## **Сохранение SecurityContext между запросами**

В зависимости от типа приложения может потребоваться стратегия хранения контекста безопасности между операциями пользователя. В типичном веб-приложении, пользователь регистрируется один раз и впоследствии идентифицируется по Id сессии. Сервер кэширует информацию принципала в течение сессии. В Spring Security, ответственность за хранение SecurityContext между запросами ложится на SecurityContextPersistenceFilter, который по умолчанию хранит контекст как атрибут HttpSession между запросами HTTP. Он восстанавливает контекст в SecurityContextHolder для каждого запроса и, что самое главное, очищает SecurityContextHolder после завершения запроса. Вы не должны напрямую взаимодействовать с HttpSession для решения задач безопасности. Для этого нет никаких оснований, всегда вместо этого используйте SecurityContextHolder.

Многие другие типы приложений(например, RESTful веб-сервисы без сохранения состояния) не используют HTTP сессии и будут требовать аутентификации при каждом запросе. Тем не менее, все равно очень важно, чтобы SecurityContextPersistenceFilter входил в цепочку, чтобы SecurityContextHolder очищался после каждого запроса.

**Примечание** В приложение, которое получает конкурентные запросы в одной сессии, один и тот же экземпляр SecurityContext будет разделяться между потоками. Несмотря на то, что используется ThreadLocal, это будет один и тот же экземпляр, что извлекается из HttpSession для каждого потока. Это будет иметь значение если вы захотите временно изменить контекст в выполняющемся потоке. Если Вы просто используете SecurityContextHolder.getContext().SetAuthentication(anAuthentication), то объект Authentication изменится во всех параллельных потоках, которые разделяют один и тот же экземпляр SecurityContext. Вы можете настроить поведение SecurityContextPersistenceFilter чтобы он создавал совершенно новый экземпляр SecurityContext для каждого запроса, чтобы один поток не влиял на другой. Альтернативный вариант, это создать новый экземпляр в том месте где требуется временно изменить контекст. Метод SecurityContextHolder.createEmptyContext() всегда возвращает новый экземпляр контекста.

# Управление доступом (Авторизация) в Spring Security

В Spring Security основным интерфейсом, отвечающим за принятие решений в области контроля доступа, является AccessDecisionManager. У него имеется решающий метод, который принимает объект Authentication, представляющий принципала запрашивающего доступ, "объект безопасности" (см. ниже), а также список атрибутов метаданных безопасности, которые применяются к объекту (например, список ролей которым разрешен доступ).

## **Безопасность и Советы AOP**

Если вы хорошо знакомы с [AOP](https://ru.wikipedia.org/wiki/AOP), то должны знать что существуют различные виды советов: *before*, *after*, *throws* и *around*. Совет *around* очень полезен, потому что советчик может выбирать, следует или нет осуществить вызов метода, следует или нет изменить отклик, следует или нет пробросить исключение. Spring Security предоставляет *around* советы как для вызова методов, так и для веб-запросов. Для вызова методов совет *around* реализуется с помощью стандартного [AOP](https://ru.wikipedia.org/wiki/AOP) модуля Spring'а, а для веб-запросов с помощью стандартного фильтра.

Для тех, кто не знаком с [AOP](https://ru.wikipedia.org/wiki/AOP), главное понять, что Spring Security может защищать вызовы методов так же хорошо, как и веб-запросы. Для большинства людей важно обеспечить безопасность вызова методов на уровне сервисов. Потому что на уровне сервисов сосредоточено большинство бизнес-логики нынешнего поколения J2EE приложений. Если вам просто нужно обеспечить безопасность при вызове методов на уровне сервисов, то стандартный Spring [AOP](https://ru.wikipedia.org/wiki/AOP) будет весьма уместным. Если вам нужно обеспечить безопасность непосредственно объектов предметной области, то вероятно следует рассмотреть вариант использования [AspectJ](https://ru.wikipedia.org/wiki/AspectJ" \o "w:AspectJ).

Вы можете выбрать метод авторизации с использованием [AspectJ](https://ru.wikipedia.org/wiki/AspectJ" \o "w:AspectJ) или Spring [AOP](https://ru.wikipedia.org/wiki/AOP), или вы можете выбрать авторизацию веб-запросов с помощью фильтров. Вы можете использовать ноль, один, два или даже все три эти подхода вместе. Типичный шаблон использования выглядит следующим образом: использование авторизации для части веб-запросов, в сочетании авторизаций вызова методов уровня сервисов с помощью Spring AOP .

