Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №6
Spring Security.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Введение в Spring Security

Spring Security – это фреймворк обеспечения безопасности, предоставляющий возможность декларативного управления безопасностью приложений на основе фреймворка Spring. Фреймворк Spring Security представляет собой всеобъемлющее решение по обеспечению безопасности, реализующее возможность аутентификации и авторизации как на уровне вебзапросов, так и на уровне вызовов методов. Опираясь на возможности фреймворка Spring Framework, Spring Security в полной мере использует поддержку внедрения зависимостей и аспектно-ориентированного программирования.

Проект Spring Security начинал разрабатываться под названием Acegi Security . Acegi – это мощный фреймворк обеспечения безопасности, но он имел один существенный недостаток – большой объем конфигурационного файла в формате XML. Не будем приводить громоздкий пример с демонстрацией примера такого конфигурационного файла, но упомянем, что типичная конфигурация Acegi занимала несколько сотен строк разметки XML. В версии 2.0 фреймворк Acegi Security был переименован в Spring Security. Но версия 2.0 отличается не только названием. В Spring Security 2.0 появилось новое пространство имен XML, связанное с безопасностью и предназначенное для настройки системы безопасности в Spring. Новое пространство имен, наряду с аннотациями и обоснованными настройками по умолчанию, позволило сократить объем конфигурации безопасности с сотен до десятков и даже единиц строк. В Spring Security 3.0, была добавлена поддержка языка выражений SpEL, что еще больше упростило настройку безопасности.

Фреймворк Spring Security обеспечивает безопасность с двух сторон. Для ограничения доступа и обеспечения безопасности на уровне запросов в Spring Security используются сервлет-фильтры. А для обеспечения безопасности на уровне вызовов методов с использованием Spring AOP фреймворк Spring Security предоставляет объекты-обертки и позволяет применять советы, гарантирующие авторизацию пользователей.

1.1 Обзор Spring Security

Независимо от типа приложения, которое предполагается обезопасить с применением Spring Security, в первую очередь следует добавить модули Spring Security в библиотеку классов (classpath) приложения. Версия Spring Security 3.0 делится на восемь модулей , которые перечислены в таблице далее.

Модуль	Описание
ACL	Обеспечивает поддержку безопасности доменных объектов с использованием списков управления доступом (Access Control Lists, ACL)

CAS Client	Обеспечивает интеграцию с централизованной службой аутентификации JA-SIG (Central Authentication Service, CAS)
Configuration	Обеспечивает поддержку пространства имен XML
Core	Основная библиотека Spring Security
LDAP	Обеспечивает поддержку аутентификации с использованием облегченного протокола доступа к каталогу (Lightweight Directory Access Protocol, LDAP)
OpenID	Обеспечивает интеграцию с децентрализованной службой OpenID
Tag Library	Включает множество тегов JSP для обеспечения безопасности на уровне представлений
Web	Обеспечивает поддержку безопасности веб-приложений с применением фильтров

В библиотеку классов приложения требуется включить, как минимум, модули Core и Configuration. Фреймворк Spring Security часто используется для обеспечения безопасности веб-приложений.

Теперь можно приступать к декларативной настройке безопасности в Spring Security. Для начала познакомимся с пространством имен XML, предоставляемым фреймворком Spring Security.

1.2 Использование конфигурационного пространства Spring Security

Когда фреймворк Spring Security еще носил название Acegi Security, все элементы поддержки безопасности настраивались как компоненты в контексте приложения Spring. Типичный конфигурационный файл Acegi мог содержать десятки объявлений компонентов <bean> и располагаться на множестве страниц.

Фреймворк Spring Security поддерживает специализированное пространство имен, существенно упрощающее настройку безопасности в Spring. Это новое пространство имен, наряду с обоснованными настройками по умолчанию, уменьшает размер типичного конфигурационного XML-файла с сотен до десятков строк. Единственное, что необходимо сделать, чтобы получить возможность пользоваться новым пространством имен, – подключить его в XML-файле, добавив его объявление, как показано далее:

```
<beans xmlns="http://www.springframework.org/schema/security"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                  http://www.springframework.org/schema/security
                  http://www.springframework.org/schema/security/spring-security-
3.0.xsd">
```

```
<!-- Здесь располагаются элементы с префиксом security: -->
</beans>
```

В нашем приложении мы поместим все настройки безопасности в отдельный конфигурационный файл с именем security.xml. Поскольку в этом файле все элементы будут принадлежать специализированному пространству имен, его можно объявить пространством имен по умолчанию, как показано ниже

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/security
            http://www.springframework.org/schema/security/spring-security-
3.0.xsd">

