпадает ли пароль с существующим закодированным значением.

Для аутентификации типа Basic клиенту достаточно отправить имя пользователя и пароль через заголовок HTTP Authorization. В значении заголовка клиент добавляет префикс Basic, за которым следует строка, содержащая имя пользователя и пароль, разделенные двоеточием (😃 в кодировке Base64.

AuthenticationProvider определяет логику аутентификации, делегируя управление пользователями и паролями. Реализация AuthenticationProvider по умолчанию использует реализации по умолчанию, предоставленные для UserDetailsService и PasswordEncoder.

Пример ex1

Переопределение конфигураций по умолчанию. Настройка управления данными пользователя

Пример ex2

Переопределение конфигураций по умолчанию. Применение авторизации на уровне конечной точки

Чтобы настроить обработку аутентификации и авторизации, нужно определить bean-компонент типа SecurityFilterChain.

Пример ex3

В этом примере мы использовали два метода конфигурации:

- httpBasic() помог нам настроить способ аутентификации. Вызвав этот метод, вы указали приложению принять HTTP Basic в качестве способа аутентификации;
- authorizeHttpRequests() помог нам настроить правила авторизации на уровне конечной точки. Вызвав этот метод, указали приложению, как авторизовать запросы, полученные на определенных конечных точках.

Для обоих методов вам нужно было использовать объект Customizer в качестве параметра. Customizer – это контракт, который вы реализуете для определения настройки любого элемента Spring Security: аутентификация, авторизация или определенные механизмы защиты, такие как CSRF или CORS.

Переопределение конфигураций по умолчанию. Определение пользовательской логики аутентификации

AuthenticationProvider реализует логику аутентификации и делегирует UserDetailsService и PasswordEncoder для управления пользователями и паролями.

Пример ex4

Метод authenticate(Authentication authentication) представляет всю логику аутентификации.

Можно зарегистрировать AuthenticationProvider в классе конфигурации с помощью метода HttpSecurity authenticationProvider().

Переопределение конфигураций по умолчанию. Использование нескольких классов конфигурации

В предыдущих примерах мы использовали только класс конфигурации. Но хорошей практикой является разделение ответственностей даже для классов конфигурации. Хорошей практикой является наличие только одного класса для каждой ответственности.

Пример ex5

- Spring Boot предоставляет ряд конфигураций по умолчанию при добавлении Spring Security к зависимостям приложения.

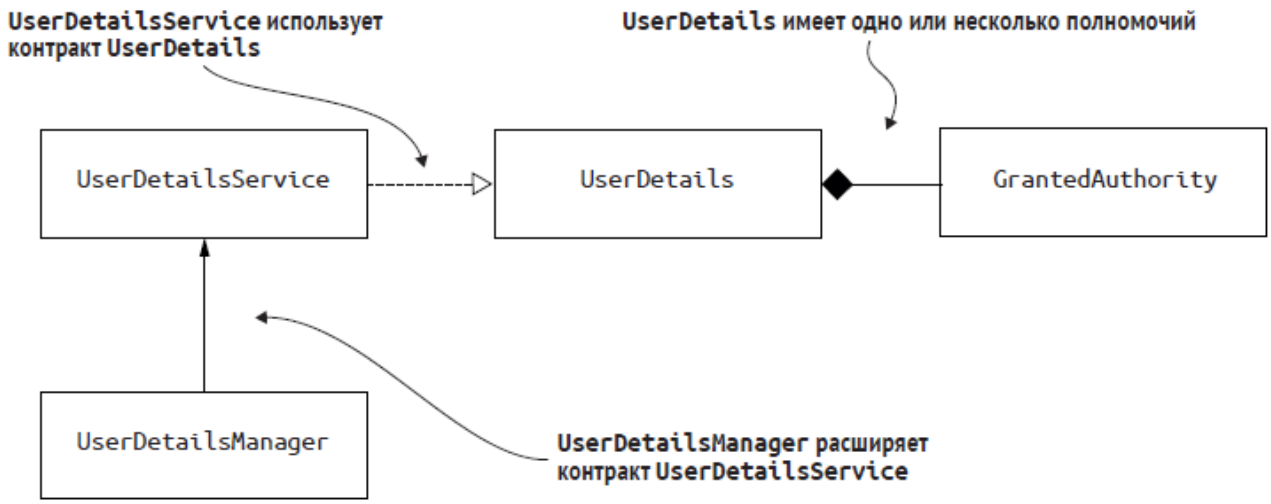
- Вы реализуете следующие базовые компоненты для аутентификации и авторизации: UserDetailsService, PasswordEncoder и AuthenticationProvider.
- Вы можете определить пользователей с помощью класса User. У пользователя должны быть как минимум имя пользователя, пароль и полномочия. Полномочия – это действия, которые вы разрешаете пользователю выполнять в контексте приложения.
- Простая реализация UserDetailsService, которую предоставляет Spring Security, – это InMemoryUserDetailsManager. Вы можете добавлять пользователей в такой экземпляр UserDetailsService для управления пользователем в памяти приложения.
- NoOpPasswordEncoder – это реализация контракта PasswordEncoder, который использует пароли в открытом тексте. Эта реализация хороша для изучения примеров и (возможно) проверки работоспособности идей, но не для приложений, готовых к производству.
- Вы можете использовать контракт AuthenticationProvider для реализации пользовательской логики аутентификации в приложении.
- Существует несколько способов написания конфигураций, но в рамках приложения следует придерживаться одного выбранного подхода. Это помогает сделать ваш код чище и проще для понимания.

Настройка аутентификации. Управление пользователями

- UserDetails – описывает пользователя для Spring Security;
- GrantedAuthority – позволяет нам определять действия, которые может выполнять пользователь;
- UserDetailsManager – расширяет контракт UserDetailsService. Помимо унаследованного поведения, он также описывает такие действия, как создание пользователя и изменение или удаление пароля пользователя.