## **Защищенные Объекты и AbstractSecurityInterceptor**

Что такое "защищенный объект" в общем смысле? Spring Security использует этот термин для обозначения любого объекта, к которому могут применяться механизмы обеспечения безопасности (например, авторизация). Наиболее распространенными примерами являются вызовы методов и веб-запросы.

Каждый поддерживамый защищенный объект имеет свой собственный класс перехватчик, который является подклассом AbstractSecurityInterceptor. Важно отметить, что в тот момент, когда вызывается AbstractSecurityInterceptor, SecurityContextHolder будет содержать корректный объект Authentication, если принципал прошел аутентификацию.

AbstractSecurityInterceptor обеспечивает последовательный рабочий процесс для обработки запросов к защищенному объекту, обычно:

1. Поиск "конфигурационных атрибутов", связанных с текущим запросом
2. Отправка защищенного объекта, текущего Authentication объекта и конфигурационных атрибутов объекту AccessDecisionManager для принятия решение об авторизации
3. Опциональное изменение Authentication под которым происходит вызов
4. Разрешение выполнения вызова безопасного объекта (при условии что доступ был разрешен)
5. При наличии в конфигурации, вызов AfterInvocationManager, сразу после возврата из вызова защищенного объекта.

### **Что такое конфигурационные атрибуты**

"Конфигурационный атрибут" можно рассматривать как String, которая имеет особое значение для классов, используемых AbstractSecurityInterceptor. Они представляются интерфейсом ConfigAttribute каркаса Spring Security. Они могут быть простыми именами ролей или иметь более сложный смысл, в зависимости от возможностей реализации AccessDecisionManager. AbstractSecurityInterceptor настроен так, чтобы использовать SecurityMetadataSource, который он использует для поиска атрибутов защищенного объекта. Обычно эта конфигурация будет скрыта от пользователя. Конфигурационные атрибуты устанавливаются в виде аннотаций для защищенных методов или атрибутов доступа для защищенных URL. Например, когда мы видим что-то вроде <intercept-url pattern='/secure/\*\*' access='ROLE\_A,ROLE\_B'/> в конфигурационном файле, это говорит о том, что конфигурационные атрибуты ROLE\_A и ROLE\_B должны применяться к веб-запросам, соответствующим заданному шаблону. На практике, с конфигурацией по умолчанию для AccessDecisionManager, это будет означает, что доступ получит каждый, кто имеет GrantedAuthority совпадающий с одним из этих двух атрибутов. Строго говоря, это просто атрибуты и их интерпретация зависит от реализации AccessDecisionManager. Префикс ROLE\_ используется как маркер, чтобы показать что эти атрибуты обозначают роли пользователей и будут использованы классом RoleVoter каркаса Spring Security. Это имеет смысл только тогда, когда используется AccessDecisionManager основанный на voter. В следующей главе мы увидим как реализован AccessDecisionManager.

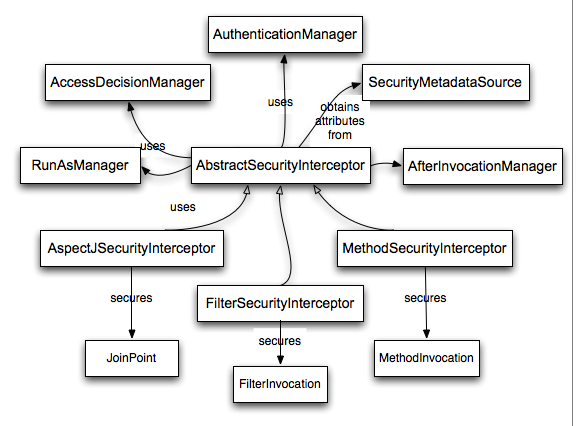
### **RunAsManager**

Предположим, что AccessDecisionManager выдает разрешение на выполнение какому-либо запросу, тогда AbstractSecurityInterceptor, как правило, просто передаст данный запрос на обработку. Надо сказать, что в редких случаях пользователи могут захотеть заменить один Authentication объект в SecurityContext на другой Authentication объект, который будет обработан AccessDecisionManager с помощью вызова RunAsManager. Это может оказаться полезным в некоторых нетиповых ситуациях, например, если методу уровня сервисов нужно вызвать удаленную систему и идентифицироваться в ней с другими данными. Так как Spring Security автоматически распространяет аутентификационные данные с одного сервера на другой (предполагается, что вы пользуетесь правильно настроенными RMI или HttpInvoker клиентами протокола удаленного вызова), то эта возможность может быть полезной.