<!-- Здесь располагаются элементы без префикса security: -->
</beans:beans>
```

Объявив пространство имен элементов настройки безопасности пространством имен по умолчанию, можно избежать необходимости использовать префикс security: во всех элементах.

Теперь, когда появилась возможность аккуратно расположить все настройки безопасности в отдельном месте, можно приступать к добавлению в наше простое приложение настроек безопасности на уровне веб-запросов.

2 Безопасность веб-запросов

Все взаимодействия с веб-приложениями на языке Java начинаются в компоненте HttpServletRequest. И коль скоро средством доступа к веб-приложению является запрос, то с него и следует начинать обеспечивать безопасность.

Обычно настройка безопасности на уровне запросов начинается с объявления одного или более шаблонов URL-адресов, требующих некоторых привилегий, и ограничения доступа к содержимому по этим адресам для пользователей, не обладающих необходимыми привилегиями. Более того, доступ к некоторым URL можно также ограничить протоколом HTTPS.

Прежде чем ограничивать доступ пользователям, не обладающим необходимыми привилегиями, необходимо предусмотреть возможность определять, кто пользуется приложением. Поэтому приложение должно аутентифицировать пользователя, предложив ему идентифицировать себя.

Фреймворк Spring Security поддерживает эти и многие другие способы обеспечения безопасности на уровне запросов. Но для начала следует настроить сервлет-фильтры, предоставляющие разнообразные возможности.

2.1 Сервлет-фильтры

Для обеспечения различных аспектов безопасности Spring Security использует несколько сервлет-фильтров . Кому-то может показаться, что это означает необходимость добавлять в конфигурационный файл web.xml приложения несколько элементов <filter> . Но это не так – благодаря волшебству Spring достаточно настроить только один фильтр. В частности, достаточно добавить следующий элемент <filter>:

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

DelegatingFilterProxy – это специальный сервлет-фильтр, не имеющий самостоятельного значения, так как он просто делегирует фильтрацию реализации интерфейса javax.servlet.Filter, зарегистрированной в контексте приложения Spring в виде компонента, как показано на рис. 1.

Рисунок 1 — Объекты-прокси DelegatingFilterProxy, обеспечивающие фильтрацию, делегируют фильтрацию компонентам в контексте приложения Spring

Чтобы задействовать фильтры из фреймворка Spring Security, в них необходимо внедрить какие-то другие компоненты. Однако в фильтры, зарегистрированные в файле web.xml, нельзя внедрить компоненты. Но с помощью DelegatingFilterProxy можно настроить фактические фильтры в Spring, воспользовавшись преимуществом поддержки внедрения зависимостей.

Значение в элементе <filter-name> играет важную роль. Это имя будет использоваться для поиска компонента фильтра в контексте приложения Spring. Фреймворк Spring Security автоматически создаст компонент фильтра с идентификатором springSecurityFilterChain, потому что это имя указано в файле web.xml.

Что касается самого компонента springSecurityFilterChain, это еще один специальный фильтр, известный как FilterChainProxy . Это единый фильтр, объединяющий в себе один или более других фильтров. Для обеспечения различных возможностей фреймворк Spring Security опирается на разные сервлет-фильтры. Но вам практически никогда не придется задумываться об этой особенности, потому что вам едва ли когда-нибудь потребуется явно

объявлять компонент springSecurityFilterChain или любые другие фильтры, объединяемые в нем. Фреймворк Spring Security автоматически создает все необходимые компоненты при настройке элемента <http>, о чем рассказывается ниже.

2.1.1 Минимальная настройка безопасности

В ранних версиях Spring Security для настройки основных механизмов системы безопасности требовалось добавлять в конфигурационные XML-файлы огромное количество элементов. Но в последних версиях Spring Security достаточно добавить следующий фрагмент:

```
<http auto-config="true">
        <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

Эти три строчки обеспечивают перехват запросов для всех URL (как следует из значения атрибута pattern элемента <intercept-url>) и разрешают доступ к содержимому только аутентифицированным пользователям, обладающим привилегией ROLE_USER. Элемент <http> автоматически настраивает компонент FilterChainProxy (которому делегирует фильтрацию объект DelegatingFilterProxy, настроенный в файле web.xml) и все остальные компоненты в цепочке, реализующие фильтрацию.

В дополнение к этим компонентам фильтров можно получить дополнительные возможности, установив атрибут auto-config в значение true. Автоматическая настройка включает в приложение страницу аутентификации, поддержку аутентификации средствами протокола HTTP и поддержку завершения сеанса работы. В действительности установка атрибута auto-config в значение true эквивалентна явной настройке этих возможностей, как показано ниже:

```
<http>
        <form-login />
        <http-basic />
        <logout />
        <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

Познакомимся с ними поближе, чтобы понять, как ими пользоваться.

2.1.2 Аутентификация с помощью формы

Одним из преимуществ установки атрибута auto-config в значение true является автоматическое создание страницы аутентификации фреймворком Spring Security.

Адрес URL автоматически сгенерированной формы аутентификации складывается из базового адреса URL приложения и относительного пути /login. Например, при обращении к приложению security на локальном

компьютере этот URL будет иметь вид: http://localhost:8081/security/login (подробнее смотри в примере к лекции). На первый взгляд возможность автоматического создания формы аутентификации выглядит большим преимуществом. Но эта форма слишком проста и не отличается эстетической привлекательностью, поэтому в большинстве случаев она заменяется специально созданной формой.

Чтобы добавить в приложение свою форму аутентификации, необходимо настроить элемент <form-login> :

```
<http auto-config="true" use-expressions="false">
        <form-login
                login-page="/mylogin"
                authentication-failure-url="/mylogin?error"/>
</http>
```

Атрибут login определяет новый относительный путь к странице аутентификации. В данном случае указывается, что страница аутентификации будет иметь относительный путь /mylogin, который в конечном итоге обслуживается контроллером Spring MVC. Аналогично, в случае неудачной попытки аутентификации, атрибут authentication-failure-url будет отправлять пользователя обратно на страницу аутентификации.

Кроме того, можно определить свойства username-parameter и password-parameter определяющие с каких полей ввода информация будет передаваться в качестве username и password для фреймворка:

```
<form-login
        login-page="/mylogin"
        authentication-failure-url="/login?error"
        username-parameter="username"
        password-parameter="password"/>
        <csrf/>
```

Также в последних версиях Spring Security важную роль играет своqство csrf (https://ru.wikipedia.org/wiki/Межсайтовая_подделка_запроса), которое обязательно необходимо добавлять в свойствах form-login, а также на смой форме:

```
<input type="hidden" name="${_csrf.parameterName}"
       value="${_csrf.token}"/>
```

2.1.3 Аутентификация средствами протокола HTTP

Аутентификация с использованием формы идеально подходит для людей, пользующихся приложением. Но позже, когда мы перейдем к изучению RESTful API станет понятно что, когда пользователем приложения является другое приложение и использовать форму аутентификации становится неудобно.

Механизм аутентификации, предусмотренный протоколом HTTP, обеспечивает возможность передачи информации об аутентификации

непосредственно в HTTP-запросе. Возможно, вам приходилось сталкиваться с этой возможностью прежде. В браузере эта процедура выполняется с использованием простого модального диалога.

Но это лишь внешнее проявление действия механизма в веб-браузере. В действительности запрос на аутентификацию возвращается в виде ответа HTTP 401, указывающего на необходимость передать имя пользователя и пароль в теле запроса. Эта возможность отлично подходит для REST-клиентов, позволяя им аутентифицировать себя при пользовании услугами приложения. Аутентификация средствами протокола HTTP включается с помощью элемента <http-basic> и не требует большого количества настроек – она может быть либо включена, любо выключена. Поэтому не будем больше останавливаться на этой теме и двинемся дальше, где нас ждет элемент <logout> .

2.1.4 Завершение сеанса работы

Элемент <logout> настраивает фильтр Spring Security, который будет закрывать сеанс работы с пользователем. При использовании с настройками по умолчанию элемент <logout> отображается на адрес /logout. Но, можно и переопределить адрес, используемый для завершения сеанса, для этого необходимо переопределить URL фильтра, как это делалось для формы аутентификации. Для этого следует определить атрибут logout-url:

<logout logout-url="/mylogout"/>

На этом завершается обсуждение возможностей, предоставляемых автоматически. Но исследование Spring Security не заканчивается. Познакомимся поближе с элементом <intercept-url> и посмотрим, как с его помощью управлять доступом на уровне запросов.

2.2 Перехват запросов

В предыдущем разделе был представлен простой пример использования элемента <intercept-url> . Но мы узнали о нем совсем немного… пока.

Элемент <intercept-url> – это первая линия обороны в системе безопасности. Его атрибут pattern определяет шаблон URL, который будет сопоставляться с входящими запросами. Если какой-либо запрос совпадет с шаблоном, к нему будут применены правила безопасности, определяемые элементом <intercept-url>.

Вернемся к определению элемента <intercept-url>, представленному выше:

<intercept-url pattern="/**" access="ROLE_USER" />

По умолчанию в атрибуте pattern указывается шаблон пути в стиле утилиты Ant. Но если атрибуту path-type элемента <http> присвоить значение regex, в качестве шаблона пути можно будет использовать регулярное выражение.

В данном случае в атрибуте pattern задано значение /**, указывающее, что перехватываться должны все запросы, независимо от URL, и для получения доступа к содержимому пользователь должен обладать привилегией ROLE_USER. Шаблон /** имеет широкую область значений и иногда бывает необходимо ограничить ее.

Представьте, что в нашем приложении имеются специальные страницы, которые должны быть доступны только администраторам. Чтобы ограничить доступ к ним, можно добавить следующий элемент <intercept-url> перед уже имеющимся:

```
<intercept-url pattern="/admin/**" access="ROLE_ADMIN" />
```

Наш первый элемент <intercept-url> гарантирует, что к большей части приложения смогут обращаться только пользователи, обладающие привилегией ROLE_USER, а данный элемент <intercept-url> ограничивает доступ к ветви /admin в иерархии сайта, допуская к ней только пользователей с привилегией ROLE_ADMIN. В файле конфигурации допускается использовать любое количество элементов <intercept-url>, чтобы обезопасить различные пути в веб-приложении. Но важно помнить, что правила, определяемые элементами <intercept-url>, применяются в направлении сверху вниз. То есть этот новый элемент <intercept-url> должен находиться перед оригинальным элементом, иначе шаблон пути /** получит более высокий приоритет.

2.2.1 Настройка безопасности с применением выражений Spring

Настройка требуемых привилегий реализуется достаточно просто, но выглядит несколько однобоко. А что, если потребуется выразить ограничения, опираясь не только на имеющиеся привилегии?

Начиная с версии 3.0, фреймворк Spring Security предоставляет возможность использовать SpEL выражения для определения правил доступа. Чтобы воспользоваться ею, необходимо определить значение true в атрибуте use-expressions элемента <http>:

```
<http auto-config="true" use-expressions="true">
 ...
</http>
```

Теперь можно использовать выражения на языке SpEL в атрибуте access. Ниже показано, как на языке SpEL потребовать наличия привилегии ROLE_ADMIN при обращении к адресам, совпадающим с шаблоном /admin/**:

```
<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')"/>
```

Этот элемент <intercept-url> полностью эквивалентен предыдущему, за исключением того, что в нем используется выражение на языке SpEL. Выражение hasRole() возвращает true, если текущий пользователь обладает указанной привилегией. Но hasRole() – лишь одно из множества

поддерживаемых выражений, имеющих отношение к безопасности. В таблице ниже перечислены все выражения SpEL, поддерживаемые фреймворком Spring Security 3.0.

Пространство имен	Назначение
authentication	Объект аутентификации пользователя
denyAll	Всегда возвращает false
hasAnyRole(list_of_roles)	true, если пользователь обладает какой-либо из привилегий, перечисленных в списке list_of_roles
hasRole(role)	true, если пользователь обладает привилегией role
hasIpAddress(IP Address)	IP-адрес пользователя (доступен только в веб-приложениях)
isAnonymous()	true, если текущий пользователь не был аутентифицирован
isAuthenticated()	true, если текущий пользователь был аутентифицирован
isFullyAuthenticated()	true, если текущий пользователь был аутентифицирован и не использовал функцию «запомнить меня»
isRememberMe()	true, если текущий пользователь был аутентифицирован автоматически
permitAll	Всегда возвращает false
principal	Основной объект, представляющий пользователя

Благодаря поддержке языка выражений SpEL доступ к страницам можно ограничивать, основываясь не только на привилегиях пользователя. Например, если доступ к страницам администрирования должен ограничиваться не только наличием привилегии ROLE_ADMIN, но и определенным IP-адресом, элемент <intercept-url> можно определить, как показано ниже:

<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>

Поддержка языка SpEL обеспечивает практически неограниченные возможности настройки безопасности.

А сейчас познакомимся с еще одной интересной особенностью элемента <intercept-url>: принудительное требование к используемому протоколу.

2.2.2 Принудительное использование протокола HTTPS

Передача данных по протоколу HTTP весьма небезопасна. Возможно, нет ничего страшного, если текст сообщений в вашем приложении будет отправляться с помощью открытого протокола HTTP. Но передача секретной информации, такой как пароли и номера кредитных карт, по протоколу HTTP может привести к большим неприятностям. Именно поэтому секретную информацию лучше отправлять в зашифрованном виде, по протоколу HTTPS.

Задействовать протокол HTTPS довольно просто. Достаточно лишь добавить символ «s» после «http» в URL и все. Правильно?

Да, это так, но при этом вся ответственность за использование протокола HTTPS возлагается на программиста. А теперь представьте, что в приложении имеются десятки и сотни ссылок и форм, в которых должен быть указан протокол HTTPS. Слишком легко забыть добавить такую важную букву «s». Слишком велика вероятность, что программист пропустит ее в одном-двух местах или по ошибке укажет протокол HTTPS там, где в нем нет необходимости.

Атрибут requires-channel элемента <intercept-url> позволяет снять эту ответственность с программиста и переложить ее на конфигурацию Spring Security.

Рассмотрим в качестве примера форму регистрации в нашем приложении. Наши простые приложения пока не просят указать номер кредитной карты или что-то жутко секретное, однако пользователи могут пожелать сохранить в тайне информацию о себе. Для этого необходимо настроить элемент <intercept-url>, описывающий доступ к адресу /form, как показано ниже:

```
<intercept-url pattern="/form" requires-channel="https"/>
```

Всякий раз, когда приложение будет получать запрос к адресу /form, фреймворк Spring Security будет обнаруживать требование https к протоколу и автоматически переадресовывать запрос на использование протокола HTTPS. Аналогично для доступа к главной странице приложения не требуется использовать протокол HTTPS, поэтому можно объявить, что доступ к ней всегда должен осуществляться по протоколу HTTP:

```
<intercept-url pattern="/home" requires-channel="http"/>
```

До настоящего момента демонстрировалось, как обезопасить веб-приложение на уровне запросов. При этом предполагалось, что основная задача системы безопасности состоит в том, чтобы помешать пользователю обращаться к определенным адресам URL, если он не имеет соответствующих привилегий. Но было бы неплохо также никогда не показывать ссылки тем пользователям, которые не смогут выполнить переход по ним. Посмотрим, что может предложить фреймворк Spring Security для безопасности представлений.

3 Безопасность на уровне представлений

Для обеспечения безопасности на уровне представлений в состав фреймворка Spring Security включена библиотека тегов JSP. Эта библиотека невелика и содержит всего три тега, которые перечислены в таблице ниже.

Тег JSP	Описание
<security:accesscontrollist>	Содержимое тега отображается, если текущий пользователь обладает одной из привилегий в указанном доменном объекте
<security:authentication>	Обеспечивает доступ к свойствам объекта аутентификации текущего пользователя
<security:authorize>	Содержимое тега отображается, если удовлетворяются указанные требования безопасности

Чтобы получить возможность использовать библиотеку тегов JSP, необходимо объявить ее в JSP-файле:

<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>

Если использовать эти теги в FreeMarker, то выглядеть это будет следующим образом:

```
<#assign security=JspTaglibs["http://www.springframework.org/security/tags"] />
```

Также для работы библиотеки тегов следует подключить зависимость spring-security-taglibs.

После объявления библиотеку можно использовать. Рассмотрим поочередно все три тега JSP, входящие в состав Spring Security, и посмотрим, как они действуют.

3.1 Доступ к информации об аутентификации

Самое простое, на что способна библиотека тегов JSP, входящая в состав Spring Security, – предоставить доступ к информации об аутентификации пользователя. Например, для многих сайтов типично выводить приветствие в заголовке страницы, указывая в нем имя пользователя. Эту информацию можно получить с помощью тега <security:authentication>. Например:

Hello <security:authentication property="principal.username" />!

Для FreeMarker:

<security:authentication property="principal.username" />!

Атрибут property определяет свойство объекта аутентификации пользователя. Перечень доступных свойств зависит от того, как был

аутентифицирован пользователь. Однако некоторые свойства, включая перечисленные в таблице ниже, являются общими и доступны всегда.

Свойство	Описание
authorities	Коллекция объектов GrantedAuthority, представляющих привиле- гии, которыми обладает пользователь
credentials	Ключевая информация, использованная для проверки подлинности пользователя (часто это пароль)
details	Дополнительная информация об аутентификации (IP-адрес, серийный номер сертификата, идентификатор сеанса и т. д.)
principal	Основной объект с информацией о пользователе

В данном примере отображаемое свойство в действительности является вложенным свойством username свойства principal.

При таком использовании, как в примере выше, тег <security: authentication> отобразит значение указанного свойства в представлении. Но если потребуется лишь присвоить значение свойства переменной, тогда достаточно просто указать имя переменной в атрибуте var :

<security:authentication property="principal.username" var="loginId"/>

Тег <security:authentication> может пригодиться во многих ситуациях, но это лишь малая часть того, что может предоставить библиотека тегов JSP в Spring Security. Посмотрим, как обеспечить отображение содержимого страницы в зависимости от привилегий пользователя.

3.2 Отображение с учетом привилегий

Иногда некоторые фрагменты представления должны или не должны отображаться, в зависимости от привилегий пользователя. Бессмысленно отображать форму аутентификации, если пользователь уже аутентифицирован, или показывать персонализированное приветствие пользователю, который еще не аутентифицирован.

Тег <security:authorize> позволяет отображать фрагменты представлений в зависимости от привилегий, которыми обладает пользователь. Продемонстрируем пример. Допустим мы хотим, чтобы некоторое сообщение было показано только для адимнистраторов.

