Capítulo 8. Olá, Spring Security

Esta seção aborda a configuração mínima para usar o Spring Security com Spring Boot. O aplicativo completo pode ser encontrado em samples/boot/helloworld.

Para sua conveniência, você pode baixar um aplicativo minimalista do Spring Boot + Spring Security clicando aqui.

8.1. Atualizando Dependências

O único passo necessário é atualizar as dependências usando Maven ou Gradle.

8.2. Iniciando o Hello Spring Security Boot

Agora você pode executar a aplicação Spring Boot usando o objetivo run do Plugin Maven.

O exemplo a seguir mostra como fazer isso (junto com o início da saída gerada ao executar o comando):

Exemplo 46. Executando a Aplicação Spring Boot

```
$ ./mvn spring-boot:run
...
INFO 23689 --- [  restartedMain] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 8e557245-73e2-4286-969a-ff57fe326336

...
```

8.3. Configuração Automática do Spring Boot

O Spring Boot automaticamente:

- **Habilita a configuração padrão do Spring Security**, que cria um filtro servlet como um bean denominado springSecurityFilterChain. Esse bean é responsável por toda a segurança (proteção das URLs da aplicação, validação do nome de usuário e senha enviados, redirecionamento para o formulário de login, entre outros) dentro da sua aplicação.
- **Cria um bean UserDetailsService** com o nome de usuário user e uma senha gerada aleatoriamente, que é registrada no console.
- **Registra o filtro** com o bean denominado springSecurityFilterChain no contêiner Servlet para cada solicitação.

Embora o Spring Boot não configure muitos aspectos, ele realiza várias ações importantes. Um resumo das funcionalidades é o seguinte:

- Exige um usuário autenticado para qualquer interação com a aplicação.
- Gera um formulário de login padrão para você.
- Permite que o usuário com o nome de usuário `user` e uma senha registrada no console se autentique por meio de autenticação baseada em formulário (no exemplo anterior, a senha é `8e557245-73e2-4286-969a-ff57fe326336`).
- Protege o armazenamento de senhas com BCrypt.
- Permite que o usuário faça logout.
- Prevenção contra ataques CSRF.
- Proteção contra Fixação de Sessão.
- Integração de Cabeçalhos de Segurança:
  - HTTP Strict Transport Security para requisições seguras.
  - Integração de `X-Content-Type-Options`.
  - Controle de Cache (pode ser substituído mais tarde pela sua aplicação para permitir o cache de recursos estáticos).
  - Integração de `X-XSS-Protection`.
  - Integração de `X-Frame-Options` para ajudar a prevenir o Clickjacking.
- Integra-se com os seguintes métodos da API Servlet:
  - HttpServletRequest#getRemoteUser()
  - HttpServletRequest#html#getUserPrincipal()
  - HttpServletRequest#html#isUserInRole(java.lang.String)
  - HttpServletRequest#html#login(java.lang.String, java.lang.String)
  - HttpServletRequest#html#logout()

Capítulo 9. Segurança Servlet: A Visão Geral

Esta seção discute a arquitetura de alto nível do Spring Security em aplicações baseadas em Servlet. Desenvolvemos essa compreensão de alto nível nas seções de **Autenticação**, **Autorização** e **Proteção contra Exploits** do referencial.

9.1. Uma Revisão dos Filtros

O suporte do Spring Security para Servlets é baseado em Filtros Servlet, portanto, é útil primeiro entender o papel dos Filtros de forma geral. A imagem abaixo mostra a camada típica dos manipuladores para uma única solicitação HTTP.

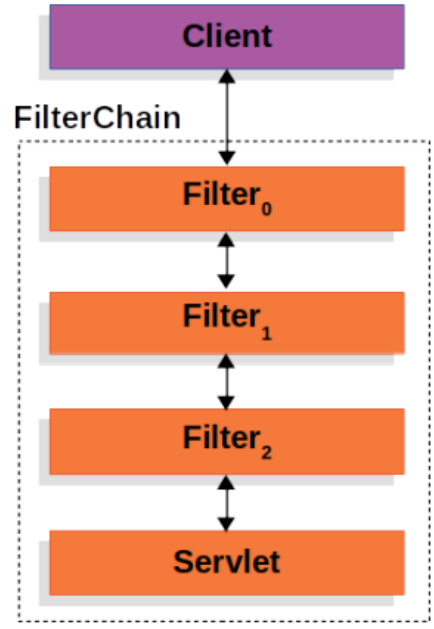


Figure 1. FilterChain

O cliente envia uma solicitação para a aplicação, e o contêiner cria um **FilterChain** que contém os **Filtros** e o **Servlet** que devem processar o `HttpServletRequest` com base no caminho da URI da solicitação. Em uma aplicação Spring MVC, o Servlet é uma instância do `DispatcherServlet`. No máximo, um Servlet pode processar uma única solicitação `HttpServletRequest` e `HttpServletResponse`. No entanto, mais de um Filtro pode ser usado para:

- **Prevenir que os Filtros ou o Servlet subsequentes sejam invocados**. Nesse caso, o Filtro geralmente escreverá o `HttpServletResponse`.
- **Modificar o `HttpServletRequest` ou `HttpServletResponse` usados pelos Filtros e Servlet subsequentes**.

O poder do Filtro vem do **FilterChain** que é passado para ele.

Exemplo 47. Exemplo de Uso do FilterChain

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    // do something before the rest of the application
    chain.doFilter(request, response); // invoke the rest of the application
    // do something after the rest of the application
}
```

Como um Filtro só afeta os Filtros e o Servlet subsequentes, a ordem em que cada Filtro é invocado é extremamente importante.

9.2. DelegatingFilterProxy

O Spring fornece uma implementação de Filtro chamada **DelegatingFilterProxy**, que permite fazer a ponte entre o ciclo de vida do contêiner Servlet e o **ApplicationContext** do Spring. O contêiner Servlet permite registrar Filtros usando seus próprios padrões, mas não reconhece os **Beans** definidos no Spring. O **DelegatingFilterProxy** pode ser registrado por meio dos mecanismos padrão do contêiner Servlet, mas delega todo o trabalho para um Bean do Spring que implementa a interface Filter.

Aqui está uma imagem de como o **DelegatingFilterProxy** se encaixa nos Filtros e no **FilterChain**.

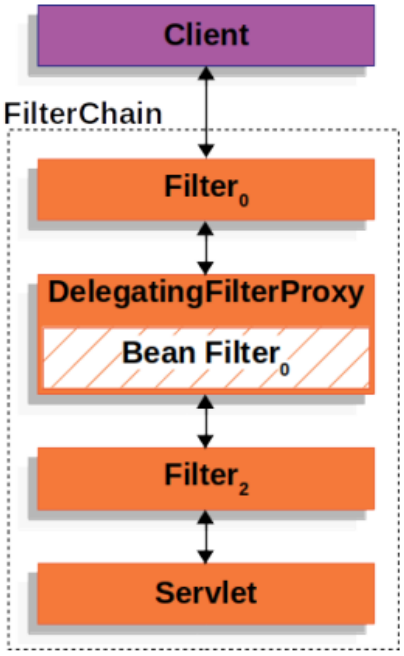


Figure 2. DelegatingFilterProxy

O **DelegatingFilterProxy** procura o Bean Filter0 no **ApplicationContext** e, em seguida, invoca o Bean Filter0. O pseudocódigo do **DelegatingFilterProxy** pode ser visto abaixo.

Exemplo 48. Pseudocódigo do DelegatingFilterProxy

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) {
    // Lazily get Filter that was registered as a Spring Bean
    // For the example in DelegatingFilterProxy delegate is an instance of Bean
Filter0
    Filter delegate = getFilterBean(someBeanName);
    // delegate work to the Spring Bean
    delegate.doFilter(request, response);
}
```

Outro benefício do **DelegatingFilterProxy** é que ele permite adiar a busca pelas instâncias de Beans de Filtro. Isso é importante porque o contêiner precisa registrar as instâncias de Filtro antes de iniciar. No entanto, o Spring normalmente usa o **ContextLoaderListener** para carregar os Beans do Spring, o que não será feito até depois de as instâncias de Filtro precisarem ser registradas.

9.3. FilterChainProxy

O suporte do Spring Security para Servlets está contido no **FilterChainProxy**. O **FilterChainProxy** é um Filtro especial fornecido pelo Spring Security que permite delegar para várias instâncias de Filtro por meio do **SecurityFilterChain**. Como o **FilterChainProxy** é um Bean, ele geralmente é encapsulado em um **DelegatingFilterProxy**.

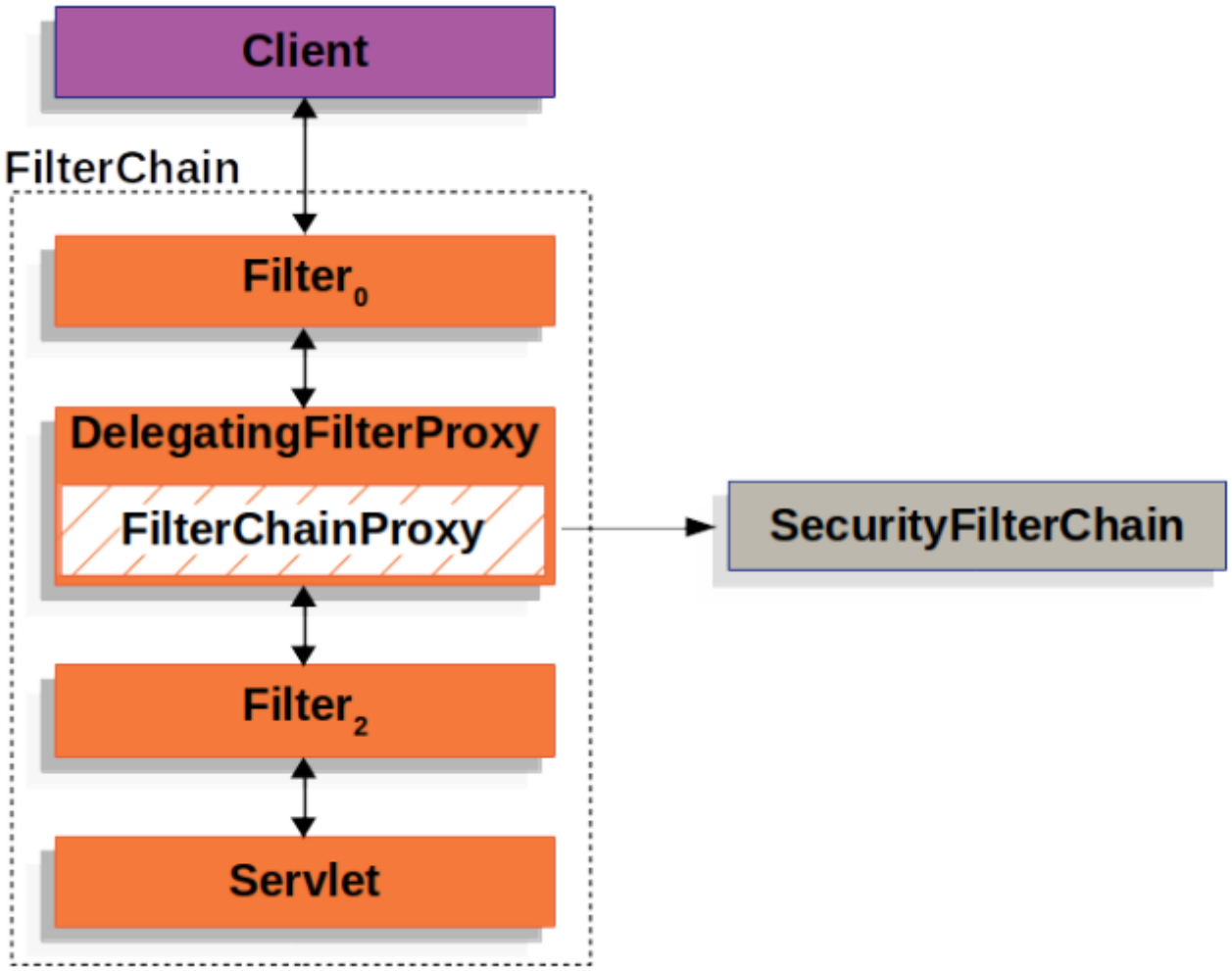


Figure 3. FilterChainProxy

9.4. SecurityFilterChain

O **SecurityFilterChain** é usado pelo **FilterChainProxy** para determinar quais Filtros do Spring Security devem ser invocados para esta solicitação.

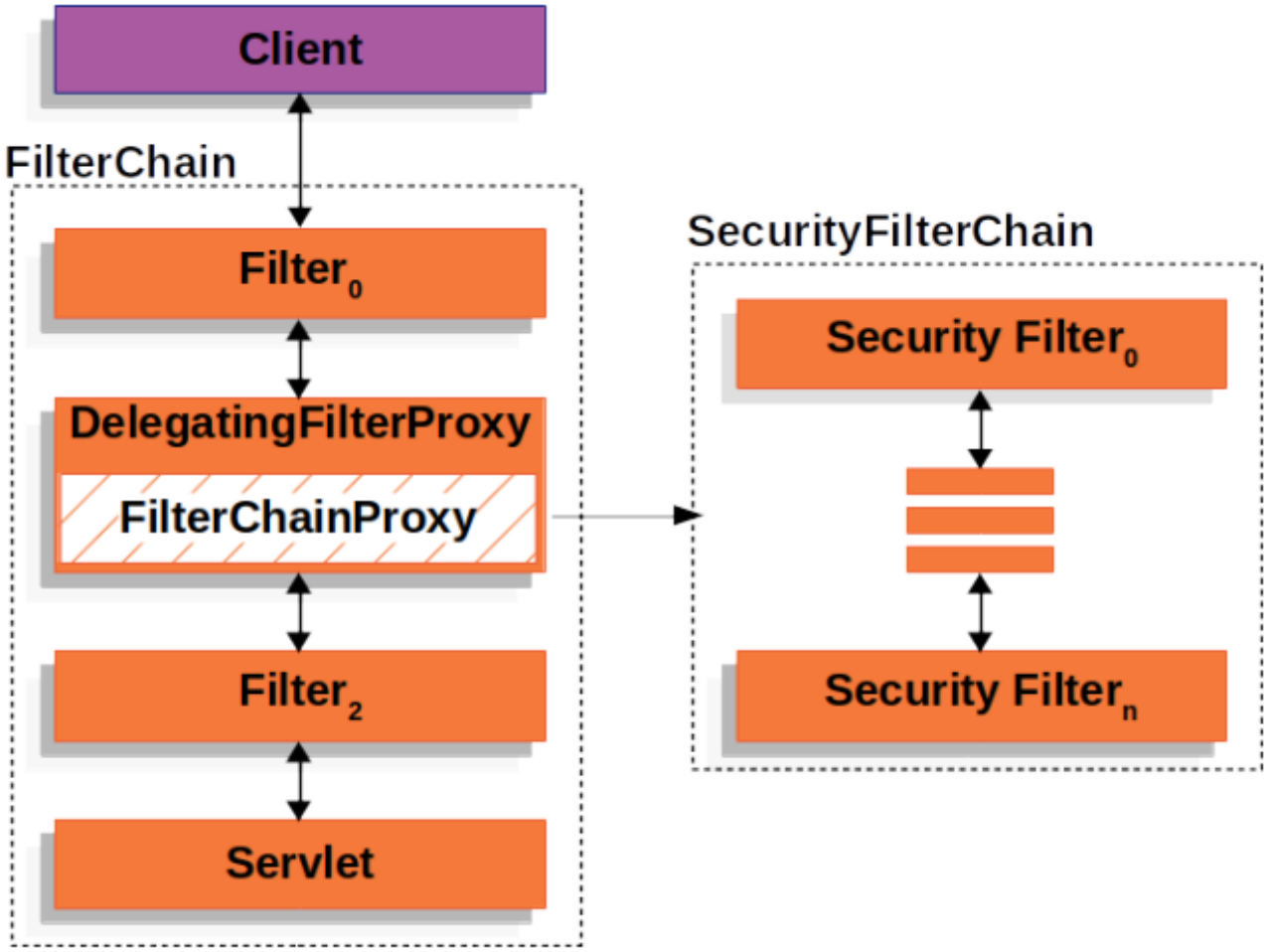


Figure 4. SecurityFilterChain

Os Filtros de Segurança no **SecurityFilterChain** são normalmente Beans, mas são registrados com o **FilterChainProxy** em vez de com o **DelegatingFilterProxy**. O **FilterChainProxy** oferece várias vantagens em relação ao registro direto no contêiner Servlet ou ao **DelegatingFilterProxy**. Primeiramente, ele fornece um ponto de partida para todo o suporte Servlet do Spring Security. Por esse motivo, se você estiver tentando solucionar problemas com o suporte Servlet do Spring Security, adicionar um ponto de depuração no **FilterChainProxy** é um ótimo lugar para começar.

Em segundo lugar, como o **FilterChainProxy** é central para o uso do Spring Security, ele pode realizar tarefas que não são vistas como opcionais. Por exemplo, ele limpa o **SecurityContext** para evitar vazamentos de memória. Ele também aplica o **HttpFirewall** do Spring Security para proteger as aplicações contra certos tipos de ataques.

Além disso, ele oferece mais flexibilidade para determinar quando um **SecurityFilterChain** deve ser invocado. Em um contêiner Servlet, os Filtros são invocados com base apenas na URL. No entanto, o **FilterChainProxy** pode determinar a invocação com base em qualquer coisa no **HttpServletRequest**, utilizando a interface **RequestMatcher**.

De fato, o **FilterChainProxy** pode ser usado para determinar qual **SecurityFilterChain** deve ser utilizado. Isso permite fornecer uma configuração totalmente separada para diferentes partes da sua aplicação.

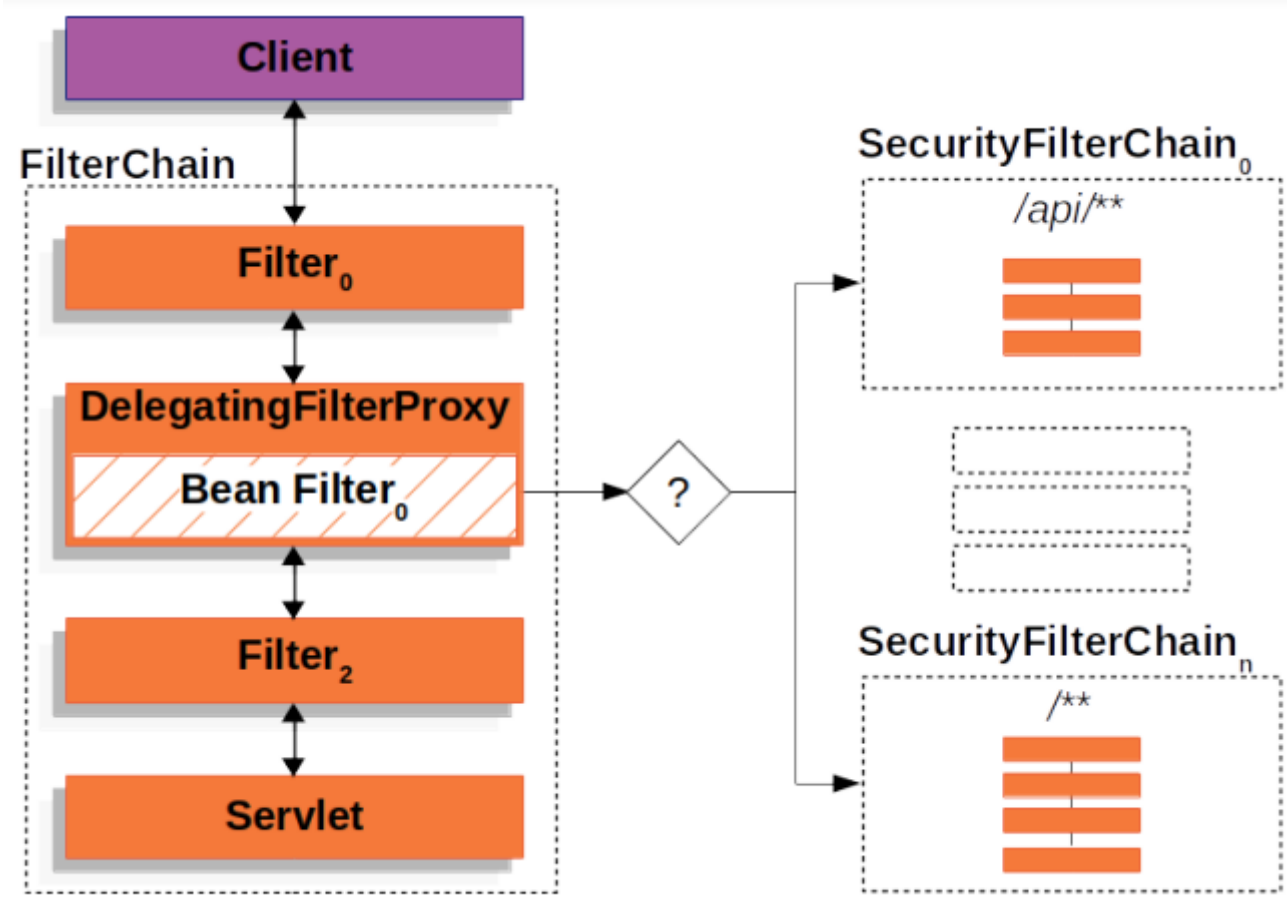


Figure 5. Multiple SecurityFilterChain

Na figura dos **Multiple SecurityFilterChain**, o **FilterChainProxy** decide qual **SecurityFilterChain** deve ser utilizado. Apenas o primeiro **SecurityFilterChain** que corresponder será invocado. Se uma URL como /api/messages/ for solicitada, ela corresponderá primeiro ao padrão de /api/** do **SecurityFilterChain0**, então apenas o **SecurityFilterChain0** será invocado, mesmo que também corresponda ao **SecurityFilterChainn**. Se uma URL como /messages/ for solicitada, ela não corresponderá ao padrão de /api/** do **SecurityFilterChain0**, então o **FilterChainProxy** continuará tentando cada **SecurityFilterChain**. Supondo que nenhum outro **SecurityFilterChain** corresponda, o **SecurityFilterChainn** será invocado.

Observe que o **SecurityFilterChain0** tem apenas três instâncias de Filtros de segurança configuradas. No entanto, o **SecurityFilterChainn** tem quatro instâncias de Filtros de segurança configuradas. É importante notar que cada **SecurityFilterChain** pode ser único e configurado isoladamente. De fato, um **SecurityFilterChain** pode ter zero Filtros de segurança se a aplicação quiser que o Spring Security ignore certas requisições.

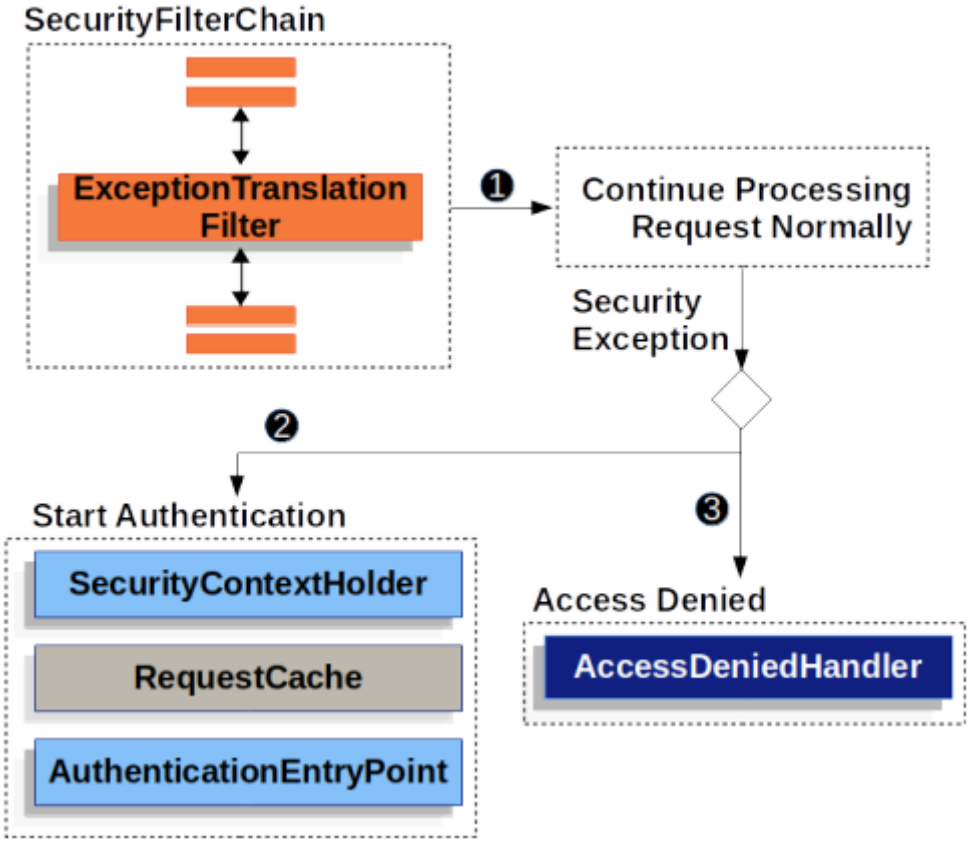
9.5. Filtros de Segurança

Os **Filtros de Segurança** são inseridos no **FilterChainProxy** com a API **SecurityFilterChain**. A ordem dos Filtros é importante. Normalmente, não é necessário saber a ordem dos Filtros do Spring Security. No entanto, há momentos em que é benéfico conhecer a ordem.

Abaixo está uma lista abrangente da ordem dos Filtros de Segurança do Spring Security:

- **ChannelProcessingFilter**
- **ConcurrentSessionFilter**
- **WebAsyncManagerIntegrationFilter**
- **SecurityContextPersistenceFilter**
- **HeaderWriterFilter**
- **CorsFilter**
- **CsrfFilter**
- **LogoutFilter**
- **OAuth2AuthorizationRequestRedirectFilter**
- **Saml2WebSsoAuthenticationRequestFilter**
- **X509AuthenticationFilter**
- **AbstractPreAuthenticatedProcessingFilter**
- **CasAuthenticationFilter**
- **OAuth2LoginAuthenticationFilter**
- **Saml2WebSsoAuthenticationFilter**
- **UsernamePasswordAuthenticationFilter**
- **ConcurrentSessionFilter**
- **OpenIDAuthenticationFilter**
- **DefaultLoginPageGeneratingFilter**
- **DefaultLogoutPageGeneratingFilter**
- **DigestAuthenticationFilter**
- **BearerTokenAuthenticationFilter**
- **BasicAuthenticationFilter**
- **RequestCacheAwareFilter**
- **SecurityContextHolderAwareRequestFilter**
- **JaasApiIntegrationFilter**
- **RememberMeAuthenticationFilter**
- **AnonymousAuthenticationFilter**
- **OAuth2AuthorizationCodeGrantFilter**
- **SessionManagementFilter**
- **ExceptionTranslationFilter**
- **FilterSecurityInterceptor**
- **SwitchUserFilter**

9.6. Tratamento de Exceções de Segurança
O **ExceptionTranslationFilter** permite a tradução das exceções **AccessDeniedException** e **AuthenticationException** em respostas HTTP.
O **ExceptionTranslationFilter** é inserido no **FilterChainProxy** como um dos Filtros de Segurança.



1. Primeiro, o **ExceptionTranslationFilter** invoca FilterChain.doFilter(request, response) para invocar o restante da aplicação.
2. Se o usuário não estiver autenticado ou ocorrer uma **AuthenticationException**, então o processo de autenticação é iniciado:
    - O **SecurityContextHolder** é limpo.
    - O **HttpServletRequest** é salvo no **RequestCache**. Quando o usuário se autentica com sucesso, o **RequestCache** é utilizado para repetir a solicitação original.
    - O **AuthenticationEntryPoint** é usado para solicitar credenciais ao cliente. Por exemplo, ele pode redirecionar para uma página de login ou enviar um cabeçalho **WWW-Authenticate**.
3. Caso contrário, se for uma **AccessDeniedException**, o acesso é negado. O **AccessDeniedHandler** é invocado para lidar com o acesso negado.