Реализация аутентификации в Spring Security

AuthenticationFilter захватывает входящий запрос и передает задачу аутентификации компоненту AuthenticationManager. Тот, в свою очередь, использует провайдер аутентификации для выполнения процесса аутентификации. Для проверки имени пользователя и пароля AuthenticationProvider задействует UserDetailsService и PasswordEncoder.



UserDetailsService извлекает данные пользователя, выполняя поиск пользователя по имени. Пользователь характеризуется контрактом UserDetails. Каждый пользователь обладает одним или несколькими полномочиями, которые отображаются интерфейсом GrantedAuthority. Для добавления таких операций, как создание, удаление или изменение пароля пользователя, применяется контракт UserDetailsManager, который расширяет интерфейс UserDetailsService нужной функциональностью

Описание пользователей с помощью контракта UserDetails

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetails.html

Методы getUsername() и getPassword() возвращают, как и ожидалось, имя пользователя и пароль. Приложение использует эти значения в процессе аутентификации, и это единственные сведения, связанные с аутентификацией из этого контракта. Все остальные пять методов относятся к авторизации пользователя для доступа к ресурсам приложения. Как правило, приложение позволяет пользователю выполнять некоторые действия, которые имеют смысл в контексте приложения. Например, пользователь должен иметь возможность читать, записывать или удалять данные. Говорят, что у пользователя имеются или отсутствуют привилегии на выполнение действия, а полномочие отражает привилегию, которую имеет пользователь. Мы реализуем метод getAuthorities() для получения ссылки на группу полномочий, предоставленных пользователю.

Кроме того, как видно из контракта UserDetails, с пользователем могут произойти следующие события:

- его учетная запись может быть просрочена;
- его учетная запись может быть заблокирована;
- его учетные данные могут быть просрочены;
- его учетная запись может быть отключена.

Детализация контракта GrantedAuthority

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/GrantedAuthority.html

Для описания полномочий в Spring Security используют интерфейс GrantedAuthority.

Альтернативный способ – использовать класс SimpleGrantedAuthority для создания экземпляров полномочий. Этот класс предлагает способ создания неизменяемых экземпляров типа GrantedAuthority. Вы указываете имя полномочия при создании экземпляра.

```
GrantedAuthority g1 = () -> "READ";
GrantedAuthority g2 = new SimpleGrantedAuthority("READ");
```

Реализация UserDetails. Реализация контракта UserDetailsService

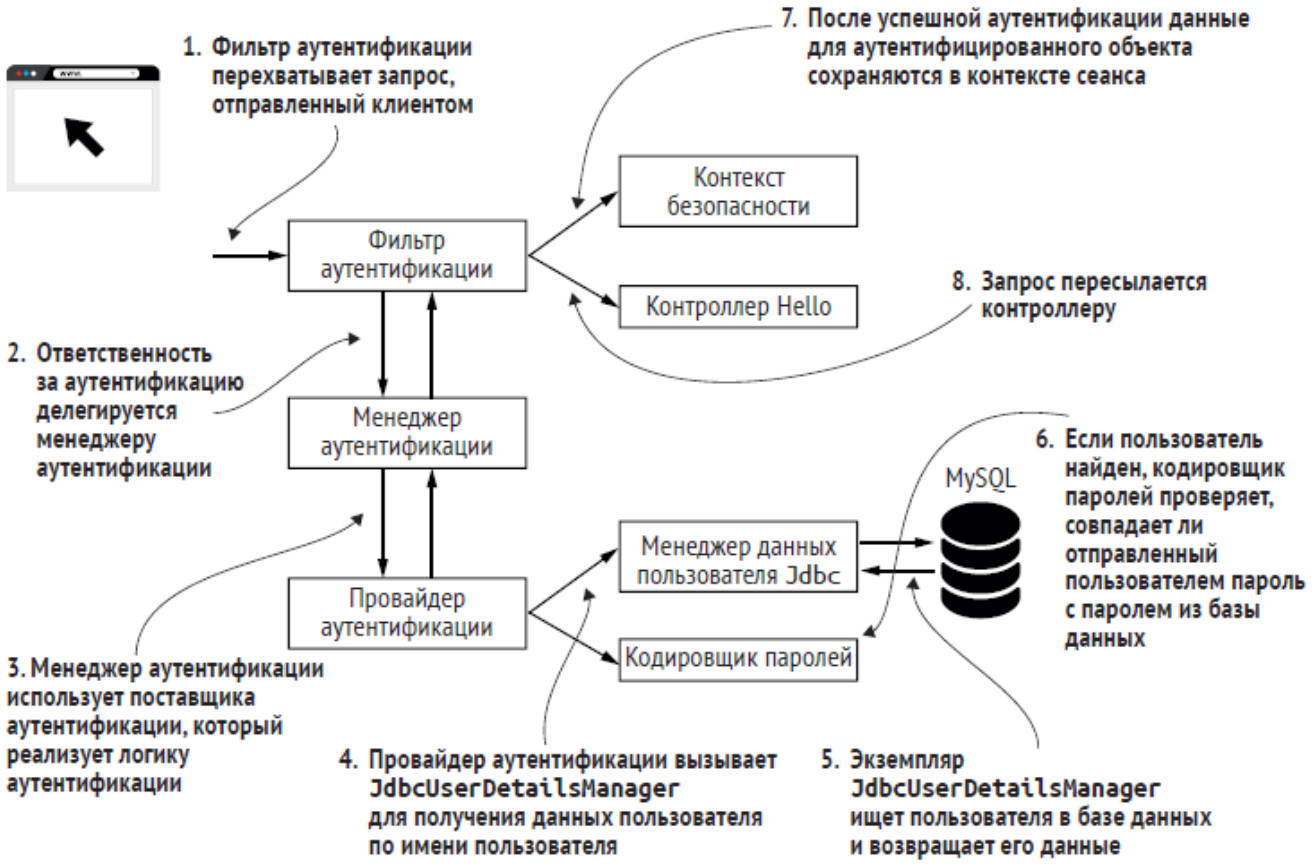
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetailsService.html

Пример ex6

Реализация аутентификации вызывает метод loadUserByUsername(String username) для получения сведений о пользователе с заданным именем. Имя пользователя, конечно же, считается уникальным. Пользователь, возвращаемый этим методом, является реализацией контракта UserDetails. Если имя пользователя не существует, метод выдает исключение UsernameNotFoundException.