```
<body>
	<h2>Welcome!</h2>
	<@security.authorize access="hasRole('ROLE_ADMIN')">
		<i>Это тайное админское сообщение</i>
	</@security.authorize>
</body>
```

В атрибуте access указано выражение на языке SpEL, результат которого определяет, будет ли отображаться тело тега <security:authorize>. Здесь используется выражение hasRole(‘ROLE_ADMIN’), проверяющее, обладает ли текущий пользователь привилегией ROLE_ADMIN. Но точно так же в атрибуте access можно использовать любые выражения на языке SpEL.

С помощью этих выражений можно реализовать весьма интересные ограничения. Например, одним типам пользователей можно показывать определенные ссылки, а другим нет, но есть одна лазейка. Разумеется, ссылка с адресом некоторой страницы не будет отображаться перед другими пользователями, но ничто не мешает им вручную ввести URL в адресной строке браузера. Но, для того чтобы исправить данную проблему, достаточно лишь добавить новый элемент <intercept-url> в настройки безопасности, ограничивающий доступ к таким URL.

<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN') />

В данном случае административные функции будут надежно заперты от посторонних. Сам URL будет доступен только одному пользователю, и с помощью jsp-тегов можно настроить, чтобы ссылки на администраторские страницы не были показаны пользователям, не обладающим соответствующими полномочиями. Но, чтобы добиться этого, пришлось объявить одно и то же выражение на языке SpEL в двух местах – в элементе <intercept-url> и в атрибуте access тега <security:authorize>. А возможно ли организовать отображение URL только при выполнении требований безопасности, определяемых для него?

Возможно, с помощью атрибута url тега <security:authorize>. В отличие от атрибута access, где ограничения определяются явно, атрибут url неявно ссылается на ограничения, накладываемые на шаблон URL. Поскольку ограничения безопасности для URL /admin уже определены в конфигурационном файле Spring Security, мы можем использовать атрибут url, как показано ниже:

<security:authorize url="/admin/**">
<spring:url value="/admin" var="admin_url" />

Admin </security:authorize>

Поскольку доступ к URL /admin может получить только аутентифицированный пользователь с привилегией ROLE_ADMIN тело тега <security: authorize> будет отображаться, только если выполняются все эти требования.

4 Аутентификация пользователей

Каждое приложение имеет свои особенности. Эта истина особенно ярко проявляется в том, как каждое приложение хранит информацию о пользователях. Иногда для этого используется реляционная база данных.

Иногда каталог LDAP. Некоторые приложения опираются на децентрализованные системы аутентификации пользователей. А некоторые могут использовать сразу несколько стратегий.

К счастью, фреймворк Spring Security обладает достаточной гибкостью и способен использовать практически любую стратегию аутентификации. Spring Security готов поддержать многие механизмы аутентификации, включая аутентификацию пользователей при использовании:

- репозитория в памяти (настраиваемого в контексте приложения Spring);
- репозитория на основе JDBC;
- репозитория на основе LDAP;
- децентрализованных систем идентификации OpenID;
- централизованной системы аутентификации (Central Authentication System, CAS);
- сертификатов X.509; провайдеров на основе JAAS.

Если ни один из этих вариантов не соответствует вашим потребностям, легко можно реализовать собственную стратегию аутентификации и внедрить ее. Рассмотрим поближе некоторые из наиболее часто используемых способов аутентификации, поддерживаемых фреймворком Spring Security.

4.1 Настройка репозитория в памяти

Простейшим из имеющихся способов аутентификации является объявление информации о пользователях непосредственно в файле конфигурации Spring. Делается это за счет настройки службы учета пользователей с помощью элемента <user-service> , имеющегося в пространстве имен Spring Security:

```
<user-service id="userService">
        <user name="admin"
password="$2y$10$dZPGF39gkohmiVwAZQw.muH8W2anLtKwqMmGqWDbQIc
dB30qF/NKe"
                authorities="ROLE_ADMIN"/>
        <user name="user"
password="$2y$10$m83AE6lNYK8l2esyc9u5buZD2nY9IQ7/z7hOiy9nOzdOLnf1
WHu1m" authorities="ROLE_USER"/>
 </user-service>
<password-encoder hash="bcrypt"/>
```

Функции службы учета пользователей фактически выполняет объект доступа к данным, который отыскивает информацию по указанному идентификатору. В случае с элементом <user-service> эта информация о пользователях объявляется внутри элемента <user-service>. Каждому

пользователю, который может пользоваться приложением, соответствует отдельный элемент <user> . Атрибуты name и password определяют, соответственно, имя пользователя и пароль. А в атрибуте authorities указывается список привилегий, перечисленных через запятую, которые определяют доступные пользователю операции.

Также здесь важен элемент password-encoder , который указывает алгоритм хэширования паролей. При использовании последних версий библиотек использование кодировщика паролей обязательно. Все еще можно указать noop (https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/DelegatingPasswordEncoder.html), но это довольно плохая практика. В данном примере используется алгоритм хэширования bcrypt (о нем можете узнать подробнее самостоятельно).

Ранее в конфигурации Spring Security мы разрешили доступ ко всем URL только пользователям с привилегией ROLE_USER. В данном случае эту привилегию получают пользователь user, а пользователь admin – нет.

Теперь служба учета пользователей готова к работе и ожидает возможности отыскать информацию для аутентификации. Осталось лишь внедрить ее в подсистему аутентификации Spring Security:

```
<authentication-manager>
    <authentication-provider user-service-ref="userService" />
</authentication-manager>
```

Элемент <authentication-manager> регистрирует компонент подсистемы аутентификации. Точнее, он регистрирует экземпляр ProviderManager, который делегирует функции аутентификации пользователей одной или более провайдерам аутентификации. В данном случае используется провайдер, опирающийся на службу учета пользователей. В приложении уже имеется служба учета пользователей. Поэтому осталось лишь внедрить ее, указав в атрибуте user-service элемента <authentication-provider>. В этом примере провайдер аутентификации и служба учета пользователей были объявлены отдельно друг от друга, а затем связаны между собой. Однако существует также возможность встроить службу учета пользователей непосредственно в провайдер аутентификации:

```
<authentication-provider>
    <user-service id="userService">
        <user name="user" password="тут хэш, но он длинный"
         authorities="ROLE_USER"/>
    ...
        </user-service>
    </authentication-provider>
```

Способ встраивания элемента <user-service> в элемент <authentication-provider> не имеет каких-либо существенных преимуществ, но для кого-то такой способ настройки может выглядеть более логичным.

Прием определения информации о пользователях в контексте приложения Spring удобно использовать во время тестирования, когда вы только приступаете к настройке системы безопасности приложения. Но в условиях нормальной эксплуатации приложения этот способ никуда не годится. Чаще всего информация о пользователях сохраняется в базе данных или на сервере каталогов. Поэтому далее посмотрим, как зарегистрировать службу учета пользователей, которая ищет информацию в реляционной базе данных.

4.2. Аутентификация с использованием базы данных

Многие приложения хранят информацию о пользователях, включая имена и пароли, в реляционной базе данных. Если в приложении предполагается хранить информацию о пользователях именно таким способом, для этой цели с успехом можно использовать элемент Spring Security <jdbc-user-service>.

Элемент <jdbc-user-service> используется так же, как и элемент <user-service>, включая его внедрение в атрибут user-service элемента <authentication-provider> или встраивание внутрь элемента <authentication-provider>. Ниже демонстрируется пример настройки <jdbc-user-service>, где определяется значение атрибута id, благодаря чему элемент может быть объявлен отдельно и затем внедрен в элемент <authentication-provider>:

```
<jdbc-user-service id="userService"
        data-source-ref="dataSource" />
```

Для извлечения информации о пользователях из базы данных элемент <jdbc-user-service> использует источник данных JDBC, внедренный посредством атрибута data-source . Без дополнительных на-строек служба учета пользователей будет извлекать информацию, используя следующий SQL-запрос:

```
select username,password,enabled
from users
where username = ?
```

И хотя сейчас мы ведем речь об аутентификации пользователя, определение привилегий пользователя тоже можно отнести к процедуре аутентификации. По умолчанию элемент <jdbc-user-service> будет использовать следующий SQL-запрос для определения привилегий пользователя с указанным именем:

```
select username,authority
from authorities
where username = ?
```

Все это хорошо, если так случилось, что информация о пользователях в базе данных приложения хранится в таблицах, соответствующих этим запросам. Но готов поспорить, что в большинстве приложений это не так.

К счастью, элемент <jdbc-user-service> легко настроить на использование любых других запросов, лучше подходящих под нужды приложения. В таблице ниже перечислены атрибуты элемента <jdbc-userservice>, которые можно использовать для настройки его поведения.

Атрибут	Описание
users-by-username-query	Запрос имени пользователя, пароля и признака «разрешен/запрещен»
authorities-by-username-query	Запрос привилегий для указанного пользователя
group-authorities-by-username-query	Запрос группы привилегий для указанного пользователя

Опишем атрибуты users-byusername-query и authorities-by-username-query следующим образом:

```
<jdbc-user-service id="userService"
        data-source-ref="dataSource"
        users-by-username-query=
                "select login, password, true from public.&#34;User&#34;where login=?"
        authorities-by-username-query=
"select login, 'ROLE_ADMIN' from public.&#34;User&#34; where login=? />
```

Здесь " - элемент кавычек для xml, используется он здесь по причине того, что для PostgreSQL требует кавычки.

В данном примере имена пользователей и пароли хранятся в таблице User, в полях login и password соответственно. Но мы не рассматривали идею возможности активации или деактивации учетных записей пользователей, предполагая, что все учетные записи активны. Поэтому SQL-запрос написан так, что в качестве признака «разрешен/запрещен» он всегда возвращает true для всех пользователей.

Мы также не позаботились о распределении пользователей приложения по разным уровням привилегий. Все пользователи обладают одинаковыми привилегиями. В действительности в примере отсутствует таблица, где хранится информация о привилегиях пользователей. Поэтому в атрибуте authorities-by-username-query указан запрос, возвращающий привилегию ROLE_USER для любых пользователей.

Использование LDAP в данном курсе не рассматривается.

4.3 Включение функции «запомнить меня»

Возможность аутентификации пользователей имеет большое значение для приложения. Но, с точки зрения пользователя, было бы неплохо, если бы приложение не всегда требовало вводить имя пользователя и пароль. Именно по этой причине многие сайты предлагают возможность «запомнить» пользователя после аутентификации, чтобы при последующих обращениях к приложению ему не приходилось повторно проходить эту процедуру.

Фреймворк Spring Security дает возможность легко и просто включить в приложение поддержку функции «запомнить меня» . Для этого достаточно всего лишь добавить элемент <remember-me> в элемент <http>:

```
<http auto-config="true" use-expressions="true">
...
<remember-me
        key="testKey"
        token-validity-seconds="2419200" />
</http>
```

Здесь демонстрируется включение поддержки функции «запомнить меня» с некоторыми дополнительными настройками. При использовании элемента <remember-me> без атрибутов автоматическая аутентификация реализуется за счет сохранения в cookie специального маркера, который остается действительным в течение двух недель. Однако в данном случае указывается, что маркер должен оставаться действительным в течение четырех недель (2 419 200 секунд).

Маркер, сохраняемый в cookie, конструируется из имени пользователя, пароля, даты истечения срока хранения и секретного ключа. Вся эта информация шифруется с применением алгоритма MD5. По умолчанию секретный ключ имеет значение SpringSecured, но здесь ему явно было присвоено значение testKey.

Достаточно просто. Теперь, когда поддержка функции «запомнить меня» включена, необходимо обеспечить некоторый способ, с помощью которого пользователи могли бы потребовать запомнить их. Для этого в запрос следует включить параметр _spring_security_ remember_me. С этим прекрасно справляется простой флажок в форме аутентификации:

```
<input id="remember_me" name="_spring_security_remember_me"
        type="checkbox"/>
<label for="remember_me" class="inline">Remember me</label>
```

До настоящего момента все внимание уделялось обеспечению безопасности на уровне веб-запросов. Поскольку фреймворк Since Spring Security чаще используется для обеспечения безопасности веб-приложений, многие забывают, что он способен обеспечивать безопасность на уровне

вызовов методов. Поэтому перейдем к изучению поддержки безопасности в Spring Security на уровне методов.

5 Защита методов

Как уже упоминалось выше, безопасность – это аспектно-ориентированное понятие. Поэтому в основе обеспечения безопасности на уровне методов в Spring Security лежит Spring AOP. Но вам едва ли придется напрямую сталкиваться с аспектами Spring Security. Все детали использования AOP, связанные с обеспечением безопасности методов, скрыты в единственном элементе: <global-method-security>. Ниже демонстрируется типичный пример использования <globalmethod-security>.

<global-method-security secured-annotations="enabled" />

Он настраивает Spring Security на обеспечение безопасности методов, отмеченных собственной аннотацией Spring Security: @Secured . Это лишь один из четырех возможных способов защиты методов, поддерживаемых фреймворком Spring Security:

- с помощью аннотации @Secured;
- с помощью аннотации JSR-250: @RolesAllowed;
- с помощью аннотаций Spring, проверяющих условия до и после вызова метода; посредством сопоставления метода с одним или более множествами точек внедрения.

Рассмотрим каждый из способов по очереди.

5.1 Защита методов с помощью аннотации @Secured

Когда в атрибуте secured-annotations элемента <global-methodsecurity> указывается значение enabled, создается множество точек внедрения, благодаря чему аспекты Spring Security получают возможность обернуть методы компонентов, отмеченные аннотацией @Secured. Например:

```
@Secured("ROLE_USER")
public void addUser(User user) {
// ...
}
```

Аннотация @Secured принимает массив строк. Каждая строка определяет привилегию, которой должен обладать пользователь, чтобы вызвать аннотированный метод. В данном случае фреймворку Spring Security сообщается, что он не должен позволять вызывать метод saveUser(), если пользователь не обладает привилегией ROLE_USER.

Если аннотации @Secured передается несколько значений, для вызова защищенного метода пользователь должен обладать хотя бы одной из указанных привилегий. Например, следующий пример использования аннотации @Secured требует, чтобы пользователь обладал привилегией ROLE_USER или ROLE_ADMIN:

```
@Secured({"ROLE_USER", "ROLE_ADMIN"})
 public void addUser(User user) {
 // ... }
```

Если защищенный метод попытается вызвать неавторизованный пользователь или пользователь, не обладающий необходимыми привилегиями, аспект, обертывающий метод, возбудит одно из исключений, реализованных в Spring Security (возможно, подкласс AuthenticationException или AccessDeniedException). В конечном счете исключение должно быть обработано приложением. Если вызов защищенного метода произойдет в процессе обработки веб-запроса, исключение будет автоматически обработано фильтрами Spring Security. Во всех остальных случаях вы должны будете написать собственный обработчик исключения.

Единственный недостаток аннотации @Secured – в том, что она является аннотацией Spring. Те, кто предпочитает использовать стандартные аннотации, могут использовать аннотацию @RolesAllowed.

5.2 Использование аннотации JSR-250 @RolesAllowed

Аннотация @RolesAllowed является практически полным эквивалентом аннотации @Secured. Единственное существенное отличие – в том, что аннотация @RolesAllowed является одной из стандартных аннотаций Java и определена в спецификации JSR-250.

Эти различия в большей степени относятся к политической стороне дела, чем к технической. Перевес в пользу стандартной аннотации @RolesAllowed может быть обусловлен необходимостью использования программного кода в контексте других фреймворков или API, обрабатывающих аннотацию. Как бы то ни было, в случае выбора аннотации @RolesAllowed необходимо включить ее поддержку, определив значение enabled в атрибуте jsr250-annotations элемента <global-method-security>:

```
<global-method-security jsr250-annotations="enabled" />
```

Хотя в данном примере был определен только атрибут jsr250- annotations, следует отметить, что он не исключает возможности определения атрибута secured-annotations. Эти два атрибута могут указываться одновременно. Более того, их можно даже использовать совместно с аннотациями фреймворка Spring, выполняемыми до и после вызова метода, о которых рассказывается далее.

5.3 Защита с помощью аннотаций, выполняемых до и после вызова

Аннотации @Secured и @RolesAllowed позволяют решить поставленную задачу, предотвращая возможность вызова методов неавторизованными пользователями, но это все, на что они способны. Иногда бывает необходимо реализовать более интересные ограничения, основанные не только на определении наличия некоторых приви-легий у пользователя.

В Spring Security 3.0 появилось несколько новых аннотаций, позволяющих использовать выражения на языке SpEL для реализации более сложных ограничений на доступ к методам. Эти новые аннотации перечислены в таблице ниже:

Аннотация	Описание
@PreAuthorize	Ограничивает доступ к методам перед их вызовом, опираясь на результат вычисления выражения
@PostAuthorize	Позволяет вызывать методы, но возбуждает исключение, если выражение возвращает значение false
@PostFilter	Позволяет вызывать методы, но фильтрует его результаты в соответствии со значением выражения
@PreFilter	Позволяет вызывать методы, но фильтрует входные данные перед фактическим вызовом метода

Примеры использования каждой из них будут представлены чуть ниже. Но прежде чем использовать, их поддержку необходимо включить в элементе <global-method-security>, указав значение enabled в атрибуте pre-post-annotations :

<global-method-security pre-post-annotations="enabled" />

После включения поддержки описываемых аннотаций можно приступать к их использованию для защиты методов. Начнем с ан- нотации @PreAuthorize.

5.3.1 Проверка условия перед вызовом метода

На первый взгляд может показаться, что аннотация @PreAuthorize является всего лишь эквивалентом аннотаций @Secured и @RolesAllowed, обладающим поддержкой выражений на языке SpEL. Фактически аннотацию @PreAuthorize можно использовать для ограничения доступа на основе привилегий, которыми обладает авторизованный пользователь:

```
@PreAuthorize("hasRole('ROLE_USER')")
public void addComment(Comment comment) {
// ...
}
```

Аннотация @PreAuthorize принимает строковый аргумент с выражением на языке SpEL. В примере выше используется функция hasRole(), предоставляемая фреймворком Spring Security, разрешающая доступ к методу, только если авторизованный пользователь обладает привилегией ROLE_USER.

Однако на языке SpEL можно выразить гораздо более сложные требования. Например, представьте, что необходимо ограничить длину сообщений, посылаемых обычным пользователем, 140 символами и не применять это ограничение к привилегированному пользователю. Аннотации

@Secured и @RolesAllowed оказались бы бесполезны в этом случае, а аннотация @PreAuthorize позволяет легко справиться с данной задачей:

```
@PreAuthorize("(hasRole('ROLE_USER') and #comment.text.length() <= 140) or hasRole('ROLE_PREMIUM')")
public void addComment(Comment comment) {
 // ...
}
```

Фрагмент #comment выражения является непосредственной ссылкой на параметр метода с тем же именем. Эта возможность позволяет фреймворку Spring Security исследовать параметры методов и использовать их в выражениях при принятии решения. Здесь проверяется длина свойства text объекта Comment, чтобы убедиться, что она не превышает допустимого значения, если метод вызывается обычным пользователем. А если пользователь является привилегированным пользователем, длина сообщения не имеет значения.

5.3.2 Проверка условия после вызова метода

Менее очевидный способ защиты методов заключается в проверке условия после вызова метода. Обычно в этом случае решение принимается на основе объекта, возвращаемого защищенным методом. Разумеется, это означает, что метод должен быть вызван и вернуть некоторое значение.

Помимо момента принятия решения, аннотация @PostAuthorize действует практически так же, как и аннотация @PreAuthorize. Например, предположим, что необходимо защитить метод getCommentById(), позволив возвращать объект Comment, только если он принадлежит авторизованному пользователю. Для этого можно было бы снабдить метод getCommentById() аннотацией @PostAuthorize, как показано ниже:

```
@PostAuthorize("returnObject.user.username == principal.username")
public Comment getCommentById(long id) {
// ...
}
```

Чтобы упростить доступ к объекту, возвращаемому защищенным методом, фреймворк Spring Security предоставляет имя returnObject в языке SpEL. В данном случае известно, что метод возвращает объект типа Comment, поэтому выражение обращается к его свойству user и извлекает из него значение его свойства username.

С другой стороны оператора сравнения (==) используется встроенный объект principal, из которого извлекается значение свойства username. Объект principal – это еще одно специальное встроенное имя, предоставляемое фреймворком Spring Security, которое ссылается на главный объект, представляющий текущего авторизованного пользователя.

Если объект User ссылается на объект Comment, значение свойства username которого совпадает со значением свойства username объекта principal, то объект Comment будет возвращен вызывающей программе. Иначе будет

возбуждено исключение AccessDeniedException и вызывающая программа не получит объекта Spittle. Имейте в виду, что, в отличие от методов, снабженных аннотацией @PreAuthorize, методы с аннотацией @PostAuthorize будут сначала выполнены, и только потом будет произведена проверка. Это означает, что перед применением аннотации @PostAuthorize необходимо убедиться в отсутствии побочных эффектов в методе, которые могут привести к нарушению системы безопасности.

5.3.3 Фильтрация после вызова метода

Иногда бывает необходимо защитить не сам метод, а данные, возвращаемые этим методом. Например, представьте, что необходимо вернуть список сообщений пользователю, но при этом список должен содержать только сообщения, которые пользователь может удалить. В этом случае можно было бы аннотировать метод, как показано ниже:

```
@PreAuthorize("hasRole('ROLE_USER")
@PostFilter("filterObject.user.username == principal.name")
public List<Comment> getABunchOfComments() {
 ...
}
```

Здесь аннотация @PreAuthorize позволяет вызывать метод лишь пользователям с привилегией ROLE_USER. Если пользователь пройдет эту проверку, метод будет вызван и вернет список сообщений. Но аннотация @PostFilter отфильтрует этот список, оставив в нем только сообщения (объекты Comment), принадлежащие пользователю.

Имя filterObject в выражении ссылается на отдельные элементы в списке (которые, как мы знаем, являются объектами Comment), возвращаемом методом. Если свойство username свойства user в этом объекте совпадает с именем авторизованного пользователя (principal.name в выражении), тогда элемент остается в отфильтрованном списке. Иначе он удаляется.

6 Заключение

Безопасность – критически важный аспект многих приложений. Фреймворк Spring Security предоставляет простые, гибкие и мощные механизмы, позволяющие обезопасить приложение.

С помощью последовательности сервлет-фильтров Spring Security может контролировать доступ к веб-ресурсам, включая контроллеры Spring MVC. А с помощью аспектов Spring Security можно организовать защиту методов. Благодаря наличию в Spring Security конфигурационного пространства имен вам не придется непосредственно сталкиваться с фильтрами или аспектами, а настройка безопасности получится краткой и выразительной.

Что касается аутентификации пользователей, Spring Security предлагает несколько вариантов. Выше было показано, как настраивать аутентификацию с

использованием хранилища информации о пользователях в памяти, в реляционной базе данных и на сервере каталогов LDAP.