Se a aplicação não lançar uma **AccessDeniedException** ou uma **AuthenticationException**, o **ExceptionTranslationFilter** não fará nada. O pseudocódigo para o **ExceptionTranslationFilter** seria algo assim:

```
try {
    filterChain.doFilter(request, response); ①
} catch (AccessDeniedException | AuthenticationException e) {
    if (!authenticated || e instanceof AuthenticationException) {
        startAuthentication(); ②
    } else {
        accessDenied(); ③
    }
}
```

1. Você se lembrará de **A Revisão dos Filtros** que invocar FilterChain.doFilter(request, response) é equivalente a invocar o restante da aplicação. Isso significa que, se outra parte da aplicação (ou seja, **FilterSecurityInterceptor** ou segurança por método) lançar uma **AuthenticationException** ou **AccessDeniedException**, ela será capturada e tratada aqui.
2. Se o usuário não estiver autenticado ou se for uma **AuthenticationException**, então **Iniciar Autenticação**.
3. Caso contrário, **Acesso Negado**.

Capítulo 10. Autenticação

O **Spring Security** fornece um suporte abrangente para **autenticação**. Esta seção aborda:

Componentes da Arquitetura

Esta seção descreve os principais componentes arquiteturais do Spring Security usados na autenticação baseada em **Servlet**. Se você precisar de fluxos concretos que explicam como essas peças se encaixam, consulte as seções específicas sobre mecanismos de autenticação.

- **SecurityContextHolder** – O **SecurityContextHolder** é onde o Spring Security armazena os detalhes de quem está autenticado.
- **SecurityContext** – Obtido a partir do **SecurityContextHolder**, contém a **Authentication** do usuário atualmente autenticado.
- **Authentication** – Pode ser a entrada para o **AuthenticationManager**, fornecendo as credenciais que um usuário utilizou para autenticação, ou o usuário autenticado atual a partir do **SecurityContext**.
- **GrantedAuthority** – Uma autoridade concedida ao principal na **Authentication** (por exemplo, **roles**, **scopes**, etc.).
- **AuthenticationManager** – A API que define como os **Filters** do Spring Security realizam a autenticação.
- **ProviderManager** – A implementação mais comum do **AuthenticationManager**.
- **AuthenticationProvider** – Utilizado pelo **ProviderManager** para realizar um tipo específico de autenticação.
- **Request Credentials with AuthenticationEntryPoint** – Utilizado para solicitar credenciais de um cliente (por exemplo, redirecionando para uma página de login, enviando uma resposta **WWW-Authenticate**, etc.).
- **AbstractAuthenticationProcessingFilter** – Um **Filter** base usado para autenticação. Ele fornece uma visão geral do fluxo de autenticação e de como os componentes trabalham juntos.

Mecanismos de Autenticação

- **Username and Password** – Como autenticar com **nome de usuário e senha**.
- **OAuth 2.0 Login** – Login com **OAuth 2.0**, incluindo **OpenID Connect** e OAuth 2.0 não padronizado (por exemplo, **GitHub**).
- **SAML 2.0 Login** – Autenticação usando **SAML 2.0**.
- **Central Authentication Server (CAS)** – Suporte ao **CAS (Central Authentication Server)**.
- **Remember Me** – Como lembrar um usuário após a expiração da sessão.
- **JAAS Authentication** – Autenticação usando **JAAS**.
- **OpenID** – Autenticação com **OpenID** (não confundir com **OpenID Connect**).
- **Pre-Authentication Scenarios** – Autenticação com um mecanismo externo, como **SiteMinder** ou segurança **Java EE**, mas ainda utilizando o Spring Security para autorização e proteção contra ataques comuns.
- **X509 Authentication** – Autenticação baseada em **X.509**.

10.1. SecurityContextHolder
No coração do modelo de autenticação do Spring Security está o **SecurityContextHolder**. Ele contém o **SecurityContext**.

O **SecurityContextHolder** é onde o Spring Security armazena os detalhes de quem está autenticado.
O Spring Security não se preocupa com a forma como o **SecurityContextHolder** é preenchido. Se ele contém um valor, então esse valor é usado como o usuário autenticado no momento.
A maneira mais simples de indicar que um usuário está autenticado é definir diretamente o **SecurityContextHolder**.
Exemplo 49. Definindo o SecurityContextHolder

```
SecurityContext context = SecurityContextHolder.createEmptyContext(); ①
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER"); ②
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context); ③
```

① Começamos criando um **SecurityContext** vazio. É importante criar uma nova instância de **SecurityContext** em vez de usar SecurityContextHolder.getContext().setAuthentication(authentication) para evitar condições de corrida entre múltiplas threads.
② Em seguida, criamos um novo objeto **Authentication**. O Spring Security não se preocupa com o tipo de implementação de **Authentication** definida no **SecurityContext**.
Aqui utilizamos **TestingAuthenticationToken** porque é muito simples.
Um cenário mais comum em produção é **UsernamePasswordAuthenticationToken(userDetails, password, authorities)**.
③ Por fim, definimos o **SecurityContext** no **SecurityContextHolder**. O Spring Security usará essas informações para autorização.
Se você deseja obter informações sobre o usuário autenticado, pode fazer isso acessando o **SecurityContextHolder**.

Exemplo 50. Acessando o Usuário Autenticado Atualmente

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
```

Por padrão, o **SecurityContextHolder** usa um **ThreadLocal** para armazenar esses detalhes, o que significa que o **SecurityContext** está sempre disponível para os métodos na mesma thread de execução, mesmo que o **SecurityContext** não seja explicitamente passado como argumento para esses métodos. Usar um **ThreadLocal** dessa forma é bastante seguro, desde que se tome cuidado para limpar a thread após o processamento da solicitação do principal atual. O **FilterChainProxy** do Spring Security garante que o **SecurityContext** seja sempre limpo.
Algumas aplicações não são totalmente adequadas para usar um **ThreadLocal**, devido à forma específica como elas lidam com threads. Por exemplo, um cliente **Swing** pode querer que todas as threads em uma **Java Virtual Machine** usem o mesmo contexto de segurança. O **SecurityContextHolder** pode ser configurado com uma estratégia na inicialização para especificar como o contexto deve ser armazenado. Para uma aplicação independente, você usaria a estratégia **SecurityContextHolder.MODE_GLOBAL**. Outras aplicações podem querer que threads geradas pela thread segura também assumam a mesma identidade de segurança. Isso é alcançado utilizando **SecurityContextHolder.MODE_INHERITABLETHREADLOCAL**.
Você pode mudar o modo do valor padrão **SecurityContextHolder.MODE_THREADLOCAL** de duas maneiras. A primeira é definindo uma propriedade do sistema, a segunda é chamando um método estático no **SecurityContextHolder**. A maioria das aplicações não precisará alterar o valor padrão, mas se for o caso, consulte a **JavaDoc** do **SecurityContextHolder** para saber mais.

10.2. SecurityContext

O **SecurityContext** é obtido a partir do **SecurityContextHolder**. O **SecurityContext** contém um objeto **Authentication**.

10.3. Authentication

O **Authentication** serve a dois principais propósitos dentro do Spring Security:

- Como entrada para o **AuthenticationManager** para fornecer as credenciais que o usuário forneceu para autenticação. Quando usado nesse cenário, o método **isAuthenticated()** retorna **false**.
- Representa o usuário atualmente autenticado. A **Authentication** atual pode ser obtida a partir do **SecurityContext**.

A **Authentication** contém:

- **principal**: identifica o usuário. Quando a autenticação é feita com um nome de usuário/senha, geralmente é uma instância de **UserDetails**.
- **credentials**: geralmente uma senha. Em muitos casos, ela será apagada após o usuário ser autenticado para garantir que não seja vazada.
- **authorities**: os **GrantedAuthoritys** são permissões de alto nível que o usuário recebe. Alguns exemplos são funções ou escopos.

10.4. GrantedAuthority

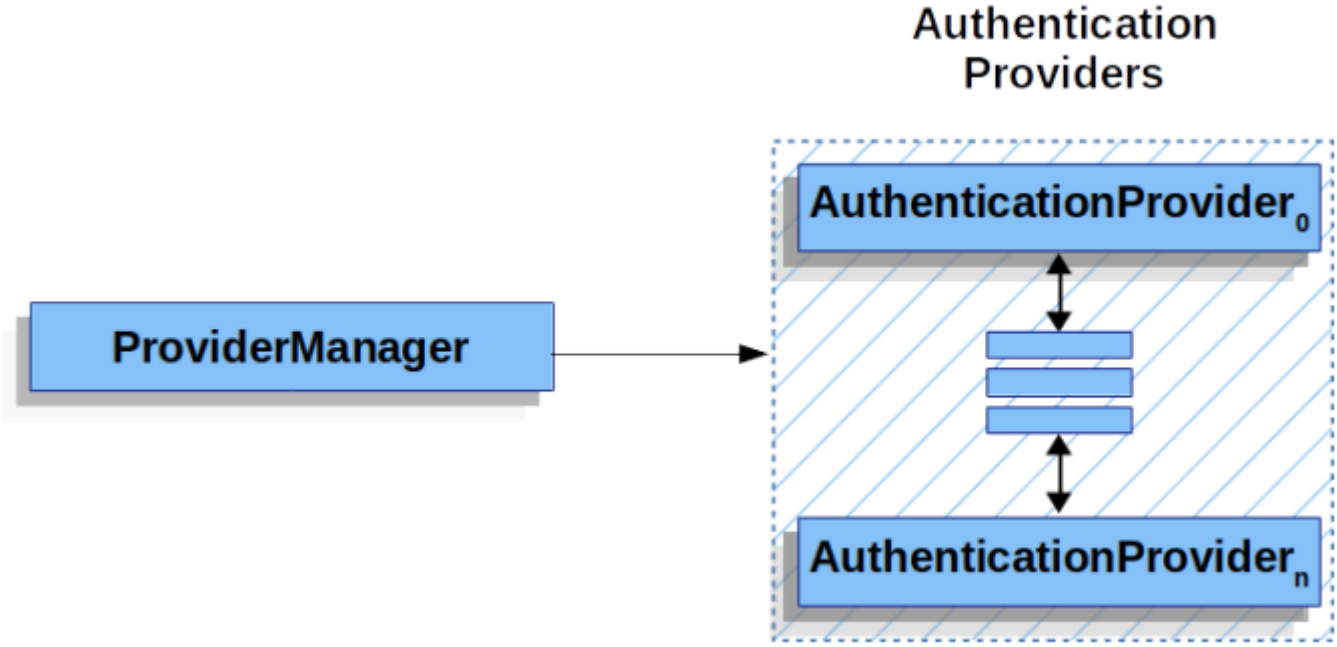
GrantedAuthoritys são permissões de alto nível que o usuário recebe. Alguns exemplos são funções ou escopos. **GrantedAuthoritys** podem ser obtidos pelo método **Authentication.getAuthorities()**. Esse método fornece uma **Collection** de objetos **GrantedAuthority**. Um **GrantedAuthority** é, como o nome sugere, uma autoridade que é concedida ao principal. Essas autoridades geralmente são "funções", como **ROLE_ADMINISTRATOR** ou **ROLE_HR_SUPERVISOR**. Essas funções são posteriormente configuradas para autorização na web, autorização de métodos e autorização de objetos de domínio. Outras partes do **Spring Security** são capazes de interpretar essas autoridades e esperam que elas estejam presentes. Ao usar autenticação baseada em nome de usuário/senha, os **GrantedAuthoritys** geralmente são carregados pelo **UserDetailsService**.
Normalmente, os objetos **GrantedAuthority** são permissões de aplicação em todo o sistema. Eles não são específicos de um dado objeto de domínio. Portanto, você provavelmente não teria um **GrantedAuthority** para representar uma permissão para o objeto **Employee** número 54, porque se houver milhares dessas autoridades, você rapidamente esgotaria a memória (ou, no mínimo, causaria um longo tempo de autenticação para o usuário). Claro, o **Spring Security** foi expressamente projetado para lidar com esse requisito comum, mas você usaria, em vez disso, as capacidades de segurança de objetos de domínio do projeto para esse propósito.

10.5. AuthenticationManager

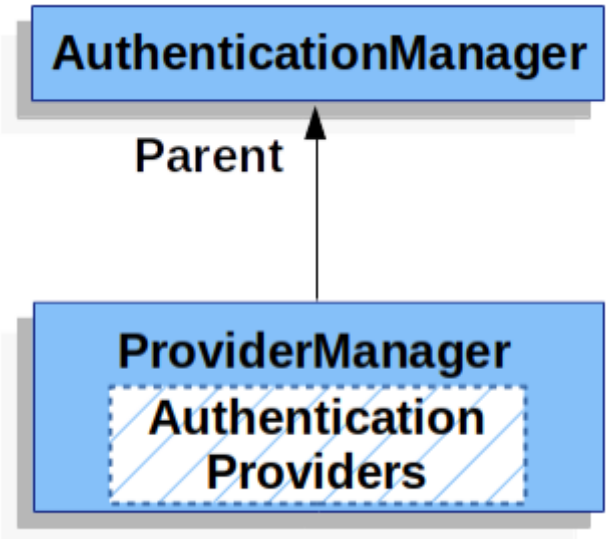
AuthenticationManager é a API que define como os filtros do **Spring Security** realizam a autenticação. A **Authentication** que é retornada é então definida no **SecurityContextHolder** pelo controlador (ou seja, pelos filtros do **Spring Security**) que invocou o **AuthenticationManager**. Se você não estiver integrando com os filtros do **Spring Security**, pode definir o **SecurityContextHolder** diretamente e não precisa usar o **AuthenticationManager**. Embora a implementação do **AuthenticationManager** possa ser qualquer coisa, a implementação mais comum é o **ProviderManager**.

10.6. ProviderManager

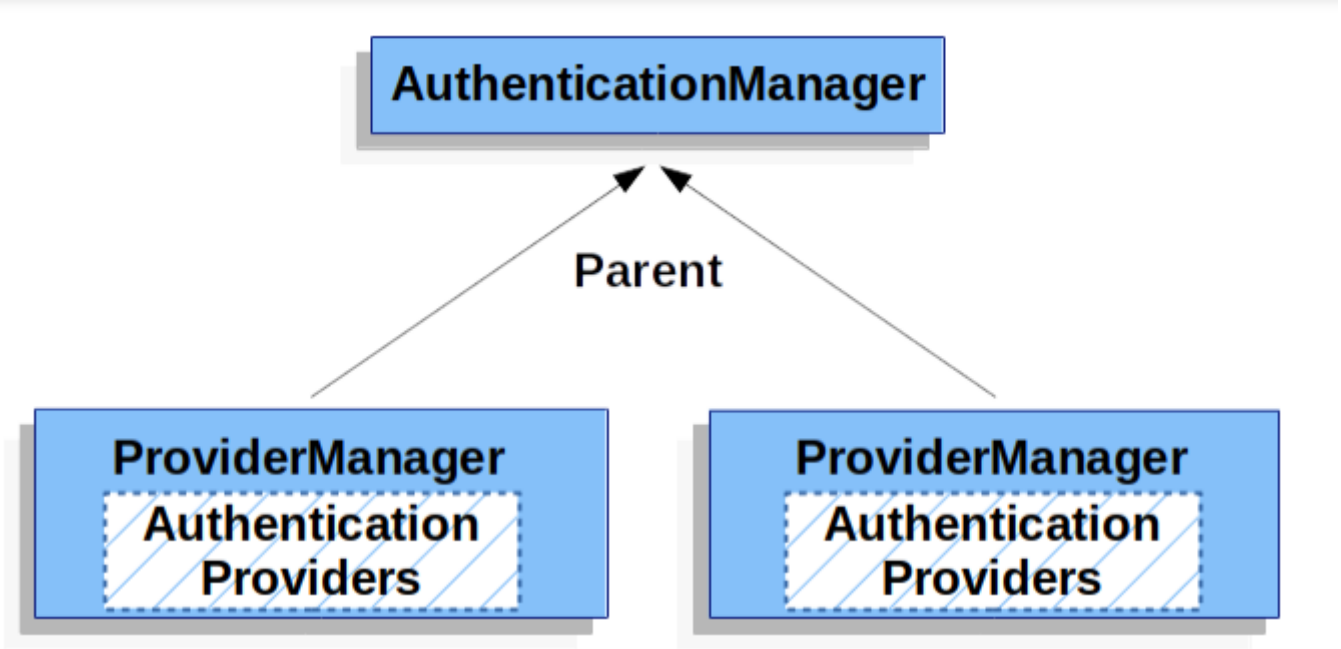
ProviderManager é a implementação mais comumente usada do **AuthenticationManager**.
O **ProviderManager** delega a uma lista de **AuthenticationProviders**. Cada **AuthenticationProvider** tem a oportunidade de indicar que a autenticação foi bem-sucedida, falhou ou indicar que não pode tomar uma decisão e permitir que um **AuthenticationProvider** subsequente decida. Se nenhum dos **AuthenticationProviders** configurados conseguir autenticar, a autenticação falhará com uma **ProviderNotFoundException**, que é uma exceção especial de **AuthenticationException** que indica que o **ProviderManager** não foi configurado para dar suporte ao tipo de autenticação que foi passado para ele.



Na prática, cada **AuthenticationProvider** sabe como realizar um tipo específico de autenticação. Por exemplo, um **AuthenticationProvider** pode ser capaz de validar um nome de usuário/senha, enquanto outro pode ser capaz de autenticar uma asserção SAML. Isso permite que cada **AuthenticationProvider** realize um tipo específico de autenticação, enquanto suporta vários tipos de autenticação e expõe apenas um único bean de **AuthenticationManager**.
O **ProviderManager** também permite configurar um **AuthenticationManager** pai opcional, que é consultado no caso de nenhum **AuthenticationProvider** conseguir realizar a autenticação. O pai pode ser qualquer tipo de **AuthenticationManager**, mas geralmente é uma instância de **ProviderManager**.



Na verdade, múltiplas instâncias de **ProviderManager** podem compartilhar o mesmo **AuthenticationManager** pai. Isso é relativamente comum em cenários onde há várias instâncias de **SecurityFilterChain** que possuem alguma autenticação em comum (o **AuthenticationManager** pai compartilhado), mas também mecanismos de autenticação diferentes (as diferentes instâncias de **ProviderManager**).



Por padrão, o **ProviderManager** tentará limpar quaisquer informações sensíveis de credenciais do objeto **Authentication** retornado por uma solicitação de autenticação bem-sucedida. Isso impede que informações como senhas sejam mantidas por mais tempo do que o necessário na **HttpSession**.
Isso pode causar problemas quando você está usando um cache de objetos de usuário, por exemplo, para melhorar o desempenho em uma aplicação sem estado. Se o **Authentication** contiver uma referência a um objeto no cache (como uma instância de **UserDetails**) e suas credenciais forem removidas, então não será mais possível autenticar contra o valor armazenado em cache. Você precisa considerar isso se estiver usando um cache. Uma solução óbvia é fazer uma cópia do objeto primeiro, seja na implementação do cache ou no **AuthenticationProvider** que cria o objeto **Authentication** retornado. Alternativamente, você pode desabilitar a propriedade **eraseCredentialsAfterAuthentication** no **ProviderManager**. Consulte a documentação para mais informações.

10.7. AuthenticationProvider

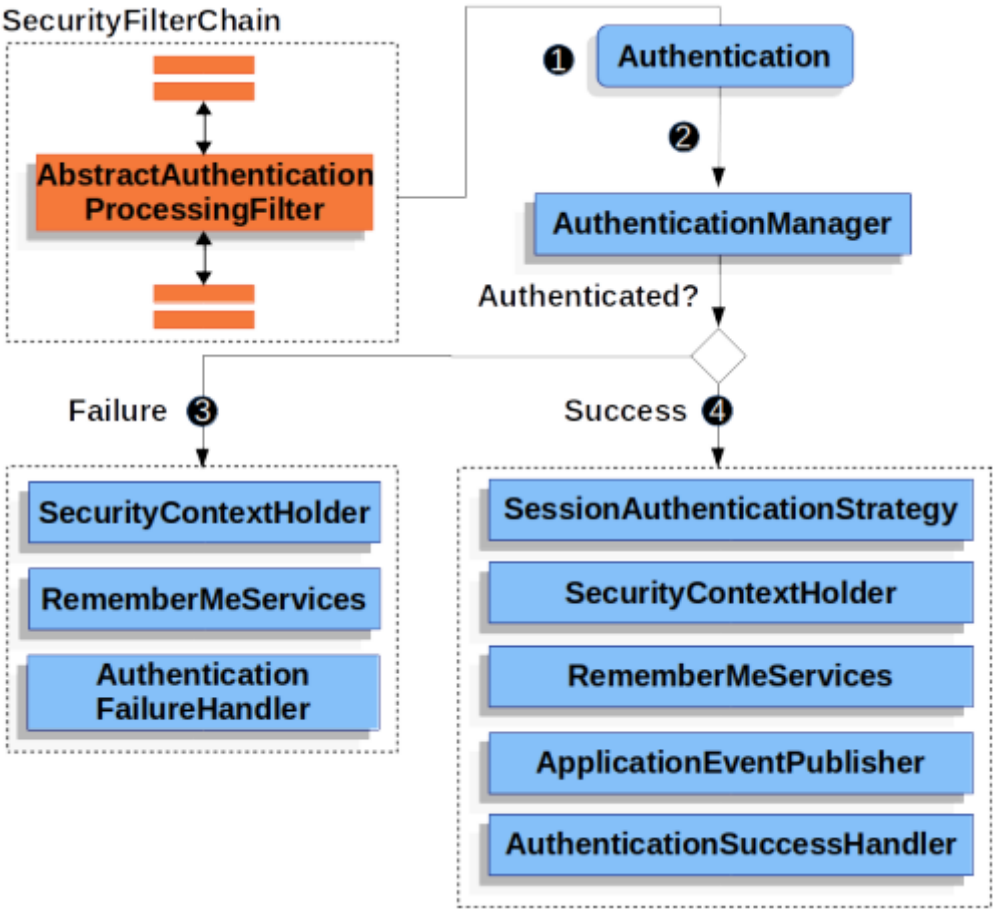
Múltiplos **AuthenticationProviders** podem ser injetados no **ProviderManager**. Cada **AuthenticationProvider** realiza um tipo específico de autenticação. Por exemplo, o **DaoAuthenticationProvider** suporta autenticação baseada em nome de usuário/senha, enquanto o **JwtAuthenticationProvider** suporta autenticação de um token JWT.

10.8. Request Credentials with AuthenticationEntryPoint

O **AuthenticationEntryPoint** é utilizado para enviar uma resposta HTTP que solicita credenciais de um cliente. Às vezes, um cliente incluirá proativamente credenciais como nome de usuário/senha para solicitar um recurso. Nesses casos, o **Spring Security** não precisa fornecer uma resposta HTTP que solicite credenciais, pois elas já estão incluídas.
Em outros casos, um cliente fará uma solicitação não autenticada a um recurso ao qual ele não está autorizado a acessar. Nesse caso, uma implementação do **AuthenticationEntryPoint** é usada para solicitar credenciais do cliente. A implementação do **AuthenticationEntryPoint** pode redirecionar para uma página de login, responder com um cabeçalho **WWW-Authenticate**, etc.

10.9. AbstractAuthenticationProcessingFilter

O **AbstractAuthenticationProcessingFilter** é usado como um filtro base para autenticar as credenciais de um usuário. Antes que as credenciais possam ser autenticadas, o **Spring Security** normalmente solicita as credenciais usando o **AuthenticationEntryPoint**.
Em seguida, o **AbstractAuthenticationProcessingFilter** pode autenticar quaisquer solicitações de autenticação que forem enviadas a ele.



Quando o usuário envia suas credenciais, o **AbstractAuthenticationProcessingFilter** cria um **Authentication** a partir do **HttpServletRequest** para ser autenticado. O tipo de **Authentication** criado depende da subclasse do **AbstractAuthenticationProcessingFilter**. Por exemplo, o **UsernamePasswordAuthenticationFilter** cria um **UsernamePasswordAuthenticationToken** a partir de um nome de usuário e senha que são enviados no **HttpServletRequest**.
Em seguida, o **Authentication** é passado para o **AuthenticationManager** para ser autenticado.
Se a autenticação falhar, ocorre o seguinte:

- O **SecurityContextHolder** é limpo.
- **RememberMeServices.loginFail** é invocado. Se o "lembre-se de mim" não estiver configurado, isso é uma operação sem efeito.
- **AuthenticationFailureHandler** é invocado.

Se a autenticação for bem-sucedida, ocorre o seguinte:

- **SessionAuthenticationStrategy** é notificado sobre o novo login.
- O **Authentication** é definido no **SecurityContextHolder**. Posteriormente, o **SecurityContextPersistenceFilter** salva o **SecurityContext** na **HttpSession**.
- **RememberMeServices.loginSuccess** é invocado. Se o "lembre-se de mim" não estiver configurado, isso é uma operação sem efeito.
- **ApplicationEventPublisher** publica um **InteractiveAuthenticationSuccessEvent**.

10.10. Autenticação com Nome de Usuário/Senha
Uma das maneiras mais comuns de autenticar um usuário é validando um nome de usuário e senha. Como tal, o **Spring Security** oferece suporte completo para autenticação com nome de usuário e senha.
Lendo o Nome de Usuário e a Senha
O **Spring Security** fornece os seguintes mecanismos integrados para ler um nome de usuário e senha a partir do **HttpServletRequest**:

- Login via Formulário
- Autenticação Básica
- Autenticação Digest

Mecanismos de Armazenamento
Cada um dos mecanismos suportados para ler um nome de usuário e senha pode utilizar qualquer um dos mecanismos de armazenamento suportados:

- Armazenamento Simples com Autenticação em Memória

- Bancos de Dados Relacionais com Autenticação JDBC
- Armazenamento personalizado com **UserDetailsService**
- Armazenamento LDAP com Autenticação LDAP

10.10.1. Login via Formulário

O **Spring Security** oferece suporte para nome de usuário e senha fornecidos por meio de um formulário HTML. Esta seção fornece detalhes sobre como a autenticação baseada em formulário funciona dentro do **Spring Security**.
Vamos dar uma olhada em como o login baseado em formulário funciona no **Spring Security**. Primeiro, vemos como o usuário é redirecionado para o formulário de login.

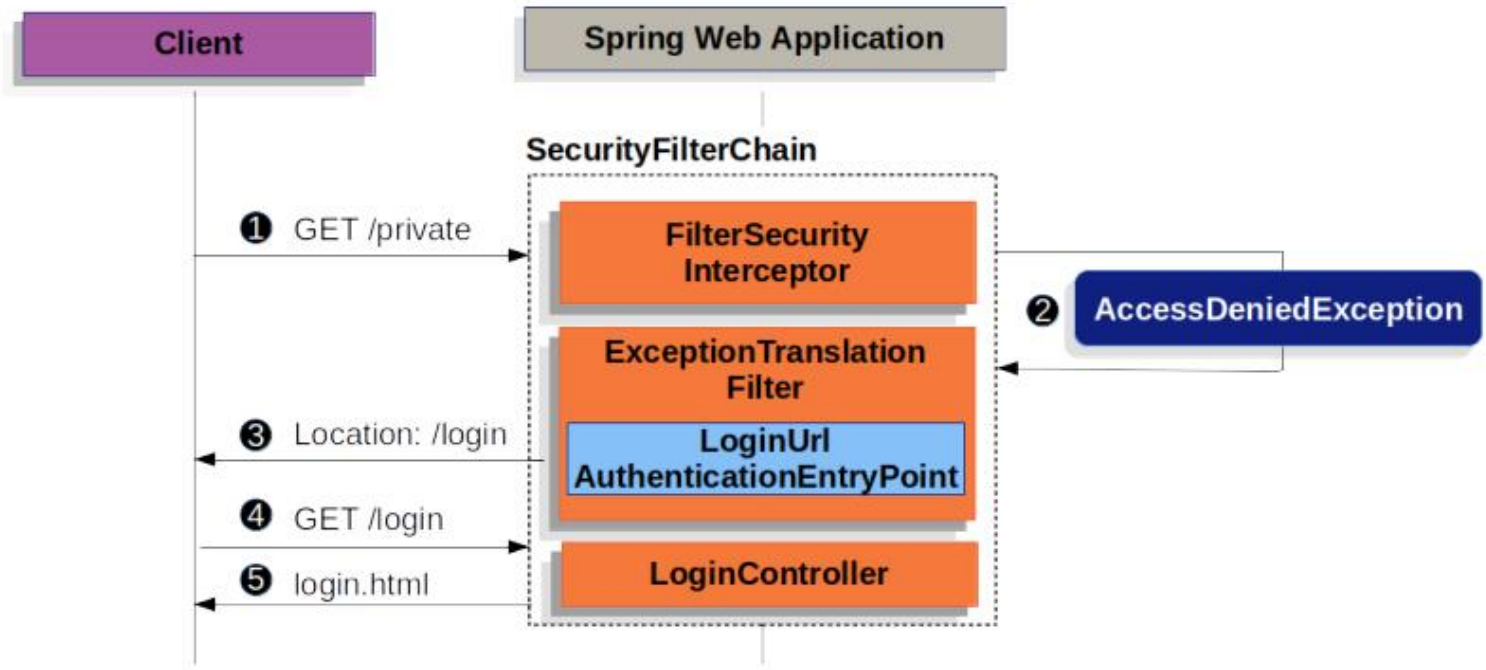


Figure 6. Redirecting to the Log In Page

A figura baseia-se em nosso diagrama de **SecurityFilterChain**.
Primeiro, um usuário faz uma solicitação não autenticada ao recurso **/private**, para o qual ele não tem autorização.
O **FilterSecurityInterceptor** do **Spring Security** indica que a solicitação não autenticada é negada ao lançar uma **AccessDeniedException**.
Como o usuário não está autenticado, o **ExceptionTranslationFilter** inicia o processo de **Iniciar Autenticação** e envia um redirecionamento para a página de login com o **AuthenticationEntryPoint** configurado. Na maioria dos casos, o **AuthenticationEntryPoint** é uma instância de **LoginUrlAuthenticationEntryPoint**.
O navegador então solicita a página de login para a qual foi redirecionado.
Algo dentro da aplicação deve renderizar a página de login.

Quando o nome de usuário e a senha são enviados, o **UsernamePasswordAuthenticationFilter** autentica o nome de usuário e a senha. O **UsernamePasswordAuthenticationFilter** estende o **AbstractAuthenticationProcessingFilter**, portanto, este diagrama deve ser muito semelhante.

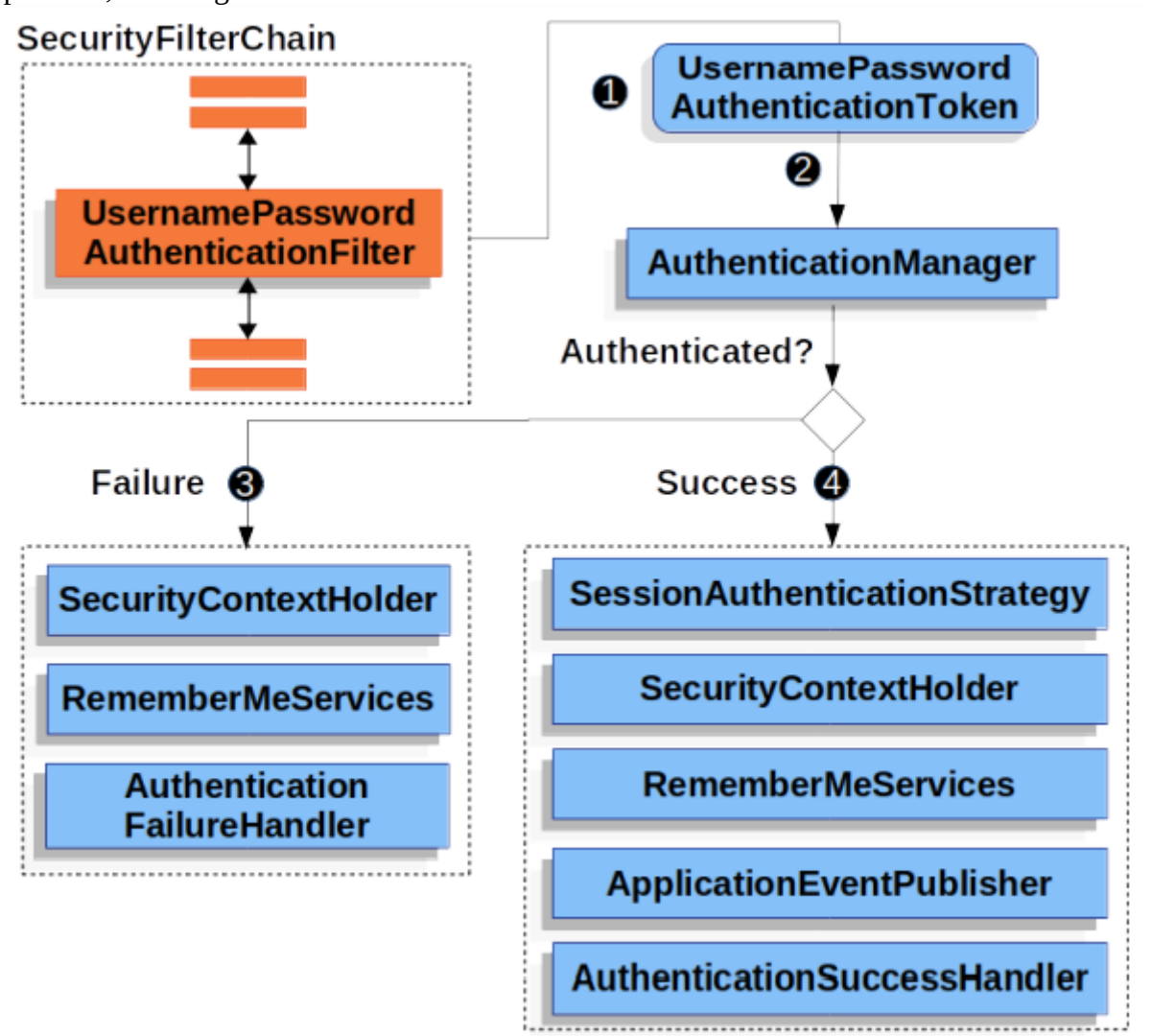


Figura 7. Autenticação de Nome de Usuário e Senha

A figura baseia-se em nosso diagrama de **SecurityFilterChain**.

Quando o usuário envia seu nome de usuário e senha, o **UsernamePasswordAuthenticationFilter** cria um **UsernamePasswordAuthenticationToken**, que é um tipo de **Authentication**, extraindo o nome de usuário e a senha do **HttpServletRequest**.

Em seguida, o **UsernamePasswordAuthenticationToken** é passado para o **AuthenticationManager** para ser autenticado. Os detalhes de como o **AuthenticationManager** funciona dependem de como as informações do usuário estão armazenadas.

Se a autenticação falhar, então **Falha**:

- O **SecurityContextHolder** é limpo.
- **RememberMeServices.loginFail** é invocado. Se o "lembrar-me" não estiver configurado, isso é uma operação sem efeito.
- **AuthenticationFailureHandler** é invocado.

Se a autenticação for bem-sucedida, então **Sucesso**:

- **SessionAuthenticationStrategy** é notificado de um novo login.
- A **Authentication** é configurada no **SecurityContextHolder**.
- **RememberMeServices.loginSuccess** é invocado. Se o "lembrar-me" não estiver configurado, isso é uma operação sem efeito.
- **ApplicationEventPublisher** publica um **InteractiveAuthenticationSuccessEvent**.
- **AuthenticationSuccessHandler** é invocado. Normalmente, isso é feito por meio de um **SimpleUrlAuthenticationSuccessHandler**, que redireciona para uma solicitação salva pelo **ExceptionTranslationFilter** quando redirecionamos para a página de login.

O login baseado em formulário do **Spring Security** é ativado por padrão. No entanto, assim que qualquer configuração baseada em servlet é fornecida, o login baseado em formulário deve ser explicitamente configurado. Uma configuração mínima e explícita em **Java** pode ser vista abaixo:

Java