Реализация контракта UserDetailsManager. Использование JdbcuserdetailsManager для управления пользователями

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/provisioning/UserDetailsManager.html



JdbcUserDetailsManager управляет пользователями в базе данных SQL. Он подключается к базе данных напрямую через JDBC. Таким образом, JdbcUserDetailsManager не зависит от какой-либо другой структуры или спецификации, связанной с подключением к базе данных.

Пример ex7

- Интерфейс UserDetails – это контракт, который вы используете для описания пользователя в Spring Security.
- Интерфейс UserDetailsService – это контракт, который вы должны реализовать в архитектуре аутентификации для описания способа, которым приложение получает данные пользователя.
- Интерфейс UserDetailsManager расширяет UserDetailsService и добавляет поведение, связанное с созданием, изменением или удалением пользователя.
- Spring Security предоставляет несколько реализаций контракта UserDetailsManager. Среди них InMemoryUserDetailsManager, JdbcUserDetailsManager и LdapUserDetailsManager.
- Преимущества класса JdbcUserDetailsManager – в прямом использовании JDBC и отсутствии привязки приложения к другим фреймворкам.

Управление паролями. Использование кодировщиков паролей

AuthenticationProvider использует PasswordEncoder для проверки пароля пользователя в процессе аутентификации.

Поскольку система не должна оперировать паролями в виде простого текста, они обычно подвергаются своего рода преобразованию, что затрудняет их чтение и кражу. Для этой задачи Spring Security определяет отдельный контракт.

Контракт PasswordEncoder

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/PasswordEncoder.html

PasswordEncoder. Он нужен, чтобы сообщить Spring Security, как проверять пароль пользователя. В процессе аутентификации PasswordEncoder решает, является ли пароль действительным. Каждая система хранит пароли в закодированной форме, обычно в виде хешей, чтобы никто не смог их прочитать. PasswordEncoder тоже может кодировать пароли. Методы encode() и matches(), которые объявляются в контракте, на самом деле являются определением его ответственности. Оба они являются частями одного и того же контракта, поскольку они тесно взаимосвязаны. Способ, которым приложение кодирует пароль, связан со способом проверки пароля.

Цель метода encode(CharSequence rawPassword) – вернуть результат преобразования заданной строки. С точки зрения функциональности Spring Security он используется для шифрования или хеширования заданного пароля. Впоследствии можно вызвать метод matches(Char Sequence rawPassword, String encodedPassword), чтобы проверить, соответствует ли закодированная строка исходному паролю. Вы будете использовать метод matches() в процессе аутентификации для проверки, соответствует ли предоставленный пароль одному из элементов набора известных учетных данных. Третий метод, называемый upgradeEncoding(CharSequence encodedPassword), по умолчанию имеет в контракте значение false. Если вы переопределяете его для возврата true, то закодированный пароль повторно кодируется для лучшей безопасности.

Реализация собственного класса PasswordEncoder

Самая простая реализация – это ко- дировщик паролей, который возвращает пароли в виде обычного текста: то есть он не выполняет никакого кодирования пароля. Операции с паролями в виде открытого текста – именно этим и занимается экземпляр NoOpPasswordEncoder.

```
public class Sha512PasswordEncoder
implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        return hashWithSHA512(rawPassword.toString());
    }

    @Override
    public boolean matches(
    CharSequence rawPassword, String encodedPassword) {
        String hashedPassword = encode(rawPassword);
```

```
        return encodedPassword.equals(hashedPassword);
    }

    private String hashWithSHA512(String input) {
        StringBuilder result = new StringBuilder();

        try {
            MessageDigest md = MessageDigest.getInstance("SHA-512");
            byte [] digested = md.digest(input.getBytes());

            for (int i = 0; i < digested.length; i++) {
                result.append(Integer.toHexString(0xFF & digested[i]));
            }
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("Bad algorithm");
        }

        return result.toString();
    }
}
```

Выбор готовых реализаций PasswordEncoder

- NoOpPasswordEncoder – не кодирует пароль, а сохраняет его в открытом виде. Мы используем эту реализацию только для примеров. Поскольку метод не хеширует пароль, вам никогда не следует использовать его в реальном сценарии;
- StandardPasswordEncoder – использует SHA-256 для хеширования пароля. Эта реализация устарела, и вам не следует использовать ее для новых проектов. Дело в том, что данный метод использует алгоритм хеширования, который больше не считается достаточно стойким, но вы можете встретить его в существующих приложениях. При обновлении существующих приложений желательно заменить этот метод на какой-то другой, более мощный кодировщик паролей;
- Pbkdf2PasswordEncoder – использует функцию деривации ключа на основе пароля 2 (PBKDF2);
- BCryptPasswordEncoder – использует функцию хеширования bcrypt для кодирования пароля;
- SCryptPasswordEncoder – использует функцию хеширования scrypt для кодирования пароля.