### **AfterInvocationManager**

После того как безопасный объект отработал и произошел возврат из него - что может означать завершение вызова метода или работы цепочки фильтров - AbstractSecurityInterceptor получает последний шанс для обработки вызова. На данном этапе AbstractSecurityInterceptor может быть заинтересован в возможности изменить возвращаемый объект. Мы можем захотеть чтобы это произошло в том случае, если решение об авторизации не могло быть принято «на пути к» вызову безопасного объекта. Будучи полностью настраиваемым с помощью плагинов, AbstractSecurityInterceptor передаст управление AfterInvocationManagerчтобы изменить возвращаемый объект, если это потребуется. Этот класс может даже полностью заменить возвращаемый объект, или выбросить исключение, или же вообще ничего не изменять, в зависимости от потребностей.

AbstractSecurityInterceptor и связанные с ним объекты показаны на рисунке 5.1, «Модель перехватчиков системы безопасности и «защищенного объекта».

[](https://ru.wikibooks.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:SpringSecurity_Interceptor_Model.png)

### **Расширение модели защищенного объекта**

Только те разработчики, которые рассматривают возможность создания полностью нового способа перехвата и авторизации запросов, должны напрямую использовать защищенные объекты. Например, можно было бы построить новый защищенный объект для системы обмена сообщениями. Все что требует обеспечения безопасности и обеспечивает способ перехвата вызовов (на подобие семантики AOP совета *around*) может быть превращено в защищенный объект. Можно сказать, что в большинстве Spring-приложений можно легко и прозрачно использовать три, в настоящее время поддержаващихся, типа защищенных объектов (MethodInvocation AOP альянса, AspectJ JoinPoint и FilterInvocation для веб-запросов).

# Локализация

Spring Security поддерживает локализацию сообщений в исключениях, которые скорее всего увидят конечные пользователи. Если ваше приложение разработано для Англоговорящих пользователей, то по умолчанию вы не должны ничего делать, все сообщения Spring Security написаны на английском языке. Если вы должны поддерживать локализацию для других языков, то все что вам нужно знать, содержится в этом разделе.

Все сообщения исключений могут быть локализованы, включая сообщения, связанные с отказом аутентификации и запрета доступа (отказы авторизации). Исключения и логи, которые предназначены для разработчиков или лиц ответсвенных за развертывание приложения (включая некорректные атрибуты, нарушения контракта интерфейса, использование некорректных конструкторов, проверки во время запуска, логгирование режима отладки) и т.д, не локализуются, а вместо этого явно написаны в коде Spring Security на английском языке.

В поставляемом файле spring-security-core-xx.jar, вы найдете пакет org.springframework.security, который в свою очередь содержит файл messages.properties. На него должна быть установлена ссылка в вашем ApplicationContext, поскольку классы Spring Security реализуют Spring интерфейс MessageSourceAware и ожидают, что в преобразователь сообщений (message resolver) будет включена зависимость из контекста приложения в момент его запуска.. Обычно все, что Вы должны сделать, это зарегистрировать bean в контексте приложения, который будет ссылаться на сообщения. Ниже приведен пример:

**<bean** id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource"**>**

**<property** name="basename" value="org/springframework/security/messages"**/>**

**</bean>**

Messages.properties назван в соответствии со стандартом пакетов ресурсов и содержит сообщения на языке по умолчанию, поддерживаемые Spring Security. Этот файл по умолчанию на английском языке. Если вы не зарегистрировали источник сообщений, Spring Security все равно будет работать правильно, и показывать английскую версию сообщений жестко заданных в коде.

Если вы хотите настроить файл messages.properties или поддерживать другие языки, то вы должны скопировать файл, переименовать в соответствии с правилами и зарегистрировать его в приведенном выше определении компонента. Файл содержит небольшое количество ключей сообщений, поэтому локализация не рассматриваться как основная инициатива. Если вы выполните локализацию этого файла, пожалуйста подумайте о том, чтобы поделиться вашей работой с сообществом. Для этого войдите в JIRA, создайте задачу и прикрепите к ней вашу версию переименованного и локализованного файла messages.properties.

В завершении нашего обсуждения локализации: Существует Spring ThreadLocal известеный как org.springframework.context.i18n.LocaleContextHolder. Вы должны установить LocaleContextHolder так, чтобы он представлял предпочтительную Locale каждого пользователя. Spring Security будет пытаться найти сообщения в соответствии с настройками Locale, который берется из ThreadLocal. Пожалуйста, обратитесь к документации Spring Framework для более подробной информациию об использовании LocaleContextHolder.