```
protected void configure(HttpSecurity http) {
    http
        // ...
        .formLogin(withDefaults());
}
```

XML

```
<http>
    <!-- ... -->
    <form-login />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        formLogin { }
    }
}
```

Nesta configuração, o **Spring Security** renderiza uma página de login padrão. A maioria das aplicações de produção exigirá um formulário de login personalizado.

A configuração abaixo demonstra como fornecer um formulário de login personalizado.

Example 52. Custom Log In Form Configuration

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .formLogin(form -> form
            .loginPage("/login")
            .permitAll()
        );
}
```

XML

```
<http>
    <!-- ... -->
    <intercept-url pattern="/login" access="permitAll" />
    <form-login login-page="/login" />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        formLogin {
            loginPage = "/login"
            permitAll()
        }
    }
}
```

Quando a página de login é especificada na configuração do **Spring Security**, você é responsável por renderizar a página. Abaixo está um template **Thymeleaf** que produz um formulário de login em HTML que atende a uma página de login em **/login**:

Example 53. Log In Form

src/main/resources/templates/login.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">
    <head>
        <title>Please Log In</title>
    </head>
    <body>
        <h1>Please Log In</h1>
        <div th:if="${param.error}">
            Invalid username and password.</div>
        <div th:if="${param.logout}">
            You have been logged out.</div>
        <form th:action="@{/login}" method="post">
            <div>
            <input type="text" name="username" placeholder="Username"/>
            </div>
            <div>
            <input type="password" name="password" placeholder="Password"/>
            </div>
            <input type="submit" value="Log in" />
        </form>
    </body>
</html>
```

Há alguns pontos chave sobre o formulário HTML padrão: • O formulário deve realizar um post para /login • O formulário precisará incluir um Token CSRF, que é incluído automaticamente pelo Thymeleaf. • O formulário deve especificar o nome de usuário em um parâmetro chamado username • O formulário deve especificar a senha em um parâmetro chamado password • Se o parâmetro HTTP error for encontrado, isso indica que o usuário falhou em fornecer um nome de usuário / senha válidos • Se o parâmetro HTTP logout for encontrado, isso indica que o usuário fez logout com sucesso Muitos usuários não precisarão de muito mais do que personalizar a página de login. No entanto, se necessário, tudo o que foi mencionado pode ser personalizado com configurações adicionais. Se você estiver usando Spring MVC, será necessário um controlador que mapeie o GET /login para o template de login que criamos. Abaixo, um exemplo mínimo de LoginController pode ser visto:

src/main/java/example/LoginController.java

```
@Controller
class LoginController {
    @GetMapping("/login")
    String login() {
        return "login";
    }
}
```

10.10.2. Autenticação Básica

Esta seção fornece detalhes sobre como o Spring Security oferece suporte à Autenticação HTTP Básica para aplicações baseadas em servlets.

Vamos ver como a Autenticação HTTP Básica funciona dentro do Spring Security. Primeiro, vemos o cabeçalho WWW-Authenticate sendo enviado de volta para um cliente não autenticado.

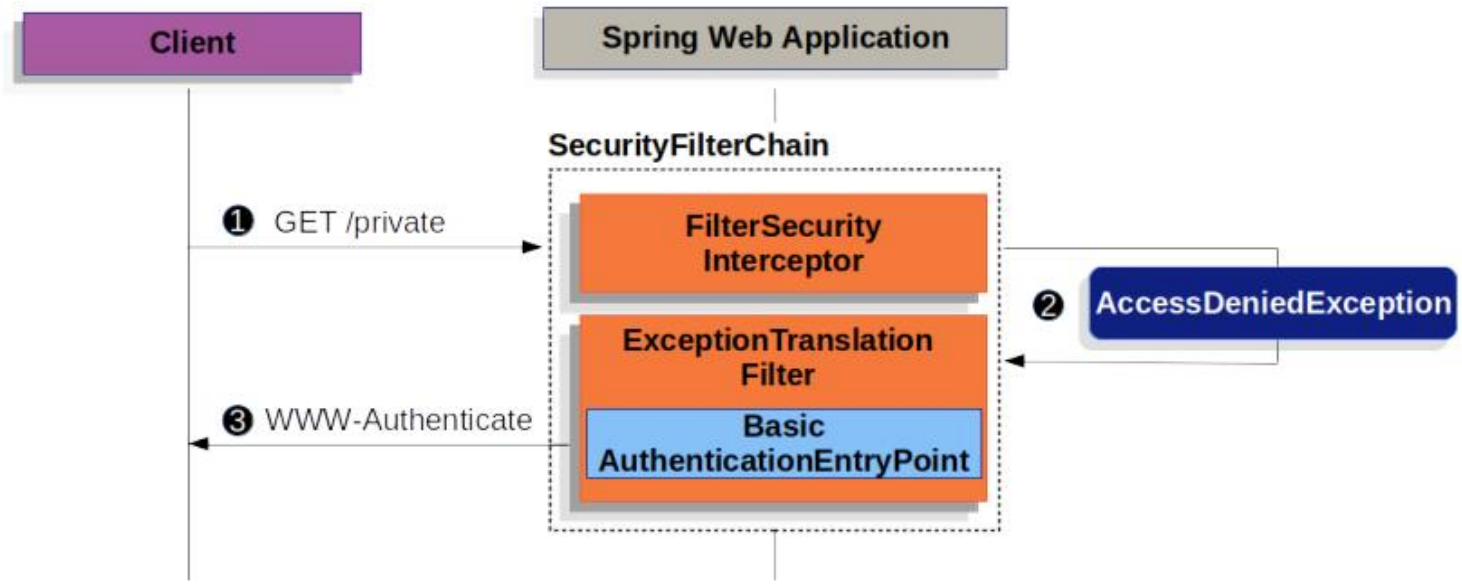


Figure 8. Sending WWW-Authenticate Header

A figura é baseada no nosso diagrama de SecurityFilterChain.
Primeiro, um usuário faz uma solicitação não autenticada para o recurso /private, para o qual ele não tem autorização. O FilterSecurityInterceptor do Spring Security indica que a solicitação não autenticada é negada, lançando uma AccessDeniedException.
Como o usuário não está autenticado, o ExceptionTranslationFilter inicia a autenticação. O AuthenticationEntryPoint configurado é uma instância do BasicAuthenticationEntryPoint, que envia um cabeçalho WWW-Authenticate. O RequestCache é tipicamente um NullRequestCache, que não salva a solicitação, pois o cliente é capaz de reenviar as solicitações que originalmente fez.
Quando um cliente recebe o cabeçalho WWW-Authenticate, ele sabe que deve tentar novamente com um nome de usuário e senha. Abaixo está o fluxo para o processamento do nome de usuário e senha.

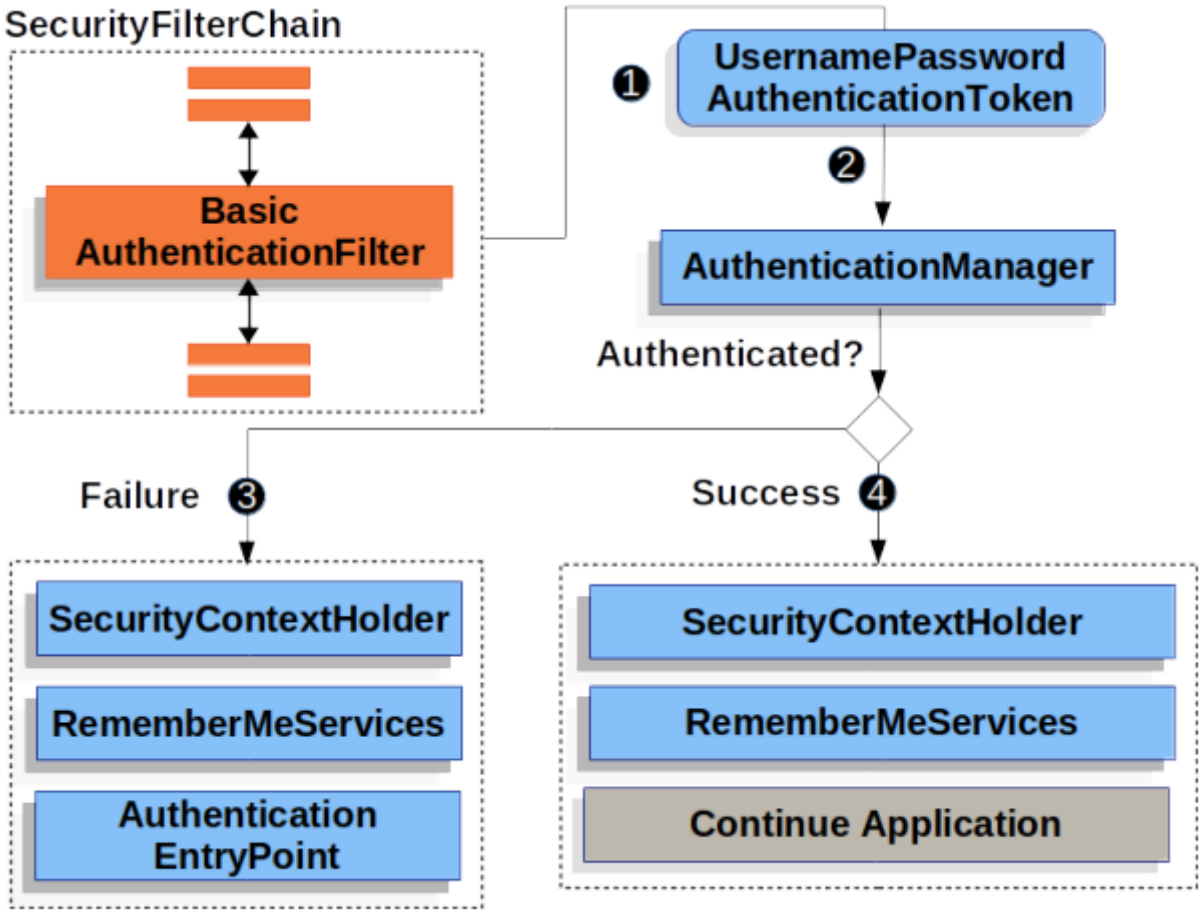


Figure 9. Authenticating Username and Password

A figura é baseada no nosso diagrama de SecurityFilterChain.

Quando o usuário envia seu nome de usuário e senha, o **UsernamePasswordAuthenticationFilter** cria um **UsernamePasswordAuthenticationToken**, que é um tipo de **Authentication**, extraindo o nome de usuário e a senha do **HttpServletRequest**.
Em seguida, o **UsernamePasswordAuthenticationToken** é passado para o **AuthenticationManager** para ser autenticado. Os detalhes de como o **AuthenticationManager** se comporta dependem de como as informações do usuário são armazenadas.
Se a autenticação falhar, ocorre a seguinte sequência de falhas:

- O **SecurityContextHolder** é limpo.
- **RememberMeServices.loginFail** é invocado. Se a funcionalidade "Lembrar-me" não estiver configurada, isso será uma operação sem efeito (no-op).
- O **AuthenticationEntryPoint** é invocado para acionar o envio do cabeçalho **WWW-Authenticate** novamente.

Se a autenticação for bem-sucedida, ocorre a seguinte sequência de sucesso:

- A **Authentication** é configurada no **SecurityContextHolder**.
- **RememberMeServices.loginSuccess** é invocado. Se a funcionalidade "Lembrar-me" não estiver configurada, isso será uma operação sem efeito (no-op).
- O **BasicAuthenticationFilter** invoca **FilterChain.doFilter(request, response)** para continuar com o restante da lógica da aplicação.

O suporte de autenticação HTTP Basic do Spring Security está habilitado por padrão. No entanto, assim que qualquer configuração baseada em servlet for fornecida, a autenticação HTTP Basic precisa ser configurada explicitamente.
Abaixo está uma configuração mínima explícita para HTTP Basic:
Exemplo 55. Configuração explícita para HTTP Basic

Java

```
protected void configure(HttpSecurity http) {
    http
        // ...
        .httpBasic(withDefaults());
}
```

XML

```
<http>
    <!-- ... -->
    <http-basic />
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        httpBasic { }
    }
}
```

10.10.3. Autenticação Digest
Esta seção fornece detalhes sobre como o Spring Security oferece suporte à Autenticação Digest, que é fornecida pelo **DigestAuthenticationFilter**.
Você **não deve usar a Autenticação Digest em aplicativos modernos**, pois ela não é considerada segura. O problema mais óbvio é que você deve armazenar suas senhas em formato de texto simples, criptografado ou em formato MD5. Todos esses formatos de armazenamento são considerados inseguros. Em vez disso, você deve armazenar credenciais usando um hash de senha adaptativo de mão única (por exemplo, bCrypt, PBKDF2, SCrypt, etc.), o que não é suportado pela Autenticação Digest.
A Autenticação Digest tenta resolver muitas das fraquezas da autenticação Básica, especificamente garantindo que as credenciais nunca sejam enviadas em texto claro pela rede. Muitos navegadores suportam a Autenticação Digest.

O padrão que rege a Autenticação Digest HTTP é definido pelo **RFC 2617**, que atualiza uma versão anterior do padrão de Autenticação Digest prescrito pelo **RFC 2069**. A maioria dos agentes de usuário implementa o **RFC 2617**. O suporte do Spring Security para Autenticação Digest é compatível com o "auth" (qualidade de proteção, qop) prescrito pelo **RFC 2617**, que também fornece compatibilidade retroativa com o **RFC 2069**. A Autenticação Digest foi vista como uma opção mais atraente se você precisasse usar HTTP não criptografado (ou seja, sem TLS/HTTPS) e quisesse maximizar a segurança do processo de autenticação. No entanto, todos devem usar **HTTPS**.

Central para a Autenticação Digest está o "**nonce**". Este é um valor gerado pelo servidor. O nonce do Spring Security adota o seguinte formato:

Example 56. Digest Syntax

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))
expirationTime:   The date and time when the nonce expires, expressed in
milliseconds
key:              A private key to prevent modification of the nonce token
```

Você precisará garantir que configure o armazenamento de senha em texto simples inseguro usando o **NoOpPasswordEncoder**. A seguir, é fornecido um exemplo de configuração da Autenticação Digest com Configuração Java:

Java

```
@Autowired
UserDetailsService userDetailsService;

DigestAuthenticationEntryPoint entryPoint() {
    DigestAuthenticationEntryPoint result = new DigestAuthenticationEntryPoint();
    result.setRealmName("My App Relam");
    result.setKey("3028472b-da34-4501-bfd8-a355c42bdf92");
}

DigestAuthenticationFilter digestAuthenticationFilter() {
    DigestAuthenticationFilter result = new DigestAuthenticationFilter();
    result.setUserDetailsService(userDetailsService);
    result.setAuthenticationEntryPoint(entryPoint());
}

protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .exceptionHandling(e ->
e.authenticationEntryPoint(authenticationEntryPoint()))
        .addFilterBefore(digestFilter());
}
```

XML

```
<b:bean id="digestFilter"

class="org.springframework.security.web.authentication.www.DigestAuthenticationFil
ter"
    p:userDetailsService-ref="jdbcDaoImpl"
    p:authenticationEntryPoint-ref="digestEntryPoint"
/>

<b:bean id="digestEntryPoint"

class="org.springframework.security.web.authentication.www.DigestAuthenticationEnt
ryPoint"
    p:realmName="My App Relam"
    p:key="3028472b-da34-4501-bfd8-a355c42bdf92"
/>

<http>
    <!-- ... -->
    <custom-filter ref="userFilter" position="DIGEST_AUTH_FILTER"/>
</http>
```

10.10.4. **Autenticação em Memória**

O **InMemoryUserDetailsManager** do Spring Security implementa o **UserDetailsService** para fornecer suporte à autenticação baseada em nome de usuário/senha que é recuperada da memória. O **InMemoryUserDetailsManager** gerencia os **UserDetails** implementando a interface **UserDetailsManager**.
A autenticação baseada em **UserDetails** é utilizada pelo **Spring Security** quando configurado para aceitar um nome de usuário/senha para autenticação.
Neste exemplo, usamos o **Spring Boot CLI** para codificar a senha "password" e obter a senha codificada:
{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW.

Example 58. InMemoryUserDetailsManager Java Configuration

Java

```
@Bean
public UserDetailsService users() {
    UserDetails user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

XML

```
<user-service>
    <user name="user"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER" />
    <user name="admin"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER,ROLE_ADMIN" />
</user-service>
```

Kotlin

```
@Bean
fun users(): UserDetailsService {
    val user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build()
    val admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build()
    return InMemoryUserDetailsManager(user, admin)
}
```

Os exemplos acima armazenam as senhas em um formato seguro, mas deixam a desejar em termos de experiência para começar.
No exemplo abaixo, usamos o **User.withDefaultPasswordEncoder** para garantir que a senha armazenada em memória esteja protegida. No entanto, isso não protege a senha contra a obtenção dela por meio da descompilação do código-fonte. Por essa razão, **User.withDefaultPasswordEncoder** deve ser usado apenas para "iniciar" e não é recomendado para produção.

Example 59. InMemoryUserDetailsManager with User.withDefaultPasswordEncoder

Java

```
@Bean
public UserDetailsService users() {
    // The builder will ensure the passwords are encoded before saving in memory
    UserBuilder users = User.withDefaultPasswordEncoder();
    UserDetails user = users
        .username("user")
        .password("password")
        .roles("USER")
        .build();
    UserDetails user = users
        .username("admin")
        .password("password")
        .roles("USER", "ADMIN")
        .build();
    return new InMemoryUserDetailsManager(user, admin);
}
```

Kotlin

```
@Bean
fun users(): UserDetailsService {
    // The builder will ensure the passwords are encoded before saving in memory
    val users = User.withDefaultPasswordEncoder()
    val user = users
        .username("user")
        .password("password")
        .roles("USER")
        .build()
    val admin = users
        .username("admin")
        .password("password")
        .roles("USER", "ADMIN")
        .build()
    return InMemoryUserDetailsManager(user, admin)
}
```

Não há uma maneira simples de usar **User.withDefaultPasswordEncoder** com configuração baseada em XML. Para demonstrações ou apenas para começar, você pode optar por prefixar a senha com **{noop}** para indicar que nenhum tipo de codificação deve ser usado.

Example 60. <user-service> {noop} XML Configuration

```
<user-service>
    <user name="user"
        password="{noop}password"
        authorities="ROLE_USER" />
    <user name="admin"
        password="{noop}password"
        authorities="ROLE_USER,ROLE_ADMIN" />
</user-service>
```

10.10.5. **Autenticação JDBC**

O **JdbcDaoImpl** do Spring Security implementa o **UserDetailsService** para fornecer suporte à autenticação baseada em nome de usuário/senha, que é recuperada usando JDBC. O **JdbcUserDetailsManager** estende o **JdbcDaoImpl** para fornecer gerenciamento de **UserDetails** através da interface **UserDetailsManager**. A autenticação baseada em **UserDetails** é usada pelo Spring Security quando configurado para aceitar nome de usuário/senha para autenticação.

Nas próximas seções, discutiremos:

- O **Esquema Padrão** usado pela Autenticação JDBC do Spring Security
- Como configurar um **DataSource**
- O **Bean JdbcUserDetailsManager**

Esquema Padrão

O Spring Security fornece consultas padrão para autenticação baseada em JDBC. Esta seção fornece os esquemas padrão correspondentes usados com as consultas padrão. Será necessário ajustar o esquema para corresponder a qualquer personalização nas consultas e no dialeto do banco de dados que você está utilizando.

Esquema de Usuário

O **JdbcDaoImpl** requer tabelas para carregar a senha, o status da conta (habilitada ou desabilitada) e uma lista de autoridades (papéis) para o usuário. O esquema padrão necessário pode ser encontrado abaixo.

Nota: O esquema padrão também está exposto como um recurso de classe com o nome **org/springframework/security/core/userdetails/jdbc/users.ddl**.

Example 61. Default User Schema

```
create table users(
    username varchar_ignorecase(50) not null primary key,
    password varchar_ignorecase(50) not null,
    enabled boolean not null
);

create table authorities (
    username varchar_ignorecase(50) not null,
    authority varchar_ignorecase(50) not null,
    constraint fk_authorities_users foreign key(username) references
users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

O Oracle é uma escolha popular de banco de dados, mas requer um esquema ligeiramente diferente. Você pode encontrar o **Esquema Padrão do Oracle** para usuários abaixo.

Example 62. Default User Schema for Oracle Databases

```
CREATE TABLE USERS (
    USERNAME NVARCHAR2(128) PRIMARY KEY,
    PASSWORD NVARCHAR2(128) NOT NULL,
    ENABLED CHAR(1) CHECK (ENABLED IN ('Y','N') ) NOT NULL
);


CREATE TABLE AUTHORITIES (
    USERNAME NVARCHAR2(128) NOT NULL,
    AUTHORITY NVARCHAR2(128) NOT NULL
);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_UNIQUE UNIQUE (USERNAME,
AUTHORITY);
ALTER TABLE AUTHORITIES ADD CONSTRAINT AUTHORITIES_FK1 FOREIGN KEY (USERNAME)
REFERENCES USERS (USERNAME) ENABLE;
```

Esquema de Grupos

Se sua aplicação estiver utilizando grupos, será necessário fornecer o esquema correspondente. O esquema padrão para grupos pode ser encontrado abaixo.