DelegatingPasswordEncoder

DelegatingPasswordEncoder – это реализация интерфейса PasswordEncoder, которая вместо своего алгоритма кодирования делегирует другому экземпляру реализации того же контракта. Хеш начинается с префикса, именующего алгоритм, используемый для определения этого хеша. DelegatingPasswordEncoder выполняет делегирование правильной реализации PasswordEncoder на основе префикса пароля.

```
@Configuration
public class ProjectConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
```

```
        Map<String, PasswordEncoder> encoders = new HashMap<>();

        encoders.put("noop", NoOpPasswordEncoder.getInstance());
        encoders.put("bcrypt", new BCryptPasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());

        return new DelegatingPasswordEncoder("bcrypt", encoders);
    }
}
```

Использование модуля Spring Security Crypto

две основных функций из SSCM:

- генераторы ключей – объекты, используемые для генерации ключей для алгоритмов хеширования и шифрования;
- шифровальщики – объекты, используемые для шифрования и дешифрования данных.

Генераторы ключей

Два интерфейса представляют два основных типа генераторов ключей: BytesKeyGenerator и StringKeyGenerator. Мы можем постро- ить их напрямую, используя класс-фабрику KeyGenerators. Существует генератор строковых ключей, представленный контрактом StringKeyGenerator, для получения ключа в виде строки. Обычно этот ключ используют как значение соли для алгоритма хеширования или шифрования.

```
StringKeyGenerator keyGenerator = KeyGenerators.string();
String salt = keyGenerator.generateKey();
```

Генератор создает 8-байтовый ключ и кодирует его как шестнад- цатеричную строку. Метод возвращает результат этих операций как строку.

```
BytesKeyGenerator keyGenerator = KeyGenerators.secureRandom();
byte [] key = keyGenerator.generateKey();
int keyLength = keyGenerator.getKeyLength();
```

Генератор ключей генерирует ключи длиной 8 байт. Изменить можно передав нужную длину в метод `secureRandom`.

Ключи, сгенерированные BytesKeyGenerator, созданным с помощью метода KeyGenerators.secureRandom(), уникальны для каждого вызова метода generateKey(). В некоторых случаях предпочтительна реализация, которая возвращает одно и то же значение ключа для каждого вызова одного и того же генератора ключей. В этом случае мы можем создать BytesKeyGenerator с помощью метода KeyGenerators.shared(int length).

```
BytesKeyGenerator keyGenerator = KeyGenerators.shared(16);
byte [] key1 = keyGenerator.generateKey();
byte [] key2 = keyGenerator.generateKey();
```

Шифрование и дешифровка секретов с помощью шифраторов

Шифратор (encryptor) – это объект, который реализует алгоритм шифрования.

Существует два типа шифраторов, определенных SSCM: BytesEncryptor и TextEncryptor. Хотя у них схожие обязанности, они обрабатывают разные типы данных.

Фабричный класс Encryptors предлагает несколько вариантов создания шифраторов. Для BytesEncryptor мы могли бы использовать методы Encryptors.standard() или Encryptors.stronger()

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";
BytesEncryptor e = Encryptors.standard(password, salt);
byte [] encrypted = e.encrypt(valueToEncrypt.getBytes());
byte [] decrypted = e.decrypt(encrypted);
```

стандартный шифратор байтов использует 256-бай- товый алгоритм AES для шифрования входных данных. Для созда- ния более сильного экземпляра шифратора байтов нужно вызвать метод Encryptors.stronger().

```
BytesEncryptor e = Encryptors.stronger(password, salt);
```

TextEncryptors бывают трех основных типов. Их создают, вызывая Encryptors.text() или Encryptors.delux(). Помимо этих методов для создания шифраторов, есть также метод, который возвращает фиктивный TextEncryptor, который не шифрует значение Encryptors.noOpText().

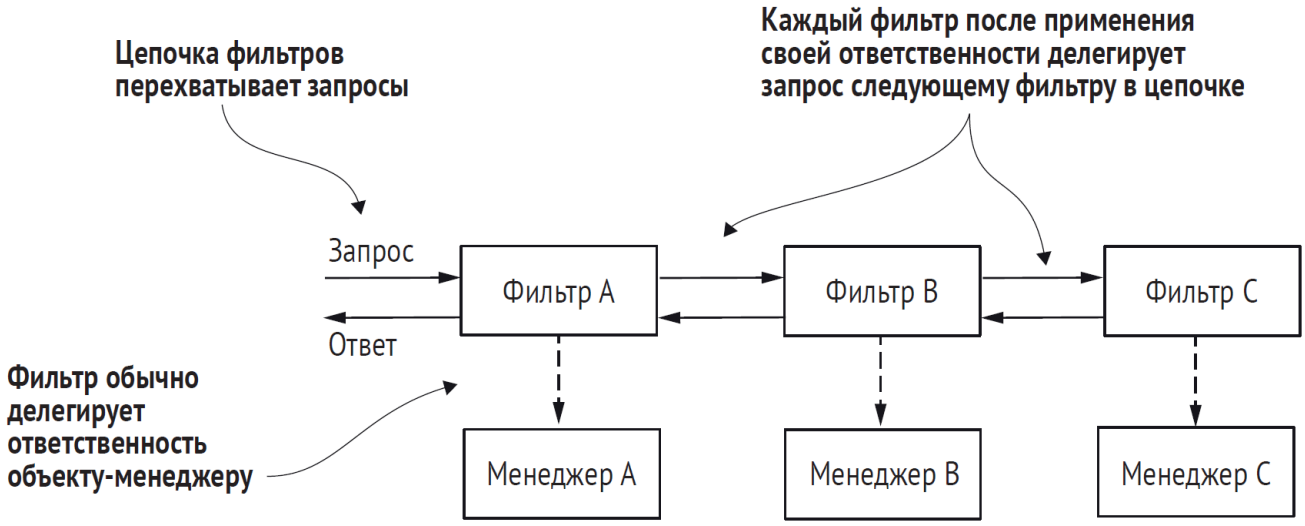
```
String valueToEncrypt = "HELLO";
TextEncryptor e = Encryptors.noOpText();
String encrypted = e.encrypt(valueToEncrypt);
```

Шифратор Encryptors.text() использует метод Encryptors.standard() для управления операцией шифрования, в то время как ме- тод Encryptors.delux() использует экземпляр Encryptors.stronger()

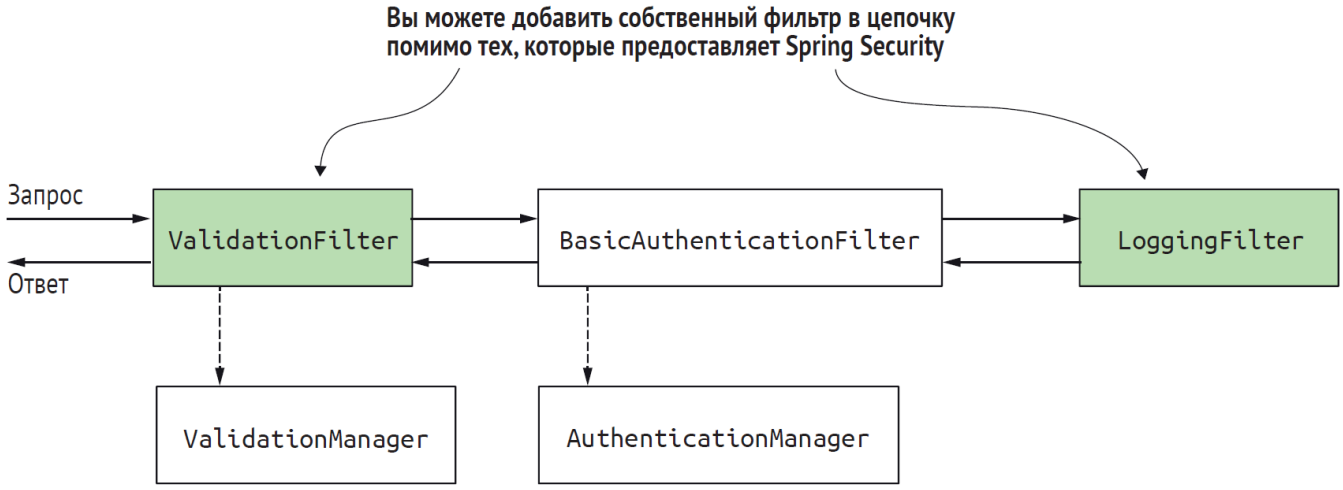
```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";
TextEncryptor e = Encryptors.text(password, salt);
```