Example 63. Default Group Schema

```
create table groups (
    id bigint generated by default as identity(start with 0) primary key,
    group_name varchar_ignorecase(50) not null
);

create table group_authorities (
    group_id bigint not null,
    authority varchar(50) not null,
    constraint fk_group_authorities_group foreign key(group_id) references
groups(id)
);

create table group_members (
    id bigint generated by default as identity(start with 0) primary key,
    username varchar(50) not null,
    group_id bigint not null,
    constraint fk_group_members_group foreign key(group_id) references groups(id)
);
```

Configurando um DataSource

Antes de configurarmos o JdbcUserDetailsManager, precisamos criar um DataSource. No nosso exemplo, configuraremos um DataSource embutido que será inicializado com o esquema de usuário padrão.

Example 64. Embedded Data Source

Java

```
@Bean
DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(H2)

.addScript("classpath:org/springframework/security/core/userdetails/jdbc/users.ddl
")
        .build();
}
```

XML

```
<jdbc:embedded-database>
    <jdbc:script
location="classpath:org/springframework/security/core/userdetails/jdbc/users.ddl"/
>
</jdbc:embedded-database>
```

Kotlin

```
@Bean
fun dataSource(): DataSource {
    return EmbeddedDatabaseBuilder()
        .setType(H2)

.addScript("classpath:org/springframework/security/core/userdetails/jdbc/users.ddl
")
        .build()
}
```

JdbcUserDetailsManager Bean

Neste exemplo, usamos o **Spring Boot CLI** para codificar a senha "password" e obter a senha codificada {bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW.
Consulte a seção **PasswordEncoder** para mais detalhes sobre como armazenar senhas de forma segura.

Example 65. JdbcUserDetailsManager

Java

```
@Bean
UserDetailsManager users(DataSource dataSource) {
    UserDetails user = User.builder()
        .username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER")
        .build();
    UserDetails admin = User.builder()
        .username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
        .roles("USER", "ADMIN")
        .build();
    JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
    users.createUser()
}
```

XML

```
<jdbc-user-service>
    <user name="user"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER" />
    <user name="admin"

password="{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW"
        authorities="ROLE_USER,ROLE_ADMIN" />
</jdbc-user-service>
```

Kotlin

```
@Bean
fun users(dataSource: DataSource): UserDetailsManager {
    val user = User.builder()
            .username("user")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
            .roles("USER")
            .build();
    val admin = User.builder()
            .username("admin")

.password("{bcrypt}$2a$10\$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")
            .roles("USER", "ADMIN")
            .build();
    val users = JdbcUserDetailsManager(dataSource)
    users.createUser(user)
    users.createUser(admin)
    return users
}
```

UserDetails

O **UserDetails** é retornado pelo **UserDetailsService**. O **DaoAuthenticationProvider** valida o **UserDetails** e, em seguida, retorna um **Authentication** cujo **principal** é o próprio **UserDetails** retornado pelo **UserDetailsService** configurado.

UserDetailsService

O **UserDetailsService** é utilizado pelo **DaoAuthenticationProvider** para recuperar o nome de usuário, senha e outros atributos necessários para a autenticação baseada em nome de usuário e senha. O **Spring Security** fornece implementações de **UserDetailsService** baseadas em **memória** e **JDBC**.

Você pode definir um processo de autenticação personalizado expondo um **UserDetailsService** personalizado como um **bean**.

Por exemplo, a seguinte configuração personaliza a autenticação, assumindo que CustomUserDetailsService implementa **UserDetailsService**:

Nota: Isso só será utilizado se o **AuthenticationManagerBuilder** não tiver sido populado e nenhum **AuthenticationProviderBean** tiver sido definido.

Example 66. Custom UserDetailsService Bean

Java

```
@Bean
CustomUserDetailsService customUserDetailsService() {
    return new CustomUserDetailsService();
}
```

XML

```
<b:bean class="example.CustomUserDetailsService"/>
```

Kotlin

```
@Bean
fun customUserDetailsService() = CustomUserDetailsService()
```

PasswordEncoder

O suporte do **Spring Security** para aplicações servlet permite armazenar senhas de forma segura por meio da integração com **PasswordEncoder**. Para personalizar a implementação do **PasswordEncoder** utilizada pelo **Spring Security**, basta expor um **PasswordEncoder Bean**.

DaoAuthenticationProvider

O **DaoAuthenticationProvider** é uma implementação de **AuthenticationProvider** que utiliza um **UserDetailsService** e um **PasswordEncoder** para autenticar um nome de usuário e senha.

Para entender como o **DaoAuthenticationProvider** funciona dentro do **Spring Security**, a figura explica os detalhes sobre como o **AuthenticationManager** processa a autenticação, desde a leitura do nome de usuário e senha até a validação das credenciais.

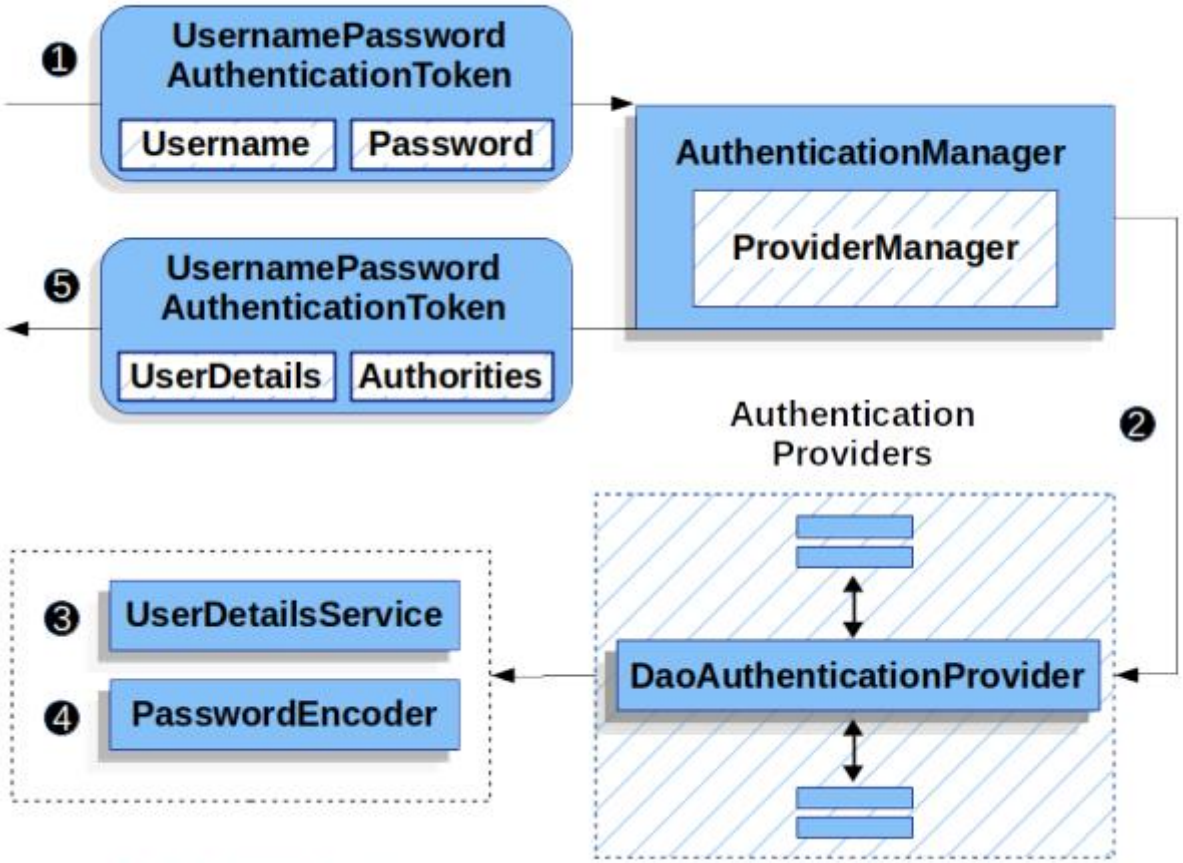


Figure 10. DaoAuthenticationProvider Usage

LDAP Authentication

O **LDAP** é frequentemente usado por organizações como um repositório central para informações de usuários e como um serviço de autenticação. Ele também pode armazenar informações sobre os papéis (roles) dos usuários da aplicação.

O **Spring Security** suporta autenticação baseada em **LDAP**, permitindo a autenticação via **username/password**. No entanto, ele não se integra com o **UserDetailsService**, pois no processo de **bind authentication**, o servidor **LDAP** não retorna a senha, impossibilitando a aplicação de validar a senha diretamente.

Cenários e Configuração do LDAP no Spring Security

Os servidores **LDAP** podem ser configurados de diversas maneiras, e o **Spring Security** oferece um provedor **LDAP** totalmente configurável. Ele usa interfaces estratégicas separadas para autenticação e recuperação de papéis, fornecendo implementações padrão que podem ser configuradas para lidar com diferentes cenários.

Pré-requisitos

Antes de integrar o **LDAP** ao **Spring Security**, é recomendado ter conhecimento básico sobre **LDAP**. Um bom guia introdutório pode ser encontrado em:

Guia sobre LDAP

Além disso, conhecer as APIs **JNDI** usadas para acessar **LDAP** no **Java** pode ser útil. O **Spring Security** utiliza amplamente o **Spring LDAP**, então, caso haja necessidade de personalizações, vale a pena se familiarizar com esse projeto.

Configuração de um Servidor LDAP Embutido

Para testar a autenticação via **LDAP**, a abordagem mais simples é configurar um **servidor LDAP embutido**. O **Spring Security** suporta dois tipos de servidores embutidos:

- **Embedded UnboundID Server**
- **Embedded ApacheDS Server**

Nos exemplos a seguir, um arquivo **users.ldif** será exposto como um recurso na **classpath**, inicializando o servidor **LDAP embutido** com os usuários **user** e **admin**, ambos com a senha **password**.

users.ldif

```
dn: ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: groups
```

```
dn: ou=people,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Rod Johnson
sn: Johnson
uid: admin
userPassword: password

dn: uid=user,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Dianne Emu
sn: Emu
uid: user
userPassword: password

dn: cn=user,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: user
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
uniqueMember: uid=user,ou=people,dc=springframework,dc=org

dn: cn=admin,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: admin
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
```

Servidor UnboundID Embutido

Se você deseja usar o UnboundID, especifique as seguintes dependências:

Example 67. UnboundID Dependencies

Maven

```
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
    <version>4.0.14</version>
    <scope>runtime</scope>
</dependency>
```

Gradle

```
depenendencies {
    runtimeOnly "com.unboundid:unboundid-ldapsdk:4.0.14"
}
```

Você pode então configurar o Servidor LDAP Embutido

Example 68. Embedded LDAP Server Configuration

Java

```
@Bean
UnboundIdContainer ldapContainer() {
    return new UnboundIdContainer("dc=springframework,dc=org",
                "classpath:users.ldif");
}
```

XML

```
<b:bean class="org.springframework.security.ldap.server.UnboundIdContainer"
    c:defaultPartitionSuffix="dc=springframework,dc=org"
    c:ldif="classpath:users.ldif"/>
```

Kotlin

```
@Bean
fun ldapContainer(): UnboundIdContainer {
    return UnboundIdContainer("dc=springframework,dc=org","classpath:users.ldif")
}
```

Servidor ApacheDS Embutido

O Spring Security usa o ApacheDS 1.x, que não é mais mantido. Infelizmente, o ApacheDS 2.x lançou apenas versões de marco, sem uma versão estável. Assim que uma versão estável do ApacheDS 2.x estiver disponível, consideraremos a atualização.

Se você deseja usar o ApacheDS, adicione as seguintes dependências:

Exemplo 69. Dependências do ApacheDS

Maven

```
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-core</artifactId>
    <version>1.5.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-server-jndi</artifactId>
    <version>1.5.5</version>
    <scope>runtime</scope>
</dependency>
```

Gradle

```
depenendencies {
    runtimeOnly "org.apache.directory.server:apacheds-core:1.5.5"
    runtimeOnly "org.apache.directory.server:apacheds-server-jndi:1.5.5"
}
```

Você pode então configurar o Servidor LDAP Embutido.

Example 70. Embedded LDAP Server Configuration

Java

```
@Bean
ApacheDSContainer ldapContainer() {
    return new ApacheDSContainer("dc=springframework,dc=org",
                "classpath:users.ldif");
}
```

XML

```
<b:bean class="org.springframework.security.ldap.server.ApacheDSContainer"
    c:defaultPartitionSuffix="dc=springframework,dc=org"
    c:ldif="classpath:users.ldif"/>
```

Kotlin

```
@Bean
fun ldapContainer(): ApacheDSContainer {
    return ApacheDSContainer("dc=springframework,dc=org", "classpath:users.ldif")
}
```

Fonte de Contexto LDAP

Depois de ter um servidor LDAP para apontar sua configuração, você precisa configurar o Spring Security para apontar para um servidor LDAP que deve ser usado para autenticar os usuários. Isso é feito criando uma Fonte de Contexto LDAP, que é equivalente a um DataSource JDBC.

Example 71. LDAP Context Source

Java

```
ContextSource contextSource(UnboundIdContainer container) {
    return new
DefaultSpringSecurityContextSource("ldap://localhost:53389/dc=springframework,dc=o
rg");
}
```

XML

```
<ldap-server
    url="ldap://localhost:53389/dc=springframework,dc=org" />
```

Kotlin

```
fun contextSource(container: UnboundIdContainer): ContextSource {
    return
DefaultSpringSecurityContextSource("ldap://localhost:53389/dc=springframework,dc=o
rg")
}
```

Autenticação

O suporte do Spring Security para LDAP não usa o UserDetailsService porque a autenticação de bind LDAP não permite que os clientes leiam a senha ou até mesmo uma versão criptografada da senha. Isso significa que não há como a senha ser lida e então autenticada pelo Spring Security.
Por esse motivo, o suporte LDAP é implementado usando a interface LdapAuthenticator. O LdapAuthenticator também é responsável por recuperar quaisquer atributos de usuário necessários. Isso ocorre porque as permissões sobre os atributos podem depender do tipo de autenticação sendo usado. Por exemplo, se o bind for feito como o próprio usuário, pode ser necessário ler esses atributos com as permissões do próprio usuário.
Existem duas implementações de LdapAuthenticator fornecidas pelo Spring Security:
• Usando Autenticação por Bind
• Usando Autenticação por Senha

Usando Autenticação por Bind

A autenticação por bind é o mecanismo mais comum para autenticar usuários com LDAP. Na autenticação por bind, as credenciais do usuário (ou seja, nome de usuário/senha) são enviadas para o servidor LDAP, que as autentica. A vantagem de usar a autenticação por bind é que os segredos do usuário (ou seja, a senha) não precisam ser expostos aos clientes, o que ajuda a protegê-los contra vazamentos.
Um exemplo de configuração de autenticação por bind pode ser encontrado abaixo.

Example 72. Bind Authentication

Java

```
@Bean
BindAuthenticator authenticator(BaseLdapPathContextSource contextSource) {
    BindAuthenticator authenticator = new BindAuthenticator(contextSource);
    authenticator.setUserDnPatterns(new String[] { "uid={0},ou=people" });
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
    user-dn-pattern="uid={0},ou=people"/>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource): BindAuthenticator {
    val authenticator = BindAuthenticator(contextSource)
    authenticator.setUserDnPatterns(arrayOf("uid={0},ou=people"))
    return authenticator
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

Este exemplo simples obteria o DN do usuário substituindo o nome de login do usuário no padrão fornecido e tentando realizar o bind como esse usuário com a senha de login. Isso é aceitável se todos os seus usuários estiverem armazenados sob um único nó no diretório. Se, em vez disso, você quiser configurar um filtro de pesquisa LDAP para localizar o usuário, você poderia usar o seguinte:

Example 73. Bind Authentication with Search Filter

Java

```
@Bean
BindAuthenticator authenticator(BaseLdapPathContextSource contextSource) {
    String searchBase = "ou=people";
    String filter = "(uid={0})";
    FilterBasedLdapUserSearch search =
        new FilterBasedLdapUserSearch(searchBase, filter, contextSource);
    BindAuthenticator authenticator = new BindAuthenticator(contextSource);
    authenticator.setUserSearch(search);
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-search-filter="(uid={0})"
    user-search-base="ou=people"/>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource): BindAuthenticator {
    val searchBase = "ou=people"
    val filter = "(uid={0})"
    val search = FilterBasedLdapUserSearch(searchBase, filter, contextSource)
    val authenticator = BindAuthenticator(contextSource)
    authenticator.setUserSearch(search)
    return authenticator
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

Se usado com a definição de ContextSource acima, isso realizaria uma pesquisa sob o DN ou=people,dc=springframework,dc=org utilizando (uid={0}) como filtro. Novamente, o nome de login do usuário é substituído pelo parâmetro no nome do filtro, então ele buscará uma entrada com o atributo uid igual ao nome de usuário. Se uma base de pesquisa de usuário não for fornecida, a pesquisa será realizada a partir da raiz.

Usando Autenticação por Senha
A comparação de senhas ocorre quando a senha fornecida pelo usuário é comparada com a armazenada no repositório. Isso pode ser feito recuperando o valor do atributo de senha e verificando-o localmente ou realizando uma operação LDAP de "comparação", onde a senha fornecida é enviada ao servidor para comparação e o valor real da senha nunca é recuperado. Uma comparação LDAP não pode ser feita quando a senha está devidamente criptografada com um sal aleatório.

Example 74. Minimal Password Compare Configuration

Java

```
@Bean
PasswordComparisonAuthenticator authenticator(BaseLdapPathContextSource
contextSource) {
    return new PasswordComparisonAuthenticator(contextSource);
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-dn-pattern="uid={0},ou=people">
    <password-compare />
</ldap-authentication-provider>
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource):
PasswordComparisonAuthenticator {
    return PasswordComparisonAuthenticator(contextSource)
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

Uma configuração mais avançada com algumas personalizações pode ser encontrada abaixo.

Example 75. Password Compare Configuration

Java

```
@Bean
PasswordComparisonAuthenticator authenticator(BaseLdapPathContextSource
contextSource) {
    PasswordComparisonAuthenticator authenticator =
        new PasswordComparisonAuthenticator(contextSource);
    authenticator.setPasswordAttributeName("pwd"); ①
    authenticator.setPasswordEncoder(new BCryptPasswordEncoder()); ②
    return authenticator;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator)
{
    return new LdapAuthenticationProvider(authenticator);
}
```

XML

```
<ldap-authentication-provider
        user-dn-pattern="uid={0},ou=people">
    <password-compare password-attribute="pwd"> ①
        <password-encoder ref="passwordEncoder" /> ②
    </password-compare>
</ldap-authentication-provider>
<b:bean id="passwordEncoder"
    class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />
```

Kotlin

```
@Bean
fun authenticator(contextSource: BaseLdapPathContextSource):
PasswordComparisonAuthenticator {
    val authenticator = PasswordComparisonAuthenticator(contextSource)
    authenticator.setPasswordAttributeName("pwd") ①
    authenticator.setPasswordEncoder(BCryptPasswordEncoder()) ②
    return authenticator
}
```

```
@Bean
fun authenticationProvider(authenticator: LdapAuthenticator):
LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator)
}
```

① Especificar o atributo de senha como pwd
② Usar BCryptPasswordEncoder

LdapAuthoritiesPopulator

O **LdapAuthoritiesPopulator** do Spring Security é usado para determinar quais autoridades são retornadas para o usuário.

Example 76. Minimal Password Compare Configuration

Java

```
@Bean
LdapAuthoritiesPopulator authorities(BaseLdapPathContextSource contextSource) {
    String groupSearchBase = "";
    DefaultLdapAuthoritiesPopulator authorities =
        new DefaultLdapAuthoritiesPopulator(contextSource, groupSearchBase);
    authorities.setGroupSearchFilter("member={0}");
    return authorities;
}

@Bean
LdapAuthenticationProvider authenticationProvider(LdapAuthenticator authenticator,
LdapAuthoritiesPopulator authorities) {
    return new LdapAuthenticationProvider(authenticator, authorities);
}
```

XML

```
<ldap-authentication-provider
    user-dn-pattern="uid={0},ou=people"
    group-search-filter="member={0}"/>
```

Kotlin

```
@Bean
fun authorities(contextSource: BaseLdapPathContextSource):
LdapAuthoritiesPopulator {
    val groupSearchBase = ""
    val authorities = DefaultLdapAuthoritiesPopulator(contextSource,
groupSearchBase)
    authorities.setGroupSearchFilter("member={0}")
    return authorities
}

@Bean
fun authenticationProvider(authenticator: LdapAuthenticator, authorities:
LdapAuthoritiesPopulator): LdapAuthenticationProvider {
    return LdapAuthenticationProvider(authenticator, authorities)
}
```

Active Directory

O **Active Directory** oferece opções de autenticação próprias e não padrão, e o padrão de uso normal não se encaixa muito bem com o **LdapAuthenticationProvider** padrão. Normalmente, a autenticação é realizada usando o nome de usuário do domínio (no formato **user@domain**), em vez de usar um nome distinto de LDAP. Para facilitar isso, o Spring Security possui um provedor de autenticação personalizado para uma configuração típica do Active Directory. Configurar o **ActiveDirectoryLdapAuthenticationProvider** é bastante simples. Você só precisa fornecer o nome do domínio e uma URL LDAP, fornecendo o endereço do servidor. Um exemplo de configuração pode ser visto abaixo:

Example 77. Example Active Directory Configuration

Java

```
@Bean
ActiveDirectoryLdapAuthenticationProvider authenticationProvider() {
    return new ActiveDirectoryLdapAuthenticationProvider("example.com",
"ldap://company.example.com/");
}
```

XML

```
<bean id="authenticationProvider"

class="org.springframework.security.ldap.authentication.ad.ActiveDirectoryLdapAuth
enticationProvider">
    <constructor-arg value="example.com" />
    <constructor-arg value="ldap://company.example.com/" />
</bean>
```

Kotlin

```
@Bean
fun authenticationProvider(): ActiveDirectoryLdapAuthenticationProvider {
    return ActiveDirectoryLdapAuthenticationProvider("example.com",
"ldap://company.example.com/")
}
```

10.11. Gerenciamento de Sessões

A funcionalidade relacionada à sessão HTTP é gerenciada por uma combinação do **SessionManagementFilter** e da interface **SessionAuthenticationStrategy**, para a qual o filtro delega. O uso típico inclui proteção contra ataques de fixação de sessão, detecção de timeouts de sessão e restrições sobre quantas sessões um usuário autenticado pode ter abertas simultaneamente.

10.11.1. Detectando Timeouts

Você pode configurar o Spring Security para detectar a submissão de um ID de sessão inválido e redirecionar o usuário para uma URL apropriada. Isso é realizado através do elemento **session-management**:

```
<http>
  ...
  <session-management invalid-session-url="/invalidSession.htm" />
</http>
```

Observe que, se você usar esse mecanismo para detectar timeouts de sessão, ele pode erroneamente relatar um erro se o usuário fizer logout e depois se logar novamente sem fechar o navegador. Isso ocorre porque o cookie de sessão não é apagado quando você invalida a sessão e será reenviado, mesmo que o usuário tenha feito logout. Você pode ser capaz de excluir explicitamente o cookie **JSESSIONID** ao fazer logout, por exemplo, usando a seguinte sintaxe no manipulador de logout:

```
<http>
  <logout delete-cookies="JSESSIONID" />
</http>
```

Infelizmente, isso não pode ser garantido para funcionar em todos os containers de servlet, então você precisará testar em seu ambiente.

Nota:

Se você estiver executando sua aplicação atrás de um proxy, também pode ser possível remover o cookie de sessão configurando o servidor proxy. Por exemplo, usando o **mod_headers** do Apache HTTPD, a seguinte diretiva excluiria o cookie **JSESSIONID** expirando-o na resposta a uma solicitação de logout (supondo que a aplicação esteja implantada no caminho **/tutorial**):

```
<LocationMatch "/tutorial/logout">
  Header always set Set-Cookie "JSESSIONID=;Path=/tutorial;Expires=Thu, 01 Jan 1970 00:00:00 GMT"
</LocationMatch>
```

10.11.2. Controle de Sessões Concorrentes

Se você deseja colocar restrições sobre a capacidade de um único usuário de fazer login em sua aplicação, o Spring Security suporta isso de forma simples com as seguintes adições. Primeiro, você precisa adicionar o seguinte ouvinte ao seu arquivo **web.xml** para manter o Spring Security atualizado sobre os eventos do ciclo de vida da sessão:

```
<listener>
<listener-class>
  org.springframework.security.web.session.HttpSessionEventPublisher
</listener-class>
</listener>
```

Em seguida, adicione as seguintes linhas ao seu contexto de aplicação:

```
<http>
...
<session-management>
  <concurrency-control max-sessions="1" />
</session-management>
</http>
```

Isso impedirá que um usuário faça login várias vezes – um segundo login fará com que o primeiro seja invalidado. Normalmente, você preferiria impedir o segundo login, caso em que pode usar

```
<http>
```

```
...
<session-management>
  <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
</session-management>
</http>
```

O segundo login será então rejeitado. Por "rejeitado", queremos dizer que o usuário será redirecionado para o URL de falha de autenticação se o login baseado em formulário estiver sendo utilizado. Se a segunda autenticação ocorrer através de outro mecanismo não interativo, como "lembrar-me", um erro "não autorizado" (401) será enviado para o cliente. Caso contrário, se você quiser usar uma página de erro, pode adicionar o atributo session-authentication-error-url ao elemento de gerenciamento de sessão.
Se você estiver usando um filtro de autenticação personalizado para login baseado em formulário, então terá que configurar explicitamente o suporte ao controle de sessões simultâneas. Mais detalhes podem ser encontrados no capítulo de Gerenciamento de Sessões.

10.11.3. Proteção contra Ataques de Fixação de Sessão

Ataques de fixação de sessão são um risco potencial onde um atacante malicioso pode criar uma sessão acessando um site e, em seguida, persuadir outro usuário a fazer login com a mesma sessão (por exemplo, enviando um link contendo o identificador da sessão como um parâmetro). O Spring Security protege contra isso automaticamente, criando uma nova sessão ou alterando o ID da sessão quando um usuário faz login. Se você não exigir essa proteção, ou se ela entrar em conflito com algum outro requisito, pode controlar o comportamento usando o atributo session-fixation-protection no <session-management>, que tem quatro opções: • none - Não faz nada. A sessão original será mantida. • newSession - Cria uma nova sessão "limpa", sem copiar os dados da sessão existente (atributos relacionados ao Spring Security ainda serão copiados). • migrateSession - Cria uma nova sessão e copia todos os atributos da sessão existente para a nova sessão. Esta é a opção padrão em containers Servlet 3.0 ou mais antigos. • changeSessionId - Não cria uma nova sessão. Em vez disso, usa a proteção de fixação de sessão fornecida pelo container Servlet (HttpServletRequest#changeSessionId()). Esta opção está disponível apenas em containers Servlet 3.1 (Java EE 7) e mais novos. Especificá-la em containers mais antigos resultará em uma exceção. Esta é a opção padrão em containers Servlet 3.1 e mais novos.
Quando a proteção de fixação de sessão ocorre, resulta em um evento SessionFixationProtectionEvent sendo publicado no contexto da aplicação. Se você usar changeSessionId, essa proteção também fará com que quaisquer listeners de javax.servlet.http.HttpSessionIdListener sejam notificados, então use cautela se o seu código escutar ambos os eventos. Consulte o capítulo de Gerenciamento de Sessões para mais informações.

10.11.4. SessionManagementFilter

O SessionManagementFilter verifica o conteúdo do SecurityContextRepository em comparação com o conteúdo atual do SecurityContextHolder para determinar se um usuário foi autenticado durante a solicitação atual, tipicamente por um mecanismo de autenticação não interativo, como pré-autenticação ou "lembrar-me". Se o repositório contiver um contexto de segurança, o filtro não faz nada. Se não contiver, e o SecurityContext local do thread contiver um objeto de Autenticação (não anônimo), o filtro assume que o usuário foi autenticado por um filtro anterior na pilha. Ele então invoca o SessionAuthenticationStrategy configurado.
Se o usuário não estiver autenticado, o filtro verificará se um ID de sessão inválido foi solicitado (por exemplo, devido a um tempo limite) e invocará o InvalidSessionStrategy configurado, se houver um. O comportamento mais comum é redirecionar para um URL fixo, e isso é encapsulado na implementação padrão SimpleRedirectInvalidSessionStrategy. O último também é utilizado ao configurar um URL de sessão inválida através do namespace, conforme descrito anteriormente.

10.11.5. SessionAuthenticationStrategy

O SessionAuthenticationStrategy é usado tanto pelo SessionManagementFilter quanto pelo AbstractAuthenticationProcessingFilter, portanto, se você estiver usando uma classe de login de formulário personalizada, por exemplo, precisará injetá-la em ambos. Nesse caso, uma configuração típica, combinando o namespace e beans personalizados, pode se parecer com isso:

```
<http>
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />
<session-management session-authentication-strategy-ref="sas"/>
</http>
<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter"
>
  <beans:property name="sessionAuthenticationStrategy" ref="sas" />
  ...
```

```
</beans:bean>
<beans:bean id="sas" class=
"org.springframework.security.web.authentication.session.SessionFixationProtectionStra
tegy" />
```

Observe que o uso do padrão, SessionFixationProtectionStrategy, pode causar problemas se você estiver armazenando beans na sessão que implementam HttpSessionBindingListener, incluindo beans de escopo de sessão do Spring. Consulte a Javadoc desta classe para mais informações.

10.11.6. Controle de Concurrency

O Spring Security é capaz de impedir que um principal se autentique simultaneamente na mesma aplicação mais vezes do que um número especificado. Muitos fornecedores de software aproveitam isso para impor licenciamento, enquanto administradores de rede gostam dessa funcionalidade porque ajuda a evitar que as pessoas compartilhem nomes de login. Você pode, por exemplo, impedir que o usuário "Batman" faça login na aplicação web a partir de duas sessões diferentes. Você pode expirar o login anterior ou relatar um erro quando ele tentar fazer login novamente, impedindo o segundo login. Observe que, se você estiver usando a segunda abordagem, um usuário que não tenha feito logout explicitamente (mas que tenha fechado o navegador, por exemplo) não poderá fazer login novamente até que a sessão original expire.

O controle de concurrency é suportado pelo namespace, portanto, consulte o capítulo anterior do namespace para a configuração mais simples. Às vezes, no entanto, você precisará personalizar as coisas. A implementação usa uma versão especializada do SessionAuthenticationStrategy, chamada ConcurrentSessionControlAuthenticationStrategy.
Z

Anteriormente, a verificação de autenticação simultânea era feita pelo ProviderManager, que poderia ser injetado com um ConcurrentSessionController. O último verificava se o usuário estava tentando exceder o número de sessões permitidas. No entanto, essa abordagem exigia que uma sessão HTTP fosse criada com antecedência, o que não é desejável. No Spring Security 3, o usuário é primeiro autenticado pelo AuthenticationManager e, uma vez autenticado com sucesso, uma sessão é criada e é feita a verificação se ele pode ter outra sessão aberta.

Para usar o suporte a sessões simultâneas, você precisará adicionar o seguinte ao seu web.xml:

```
<listener>
  <listener-class>
  org.springframework.security.web.session.HttpSessionEventPublisher
  </listener-class>
</listener>
```

Além disso, você precisará adicionar o ConcurrentSessionFilter ao seu FilterChainProxy. O ConcurrentSessionFilter requer dois argumentos no construtor: sessionRegistry, que geralmente aponta para uma instância de SessionRegistryImpl, e sessionInformationExpiredStrategy, que define a estratégia a ser aplicada quando uma sessão expirar. Uma configuração usando o namespace para criar o FilterChainProxy e outros beans padrão pode ser semelhante a esta:

```
<http>
<custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter" />
<custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />
<session-management session-authentication-strategy-ref="sas"/>
</http>
<beans:bean id="redirectSessionInformationExpiredStrategy",
class="org.springframework.security.web.session.SimpleRedirectSessionInformationExpire
dStrategy">
<beans:constructor-arg name="invalidSessionUrl" value="/session-expired.htm" />
</beans:bean>
<beans:bean id="concurrencyFilter"
class="org.springframework.security.web.session.ConcurrentSessionFilter">
<beans:constructor-arg name="sessionRegistry" ref="sessionRegistry" />
<beans:constructor-arg name="sessionInformationExpiredStrategy"
ref="redirectSessionInformationExpiredStrategy" />
</beans:bean>
<beans:bean id="myAuthFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter"
>
<beans:property name="sessionAuthenticationStrategy" ref="sas" />
```

```
<beans:property name="authenticationManager" ref="authenticationManager" />
</beans:bean>
<beans:bean id="sas"
class="org.springframework.security.web.authentication.session.CompositeSessionAuthent
icationStrategy">
<beans:constructor-arg>
  <beans:list>
  <beans:bean
class="org.springframework.security.web.authentication.session.ConcurrentSessionContro
lAuthenticationStrategy">
  <beans:constructor-arg ref="sessionRegistry"/>
  <beans:property name="maximumSessions" value="1" />
  <beans:property name="exceptionIfMaximumExceeded" value="true" />
  </beans:bean>
  <beans:bean
class="org.springframework.security.web.authentication.session.SessionFixationProtecti
onStrategy">
  </beans:bean>
  <beans:bean
class="org.springframework.security.web.authentication.session.RegisterSessionAuthenti
cationStrategy">
  <beans:constructor-arg ref="sessionRegistry"/>
  </beans:bean>
  </beans:list>
</beans:constructor-arg>
</beans:bean>
<beans:bean id="sessionRegistry"
  class="org.springframework.security.core.session.SessionRegistryImpl" />
```

Adicionar o listener ao web.xml faz com que um ApplicationEvent seja publicado para o ApplicationContext do Spring toda vez que uma HttpSession começa ou termina. Isso é fundamental, pois permite que o SessionRegistryImpl seja notificado quando uma sessão termina. Sem isso, um usuário nunca poderá fazer login novamente uma vez que tenha excedido o número de sessões permitidas, mesmo que tenha feito logout de outra sessão ou ela tenha expirado.

Consultando o SessionRegistry para usuários autenticados atualmente e suas sessões

Configurar o controle de concorrência, seja por meio do namespace ou usando beans comuns, tem o efeito colateral útil de fornecer uma referência ao SessionRegistry, que você pode usar diretamente dentro de sua aplicação. Então, mesmo que você não queira restringir o número de sessões que um usuário pode ter, pode valer a pena configurar a infraestrutura de qualquer forma. Você pode definir a propriedade maximumSession como -1 para permitir sessões ilimitadas. Se estiver usando o namespace, pode configurar um alias para o SessionRegistry criado internamente usando o atributo session-registry-alias, fornecendo uma referência que pode ser injetada nos seus próprios beans.

O método getAllPrincipals() fornece uma lista dos usuários autenticados atualmente. Você pode listar as sessões de um usuário chamando o método getAllSessions(Object principal, boolean includeExpiredSessions), que retorna uma lista de objetos SessionInformation. Você também pode expirar a sessão de um usuário chamando expireNow() em uma instância de SessionInformation. Quando o usuário retornar à aplicação, será impedido de prosseguir. Esses métodos podem ser úteis, por exemplo, em uma aplicação de administração. Veja a Javadoc para mais informações.

10.12. Autenticação "Remember-Me"

10.12.1. Visão Geral

A autenticação "remember-me" ou login persistente refere-se a sites que podem lembrar a identidade de um principal entre sessões. Isso é tipicamente realizado enviando um cookie para o navegador, sendo que o cookie é detectado durante sessões futuras, causando o login automático. O Spring Security fornece os ganchos necessários para que essas operações ocorram e possui duas implementações concretas de "remember-me". Uma usa hashing para preservar a segurança dos tokens baseados em cookies e a outra usa um banco de dados ou outro mecanismo de armazenamento persistente para armazenar os tokens gerados.

Observe que ambas as implementações requerem um UserDetailsService. Se você estiver usando um provedor de autenticação que não usa um UserDetailsService (por exemplo, o provedor LDAP), isso não funcionará, a menos que você também tenha um bean UserDetailsService no contexto da sua aplicação.

10.12.2. Abordagem Simples com Token Baseado em Hash

Essa abordagem usa hashing para alcançar uma estratégia de "remember-me" útil. Essencialmente, um cookie é enviado para o navegador após uma autenticação interativa bem-sucedida, sendo que o cookie é composto da seguinte forma:

base64(username + ":" + expirationTime + ":" +
md5Hex(username + ":" + expirationTime + ":" password + ":" + key))
username: As identifiable to the UserDetailsService
password: That matches the one in the retrieved UserDetails
expirationTime: The date and time when the remember-me token expires, expressed in
milliseconds
key: A private key to prevent modification of the remember-me token

Assim, o token "remember-me" é válido apenas pelo período especificado, desde que o nome de usuário, a senha e a chave não mudem. Notavelmente, isso apresenta uma possível questão de segurança, pois um token "remember-me" capturado poderá ser usado de qualquer agente de usuário até que o token expire. Esse é o mesmo problema encontrado com a autenticação por digest. Se um principal souber que o token foi capturado, ele pode facilmente alterar sua senha e invalidar imediatamente todos os tokens "remember-me" emitidos. Se for necessário maior segurança, você deve usar a abordagem descrita na próxima seção. Como alternativa, os serviços "remember-me" simplesmente não devem ser usados.

Se você está familiarizado com os tópicos discutidos no capítulo sobre configuração do namespace, pode ativar a autenticação "remember-me" simplesmente adicionando o elemento <remember-me>:

```
<http>
...
<remember-me key="myAppKey"/>
</http>
```

O UserDetailsService normalmente será selecionado automaticamente. Se você tiver mais de um no seu contexto de aplicação, será necessário especificar qual deve ser utilizado com o atributo user-service-ref, onde o valor é o nome do seu bean UserDetailsService.

10.12.3. Abordagem de Token Persistente

Essa abordagem é baseada no artigo http://jaspan.com/improved_persistent_login_cookie_best_practice com algumas modificações menores [3]. Para usar essa abordagem com configuração de namespace, você deve fornecer uma referência para o datasource:

```
<http>
...
<remember-me data-source-ref="someDataSource"/>
</http>
```

O banco de dados deve conter uma tabela persistent_logins, criada usando o seguinte SQL (ou equivalente):

```
create table persistent_logins (username varchar(64) not null,   series varchar(64) primary key,   token varchar(64) not
null,   last_used timestamp not null)
```

10.12.4. Interfaces e Implementações de Lembrete de Sessão

O recurso "Remember-me" é utilizado com o UsernamePasswordAuthenticationFilter e implementado por meio de ganchos na superclasse AbstractAuthenticationProcessingFilter. Ele também é utilizado dentro do BasicAuthenticationFilter. Os ganchos irão invocar um serviço concreto de "RememberMeServices" nos momentos apropriados. A interface se parece com isto:

```
Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);
void loginFail(HttpServletRequest request, HttpServletResponse response);
void loginSuccess(HttpServletRequest request, HttpServletResponse response,
  Authentication successfulAuthentication);
```

Consulte a Javadoc para uma discussão mais detalhada sobre o que os métodos fazem, embora neste momento seja importante observar que o AbstractAuthenticationProcessingFilter só chama os métodos loginFail() e loginSuccess(). O método autoLogin() é chamado pelo RememberMeAuthenticationFilter sempre que o SecurityContextHolder não contém uma Authentication. Essa interface, portanto, fornece à implementação de "remember-me" a notificação adequada de eventos

relacionados à autenticação e delega para a implementação sempre que um pedido da web candidato possa conter um cookie e desejar ser lembrado. Esse design permite várias estratégias de implementação de "remember-me". Vimos acima que o Spring Security oferece duas implementações. Vamos analisá-las a seguir.

TokenBasedRememberMeServices

Essa implementação suporta a abordagem mais simples descrita na **Abordagem de Token Baseado em Hash Simples**. O TokenBasedRememberMeServices gera um RememberMeAuthenticationToken, que é processado pelo RememberMeAuthenticationProvider. Uma chave é compartilhada entre esse provedor de autenticação e o TokenBasedRememberMeServices. Além disso, o TokenBasedRememberMeServices requer um UserDetailsService, do qual pode recuperar o nome de usuário e a senha para fins de comparação de assinatura, e gerar o RememberMeAuthenticationToken para conter as GrantedAuthority corretas. Um comando de logout deve ser fornecido pela aplicação para invalidar o cookie, caso o usuário solicite. O TokenBasedRememberMeServices também implementa a interface LogoutHandler do Spring Security, podendo ser usado com o LogoutFilter para limpar o cookie automaticamente.

Os beans necessários no contexto de aplicação para habilitar os serviços de "remember-me" são os seguintes:

```
<bean id="rememberMeFilter" class=
"org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFi
lter">
<property name="rememberMeServices" ref="rememberMeServices"/>
<property name="authenticationManager" ref="theAuthenticationManager" />
</bean>
<bean id="rememberMeServices" class=
"org.springframework.security.web.authentication.rememberme.TokenBasedRememberMeServic
es">
<property name="userDetailsService" ref="myUserDetailsService"/>
<property name="key" value="springRocks"/>
</bean>
<bean id="rememberMeAuthenticationProvider" class=
"org.springframework.security.authentication.RememberMeAuthenticationProvider">
<property name="key" value="springRocks"/>
</bean>
```

Não se esqueça de adicionar sua implementação de RememberMeServices à propriedade UsernamePasswordAuthenticationFilter.setRememberMeServices(), incluir o RememberMeAuthenticationProvider na lista de provedores do AuthenticationManager.setProviders() e adicionar o RememberMeAuthenticationFilter no seu FilterChainProxy (geralmente imediatamente após o seu UsernamePasswordAuthenticationFilter).

PersistentTokenBasedRememberMeServices

Essa classe pode ser usada da mesma forma que o TokenBasedRememberMeServices, mas ela precisa ser configurada com um PersistentTokenRepository para armazenar os tokens. Existem duas implementações padrão:

- **InMemoryTokenRepositoryImpl**: destinada apenas para testes.
- **JdbcTokenRepositoryImpl**: armazena os tokens em um banco de dados.

O esquema do banco de dados é descrito anteriormente na **Abordagem de Token Persistente**.

10.13. Suporte ao OpenID

O namespace oferece suporte ao login OpenID, que pode ser usado em vez do login baseado em formulário normal ou adicionalmente a ele, com uma simples alteração:

```
<http>
<intercept-url pattern="/**" access="ROLE_USER" />
<openid-login />
</http>
```

Você deve então se registrar com um provedor OpenID (como myopenid.com) e adicionar as informações do usuário ao seu <user-service> na memória:

```
<user name="https://jimi.hendrix.myopenid.com/" authorities="ROLE_USER" />
```

Você deverá conseguir fazer login usando o site myopenid.com para autenticar. Também é possível selecionar um bean específico do UserDetailsService para ser usado com o OpenID configurando o atributo user-service-ref no elemento openid-login. Observe que omitimos o atributo password na configuração do usuário acima, pois esse conjunto de dados de usuário está sendo usado apenas para carregar as autoridades do usuário. Uma senha aleatória

será gerada internamente, impedindo que você use acidentalmente esses dados de usuário como uma fonte de autenticação em outras partes da sua configuração.

10.13.1. Troca de Atributos

Suporte para a troca de atributos do OpenID. Como exemplo, a seguinte configuração tentaria recuperar o e-mail e o nome completo do provedor OpenID, para uso no aplicativo:

```
<openid-login>
<attribute-exchange>
  <openid-attribute name="email" type="https://axschema.org/contact/email"
required="true"/>
  <openid-attribute name="name" type="https://axschema.org/namePerson"/>
</attribute-exchange>
</openid-login>
```

O "tipo" de cada atributo OpenID é um URI, determinado por um esquema específico, neste caso, https://axschema.org/. Se um atributo precisar ser recuperado para a autenticação bem-sucedida, o atributo necessário pode ser configurado. O esquema exato e os atributos suportados dependerão do seu provedor OpenID. Os valores dos atributos são retornados como parte do processo de autenticação e podem ser acessados posteriormente usando o seguinte código:

```
OpenIDAuthenticationToken token =
  (OpenIDAuthenticationToken)SecurityContextHolder.getContext().getAuthentication();
List<OpenIDAttribute> attributes = token.getAttributes();
```

Podemos obter o OpenIDAuthenticationToken do SecurityContextHolder. O OpenIDAttribute contém o tipo de atributo e o valor recuperado (ou valores, no caso de atributos multivalorados). Você pode fornecer múltiplos elementos de troca de atributo, usando um atributo identifier-matcher em cada um deles. Esse atributo contém uma expressão regular que será comparada ao identificador OpenID fornecido pelo usuário. Veja o exemplo de aplicação OpenID no código-fonte para uma configuração de exemplo, fornecendo listas de atributos diferentes para os provedores Google, Yahoo e MyOpenID.

10.14. Autenticação Anônima

10.14.1. Visão Geral

Geralmente, é considerado uma boa prática de segurança adotar o princípio do "negar por padrão", onde você especifica explicitamente o que é permitido e nega tudo o mais. Definir o que é acessível para usuários não autenticados é uma situação semelhante, especialmente para aplicações web. Muitos sites exigem que os usuários sejam autenticados para acessar qualquer coisa, exceto algumas URLs (por exemplo, as páginas iniciais e de login). Nesse caso, é mais fácil definir atributos de configuração de acesso para essas URLs específicas, em vez de ter que configurar para cada recurso protegido. Em outras palavras, às vezes é conveniente dizer que a ROLE_SOMETHING é necessária por padrão e permitir apenas algumas exceções a essa regra, como as páginas de login, logout e home de uma aplicação. Você também poderia omitir essas páginas da cadeia de filtros completamente, assim ignorando as verificações de controle de acesso, mas isso pode ser indesejável por outros motivos, especialmente se as páginas se comportarem de maneira diferente para usuários autenticados.

Isso é o que queremos dizer com autenticação anônima. Note que não há uma diferença conceitual real entre um usuário que está "autenticado anonimamente" e um usuário não autenticado. A autenticação anônima do Spring Security simplesmente oferece uma maneira mais conveniente de configurar os atributos de controle de acesso. Chamadas para a API do servlet, como getCallerPrincipal, por exemplo, ainda retornarão null, mesmo que haja um objeto de autenticação anônima no SecurityContextHolder.

Existem outras situações onde a autenticação anônima é útil, como quando um interceptor de auditoria consulta o SecurityContextHolder para identificar qual principal foi responsável por uma operação dada. Classes podem ser criadas de maneira mais robusta se souberem que o SecurityContextHolder sempre contém um objeto Authentication e nunca null.

10.14.2. Configuração

O suporte à autenticação anônima é fornecido automaticamente ao usar a configuração HTTP do Spring Security 3.0 e pode ser personalizado (ou desabilitado) usando o elemento `<anonymous>`. Você não precisa configurar os beans descritos aqui, a menos que esteja usando configuração tradicional de beans. São três as classes que juntas fornecem o recurso de autenticação anônima:

- **AnonymousAuthenticationToken** é uma implementação de Authentication e armazena as GrantedAuthoritys que se aplicam ao principal anônimo.
- Há um **AnonymousAuthenticationProvider** correspondente, que é encadeado no ProviderManager, permitindo que os AnonymousAuthenticationTokens sejam aceitos.

- Por fim, existe um **AnonymousAuthenticationFilter**, que é encadeado após os mecanismos de autenticação normais e adiciona automaticamente um AnonymousAuthenticationToken ao SecurityContextHolder se não houver uma autenticação existente lá.

A definição do filtro e do provedor de autenticação aparece da seguinte forma:

```
<bean id="anonymousAuthFilter"
class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">
<property name="key" value="foobar"/>
<property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>
<bean id="anonymousAuthenticationProvider"
class="org.springframework.security.authentication.AnonymousAuthenticationProvider">
<property name="key" value="foobar"/>
</bean>
```

A chave é compartilhada entre o filtro e o provedor de autenticação, para que os tokens criados pelo primeiro sejam aceitos pelo último [4]. O atributo **userAttribute** é expresso na forma de usernameInTheAuthenticationToken, grantedAuthority[,grantedAuthority]. Esta é a mesma sintaxe usada após o sinal de igual para a propriedade **userMap** do **InMemoryDaoImpl**.

Como explicado anteriormente, o benefício da autenticação anônima é que todos os padrões de URI podem ter segurança aplicada a eles. Por exemplo:

```
<bean id="filterSecurityInterceptor"
class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
<property name="authenticationManager" ref="authenticationManager"/>
<property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
<property name="securityMetadata">
  <security:filter-security-metadata-source>
  <security:intercept-url pattern='/index.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
  <security:intercept-url pattern='/hello.htm' access='ROLE_ANONYMOUS,ROLE_USER'/>
  <security:intercept-url pattern='/logoff.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
  <security:intercept-url pattern='/login.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
  <security:intercept-url pattern='/**' access='ROLE_USER'/>
  </security:filter-security-metadata-source>" +
</property>
</bean>
```

10.14.3. **AuthenticationTrustResolver**

Concluindo a discussão sobre autenticação anônima, temos a interface **AuthenticationTrustResolver** e sua implementação correspondente, **AuthenticationTrustResolverImpl**. Esta interface fornece o método **isAnonymous(Authentication)**, que permite que classes interessadas levem em consideração esse tipo especial de status de autenticação. O **ExceptionTranslationFilter** utiliza essa interface no processamento de **AccessDeniedException**. Se uma **AccessDeniedException** for lançada e a autenticação for do tipo anônimo, em vez de retornar uma resposta 403 (proibido), o filtro irá iniciar o **AuthenticationEntryPoint**, permitindo que o principal se autentique corretamente.

Essa distinção é necessária, caso contrário, os principais seriam sempre considerados "autenticados" e nunca teriam a oportunidade de fazer login por meio de um formulário, autenticação básica, digest ou outro mecanismo de autenticação normal.

Você frequentemente verá o atributo **ROLE_ANONYMOUS** na configuração do interceptor acima sendo substituído por **IS_AUTHENTICATED_ANONYMOUSLY**, que é efetivamente a mesma coisa ao definir controles de acesso. Este é um exemplo do uso do **AuthenticatedVoter**, que veremos no capítulo sobre autorização. Ele usa um **AuthenticationTrustResolver** para processar esse atributo de configuração específico e conceder acesso a usuários anônimos. A abordagem do **AuthenticatedVoter** é mais poderosa, pois permite diferenciar entre usuários anônimos, lembrados e totalmente autenticados. Se você não precisar dessa funcionalidade, pode continuar com **ROLE_ANONYMOUS**, que será processado pelo **RoleVoter** padrão do Spring Security.

10.15. **Cenários de Pré-Autenticação**

Existem situações em que você deseja usar o Spring Security para autorização, mas o usuário já foi autenticado de forma confiável por algum sistema externo antes de acessar a aplicação. Chamamos essas situações de "cenários de pré-autenticação". Exemplos incluem X.509, Siteminder e autenticação pelo contêiner Java EE no qual a aplicação está sendo executada.

Quando se usa a pré-autenticação, o Spring Security precisa:

- Identificar o usuário que está fazendo a solicitação.

- Obter as autoridades do usuário.

Os detalhes dependem do mecanismo de autenticação externo. O usuário pode ser identificado pelas informações do certificado no caso de X.509, ou por um cabeçalho de solicitação HTTP no caso de Siteminder. Se depender da autenticação do contêiner, o usuário será identificado chamando o método **getUserPrincipal()** na solicitação HTTP de entrada. Em alguns casos, o mecanismo externo pode fornecer informações sobre as funções/autoridades do usuário, mas em outros casos, as autoridades devem ser obtidas de uma fonte separada, como um **UserDetailsService**.

10.15.1. **Classes do Framework de Pré-Autenticação**

Como a maioria dos mecanismos de pré-autenticação segue o mesmo padrão, o Spring Security possui um conjunto de classes que fornecem um framework interno para implementar provedores de autenticação pré-autenticados. Isso elimina a duplicação e permite que novas implementações sejam adicionadas de forma estruturada, sem a necessidade de escrever tudo do zero. Você não precisa conhecer essas classes se deseja usar algo como a autenticação X.509, pois ela já possui uma opção de configuração de namespace que é mais simples de usar e começar. Se você precisar usar configuração explícita de beans ou planeja escrever sua própria implementação, então entender como as implementações fornecidas funcionam será útil. Você encontrará classes em **org.springframework.security.web.authentication.preauth**. Aqui, fornecemos apenas um esboço, então consulte o Javadoc e o código-fonte conforme necessário.

AbstractPreAuthenticatedProcessingFilter

Esta classe verificará o conteúdo atual do contexto de segurança e, se estiver vazio, tentará extrair informações do usuário da solicitação HTTP e enviá-las ao **AuthenticationManager**. As subclasses sobrescrevem os seguintes métodos para obter essas informações:

```
protected abstract Object getPreAuthenticatedPrincipal(HttpServletRequest request);
protected abstract Object getPreAuthenticatedCredentials(HttpServletRequest request);
```

Após chamar esses métodos, o filtro criará um **PreAuthenticatedAuthenticationToken** contendo os dados retornados e os submeterá à autenticação. Por "autenticação", aqui estamos nos referindo ao processamento adicional para, talvez, carregar as autoridades do usuário, mas a arquitetura padrão de autenticação do Spring Security é seguida.

Como outros filtros de autenticação do Spring Security, o filtro de pré-autenticação possui uma propriedade **authenticationDetailsSource**, que por padrão criará um objeto **WebAuthenticationDetails** para armazenar informações adicionais, como o identificador de sessão e o endereço IP de origem na propriedade **details** do objeto **Authentication**. Em casos onde as informações sobre o papel do usuário podem ser obtidas pelo mecanismo de pré-autenticação, esses dados também são armazenados nesta propriedade, com os detalhes implementando a interface **GrantedAuthoritiesContainer**. Isso permite que o provedor de autenticação leia as autoridades que foram alocadas externamente ao usuário. Vamos ver um exemplo concreto a seguir.

J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource

Se o filtro for configurado com uma **authenticationDetailsSource** que é uma instância desta classe, as informações de autoridade são obtidas chamando o método **isUserInRole(String role)** para cada um de um conjunto pré-determinado de "papéis mapeáveis". A classe obtém esses papéis de um **MappableAttributesRetriever** configurado. Implementações possíveis incluem codificar uma lista no contexto da aplicação ou ler as informações de papel do elemento **<security-role>** em um arquivo **web.xml**. O exemplo de aplicação de pré-autenticação usa a segunda abordagem.

Há uma etapa adicional onde os papéis (ou atributos) são mapeados para objetos **GrantedAuthority** do Spring Security usando um **Attributes2GrantedAuthoritiesMapper** configurado. O padrão apenas adicionará o prefixo **ROLE_** aos nomes, mas ele oferece controle total sobre o comportamento.

PreAuthenticatedAuthenticationProvider

O provedor de pré-autenticação tem pouco mais a fazer do que carregar o objeto **UserDetails** para o usuário. Ele faz isso delegando a tarefa para um **AuthenticationUserDetailsService**. Este último é semelhante ao **UserDetailsService** padrão, mas recebe um objeto **Authentication** em vez de apenas o nome de usuário.

```
public interface AuthenticationUserDetailsService {
  UserDetails loadUserDetails(Authentication token) throws
UsernameNotFoundException;
}
```

Esta interface pode ter outros usos, mas no caso de pré-autenticação, ela permite o acesso às autoridades que foram embaladas no objeto **Authentication**, como vimos na seção anterior. A classe

PreAuthenticatedGrantedAuthoritiesUserDetailsService faz isso. Alternativamente, ela pode delegar para um **UserDetailsService** padrão por meio da implementação **UserDetailsByNameServiceWrapper**.

Http403ForbiddenEntryPoint

O **AuthenticationEntryPoint** é responsável por iniciar o processo de autenticação para um usuário não autenticado (quando ele tenta acessar um recurso protegido), mas no caso de pré-autenticação isso não se aplica. Você só configuraria o **ExceptionTranslationFilter** com uma instância desta classe se não estiver usando pré-autenticação em combinação com outros mecanismos de autenticação. Ele será chamado se o usuário for rejeitado pelo **AbstractPreAuthenticatedProcessingFilter**, resultando em uma autenticação nula. Ele sempre retorna um código de resposta **403 Forbidden** se chamado.

10.15.2. Implementações Concretas

A autenticação **X.509** é abordada em seu próprio capítulo. Aqui, veremos algumas classes que fornecem suporte para outros cenários de pré-autenticação.

Autenticação via Cabeçalho de Requisição (Siteminder)

Um sistema de autenticação externo pode fornecer informações para a aplicação definindo cabeçalhos específicos na requisição HTTP. Um exemplo bem conhecido disso é o **Siteminder**, que passa o nome de usuário em um cabeçalho chamado **SM_USER**. Esse mecanismo é suportado pela classe **RequestHeaderAuthenticationFilter**, que simplesmente extrai o nome de usuário do cabeçalho. Ele usa **SM_USER** como nome do cabeçalho por padrão. Consulte a Javadoc para mais detalhes.

Nota: Ao usar um sistema como esse, o framework não realiza nenhuma verificação de autenticação, e é extremamente importante que o sistema externo esteja configurado corretamente e proteja todo o acesso à aplicação. Se um atacante for capaz de forjar os cabeçalhos em sua requisição original sem que isso seja detectado, ele poderia potencialmente escolher qualquer nome de usuário que desejasse.

Exemplo de Configuração do Siteminder

Uma configuração típica usando esse filtro seria a seguinte:

```
<security:http>
<!-- Additional http configuration omitted -->
<security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />
</security:http>
<bean id="siteminderFilter"
class="org.springframework.security.web.authentication.preauth.RequestHeaderAuthentica
tionFilter">
<property name="principalRequestHeader" value="SM_USER"/>
<property name="authenticationManager" ref="authenticationManager" />
</bean>
<bean id="preauthAuthProvider"
class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthent
icationProvider">
<property name="preAuthenticatedUserDetailsService">
  <bean id="userDetailsServiceWrapper"
class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
  <property name="userDetailsService" ref="userDetailsService"/>
  </bean>
</property>
</bean>
<security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="preauthAuthProvider" />
</security:authentication-manager>
```

Autenticação no Contêiner Java EE

A classe **J2eePreAuthenticatedProcessingFilter** irá extrair o nome de usuário da propriedade **userPrincipal** do **HttpServletRequest**. O uso desse filtro normalmente seria combinado com o uso de funções de **Java EE roles** descritas anteriormente em **J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource**. Há um aplicativo de exemplo no código-fonte que usa essa abordagem, então, se você estiver interessado, pode acessar o repositório do GitHub e examinar o arquivo de contexto da aplicação na pasta **samples/xml/preauth**.

10.16. Java Authentication and Authorization Service (JAAS) Provider

10.16.1. Visão Geral

O **Spring Security** oferece um pacote capaz de delegar as requisições de autenticação para o **Java Authentication and Authorization Service (JAAS)**. Este pacote é discutido com mais detalhes abaixo.

10.16.2. AbstractJaasAuthenticationProvider

O **AbstractJaasAuthenticationProvider** é a base para as implementações fornecidas pelo **JAAS AuthenticationProvider**. Subclasses precisam implementar um método que cria o **LoginContext**. O **AbstractJaasAuthenticationProvider** possui diversas dependências que podem ser injetadas e que são discutidas abaixo.

JAAS CallbackHandler

A maioria dos **LoginModules** do **JAAS** requer um **callback** de algum tipo. Esses **callbacks** são normalmente usados para obter o nome de usuário e senha do usuário. Em uma implementação **Spring Security**, a interação do usuário será gerenciada pelo próprio **Spring Security** (via o mecanismo de autenticação). Portanto, quando a requisição de autenticação for delegada ao **JAAS**, o mecanismo de autenticação do **Spring Security** já terá populado completamente um objeto **Authentication**, contendo todas as informações necessárias pelo **JAAS LoginModule**. Assim, o pacote **JAAS** para **Spring Security** fornece dois manipuladores de callback padrão: **JaasNameCallbackHandler** e **JaasPasswordCallbackHandler**. Cada um desses manipuladores de callback implementa a interface **JaasAuthenticationCallbackHandler**. Em muitos casos, esses manipuladores de callback podem ser usados diretamente sem a necessidade de entender o funcionamento interno.

Para quem precisa de controle total sobre o comportamento dos **callbacks**, o **AbstractJaasAuthenticationProvider** envolve esses manipuladores com o **InternalCallbackHandler**. O **InternalCallbackHandler** é a classe que implementa a interface padrão **CallbackHandler** do **JAAS**. Cada vez que o **JAAS LoginModule** for utilizado, ele recebe uma lista de **InternalCallbackHandlers** configurados no contexto da aplicação. Se o **LoginModule** solicitar um **callback**, ele será passado para os manipuladores de **JaasAuthenticationCallbackHandler** envolvidos.

JAAS AuthorityGranter

O **JAAS** trabalha com **principals**. Até "roles" (funções ou papéis) são representados como **principals** no **JAAS**. O **Spring Security**, por outro lado, trabalha com objetos **Authentication**, que contêm um único **principal** e várias **GrantedAuthorities**. Para facilitar a correspondência entre esses conceitos, o pacote **JAAS** do **Spring Security** inclui a interface **AuthorityGranter**.

Um **AuthorityGranter** é responsável por inspecionar um **principal** do **JAAS** e retornar um conjunto de **Strings**, representando as autoridades atribuídas ao **principal**. Para cada string de autoridade retornada, o **AbstractJaasAuthenticationProvider** cria um **JaasGrantedAuthority** (que implementa a interface **GrantedAuthority** do **Spring Security**) contendo a string de autoridade e o **JAAS principal** passado ao **AuthorityGranter**. O **AbstractJaasAuthenticationProvider** obtém os **principals** do **JAAS** autenticando com sucesso as credenciais do usuário usando o **JAAS LoginModule** e acessando o **LoginContext** retornado. Uma chamada para **LoginContext.getSubject().getPrincipals()** é feita, com cada **principal** resultante sendo passado para cada **AuthorityGranter** definido na propriedade **setAuthorityGranters(List)** do **AbstractJaasAuthenticationProvider**.

O **Spring Security** não inclui **AuthorityGranters** de produção, dado que cada **JAAS principal** tem um significado específico da implementação. No entanto, há um **TestAuthorityGranter** nos testes unitários que demonstra uma implementação simples de **AuthorityGranter**.

10.16.3. DefaultJaasAuthenticationProvider

O **DefaultJaasAuthenticationProvider** permite que um objeto de **JAAS Configuration** seja injetado como dependência. Ele cria um **LoginContext** usando a **JAAS Configuration** injetada. Isso significa que o **DefaultJaasAuthenticationProvider** não é vinculado a uma implementação específica da **Configuration**, como o **JaasAuthenticationProvider** é.

InMemoryConfiguration

Para facilitar a injeção de uma **Configuration** no **DefaultJaasAuthenticationProvider**, uma implementação padrão em memória chamada **InMemoryConfiguration** é fornecida. O construtor da implementação aceita um **Map**, onde cada chave representa um nome de configuração de login e o valor representa um **Array** de **AppConfigurationEntry**. A **InMemoryConfiguration** também suporta um **Array** padrão de **AppConfigurationEntry** que será usado se nenhum mapeamento for encontrado no **Map** fornecido.

Exemplo de Configuração do DefaultJaasAuthenticationProvider

Embora a configuração do **InMemoryConfiguration** no **Spring** possa ser mais verbosa do que os arquivos padrão de configuração do **JAAS**, usá-la em conjunto com o **DefaultJaasAuthenticationProvider** é mais flexível do que o **JaasAuthenticationProvider**, já que não depende da implementação padrão de **Configuration**.

Um exemplo de configuração do **DefaultJaasAuthenticationProvider** usando **InMemoryConfiguration** é mostrado abaixo. Note que implementações personalizadas de **Configuration** também podem ser facilmente injetadas no **DefaultJaasAuthenticationProvider**.

```
<bean id="jaasAuthProvider"
class="org.springframework.security.authentication.jaas.DefaultJaasAuthenticationProvi
der">
<property name="configuration">
<bean
class="org.springframework.security.authentication.jaas.memory.InMemoryConfiguration">
<constructor-arg>
  <map>
  <!--
  SPRINGSECURITY is the default loginContextName
  for AbstractJaasAuthenticationProvider
  -->
  <entry key="SPRINGSECURITY">
  <array>
  <bean class="javax.security.auth.login.AppConfigurationEntry">
  <constructor-arg value="sample.SampleLoginModule" />
  <constructor-arg>
  <util:constant static-field=
"javax.security.auth.login.AppConfigurationEntry$LoginModuleControlFlag.REQUIRED"/>
  </constructor-arg>
  <constructor-arg>
  <map></map>
  </constructor-arg>
  </bean>
  </array>
  </entry>
  </map>
  </constructor-arg>
</bean>
</property>
<property name="authorityGranters">
<list>
  <!-- You will need to write your own implementation of AuthorityGranter -->
  <bean
class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
</list>
</property>
</bean>
```

10.16.4. JaasAuthenticationProvider

O **JaasAuthenticationProvider** assume que a configuração padrão é uma instância de **ConfigFile**. Essa suposição é feita com o objetivo de tentar atualizar a configuração. O **JaasAuthenticationProvider** então usa a configuração padrão para criar o **LoginContext**.

Vamos assumir que temos um arquivo de configuração de login **JAAS**, **/WEB-INF/login.conf**, com o seguinte conteúdo:

```
JAASTest {
  sample.SampleLoginModule required;
};
```

Como todos os beans do Spring Security, o **JaasAuthenticationProvider** é configurado através do contexto da aplicação. As seguintes definições corresponderiam ao arquivo de configuração de login **JAAS** mencionado acima:

```
<bean id="jaasAuthenticationProvider"
class="org.springframework.security.authentication.jaas.JaasAuthenticationProvider">
<property name="loginConfig" value="/WEB-INF/login.conf"/>
<property name="loginContextName" value="JAASTest"/>
<property name="callbackHandlers">
```

```
<list>
<bean
  class="org.springframework.security.authentication.jaas.JaasNameCallbackHandler"/>
<bean
class="org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler"/>
</list>
</property>
<property name="authorityGranters">
  <list>
  <bean
class="org.springframework.security.authentication.jaas.TestAuthorityGranter"/>
  </list>
</property>
</bean>
```

10.16.5. Executando como um **Subject**

Se configurado, o **JaasApiIntegrationFilter** tentará executar como o **Subject** no **JaasAuthenticationToken**. Isso significa que o **Subject** pode ser acessado usando:

```
Subject subject = Subject.getSubject(AccessController.getContext());
```

Esta integração pode ser facilmente configurada utilizando o atributo jaas-api-provision. Esse recurso é útil quando se integra com APIs legadas ou externas que dependem do preenchimento do Subject do JAAS.

10.17. Autenticação CAS

10.17.1. Visão geral

O JA-SIG produz um sistema de autenticação única corporativa conhecido como CAS. Ao contrário de outras iniciativas, o Central Authentication Service do JA-SIG é open-source, amplamente utilizado, simples de entender, independente de plataforma e suporta capacidades de proxy. O Spring Security oferece suporte total ao CAS e fornece um caminho fácil de migração de implantações de Spring Security de uma única aplicação para implantações de múltiplas aplicações protegidas por um servidor CAS corporativo.

Você pode aprender mais sobre o CAS em https://www.apereo.org. Você também precisará visitar este site para baixar os arquivos do servidor CAS.

10.17.2. Como o CAS funciona

Embora o site do CAS contenha documentos detalhados sobre a arquitetura do CAS, apresentamos a visão geral novamente aqui, no contexto do Spring Security. O Spring Security 3.x oferece suporte ao CAS 3. No momento da escrita, o servidor CAS estava na versão 3.4.

Em algum lugar na sua empresa, você precisará configurar um servidor CAS. O servidor CAS é simplesmente um arquivo WAR padrão, então não há nada difícil em configurar seu servidor. Dentro do arquivo WAR, você personalizará as páginas de login e outras páginas de autenticação única exibidas para os usuários.

Ao implantar um servidor CAS 3.4, você também precisará especificar um AuthenticationHandler no deployerConfigContext.xml incluído com o CAS. O AuthenticationHandler tem um método simples que retorna um booleano indicando se um conjunto de credenciais é válido. Sua implementação do AuthenticationHandler precisará se conectar a algum tipo de repositório de autenticação, como um servidor LDAP ou banco de dados. O próprio CAS inclui diversos AuthenticationHandlers prontos para ajudar nisso. Quando você baixar e implantar o arquivo WAR do servidor, ele estará configurado para autenticar com sucesso usuários que insiram uma senha correspondente ao nome de usuário, o que é útil para testes.

Além do servidor CAS, os outros principais componentes são, claro, as aplicações web seguras implantadas em sua empresa. Essas aplicações web são conhecidas como "serviços". Existem três tipos de serviços: aqueles que autenticam tickets de serviço, aqueles que podem obter tickets de proxy e aqueles que autenticam tickets de proxy. A autenticação de um ticket de proxy difere porque a lista de proxies deve ser validada e, muitas vezes, um ticket de proxy pode ser reutilizado.

Sequência de interação do Spring Security e CAS

A interação básica entre um navegador web, o servidor CAS e um serviço protegido pelo Spring Security é a seguinte:

- O usuário da web está navegando nas páginas públicas do serviço. O CAS ou o Spring Security não estão envolvidos.
- O usuário eventualmente solicita uma página que seja segura ou tenha algum dos beans que utiliza sendo seguro. O ExceptionTranslationFilter do Spring Security detectará a AccessDeniedException ou a AuthenticationException.
- Como o objeto Authentication do usuário (ou sua falta) causou uma AuthenticationException, o ExceptionTranslationFilter chamará o AuthenticationEntryPoint configurado. Se estiver usando o CAS, isso será a classe CasAuthenticationEntryPoint.

- O CasAuthenticationEntryPoint redirecionará o navegador do usuário para o servidor CAS. Ele também indicará um parâmetro de serviço, que é a URL de callback do serviço Spring Security (sua aplicação). Por exemplo, a URL para a qual o navegador será redirecionado pode ser https://my.company.com/cas/login?service=https%3A%2F%2Fserver3.company.com%2Fwebapp%2Flogin/cas.
- Depois que o navegador do usuário for redirecionado para o CAS, ele será solicitado a informar seu nome de usuário e senha. Se o usuário apresentar um cookie de sessão que indique que ele já fez login anteriormente, não será solicitado que ele faça login novamente (há uma exceção para esse procedimento, que será abordada mais adiante). O CAS usará o PasswordHandler (ou AuthenticationHandler se estiver usando o CAS 3.0) mencionado acima para decidir se o nome de usuário e a senha são válidos.
- Após o login bem-sucedido, o CAS redirecionará o navegador do usuário de volta para o serviço original. Ele também incluirá um parâmetro de ticket, que é uma string opaca representando o "ticket de serviço". Continuando o exemplo anterior, a URL para a qual o navegador é redirecionado pode ser https://server3.company.com/webapp/login/cas?ticket=ST-0-ER94xMJmn6pha35CQRoZ.

A tradução segue a estrutura do texto técnico de maneira concisa. Se precisar de mais detalhes ou continuar com mais traduções, posso ajudar!

```
<bean id="serviceProperties"
  class="org.springframework.security.cas.ServiceProperties">
<property name="service"
  value="https://localhost:8443/cas-sample/login/cas"/>
<property name="sendRenew" value="false"/>
</bean>
```

O serviço deve ser igual a uma URL que será monitorada pelo CasAuthenticationFilter. O parâmetro sendRenew tem o valor padrão de falso, mas deve ser configurado como verdadeiro se sua aplicação for particularmente sensível. O que esse parâmetro faz é informar ao serviço de login do CAS que o login de autenticação única (single sign-on) não é aceitável. Em vez disso, o usuário precisará reintroduzir seu nome de usuário e senha para obter acesso ao serviço.

Os seguintes beans devem ser configurados para iniciar o processo de autenticação CAS (supondo que você esteja utilizando uma configuração baseada em namespace):

```
<security:http entry-point-ref="casEntryPoint">
...
<security:custom-filter position="CAS_FILTER" ref="casFilter" />
</security:http>
<bean id="casFilter"
  class="org.springframework.security.cas.web.CasAuthenticationFilter">
<property name="authenticationManager" ref="authenticationManager"/>
</bean>
<bean id="casEntryPoint"
  class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
<property name="loginUrl" value="https://localhost:9443/cas/login"/>
<property name="serviceProperties" ref="serviceProperties"/>
</bean>
```

Para o CAS funcionar, o ExceptionTranslationFilter deve ter a propriedade authenticationEntryPoint configurada para o bean CasAuthenticationEntryPoint. Isso pode ser feito facilmente usando o entry-point-ref, como mostrado no exemplo acima. O CasAuthenticationEntryPoint deve se referir ao bean ServiceProperties (discutido acima), que fornece a URL do servidor de login CAS da empresa. É para esse servidor que o navegador do usuário será redirecionado.
O CasAuthenticationFilter possui propriedades muito semelhantes às do UsernamePasswordAuthenticationFilter (usado para logins baseados em formulários). Você pode usar essas propriedades para personalizar comportamentos como o sucesso e a falha da autenticação.

Em seguida, você precisa adicionar um CasAuthenticationProvider e seus colaboradores.

```
<security:authentication-manager alias="authenticationManager">
<security:authentication-provider ref="casAuthenticationProvider" />
</security:authentication-manager>
<bean id="casAuthenticationProvider"
  class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
<property name="authenticationUserDetailsService">
  <bean
class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
  <constructor-arg ref="userService" />
  </bean>
</property>
<property name="serviceProperties" ref="serviceProperties" />
<property name="ticketValidator">
  <bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
  <constructor-arg index="0" value="https://localhost:9443/cas" />
  </bean>
</property>
<property name="key" value="an_id_for_this_auth_provider_only"/>
</bean>
<security:user-service id="userService">
<!-- Password is prefixed with {noop} to indicate to DelegatingPasswordEncoder that
NoOpPasswordEncoder should be used.
This is not safe for production, but makes reading
in samples easier.
Normally passwords should be hashed using BCrypt -->
<security:user name="joe" password="{noop}joe" authorities="ROLE_USER" />
...
</security:user-service>
```

O CasAuthenticationProvider usa uma instância do UserDetailsService para carregar as autoridades de um usuário, uma vez que ele tenha sido autenticado pelo CAS. Mostramos uma configuração simples na memória aqui. Note que o CasAuthenticationProvider não utiliza a senha para autenticação, mas sim as autoridades.
Os beans são todos razoavelmente autoexplicativos se você consultar a seção "Como o CAS Funciona".
Isso completa a configuração mais básica para o CAS. Se você não cometeu erros, sua aplicação web deverá funcionar corretamente dentro do framework de Single Sign-On (SSO) do CAS. Nenhuma outra parte do Spring Security precisa se preocupar com o fato de que o CAS lidou com a autenticação. Nas seções seguintes, discutiremos algumas configurações mais avançadas (opcionais).

Logout Único

O protocolo CAS suporta o Logout Único e pode ser facilmente adicionado à sua configuração do Spring Security. Abaixo estão as atualizações na configuração do Spring Security que lidam com o Logout Único.

```
<security:http entry-point-ref="casEntryPoint">
...
<security:logout logout-success-url="/cas-logout.jsp"/>
<security:custom-filter ref="requestSingleLogoutFilter" before="LOGOUT_FILTER"/>
<security:custom-filter ref="singleLogoutFilter" before="CAS_FILTER"/>
</security:http>
<!-- This filter handles a Single Logout Request from the CAS Server -->
<bean id="singleLogoutFilter"
class="org.jasig.cas.client.session.SingleSignOutFilter"/>
<!-- This filter redirects to the CAS Server to signal Single Logout should be
performed -->
<bean id="requestSingleLogoutFilter"
  class="org.springframework.security.web.authentication.logout.LogoutFilter">
<constructor-arg value="https://localhost:9443/cas/logout"/>
<constructor-arg>
  <bean class=
"org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/
>
</constructor-arg>
<property name="filterProcessesUrl" value="/logout/cas"/>
</bean>
```

O elemento logout faz o logout do usuário da aplicação local, mas não termina a sessão com o servidor CAS ou com qualquer outra aplicação na qual o usuário tenha se autenticado. O filtro requestSingleLogoutFilter permitirá que a URL /spring_security_cas_logout seja requisitada para redirecionar a aplicação para a URL de logout do servidor CAS configurado. Em seguida, o servidor CAS enviará uma solicitação de Logout Único para todos os serviços nos quais o usuário estava autenticado. O filtro singleLogoutFilter lida com a solicitação de Logout Único procurando a HttpSession em um Map estático e, então, invalidando-a.

Pode ser confuso entender por que tanto o elemento logout quanto o filtro singleLogoutFilter são necessários. É considerado uma boa prática realizar o logout localmente primeiro, já que o SingleSignOutFilter apenas armazena a HttpSession em um Map estático para depois chamá-la e invalidá-la. Com a configuração acima, o fluxo de logout seria:

- O usuário solicita /logout, o que faz o logout da aplicação local e redireciona o usuário para a página de sucesso de logout.
- A página de sucesso de logout, /cas-logout.jsp, deve instruir o usuário a clicar em um link apontando para /logout/cas para fazer o logout de todas as aplicações.
- Quando o usuário clica no link, ele é redirecionado para a URL de logout único do CAS (por exemplo, https://localhost:9443/cas/logout).
- No lado do servidor CAS, a URL de logout único do CAS envia solicitações de logout único para todos os serviços CAS. No lado do serviço CAS, o SingleSignOutFilter do JASIG processa a solicitação de logout invalidando a sessão original.

O próximo passo é adicionar o seguinte ao seu arquivo web.xml.

```
<filter>
<filter-name>characterEncodingFilter</filter-name>
<filter-class>
  org.springframework.web.filter.CharacterEncodingFilter
</filter-class>
<init-param>
  <param-name>encoding</param-name>
  <param-value>UTF-8</param-value>
</init-param>
</filter>
<filter-mapping>
<filter-name>characterEncodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
<listener-class>
  org.jasig.cas.client.session.SingleSignOutHttpSessionListener
</listener-class>
</listener>
```

Ao usar o SingleSignOutFilter, você pode encontrar alguns problemas de codificação. Portanto, é recomendado adicionar o CharacterEncodingFilter para garantir que a codificação de caracteres esteja correta ao usar o SingleSignOutFilter. Novamente, consulte a documentação do JASIG para mais detalhes. O SingleSignOutHttpSessionListener garante que, quando uma HttpSession expira, o mapeamento usado para o logout único seja removido.

Autenticando em um Serviço Stateless com CAS

Esta seção descreve como autenticar em um serviço usando CAS. Em outras palavras, ela discute como configurar um cliente que usa um serviço que se autentica com CAS. A próxima seção descreve como configurar um serviço stateless para autenticar usando CAS.

Configurando o CAS para Obter Proxy Granting Tickets

Para autenticar em um serviço stateless, a aplicação precisa obter um **proxy granting ticket** (PGT). Esta seção descreve como configurar o Spring Security para obter um PGT, construindo a configuração do **Service Ticket Authentication**.

O primeiro passo é incluir um **ProxyGrantingTicketStorage** na configuração do Spring Security. Isso é usado para armazenar os PGTs que são obtidos pelo CasAuthenticationFilter, para que possam ser usados para obter os **proxy tickets**. Abaixo está um exemplo de configuração:

```
<!--
NOTE: In a real application you should not use an in memory implementation.
You will also want to ensure to clean up expired tickets by calling
ProxyGrantingTicketStorage.cleanup()
-->
<bean id="pgtStorage"
class="org.jasig.cas.client.proxy.ProxyGrantingTicketStorageImpl"/>
```

O próximo passo é atualizar o CasAuthenticationProvider para ser capaz de obter **proxy tickets**. Para fazer isso, substitua o Cas20ServiceTicketValidator por um Cas20ProxyTicketValidator. O parâmetro proxyCallbackUrl deve ser configurado com uma URL para a qual a aplicação receberá os **PGTs**. Por fim, a configuração também deve referenciar o ProxyGrantingTicketStorage para que possa usar um **PGT** para obter os **proxy tickets**.
Abaixo está um exemplo das mudanças de configuração que devem ser feitas:

```
<bean id="casAuthenticationProvider"
  class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
  <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
  <constructor-arg value="https://localhost:9443/cas"/>
  <property name="proxyCallbackUrl"
  value="https://localhost:8443/cas-sample/login/cas/proxyreceptor"/>
  <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
  </bean>
</property>
</bean>
```

O último passo é atualizar o CasAuthenticationFilter para aceitar PGTs e armazená-los no ProxyGrantingTicketStorage. É importante que a proxyReceptorUrl corresponda à proxyCallbackUrl do Cas20ProxyTicketValidator. Um exemplo de configuração é mostrado abaixo.

```
<bean id="casFilter"
  class="org.springframework.security.cas.web.CasAuthenticationFilter">
  ...
  <property name="proxyGrantingTicketStorage" ref="pgtStorage"/>
  <property name="proxyReceptorUrl" value="/login/cas/proxyreceptor"/>
</bean>
```

Chamando um Serviço Stateless Usando um Ticket de Proxy
Agora que o Spring Security obtém PGTs, você pode usá-los para criar tickets de proxy, que podem ser usados para autenticar em um serviço sem estado. O aplicativo de exemplo CAS contém um exemplo funcional no ProxyTicketSampleServlet. O código de exemplo pode ser encontrado abaixo:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
// NOTE: The CasAuthenticationToken can also be obtained using
// SecurityContextHolder.getContext().getAuthentication()
final CasAuthenticationToken token = (CasAuthenticationToken)
request.getUserPrincipal();
// proxyTicket could be reused to make calls to the CAS service even if the
// target url differs
final String proxyTicket =
token.getAssertion().getPrincipal().getProxyTicketFor(targetUrl);
// Make a remote call using the proxy ticket
final String serviceUrl = targetUrl+"?ticket="+URLEncoder.encode(proxyTicket, "UTF8");
String proxyResponse = CommonUtils.getResponseFromServer(serviceUrl, "UTF-8");
...
}
```

Autenticação com Ticket de Proxy
O CasAuthenticationProvider distingue entre clientes com estado e clientes sem estado. Um cliente com estado é qualquer um que envie solicitações para a filterProcessUrl do CasAuthenticationFilter. Um cliente sem estado é aquele que apresenta uma solicitação de autenticação ao CasAuthenticationFilter em uma URL diferente da filterProcessUrl.
Como os protocolos de comunicação remota não têm como se apresentar dentro do contexto de uma HttpSession, não é possível confiar na prática padrão de armazenar o contexto de segurança na sessão entre as requisições. Além disso,

como o servidor CAS invalida um ticket após ele ter sido validado pelo TicketValidator, apresentar o mesmo ticket de proxy em requisições subsequentes não funcionará.
Uma opção óbvia seria não usar o CAS para clientes de protocolos remotos. No entanto, isso eliminaria muitos dos recursos desejáveis do CAS. Como uma solução intermediária, o CasAuthenticationProvider utiliza um StatelessTicketCache. Esse cache é usado exclusivamente para clientes sem estado que utilizam um principal igual a CasAuthenticationFilter.CAS_STATELESS_IDENTIFIER. O que acontece é que o CasAuthenticationProvider armazenará o CasAuthenticationToken resultante no StatelessTicketCache, indexado pelo ticket de proxy. Dessa forma, clientes de protocolos remotos podem apresentar o mesmo ticket de proxy e o CasAuthenticationProvider não precisará entrar em contato com o servidor CAS para validação (exceto na primeira requisição). Uma vez autenticado, o ticket de proxy pode ser usado para URLs diferentes do serviço de destino original.
Esta seção se baseia nas anteriores para acomodar a autenticação com ticket de proxy. O primeiro passo é especificar a autenticação de todos os artefatos, conforme mostrado abaixo.

```
<bean id="serviceProperties"
  class="org.springframework.security.cas.ServiceProperties">
...
<property name="authenticateAllArtifacts" value="true"/>
</bean>
```

O próximo passo é especificar serviceProperties e authenticationDetailsSource para o CasAuthenticationFilter.

- A propriedade serviceProperties instrui o CasAuthenticationFilter a tentar autenticar todos os artefatos, em vez de apenas aqueles presentes na filterProcessUrl.
- O ServiceAuthenticationDetailsSource cria um ServiceAuthenticationDetails que garante que a URL atual, baseada no HttpServletRequest, seja usada como a URL de serviço ao validar o ticket.

O método para gerar a URL de serviço pode ser personalizado injetando um AuthenticationDetailsSource personalizado que retorne um ServiceAuthenticationDetails customizado.

```
<bean id="casFilter"
  class="org.springframework.security.cas.web.CasAuthenticationFilter">
...
<property name="serviceProperties" ref="serviceProperties"/>
<property name="authenticationDetailsSource">
  <bean class=
"org.springframework.security.cas.web.authentication.ServiceAuthenticationDetailsSourc
e">
  <constructor-arg ref="serviceProperties"/>
  </bean>
</property>
</bean>
```

Você também precisará atualizar o CasAuthenticationProvider para lidar com tickets de proxy.
Para isso:

- Substitua o Cas20ServiceTicketValidator pelo Cas20ProxyTicketValidator.
- Configure o statelessTicketCache.
- Defina quais proxies deseja aceitar.