```
String encrypted = e.encrypt(valueToEncrypt);
String decrypted = e.decrypt(encrypted);
```

Фильтры



Security предоставляет реализации фильтров, которые вы добавля- ете в цепочку посредством настройки, но вы также можете опреде- лить пользовательские фильтры.



Реализация фильтров в архитектуре Spring Security

Фильтр аутентификации перехватывает запрос и делегирует ответственность за аутентификацию менеджеру авторизации. Если мы хотим добавить определенную логику, выполняемую перед аутентификацией, мы вставляем дополнительный фильтр перед фильтром аутентификации.

Фильтры в архитектуре Spring Security – это типичные HTTP-фильтры. Мы создаем их, реализуя интерфейс Filter из пакета jakarta.servlet. Как и для любого другого HTTP-фильтра, вам необходимо переопределить метод doFilter() для реализации его логики. Этот метод получает параметры ServletRequest, ServletResponse и Filter Chain:

- ServletRequest – представляет HTTP-запрос. Мы используем объект ServletRequest для получения сведений о запросе;

- ServletResponse – представляет HTTP-ответ. Мы используем объект ServletResponse для изменения ответа перед его отправкой обратно клиенту или далее по цепочке фильтров;
- FilterChain – представляет цепочку фильтров. Мы используем объект FilterChain для пересылки запроса следующему фильтру в цепочке.

Цепочка фильтров представляет собой набор фильтров, действу- ющих в определенном порядке. Spring Security предлагает нам не- сколько готовых реализаций фильтров и их порядок. Готовые фильтры:

- BasicAuthenticationFilter обслуживает аутентификацию HTTP Basic, если она присутствует;
- CsrfFilter заботится о защите от подделки межсайтовых за- просов (CSRF);
- CorsFilter отвечает за правила авторизации совместного ис- пользования ресурсов между источниками (CORS).

Каждый фильтр имеет порядковый номер, который определяет порядок применения фильтров к запросу.

Документация по фильтрам:

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/web/server/SecurityWebFiltersOrder.html

Добавление фильтра перед существующим фильтром / после существующего фильтра в цепочке

1. Реализовать фильтр. Создайть класс RequestValidationFilter, который проверяет наличие необходимого заголовка в запросе.
2. Добавить фильтр в цепочку фильтров в классе конфигурации, используя компонент SecurityFilterChain.

Пример ex8

Добавление фильтра на место существующего