Abaixo, você pode encontrar um exemplo das atualizações necessárias para aceitar todos os proxies.

```
<bean id="casAuthenticationProvider"
  class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
...
<property name="ticketValidator">
  <bean class="org.jasig.cas.client.validation.Cas20ProxyTicketValidator">
  <constructor-arg value="https://localhost:9443/cas"/>
  <property name="acceptAnyProxy" value="true"/>
  </bean>
</property>
<property name="statelessTicketCache">
  <bean
class="org.springframework.security.cas.authentication.EhCacheBasedTicketCache">
  <property name="cache">
```

```
  <bean class="net.sf.ehcache.Cache"
  init-method="initialise" destroy-method="dispose">
  <constructor-arg value="casTickets"/>
  <constructor-arg value="50"/>
  <constructor-arg value="true"/>
  <constructor-arg value="false"/>
  <constructor-arg value="3600"/>
  <constructor-arg value="900"/>
  </bean>
  </property>
  </bean>
</property>
</bean>
```

10.18. Autenticação X.509

10.18.1. Visão Geral

O uso mais comum da autenticação por certificado X.509 é a verificação da identidade de um servidor ao utilizar SSL, especialmente em conexões HTTPS através de um navegador. O navegador verifica automaticamente se o certificado apresentado pelo servidor foi emitido (ou seja, assinado digitalmente) por uma das autoridades certificadoras confiáveis que ele mantém.

Também é possível usar SSL com "autenticação mútua", onde o servidor solicita um certificado válido do cliente como parte do handshake SSL. O servidor autentica o cliente verificando se o certificado foi assinado por uma autoridade aceitável. Se um certificado válido for fornecido, ele pode ser obtido através da API do servlet em uma aplicação.

O módulo X.509 do Spring Security extrai o certificado usando um filtro, mapeia-o para um usuário da aplicação e carrega as permissões concedidas a esse usuário, para uso com a infraestrutura padrão do Spring Security.

Antes de utilizar essa funcionalidade com o Spring Security, é importante estar familiarizado com o uso de certificados e a configuração de autenticação do cliente no seu container de servlets. A maior parte do trabalho envolve a criação e instalação de certificados e chaves apropriados.

Por exemplo, se estiver usando o Tomcat, consulte as instruções em:

📎 Configuração SSL no Tomcat

É essencial garantir que essa configuração funcione antes de integrá-la ao Spring Security.

10.18.2. Adicionando Autenticação X.509 à Sua Aplicação Web

Habilitar a autenticação X.509 para clientes é muito simples. Basta adicionar o elemento `<x509/>` à configuração de segurança HTTP no namespace do Spring Security.

```
<http>
...
  <x509 subject-principal-regex="CN=(.*?)," user-service-ref="userService"/>;
</http>
```

O elemento possui dois atributos opcionais:

- **subject-principal-regex**: Expressão regular usada para extrair o nome de usuário do certificado. O valor extraído será passado para o UserDetailsService para carregar as permissões do usuário.
- **user-service-ref**: ID do *bean* do UserDetailsService usado com X.509, dispensável caso haja apenas um definido no contexto da aplicação.

A expressão `"CN=(.*?),"` captura o *Common Name* do certificado, como `"CN=Jimi Hendrix, OU=..."`, resultando no nome de usuário `"Jimi Hendrix"`. Também pode ser usada para capturar e-mails, como `"emailAddress=(.*?),"` extraindo `"jimi@hendrix.org"`.

Se um certificado válido for apresentado e um nome de usuário for extraído, a autenticação será bem-sucedida. Caso contrário, o contexto de segurança permanecerá vazio, permitindo combinar a autenticação X.509 com outras, como login por formulário.

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https"
secure="true"
  clientAuth="true" sslProtocol="TLS"
  keystoreFile="${catalina.home}/conf/server.jks"
  keystoreType="JKS" keystorePass="password"
  truststoreFile="${catalina.home}/conf/server.jks"
```

```
  truststoreType="JKS" truststorePass="password"
/>
```

A opção clientAuth pode ser configurada como want para permitir conexões SSL mesmo sem um certificado do cliente. No entanto, sem um certificado, o acesso a objetos protegidos exigirá outro método de autenticação, como login por formulário.

Substituição de Autenticação Run-As

O AbstractSecurityInterceptor pode substituir temporariamente o objeto Authentication no SecurityContext durante a execução de um objeto seguro, desde que a autenticação original tenha sido validada pelo AuthenticationManager e AccessDecisionManager. O RunAsManager define qual objeto de autenticação substituirá o original.

Isso permite que chamadas seguras utilizem diferentes credenciais de autenticação e autorização, facilitando a comunicação com serviços remotos. O Spring Security fornece classes auxiliares para configurar automaticamente protocolos remotos com base no SecurityContextHolder, tornando essa funcionalidade útil para chamadas a serviços web.

```
Authentication buildRunAs(Authentication authentication, Object object,
  List<ConfigAttribute> config);
boolean supports(ConfigAttribute attribute);
boolean supports(Class clazz);
```

O primeiro método retorna um objeto Authentication que substituirá a autenticação atual durante a execução do método. Se retornar null, nenhuma substituição será feita. O segundo método é utilizado pelo AbstractSecurityInterceptor para validar atributos de configuração, garantindo que o RunAsManager suporte o tipo de objeto seguro apresentado pelo interceptor.

O Spring Security fornece uma implementação concreta chamada RunAsManagerImpl. Ela retorna um RunAsUserToken sempre que um ConfigAttribute começa com RUN_AS_. Esse RunAsUserToken mantém os mesmos dados do objeto original (principal, credentials e granted authorities), mas adiciona um novo SimpleGrantedAuthority para cada RUN_AS_ detectado. Por exemplo, RUN_AS_SERVER adiciona ROLE_RUN_AS_SERVER como uma autoridade concedida.

O RunAsUserToken funciona como qualquer outro objeto de autenticação e precisa ser validado por um AuthenticationManager, geralmente via um AuthenticationProvider adequado. O RunAsImplAuthenticationProvider realiza essa autenticação, aceitando qualquer RunAsUserToken recebido.

Para evitar que código malicioso crie e utilize um RunAsUserToken indevidamente, um hash de uma chave é armazenado em todos os tokens gerados. Tanto o RunAsManagerImpl quanto o RunAsImplAuthenticationProvider são configurados no contexto de beans com essa mesma chave.

```
<bean id="runAsManager"
  class="org.springframework.security.access.intercept.RunAsManagerImpl">
<property name="key" value="my_run_as_password"/>
</bean>
<bean id="runAsAuthenticationProvider"
class="org.springframework.security.access.intercept.RunAsImplAuthenticationProvider">
<property name="key" value="my_run_as_password"/>
</bean>
```

Ao usar a mesma chave, cada RunAsUserToken pode ser validado para garantir que foi criado por um RunAsManagerImpl aprovado. O RunAsUserToken é imutável após sua criação por questões de segurança.

10.20. Tratamento de Logouts

10.20.1. Configuração de Logout em Java

Ao utilizar o WebSecurityConfigurerAdapter, as funcionalidades de logout são aplicadas automaticamente. O padrão é que o acesso à URL /logout realiza o logout do usuário, executando as seguintes ações:

- Invalidar a sessão HTTP.
- Limpar qualquer autenticação de RememberMe configurada.
- Limpar o SecurityContextHolder.
- Redirecionar para /login?logout.

Semelhante à configuração de login, você também tem várias opções para personalizar seus requisitos de logout.

```
protected void configure(HttpSecurity http) throws Exception {
  http
  .logout(logout -> logout ①
  .logoutUrl("/my/logout") ②
  .logoutSuccessUrl("/my/index") ③
  .logoutSuccessHandler(logoutSuccessHandler) ④
  .invalidateHttpSession(true) ⑤
  .addLogoutHandler(logoutHandler) ⑥
  .deleteCookies(cookieNamesToClear) ⑦
  )
  ...
}
```

① Oferece suporte para logout. Isso é aplicado automaticamente ao usar o WebSecurityConfigurerAdapter.
② A URL que aciona o logout (o padrão é /logout). Se a proteção CSRF estiver ativada (padrão), a solicitação também deve ser um POST. Consulte a JavaDoc para mais informações.
③ A URL para redirecionar após o logout (o padrão é /login?logout). Consulte a JavaDoc para mais informações.
④ Permite especificar um LogoutSuccessHandler personalizado. Se especificado, a configuração logoutSuccessUrl() será ignorada. Consulte a JavaDoc para mais informações.
⑤ Especifica se deve invalidar a HttpSession no momento do logout. O padrão é verdadeiro. Configura o SecurityContextLogoutHandler nos bastidores. Consulte a JavaDoc para mais informações.
⑥ Adiciona um LogoutHandler. O SecurityContextLogoutHandler é adicionado como o último LogoutHandler por padrão.
⑦ Permite especificar os nomes dos cookies a serem removidos no sucesso do logout. Este é um atalho para adicionar explicitamente um CookieClearingLogoutHandler.

Logouts também podem ser configurados usando a notação de Namespace XML. Consulte a documentação do elemento logout na seção XML Namespace do Spring Security para mais detalhes.

Geralmente, para personalizar a funcionalidade de logout, você pode adicionar implementações de LogoutHandler e/ou LogoutSuccessHandler. Para muitos cenários comuns, esses manipuladores são aplicados automaticamente ao usar a API fluente.

10.20.2. Configuração de Logout em XML

O elemento logout oferece suporte para realizar logout ao acessar uma URL específica. A URL de logout padrão é /logout, mas você pode definir outra usando o atributo logout-url. Mais informações sobre outros atributos disponíveis podem ser encontradas no apêndice de namespace.

10.20.3. LogoutHandler

Normalmente, as implementações de LogoutHandler são classes que podem participar do tratamento de logout. Elas devem ser invocadas para realizar a limpeza necessária, e não devem lançar exceções. Algumas implementações fornecidas incluem:

- PersistentTokenBasedRememberMeServices
- TokenBasedRememberMeServices
- CookieClearingLogoutHandler
- CsrfLogoutHandler
- SecurityContextLogoutHandler
- HeaderWriterLogoutHandler

A API fluente também oferece atalhos que fornecem as respectivas implementações de LogoutHandler nos bastidores. Por exemplo, deleteCookies() permite especificar os nomes de um ou mais cookies a serem removidos após o logout, em vez de adicionar um CookieClearingLogoutHandler.

10.20.4. LogoutSuccessHandler

O LogoutSuccessHandler é chamado após um logout bem-sucedido pelo LogoutFilter, para tratar, por exemplo, redirecionamentos ou encaminhamentos para o destino apropriado. A interface é quase idêntica ao LogoutHandler, mas pode gerar exceções. Algumas implementações fornecidas incluem:

- SimpleUrlLogoutSuccessHandler
- HttpStatusReturningLogoutSuccessHandler

Como mencionado, você não precisa especificar diretamente o SimpleUrlLogoutSuccessHandler. A API fluente oferece um atalho ao configurar o logoutSuccessUrl(), que configura automaticamente o SimpleUrlLogoutSuccessHandler. A URL fornecida será redirecionada após o logout. O padrão é /login?logout.
O HttpStatusReturningLogoutSuccessHandler pode ser útil em cenários de API REST. Em vez de redirecionar para uma URL após o logout bem-sucedido, ele permite retornar um código de status HTTP simples. Se não configurado, o código de status 200 será retornado por padrão.

10.20.5. Referências Adicionais sobre Logout

- Manipulação de Logout
- Testando Logout
- HttpServletRequest.logout()
- Interfaces e Implementações de Remember-Me
- Logout na seção CSRF Caveats
- Seção Single Logout (protocolo CAS)
- Documentação para o elemento logout na seção XML Namespace do Spring Security

[1] Também é possível obter o endereço IP do servidor usando uma consulta DNS. Isso não é atualmente suportado, mas espera-se que seja na versão futura.
[2] A autenticação por mecanismos que realizam um redirecionamento após a autenticação (como o form-login) não será detectada pelo SessionManagementFilter, pois o filtro não será invocado durante a solicitação de autenticação. A funcionalidade de gerenciamento de sessão deve ser tratada separadamente nesses casos.
[3] Essencialmente, o nome de usuário não é incluído no cookie, para evitar expor um nome de login válido desnecessariamente. Há uma discussão sobre isso na seção de comentários deste artigo.
[4] O uso da propriedade key não deve ser considerado como fornecendo segurança real. É apenas um exercício de controle. Se você estiver compartilhando um ProviderManager que contém um AnonymousAuthenticationProvider em um cenário onde o cliente autenticador pode construir o objeto Authentication (como em invocações RMI), um cliente malicioso poderia enviar um AnonymousAuthenticationToken que ele mesmo criou (com nome de usuário e lista de autoridade escolhidos). Se a chave for adivinhável ou puder ser descoberta, o token seria aceito pelo provedor anônimo. Isso não é um problema no uso normal, mas se você estiver usando RMI, é melhor usar um ProviderManager personalizado que omita o provedor anônimo, em vez de compartilhar o usado para seus mecanismos de autenticação HTTP.

Capítulo 11. Autorização

As capacidades avançadas de autorização no Spring Security representam um dos motivos mais atraentes para sua popularidade. Independentemente de como você escolhe autenticar - seja usando um mecanismo e provedor fornecido pelo Spring Security ou integrando com um contêiner ou outra autoridade de autenticação não-Spring Security - você encontrará que os serviços de autorização podem ser usados de forma consistente e simples em sua aplicação.
Nesta parte, exploraremos as diferentes implementações do AbstractSecurityInterceptor, que foram introduzidas na Parte I. Em seguida, abordaremos como afinar a autorização por meio do uso de listas de controle de acesso baseadas em domínio.

11.1. Arquitetura de Autorização

11.1.1. Autoridades

A autenticação discute como todas as implementações de Authentication armazenam uma lista de objetos GrantedAuthority. Estes representam as autoridades que foram concedidas ao principal. Os objetos GrantedAuthority são inseridos no objeto Authentication pelo AuthenticationManager e são lidos posteriormente pelos AccessDecisionManager ao tomar decisões de autorização.
GrantedAuthority é uma interface com apenas um método:

```
String getAuthority();
```

Este método permite que os AccessDecisionManager obtenham uma representação precisa em String da GrantedAuthority. Ao retornar uma representação como String, uma GrantedAuthority pode ser facilmente "lida" pela maioria dos AccessDecisionManager. Se uma GrantedAuthority não puder ser representada precisamente como uma String, a GrantedAuthority é considerada "complexa" e o método getAuthority() deve retornar null.
Um exemplo de uma GrantedAuthority "complexa" seria uma implementação que armazena uma lista de operações e limites de autoridade que se aplicam a diferentes números de contas de clientes. Representar essa GrantedAuthority complexa como uma String seria bastante difícil, e como resultado, o método getAuthority() deve retornar null. Isso indicará a qualquer AccessDecisionManager que ele precisará oferecer suporte específico para a implementação de GrantedAuthority a fim de entender seu conteúdo.

O Spring Security inclui uma implementação concreta de GrantedAuthority, a SimpleGrantedAuthority. Isso permite que qualquer String especificada pelo usuário seja convertida em uma GrantedAuthority. Todos os AuthenticationProvider incluídos na arquitetura de segurança usam SimpleGrantedAuthority para preencher o objeto Authentication.

11.1.2. Manipulação Pré-Invocação

O Spring Security fornece interceptadores que controlam o acesso a objetos seguros, como invocações de métodos ou requisições web. Uma decisão pré-invocação sobre se a invocação pode continuar é tomada pelo AccessDecisionManager.

O AccessDecisionManager

O AccessDecisionManager é chamado pelo AbstractSecurityInterceptor e é responsável por tomar as decisões finais de controle de acesso. A interface AccessDecisionManager contém três métodos:

```
void decide(Authentication authentication, Object secureObject,
  Collection<ConfigAttribute> attrs) throws AccessDeniedException;
boolean supports(ConfigAttribute attribute);
boolean supports(Class clazz);
```

O método **decide** do AccessDecisionManager recebe todas as informações relevantes necessárias para tomar uma decisão de autorização. Em particular, ao passar o objeto seguro, permite que os argumentos contidos na invocação real do objeto seguro sejam inspecionados. Por exemplo, suponha que o objeto seguro seja uma invocação de método (MethodInvocation). Seria fácil consultar a MethodInvocation para obter qualquer argumento de Customer, e então implementar alguma lógica de segurança no AccessDecisionManager para garantir que o principal tem permissão para operar sobre esse cliente. As implementações são esperadas para lançar uma AccessDeniedException caso o acesso seja negado.

O método **supports(ConfigAttribute)** é chamado pelo AbstractSecurityInterceptor no momento de inicialização para determinar se o AccessDecisionManager pode processar o ConfigAttribute passado. O método **supports(Class)** é chamado por uma implementação do interceptor de segurança para garantir que o AccessDecisionManager configurado suporta o tipo de objeto seguro que o interceptor de segurança irá apresentar.

Implementações de AccessDecisionManager Baseadas em Votação

Embora os usuários possam implementar seu próprio AccessDecisionManager para controlar todos os aspectos da autorização, o Spring Security inclui várias implementações de AccessDecisionManager baseadas em votação. O **Voting Decision Manager** ilustra as classes relevantes.

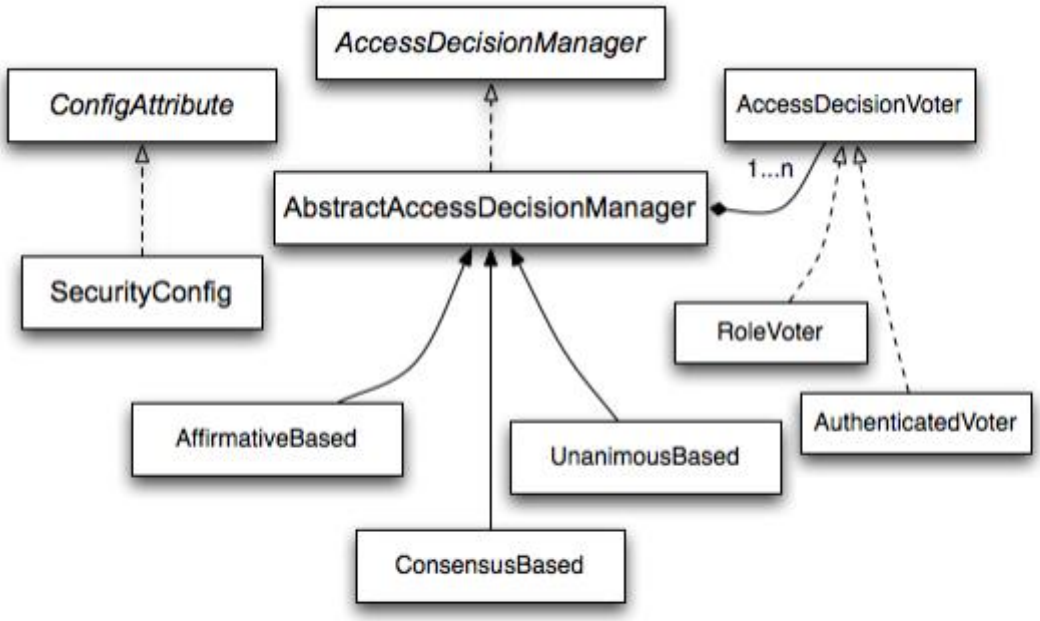


Figure 11. Voting Decision Manager

Usando essa abordagem, uma série de implementações de **AccessDecisionVoter** são consultadas para uma decisão de autorização. O **AccessDecisionManager** então decide se deve ou não lançar uma **AccessDeniedException** com base na avaliação dos votos.

A interface **AccessDecisionVoter** possui três métodos:

```
int vote(Authentication authentication, Object object, Collection<ConfigAttribute>
attrs);
boolean supports(ConfigAttribute attribute);
```

boolean supports(Class clazz);

As implementações concretas retornam um valor **int**, com valores possíveis refletidos nos campos estáticos do **AccessDecisionVoter**: **ACCESS_ABSTAIN**, **ACCESS_DENIED** e **ACCESS_GRANTED**. Uma implementação de voto retornará **ACCESS_ABSTAIN** se não tiver uma opinião sobre a decisão de autorização. Se tiver uma opinião, deverá retornar **ACCESS_DENIED** ou **ACCESS_GRANTED**.
Existem três implementações concretas de **AccessDecisionManager** fornecidas pelo Spring Security que contabilizam os votos:

- **ConsensusBased**: concede ou nega o acesso com base no consenso dos votos não-abstidos. Há propriedades para controlar o comportamento no caso de empate de votos ou quando todos os votos forem de abstinência.
- **AffirmativeBased**: concede acesso se um ou mais votos forem **ACCESS_GRANTED** (ou seja, um voto de negação será ignorado, desde que haja pelo menos um voto de concessão). Também há um parâmetro que controla o comportamento quando todos os eleitores se abstêm.
- **UnanimousBased**: concede acesso apenas se todos os votos forem **ACCESS_GRANTED**, ignorando as abstinências. Negará o acesso se houver qualquer voto de **ACCESS_DENIED**. Como as outras implementações, há um parâmetro para controlar o comportamento caso todos os eleitores se abstenham.

É possível implementar um **AccessDecisionManager** personalizado que contabilize os votos de maneira diferente. Por exemplo, votos de um determinado **AccessDecisionVoter** podem receber peso adicional, enquanto um voto de negação de um eleitor específico pode ter um efeito de veto.

RoleVoter

O **AccessDecisionVoter** mais comumente usado fornecido pelo Spring Security é o simples **RoleVoter**, que trata os atributos de configuração como nomes de funções e vota para conceder acesso se o usuário tiver sido atribuído a essa função.

- Ele votará se qualquer **ConfigAttribute** começar com o prefixo **ROLE_**. Votará para conceder acesso se houver uma **GrantedAuthority** que retorne uma representação **String** (via o método **getAuthority()**) exatamente igual a um ou mais **ConfigAttributes** começando com o prefixo **ROLE_**.
- Se não houver correspondência exata de qualquer **ConfigAttribute** começando com **ROLE_**, o **RoleVoter** votará para negar o acesso.
- Se nenhum **ConfigAttribute** começar com **ROLE_**, o eleitor se abstém.

AuthenticatedVoter

Outro eleitor que vimos implicitamente é o **AuthenticatedVoter**, que pode ser usado para diferenciar entre usuários anônimos, totalmente autenticados e usuários autenticados via **remember-me**.

- Muitos sites permitem acesso limitado com a autenticação **remember-me**, mas exigem que o usuário confirme sua identidade ao fazer login para acesso completo.
- Quando usamos o atributo **IS_AUTHENTICATED_ANONYMOUSLY** para conceder acesso anônimo, esse atributo estava sendo processado pelo **AuthenticatedVoter**.

Custom Voters

Você também pode implementar um **AccessDecisionVoter** personalizado e colocar qualquer lógica de controle de acesso que desejar nele. Isso pode ser específico para sua aplicação (relacionado à lógica de negócios) ou pode implementar alguma lógica de administração de segurança. Por exemplo, você pode encontrar um artigo no blog do Spring descrevendo como usar um eleitor para negar acesso em tempo real a usuários cujas contas foram suspensas.

After Invocation Handling

Enquanto o **AccessDecisionManager** é chamado pelo **AbstractSecurityInterceptor** antes de prosseguir com a invocação do objeto seguro, algumas aplicações precisam de uma maneira de modificar o objeto realmente retornado pela invocação do objeto seguro. Embora você possa facilmente implementar sua própria preocupação AOP para alcançar isso, o Spring Security fornece um ponto de integração conveniente com várias implementações concretas que se integram com suas capacidades de **ACL** (Access Control List).

A implementação **After Invocation** ilustra o **AfterInvocationManager** do Spring Security e suas implementações concretas.

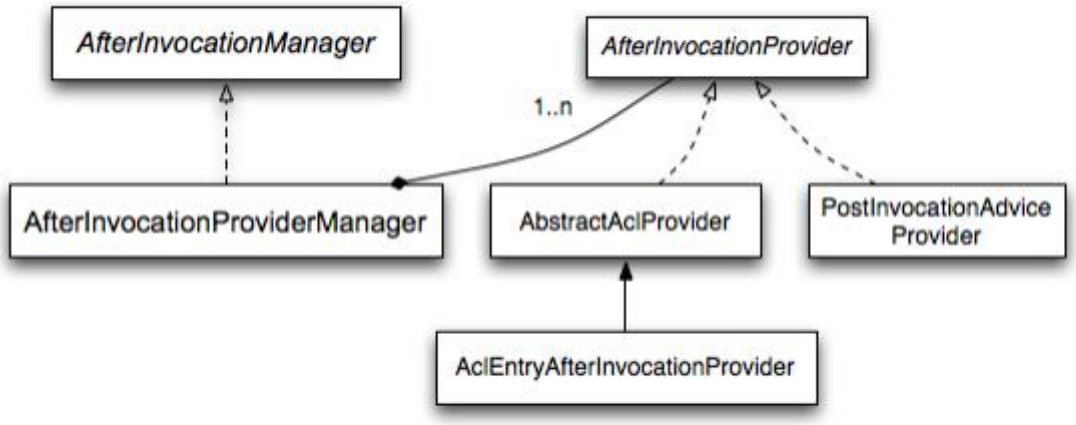


Figure 12. After Invocation Implementation

Como muitas outras partes do Spring Security, o **AfterInvocationManager** possui uma única implementação concreta, **AfterInvocationProviderManager**, que faz a sondagem de uma lista de **AfterInvocationProvider**. Cada **AfterInvocationProvider** tem permissão para modificar o objeto retornado ou lançar uma **AccessDeniedException**. Na verdade, vários provedores podem modificar o objeto, pois o resultado do provedor anterior é passado para o próximo na lista.

Esteja ciente de que, se você estiver usando o **AfterInvocationManager**, ainda precisará de atributos de configuração que permitam que o **AccessDecisionManager** do **MethodSecurityInterceptor** autorize uma operação. Se você estiver usando as implementações típicas do **AccessDecisionManager** incluídas no Spring Security, não ter atributos de configuração definidos para uma invocação de método segura específica fará com que cada **AccessDecisionVoter** se abstenha de votar. Como resultado, se a propriedade **"allowIfAllAbstainDecisions"** do **AccessDecisionManager** estiver configurada como **false**, será lançada uma **AccessDeniedException**. Você pode evitar esse possível problema de duas maneiras: (i) definindo **"allowIfAllAbstainDecisions"** como **true** (embora isso geralmente não seja recomendado) ou (ii) garantindo que haja pelo menos um atributo de configuração para o qual um **AccessDecisionVoter** vote para conceder acesso. Esta última abordagem (recomendada) geralmente é alcançada por meio de um atributo de configuração **ROLE_USER** ou **ROLE_AUTHENTICATED**.

11.1.4. Papéis Hierárquicos

É uma exigência comum que um papel específico em uma aplicação inclua automaticamente outros papéis. Por exemplo, em uma aplicação que tem os conceitos de papéis **"admin"** e **"user"**, você pode querer que um **admin** tenha permissão para fazer tudo o que um **user** pode. Para isso, você pode garantir que todos os usuários administradores também sejam atribuídos ao papel **user**. Alternativamente, você pode modificar cada restrição de acesso que exija o papel **user** para também incluir o papel **admin**. Isso pode ficar bastante complicado se você tiver muitos papéis diferentes na sua aplicação.

O uso de uma **hierarquia de papéis** permite configurar quais papéis (ou autoridades) devem incluir outros. Uma versão estendida do **RoleVoter** do Spring Security, o **RoleHierarchyVoter**, é configurada com um **RoleHierarchy**, de onde ele obtém todas as "autoridades alcançáveis" que o usuário possui. Uma configuração típica pode ser assim:

```
<bean id="roleVoter"
class="org.springframework.security.access.vote.RoleHierarchyVoter">
  <constructor-arg ref="roleHierarchy" />
</bean>
<bean id="roleHierarchy"
class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">
  <property name="hierarchy">
  <value>
  ROLE_ADMIN > ROLE_STAFF
```