Предположим, что вместо потока аутентификации HTTP Basic вы хотите реализовать какой-то другой поток. Следовательно, вместо использования имени пользователя и пароля в качестве входных учетных данных, на основе которых приложение аутентифицирует пользователя, вам нужно применить другой подход. Вот несколько примеров сценариев, с которыми вы можете столкнуться на прак- тике:

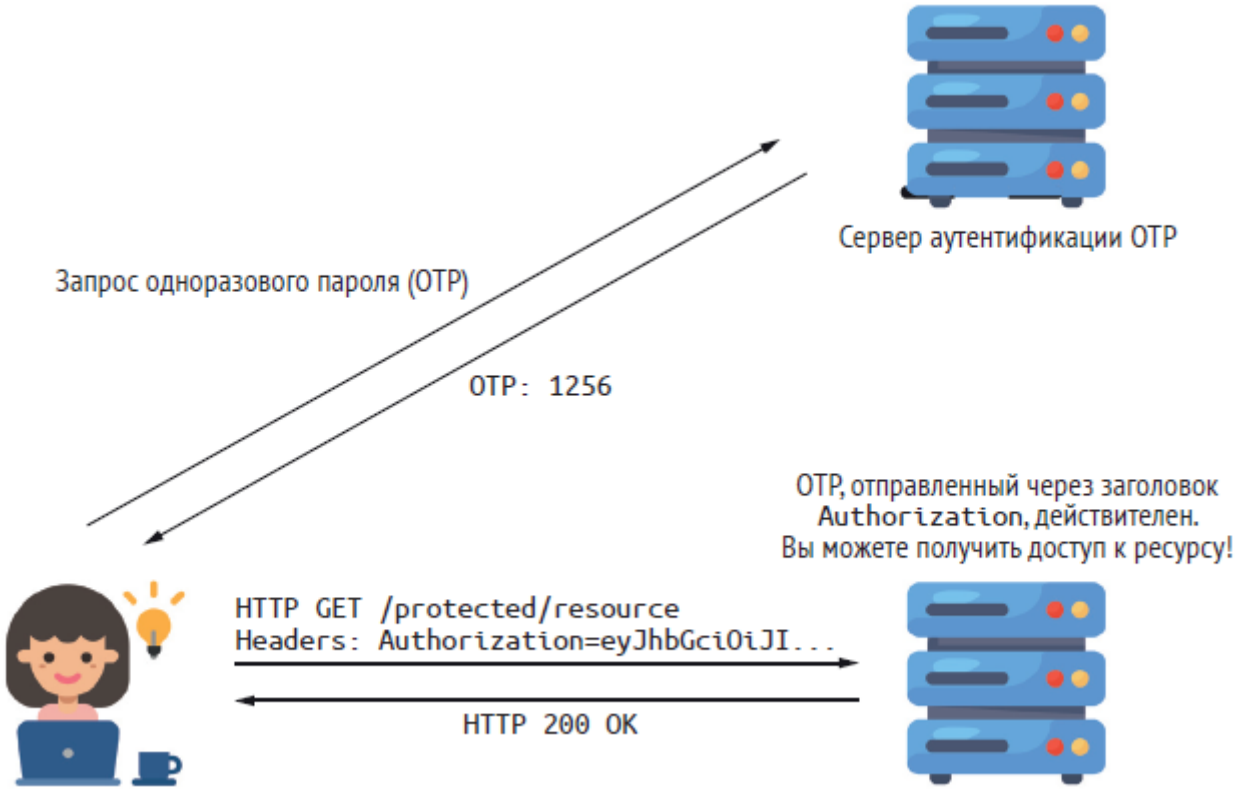
- идентификация на основе статического значения заголовка для аутентификации;
- использование симметричного ключа для подписи запроса на аутентификацию;
- использование одноразового пароля (one-time password, OTP) в процессе аутентификации.

В нашем первом сценарии (идентификация на основе статического ключа для аутентификации) клиент отправляет приложению строку в заголовке HTTP-запроса, который всегда одинаков. Приложение где-то хранит эти значения – скорее всего, в базе данных или хранилище секретов. На основе этого статического значения приложение идентифицирует клиента.

Этот подход обеспечивает слабую безопасность аутентификации, но архитекторы и разработчики часто выбирают его для аутентификации вызовов между бэкенд-приложениями из-за его простоты. У этого подхода высокое быстродействие, поскольку не нужно выполнять сложные вычисления, как в случае

применения криптографической подписи. Таким образом, статические ключи, используемые для аутентификации, представляют собой компромисс, при котором разработчики больше полагаются на инфраструктурные меры безопасности, но не оставляют конечные точки полностью незащищенными.

Пример ex9



Реализации фильтров, предоставляемые Spring Security

Пример ex10

Несколько коротких замечаний о классе OncePerRequestFilter, ко- торые могут оказаться полезными:

- он поддерживает только HTTP-запросы, но на самом деле мы всегда их используем. Преимущество в том, что он приводит типы, и мы напрямую получаем запросы как HttpServletRequest и HttpServletResponse. Напомню, что с интерфейсом Filter нам пришлось привести запрос и ответ;
- вы можете реализовать логику, чтобы решить, применяется ли фильтр. Даже если вы добавили фильтр в цепочку, вы можете решить, что он не будет применяться к определенным запросам. Вы добиваетесь этого, переопределяя метод shouldNotFilter(HttpServletRequest). По умолчанию фильтр применяется ко всем запросам;
- по умолчанию OncePerRequestFilter не применяется к асинхронным запросам или запросам на отправку ошибок. Вы можете изменить это поведение, переопределив методы shouldNotFilterAsyncDispatch() и shouldNotFilterErrorDispatch().

Реализация аутентификации

процесс аутентификации имеет только два возможных исхода:

- субъект, делающий запрос, не аутентифицирован. Пользователь не распознан, и приложение отклоняет запрос, не делегируя ответственность процессу авторизации. Обычно в этом случае клиенту возвращается статус ответа HTTP 401 Unauthorized;
- субъект, делающий запрос, аутентифицирован. Данные о запрашивающей стороне сохраняются, чтобы приложение могло использовать их для авторизации. Как вы узнаете в этой главе, SecurityContext отвечает за данные, касающиеся текущего аутентифицированного запроса.

Можно использовать контракт AuthenticationProvider для определения любой пользовательской логики аутентификации.

Представление запроса во время аутентификации

Authentication – один из основных интерфейсов, участвующих в процессе аутентификации. Интерфейс Authentication представляет событие запроса аутентификации и содержит сведения о субъекте, который запрашивает доступ к приложению.

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/Authentication.html

Пользователь, запрашивающий доступ к приложению, называется принципалом (principal).

Реализация пользовательской логики аутентификации

AuthenticationProvider в Spring Security отвечает за логику аутентификации. Реализация интерфейса AuthenticationProvider по умолчанию делегирует ответственность за поиск пользователя системы компоненту UserDetailsService. Она также использует PasswordEncoder для управления паролями в процессе аутентификации.

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/AuthenticationProvider.html

Ответственность AuthenticationProvider тесно связана с контрактом Authentication. Метод authenticate() получает объект Authentication в качестве параметра и возвращает объект Authentication. Мы реализуем метод authenticate() для определения логики аутентификации:

- метод должен выдавать исключение AuthenticationException, если аутентификация не удалась;
- если метод получает объект аутентификации, не поддерживае- мый вашей реализацией AuthenticationProvider, то он должен возвращать null. Таким образом, у нас есть возможность использовать несколько типов аутентификации, разделенных на уровне HTTP-фильтра;
- метод должен возвращать экземпляр Authentication, представляющий полностью аутентифицированный объект. Для этого экземпляра метод isAuthenticated() возвращает true, и он содержит все необходимые сведения об аутентифицированном объекте. Обычно приложение также удаляет из этого экземпляра конфиденциальные данные, такие как пароль. После успешной аутентификации пароль больше не требуется, и сохранение этих сведений может потенциально раскрыть их посторонним.

Второй метод в интерфейсе AuthenticationProvider – это supports(Class<?> authentication). Вы можете заставить его возвращать true, если текущий AuthenticationProvider поддерживает тип, предоставленный как объект Authentication.

Применение пользовательской логики аутентификации

Пример в ex11

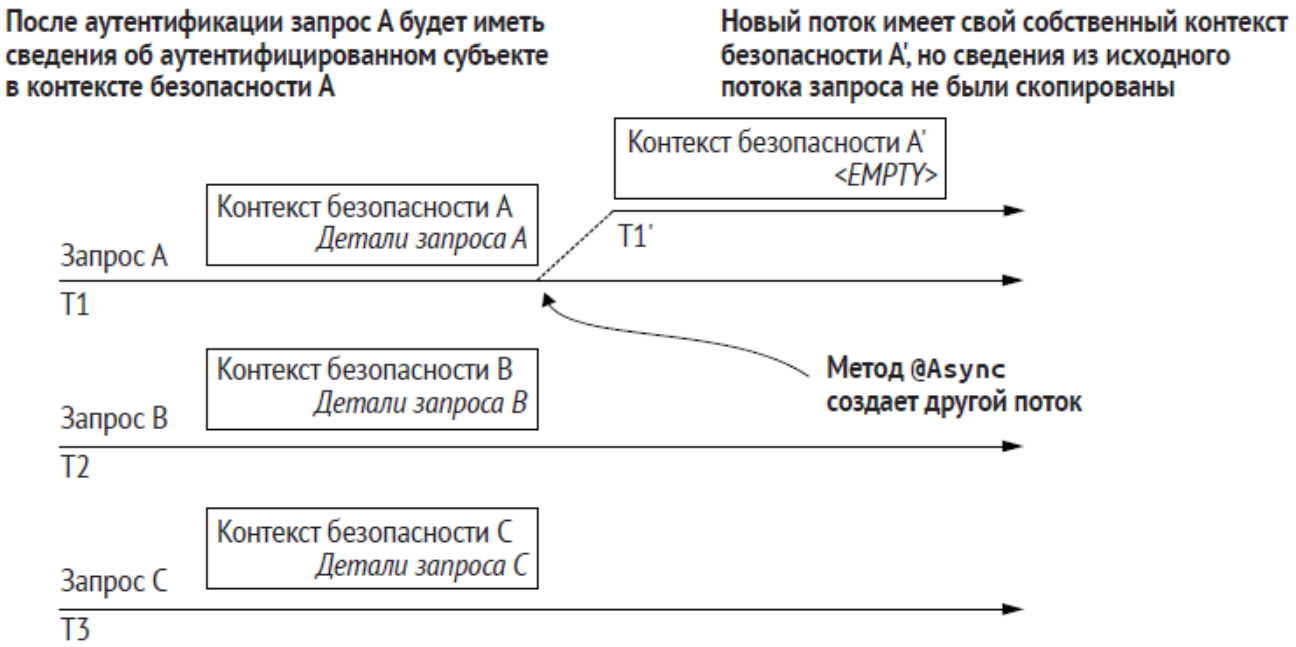
Использование SecurityContext

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/context/SecurityContext.html

Как следует из определения контракта, основная обязанность SecurityContext – хранить объект Authentication. Но как управляется сам SecurityContext? Spring Security предлагает три стратегии управления SecurityContext с объектом в роли менеджера. Он называется SecurityContextHolder:

- MODE_THREADLOCAL – позволяет каждому потоку хранить свои собственные сведения в контексте безопасности. В веб-приложении с одним потоком на запрос это распространенный подход, поскольку каждый запрос имеет отдельный поток;
- MODE_INHERITABLETHREADLOCAL – работает аналогично предыдущей стратегии, но также указывает Spring Security копировать контекст безопасности в следующий поток в случае асинхронного метода. Таким образом, можно сказать, что новый поток, запускающий метод @Async, наследует контекст безопасности. Аннотация @Async используется с методами, чтобы указать Spring вызвать аннотированный метод в отдельном потоке;
- MODE_GLOBAL – заставляет все потоки приложения видеть один и тот же экземпляр контекста безопасности.

Использование стратегии хранения для контекста безопасности



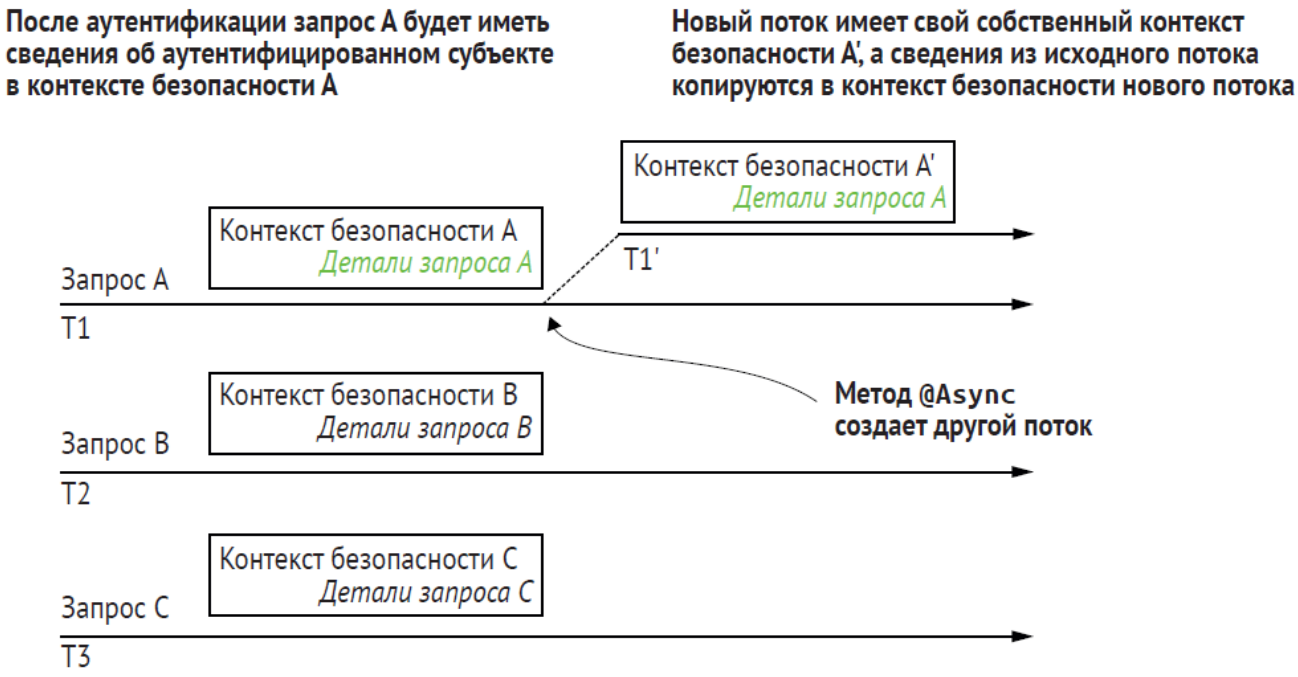
Согласно этой стратегии Spring Security использует ThreadLocal для управления контекстом. ThreadLocal – это реализация, предо- ставляемая JDK. Она работает как коллекция данных, но гарантирует, что каждый поток приложения может видеть только данные, хранящиеся в его выделенной части коллекции. Таким образом, каждый запрос имеет доступ к своему контексту безопасности. Ни один

поток не будет иметь доступа к ThreadLocal другого потока. Это означает, что в веб-приложении каждый запрос может видеть только свой собственный контекст безопасности.

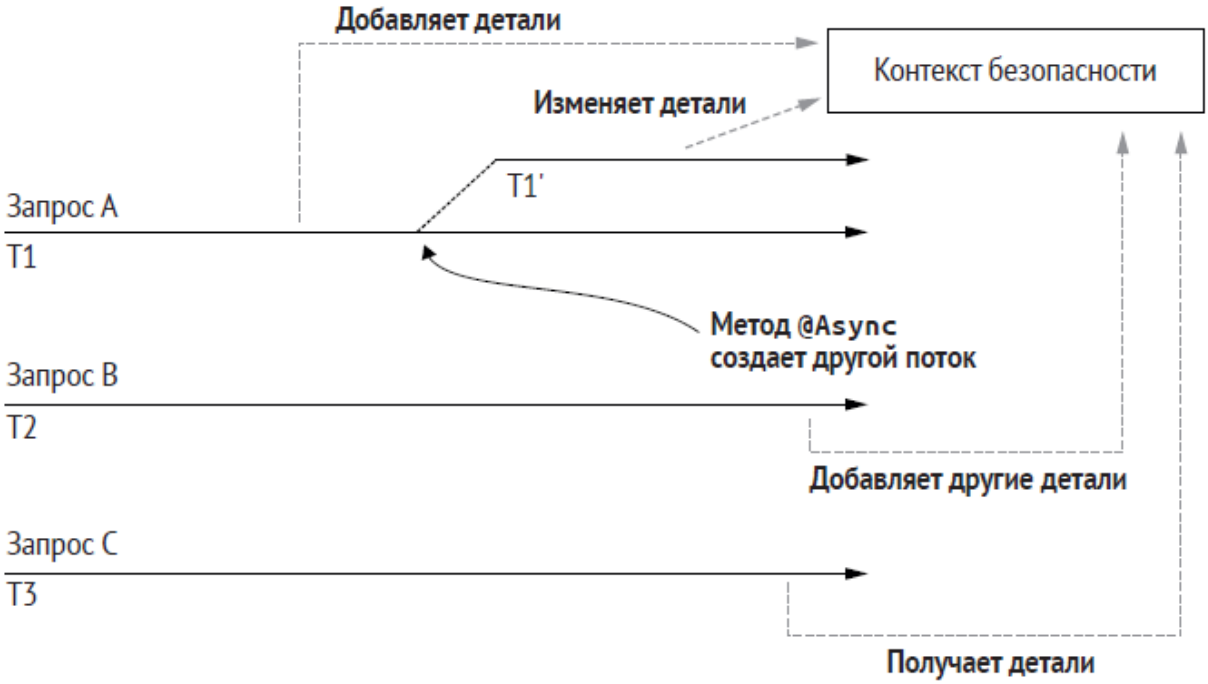
Пример в ex12

Использование стратегии хранения для асинхронных вызовов

Ситуация усложняется, если нам приходится иметь дело с не- сколькими потоками на запрос.



Использование стратегии хранения в автономных приложениях



Аутентификация на основе форм

Пример в ex14