```
  ROLE_STAFF > ROLE_USER
  ROLE_USER > ROLE_GUEST
  </value>
  </property>
</bean>
```

Aqui temos quatro papéis em uma hierarquia: **ROLE_ADMIN ⇒ ROLE_STAFF ⇒ ROLE_USER ⇒ ROLE_GUEST**. Um usuário autenticado com o **ROLE_ADMIN** se comportará como se tivesse os quatro papéis quando as restrições de segurança forem avaliadas por um **AccessDecisionManager** configurado com o **RoleHierarchyVoter** acima. O símbolo > pode ser entendido como "inclui".
As hierarquias de papéis oferecem uma maneira conveniente de simplificar os dados de configuração de controle de acesso para sua aplicação e/ou reduzir o número de autoridades que você precisa atribuir a um usuário. Para requisitos mais complexos, você pode querer definir um mapeamento lógico entre os direitos de acesso específicos exigidos pela sua aplicação e os papéis atribuídos aos usuários, realizando a tradução entre ambos ao carregar as informações do usuário.

11.2. Autorizar HttpServletRequest com o FilterSecurityInterceptor

Esta seção aprofunda a Arquitetura e Implementação do Servlet, explorando como a autorização funciona em aplicações baseadas em Servlet. O **FilterSecurityInterceptor** fornece autorização para **HttpServletRequest**. Ele é inserido no **FilterChainProxy** como um dos Filtros de Segurança.

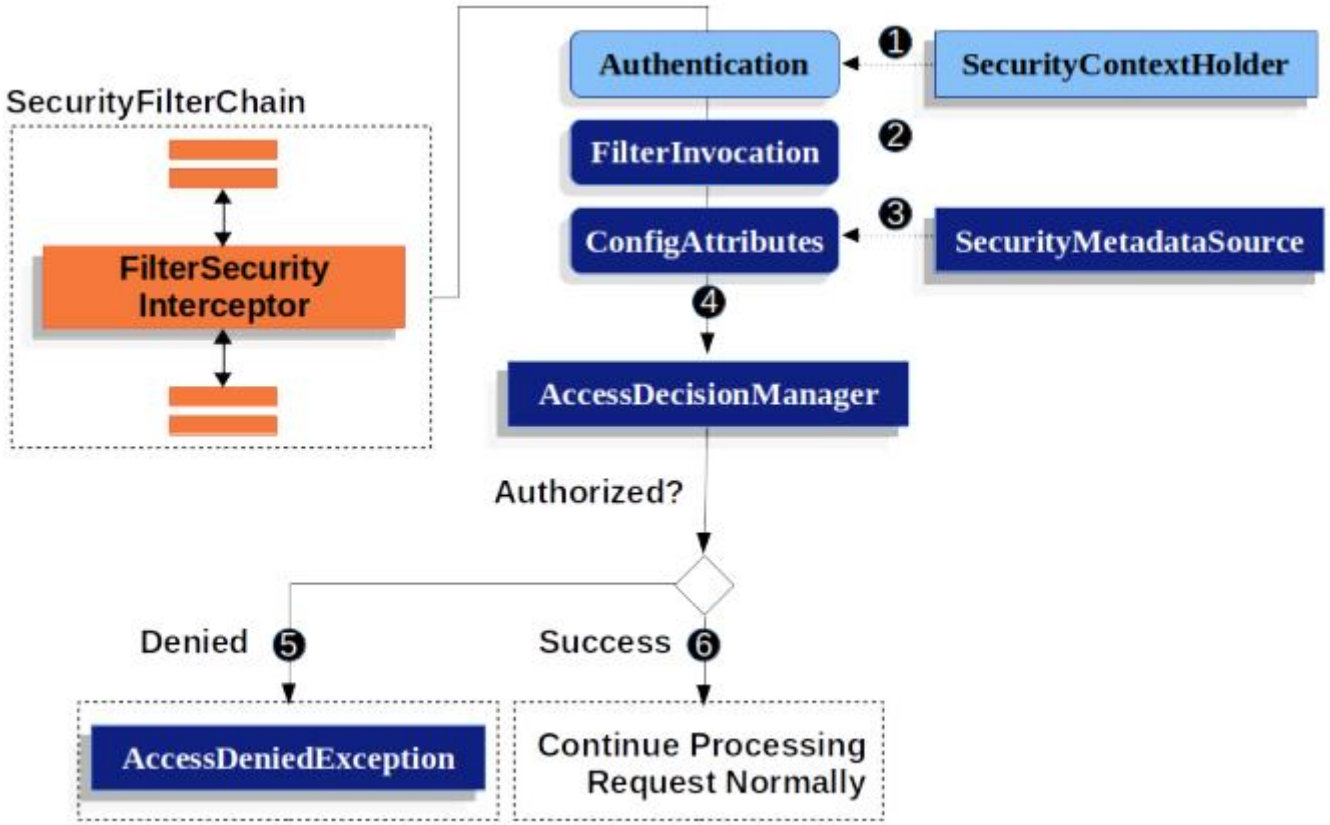


Figure 13. Authorize HttpServletRequest

• Primeiro, o **FilterSecurityInterceptor** obtém uma **Authentication** do **SecurityContextHolder**.
• Em segundo lugar, o **FilterSecurityInterceptor** cria uma **FilterInvocation** a partir do **HttpServletRequest**, **HttpServletResponse** e **FilterChain** que são passados para o **FilterSecurityInterceptor**.
• Em seguida, ele passa a **FilterInvocation** para o **SecurityMetadataSource** para obter os **ConfigAttributes**.
• Por fim, ele passa a **Authentication**, **FilterInvocation** e **ConfigAttributes** para o **AccessDecisionManager**.
◦ Se a autorização for negada, uma **AccessDeniedException** é lançada. Neste caso, o **ExceptionTranslationFilter** trata a **AccessDeniedException**.
◦ Se o acesso for concedido, o **FilterSecurityInterceptor** continua com o **FilterChain**, o que permite que a aplicação prossiga normalmente.
Por padrão, a autorização do Spring Security exigirá que todas as requisições sejam autenticadas. A configuração explícita seria algo assim:

Example 78. Every Request Must be Authenticated

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .authorizeRequests(authorize -> authorize
            .anyRequest().authenticated()
        );
}
```

XML

```
<http>
    <!-- ... -->
    <intercept-url pattern="/**" access="authenticated"/>
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
    http {
        // ...
        authorizeRequests {
            authorize(anyRequest, authenticated)
        }
    }
}
```

Podemos configurar o Spring Security para ter diferentes regras, adicionando mais regras em ordem de precedência.

Example 79. Authorize Requests

Java

```
protected void configure(HttpSecurity http) throws Exception {
    http
        // ...
        .authorizeRequests(authorize -> authorize
①
            .mvcMatchers("/resources/**", "/signup", "/about").permitAll()
②
            .mvcMatchers("/admin/**").hasRole("ADMIN")
③
            .mvcMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")
④
            .anyRequest().denyAll()
⑤
        );
}
```

XML

```
<http> ①
    <!-- ... -->
    ②
    <intercept-url pattern="/resources/**" access="permitAll"/>
    <intercept-url pattern="/signup" access="permitAll"/>
    <intercept-url pattern="/about" access="permitAll"/>

    <intercept-url pattern="/admin/**" access="hasRole('ADMIN')"/> ③
    <intercept-url pattern="/db/**" access="hasRole('ADMIN') and hasRole('DBA')"/>
④
    <intercept-url pattern="/**" access="denyAll"/> ⑤
</http>
```

Kotlin

```
fun configure(http: HttpSecurity) {
   http {
        authorizeRequests { ①
            authorize("/resources/**", permitAll) ②
            authorize("/signup", permitAll)
            authorize("/about", permitAll)

            authorize("/admin/**", hasRole("ADMIN")) ③
            authorize("/db/**", "hasRole('ADMIN') and hasRole('DBA')") ④
            authorize(anyRequest, denyAll) ⑤
        }
    }
}
```

① Existem várias regras de autorização especificadas. Cada regra é considerada na ordem em que foi declarada.
② Especificamos múltiplos padrões de URL que qualquer usuário pode acessar. Especificamente, qualquer usuário pode acessar uma solicitação se a URL começar com "/resources/", for igual a "/signup" ou for igual a "/about".
③ Qualquer URL que começar com "/admin/" será restrita a usuários que tenham o papel "ROLE_ADMIN". Você notará que, como estamos invocando o método hasRole, não precisamos especificar o prefixo "ROLE_".
④ Qualquer URL que começar com "/db/" exige que o usuário tenha tanto "ROLE_ADMIN" quanto "ROLE_DBA". Você notará que, como estamos usando a expressão hasRole, não precisamos especificar o prefixo "ROLE_".
⑤ Qualquer URL que não tenha sido previamente correspondida é negada para acesso. Esta é uma boa estratégia se você não quiser esquecer acidentalmente de atualizar suas regras de autorização.

11.3. Controle de Acesso Baseado em Expressões

O Spring Security 3.0 introduziu a capacidade de usar expressões Spring EL como mecanismo de autorização, além do uso simples de atributos de configuração e eleitores de decisão de acesso que vimos antes. O controle de acesso

baseado em expressões é construído sobre a mesma arquitetura, mas permite que a lógica booleana complicada seja encapsulada em uma única expressão.

11.3.1. Visão Geral

O Spring Security usa Spring EL para suporte a expressões e você deve examinar como isso funciona se estiver interessado em entender o tópico com mais profundidade. As expressões são avaliadas com um "objeto raiz" como parte do contexto de avaliação. O Spring Security usa classes específicas para segurança da web e de métodos como o objeto raiz, a fim de fornecer expressões incorporadas e acesso a valores, como o principal atual.

Expressões Incorporadas Comuns

A classe base para objetos raiz de expressões é SecurityExpressionRoot. Esta fornece algumas expressões comuns que estão disponíveis tanto para segurança da web quanto para segurança de métodos.

Tabela 1. Expressões incorporadas comuns

Expressão	Descrição
hasRole(String role)	Retorna verdadeiro se o principal atual tiver o papel especificado. Por exemplo, `hasRole('admin')`. Por padrão, se o papel fornecido não começar com 'ROLE_', ele será adicionado. Isso pode ser personalizado modificando o `defaultRolePrefix` no `DefaultWebSecurityExpressionHandler`.
hasAnyRole(String… roles)	Retorna verdadeiro se o principal atual tiver qualquer um dos papéis fornecidos (dado como uma lista de strings separadas por vírgula). Por exemplo, `hasAnyRole('admin', 'user')`. Por padrão, se o papel fornecido não começar com 'ROLE_', ele será adicionado. Isso pode ser personalizado modificando o `defaultRolePrefix` no `DefaultWebSecurityExpressionHandler`.
hasAuthority(String authority)	Retorna verdadeiro se o principal atual tiver a autoridade especificada. Por exemplo, `hasAuthority('read')`.
hasAnyAuthority(String… authorities)	Retorna verdadeiro se o principal atual tiver qualquer uma das autoridades fornecidas (dado como uma lista de strings separadas por vírgula). Por exemplo, `hasAnyAuthority('read', 'write')`.
principal	Permite acesso direto ao objeto principal representando o usuário atual.
authentication	Permite acesso direto ao objeto `Authentication` atual obtido do `SecurityContext`.
permitAll	Sempre avalia como verdadeiro.
denyAll	Sempre avalia como falso.
isAnonymous()	Retorna verdadeiro se o principal atual for um usuário anônimo.
isRememberMe()	Retorna verdadeiro se o principal atual for um usuário "remember-me".
isAuthenticated()	Retorna verdadeiro se o usuário não for anônimo.
isFullyAuthenticated()	Retorna verdadeiro se o usuário não for anônimo nem um usuário "remember-me".
hasPermission(Object target, Object permission)	Retorna verdadeiro se o usuário tiver acesso ao alvo fornecido para a permissão dada. Por exemplo, `hasPermission(domainObject, 'read')`.
hasPermission(Object targetId, String targetType, Object permission)	Retorna verdadeiro se o usuário tiver acesso ao alvo fornecido para a permissão dada. Por exemplo, `hasPermission(1, 'com.example.domain.Message', 'read')`.

11.3.2. Expressões de Segurança para a Web

Para usar expressões para garantir a segurança de URLs individuais, você precisará primeiro definir o atributo use-expressions no elemento <http> como verdadeiro. O Spring Security então esperará que os atributos de acesso dos elementos <intercept-url> contenham expressões Spring EL. As expressões devem avaliar para um valor Booleano, definindo se o acesso deve ser permitido ou não. Por exemplo:

```
<http>
  <intercept-url pattern="/admin*"
  access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>
  ...
</http>
```

Aqui, definimos que a área "admin" de uma aplicação (definida pelo padrão de URL) deve estar disponível apenas para usuários que possuam a autoridade concedida "admin" e cujo endereço IP corresponda a uma sub-rede local. Já vimos a expressão incorporada hasRole na seção anterior. A expressão hasIpAddress é uma expressão adicional incorporada, específica para segurança da web. Ela é definida pela classe WebSecurityExpressionRoot, cuja instância

é usada como objeto raiz da expressão ao avaliar expressões de acesso à web. Este objeto também expõe diretamente o objeto HttpServletRequest sob o nome request, permitindo que você invoque o request diretamente em uma expressão. Se expressões estiverem sendo usadas, um WebExpressionVoter será adicionado ao AccessDecisionManager, que é utilizado pelo namespace. Portanto, se você não estiver usando o namespace e quiser usar expressões, será necessário adicionar um desses à sua configuração.

Referindo-se aos Beans nas Expressões de Segurança da Web

Se você desejar estender as expressões disponíveis, pode facilmente referenciar qualquer Bean do Spring que você expuser. Por exemplo, supondo que você tenha um Bean com o nome webSecurity que contenha a seguinte assinatura de método...

```
public class WebSecurity {
  public boolean check(Authentication authentication, HttpServletRequest
request) {
  ...
  }
}
```

Você poderia se referir ao método usando:

```
<http>
  <intercept-url pattern="/user/**"
  access="@webSecurity.check(authentication,request)"/>
  ...
</http>
```

ou na configuração em Java.

```
http
  .authorizeRequests(authorize -> authorize
  .antMatchers("/user/**").access("@webSecurity.check(authentication,request)")
  ...
  )
```

Variáveis de Caminho nas Expressões de Segurança Web

Às vezes, é interessante poder se referir às variáveis de caminho dentro de uma URL. Por exemplo, considere uma aplicação RESTful que busca um usuário pelo id a partir do caminho da URL no formato /user/{userId}.

Você pode facilmente se referir à variável de caminho colocando-a no padrão. Por exemplo, se você tiver um Bean com o nome de webSecurity que contém a seguinte assinatura de método:

```
public class WebSecurity {
  public boolean checkUserId(Authentication authentication, int id) {
  ...
  }
}
```

Você poderia se referir ao método usando:

```
<http>
  <intercept-url pattern="/user/{userId}/**"
  access="@webSecurity.checkUserId(authentication,#userId)"/>
  ...
</http>
```

Ou na configuração Java

```
http
  .authorizeRequests(authorize -> authorize
.antMatchers("/user/{userId}/**").access("@webSecurity.checkUserId(authentication,#use
rId)")
  ...
```

);

Em ambas as configurações, URLs que correspondem passariam a variável de caminho (e a converteriam) para o método checkUserId. Por exemplo, se a URL fosse /user/123/resource, o id passado seria 123.

11.3.3. Expressões de Segurança de Método

A segurança de método é um pouco mais complicada do que uma simples regra de permitir ou negar. O Spring Security 3.0 introduziu novas anotações para permitir suporte abrangente ao uso de expressões.

Anotações @Pre e @Post

Existem quatro anotações que suportam atributos de expressão para permitir verificações de autorização antes e depois da invocação, além de suportar o filtro de argumentos de coleção submetidos ou valores de retorno. São elas @PreAuthorize, @PreFilter, @PostAuthorize e @PostFilter. O uso delas é ativado por meio do elemento do namespace global-method-security:

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

Controle de Acesso usando @PreAuthorize e @PostAuthorize

A anotação mais obviamente útil é a @PreAuthorize, que decide se um método pode ou não ser invocado. Por exemplo (do aplicativo de amostra "Contacts"):

@PreAuthorize("hasRole('USER')")
public void create(Contact contact);

Isso significa que o acesso será permitido apenas para usuários com a função "ROLE_USER". Obviamente, a mesma coisa poderia ser facilmente alcançada usando uma configuração tradicional e um atributo de configuração simples para a função necessária. Mas e quanto a:

@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);

Aqui, estamos realmente usando um argumento de método como parte da expressão para decidir se o usuário atual tem a permissão "admin" para o contato fornecido. A expressão integrada hasPermission() está vinculada ao módulo Spring Security ACL através do contexto da aplicação, como veremos abaixo. Você pode acessar qualquer um dos argumentos do método pelo nome, como variáveis de expressão.

Existem várias maneiras pelas quais o Spring Security pode resolver os argumentos do método. O Spring Security usa DefaultSecurityParameterNameDiscoverer para descobrir os nomes dos parâmetros. Por padrão, as seguintes opções são tentadas para um método como um todo:

- Se a anotação @P do Spring Security estiver presente em um único argumento do método, o valor será usado. Isso é útil para interfaces compiladas com um JDK anterior ao JDK 8, que não contêm informações sobre os nomes dos parâmetros. Por exemplo:

```
import org.springframework.security.access.method.P;
...
@PreAuthorize("#c.name == authentication.name")
public void doSomething(@P("c") Contact contact);
```

Nos bastidores, isso é implementado usando AnnotationParameterNameDiscoverer, que pode ser personalizado para dar suporte ao atributo de valor de qualquer anotação especificada.

- Se a anotação @Param do Spring Data estiver presente em pelo menos um parâmetro do método, o valor será usado. Isso é útil para interfaces compiladas com um JDK anterior ao JDK 8, que não contêm informações sobre os nomes dos parâmetros. Por exemplo:

```
import org.springframework.data.repository.query.Param;
...
@PreAuthorize("#n == authentication.name")
Contact findContactByName(@Param("n") String name);
```

Nos bastidores, isso é implementado usando AnnotationParameterNameDiscoverer, que pode ser personalizado para dar suporte ao atributo de valor de qualquer anotação especificada.

- Se o JDK 8 foi usado para compilar o código com o argumento -parameters e o Spring 4+ está sendo usado, então a API de reflexão padrão do JDK é usada para descobrir os nomes dos parâmetros. Isso funciona tanto para classes quanto para interfaces.
- Por último, se o código foi compilado com os símbolos de depuração, os nomes dos parâmetros serão descobertos usando esses símbolos de depuração. Isso não funcionará para interfaces, pois elas não possuem

informações de depuração sobre os nomes dos parâmetros. Para interfaces, anotações ou a abordagem do JDK 8 devem ser usadas.

Qualquer funcionalidade do Spring-EL está disponível dentro da expressão, então você também pode acessar propriedades dos argumentos. Por exemplo, se você quiser que um determinado método só permita o acesso a um usuário cujo nome de usuário corresponda ao do contato, você poderia escrever:

```
@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

Quando se usa a anotação @PostFilter, o Spring Security itera sobre a coleção retornada e remove quaisquer elementos para os quais a expressão fornecida seja falsa. O nome filterObject se refere ao objeto atual na coleção. Você também pode filtrar antes da chamada do método, usando @PreFilter, embora isso seja uma exigência menos comum. A sintaxe é a mesma, mas se houver mais de um argumento que seja do tipo coleção, você deverá selecionar um pelo nome usando a propriedade filterTarget dessa anotação.

Observe que a filtragem obviamente não é um substituto para o ajuste de suas consultas de recuperação de dados. Se você estiver filtrando grandes coleções e removendo muitas das entradas, isso provavelmente será ineficiente.

Expressões Integradas

Existem algumas expressões integradas que são específicas para segurança de métodos, que já vimos em uso acima. Os valores filterTarget e returnValue são simples o suficiente, mas o uso da expressão hasPermission() merece uma análise mais detalhada.

A interface PermissionEvaluator

As expressões hasPermission() são delegadas a uma instância de PermissionEvaluator. Ela foi projetada para fazer a ponte entre o sistema de expressões e o sistema ACL do Spring Security, permitindo que você especifique restrições de autorização sobre objetos de domínio, com base em permissões abstratas. Ela não tem dependências explícitas sobre o módulo ACL, então você pode trocá-lo por uma implementação alternativa, se necessário. A interface tem dois métodos:

```
boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission);
boolean hasPermission(Authentication authentication, Serializable targetId, String targetType, Object permission);
```

Esses métodos mapeiam diretamente para as versões disponíveis da expressão, com a exceção de que o primeiro argumento (o objeto Authentication) não é fornecido. O primeiro é usado em situações onde o objeto de domínio, ao qual o acesso está sendo controlado, já está carregado. A expressão retornará true se o usuário atual tiver a permissão fornecida para esse objeto. A segunda versão é usada em casos onde o objeto não está carregado, mas seu identificador é conhecido. Um especificador abstrato de "tipo" para o objeto de domínio também é necessário, permitindo que as permissões corretas do ACL sejam carregadas. Tradicionalmente, isso tem sido a classe Java do objeto, mas não precisa ser assim, desde que seja consistente com a maneira como as permissões são carregadas.

Para usar expressões hasPermission(), você deve configurar explicitamente um PermissionEvaluator no seu contexto de aplicação. Isso seria feito da seguinte maneira:

```
<security:global-method-security pre-post-annotations="enabled">
  <security:expression-handler ref="expressionHandler"/>
</security:global-method-security>

<bean id="expressionHandler" class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
  <property name="permissionEvaluator" ref="myPermissionEvaluator"/>
</bean>
```

Onde myPermissionEvaluator é o bean que implementa PermissionEvaluator. Normalmente, isso será a implementação do módulo ACL chamada AclPermissionEvaluator. Veja a configuração do aplicativo de amostra "Contacts" para mais detalhes.

Meta Anotações de Segurança de Método

Você pode usar meta anotações para segurança de método para tornar seu código mais legível. Isso é especialmente conveniente se você achar que está repetindo a mesma expressão complexa em todo o código. Por exemplo, considere o seguinte:

```
@PreAuthorize("#contact.name == authentication.name")
```

Em vez de repetir isso em todos os lugares, podemos criar uma meta anotação que pode ser usada no lugar.

```
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
```

```
public @interface ContactPermission {}
```

Meta anotações podem ser usadas para qualquer uma das anotações de segurança de método do Spring Security. Para permanecer em conformidade com a especificação, as anotações JSR-250 não suportam meta anotações.

11.4. Implementações de Objetos Seguros

11.4.1. Interceptor de Segurança AOP Alliance (MethodInvocation)

Antes do Spring Security 2.0, a segurança de MethodInvocation exigia muita configuração repetitiva. Agora, a abordagem recomendada para a segurança de métodos é usar a configuração via namespace. Dessa forma, os beans da infraestrutura de segurança de métodos são configurados automaticamente, e você não precisa se preocupar com as classes de implementação. Vamos fornecer uma visão geral rápida das classes envolvidas aqui.

A segurança de métodos é aplicada usando um MethodSecurityInterceptor, que protege os MethodInvocation. Dependendo da abordagem de configuração, o interceptor pode ser específico para um único bean ou compartilhado entre vários beans. O interceptor usa uma instância de MethodSecurityMetadataSource para obter os atributos de configuração aplicáveis a uma invocação de método específica.

MapBasedMethodSecurityMetadataSource é usado para armazenar os atributos de configuração, que são indexados pelos nomes dos métodos (que podem usar curingas) e será usado internamente quando os atributos forem definidos no contexto da aplicação usando os elementos <intercept-methods> ou <protect-point>. Outras implementações serão usadas para lidar com a configuração baseada em anotações.

Configuração Explícita de MethodSecurityInterceptor

Você pode, é claro, configurar um MethodSecurityInterceptor diretamente no seu contexto de aplicação para uso com um dos mecanismos de proxy do Spring AOP:

```
<bean id="bankManagerSecurity"
class="org.springframework.security.access.intercept.aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <sec:method-security-metadata-source>
      <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
      <sec:protect method="com.mycompany.BankManager.getBalance"
access="ROLE_TELLER,ROLE_SUPERVISOR"/>
    </sec:method-security-metadata-source>
  </property>
</bean>
```

11.4.2. Interceptor de Segurança AspectJ (JoinPoint)

O interceptor de segurança AspectJ é muito semelhante ao interceptor de segurança AOP Alliance discutido na seção anterior. De fato, discutiremos apenas as diferenças nesta seção.

O interceptor AspectJ é chamado AspectJSecurityInterceptor. Ao contrário do interceptor de segurança AOP Alliance, que depende do contexto da aplicação Spring para integrar o interceptor de segurança via proxying, o AspectJSecurityInterceptor é integrado via compilador AspectJ. Não seria incomum usar ambos os tipos de interceptores de segurança na mesma aplicação, com o AspectJSecurityInterceptor sendo usado para segurança de instâncias de objetos de domínio e o MethodSecurityInterceptor do AOP Alliance sendo usado para segurança da camada de serviços.

Aqui está como o AspectJSecurityInterceptor é configurado no contexto de aplicação Spring:

```
<bean id="bankManagerSecurity"
class="org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <sec:method-security-metadata-source>
      <sec:protect method="com.mycompany.BankManager.delete*" access="ROLE_SUPERVISOR"/>
      <sec:protect method="com.mycompany.BankManager.getBalance"
access="ROLE_TELLER,ROLE_SUPERVISOR"/>
    </sec:method-security-metadata-source>
  </property>
</bean>
```

Como você pode ver, além do nome da classe, o AspectJSecurityInterceptor é exatamente o mesmo que o interceptor de segurança AOP Alliance. De fato, os dois interceptores podem compartilhar o mesmo securityMetadataSource, pois o SecurityMetadataSource trabalha com java.lang.reflect.Methods em vez de uma classe específica de biblioteca AOP. Claro, suas decisões de acesso têm acesso à invocação específica da biblioteca AOP relevante (por exemplo, MethodInvocation ou JoinPoint) e, como tal, podem considerar uma série de critérios adicionais ao tomar decisões de acesso (como os argumentos do método).
A seguir, você precisará definir um aspecto AspectJ. Exemplo:

```
package org.springframework.security.samples.aspectj;

import org.springframework.security.access.intercept.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.access.intercept.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {
  private AspectJSecurityInterceptor securityInterceptor;

  pointcut domainObjectInstanceExecution(): target(PersistableEntity) && execution(public * *(..)) &&
!within(DomainObjectInstanceSecurityAspect);

  Object around(): domainObjectInstanceExecution() {
    if (this.securityInterceptor == null) {
      return proceed();
    }
    AspectJCallback callback = new AspectJCallback() {
      public Object proceedWithObject() {
        return proceed();
      }
    };
    return this.securityInterceptor.invoke(thisJoinPoint, callback);
  }

  public AspectJSecurityInterceptor getSecurityInterceptor() {
    return securityInterceptor;
  }

  public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {
    this.securityInterceptor = securityInterceptor;
  }

  public void afterPropertiesSet() throws Exception {
    if (this.securityInterceptor == null)
      throw new IllegalArgumentException("securityInterceptor required");
  }
}
```

No exemplo acima, o interceptor de segurança será aplicado a cada instância de PersistableEntity, que é uma classe abstrata não mostrada (você pode usar qualquer outra classe ou expressão de ponto de corte que desejar). Para os curiosos, o AspectJCallback é necessário porque a instrução proceed(); tem um significado especial apenas dentro de um corpo de around(). O AspectJSecurityInterceptor chama essa classe anônima AspectJCallback quando deseja que o objeto de destino continue.
Agora, você precisará configurar o Spring para carregar o aspecto e conectá-lo com o AspectJSecurityInterceptor. Uma declaração de bean que realiza isso é mostrada abaixo:

```
<bean id="domainObjectInstanceSecurityAspect"
class="security.samples.aspectj.DomainObjectInstanceSecurityAspect" factory-method="aspectOf">
  <property name="securityInterceptor" ref="bankManagerSecurity"/>
</bean>
```

Pronto! Agora você pode criar seus beans de qualquer lugar dentro da sua aplicação, usando qualquer meio que achar adequado (por exemplo, `new Person();`) e o interceptor de segurança será aplicado.

11.5. Segurança de Método

A partir da versão 2.0, o Spring Security melhorou substancialmente o suporte para adicionar segurança aos métodos da camada de serviço. Ele oferece suporte para segurança com anotações JSR-250, bem como para a anotação original `@Secured` do framework. A partir da versão 3.0, você também pode usar novas anotações baseadas em expressões. A segurança pode ser aplicada a um único bean, utilizando o elemento `intercept-methods` para decorar a declaração do bean, ou você pode proteger múltiplos beans em toda a camada de serviço usando os pointcuts no estilo AspectJ.

11.5.1. EnableGlobalMethodSecurity

Podemos habilitar a segurança baseada em anotações usando a anotação `@EnableGlobalMethodSecurity` em qualquer instância de `@Configuration`. Por exemplo, o seguinte habilitaria a anotação `@Secured` do Spring Security:

```
@EnableGlobalMethodSecurity(securedEnabled = true)

public class MethodSecurityConfig {

  // ...

}
```

Adicionar uma anotação a um método (em uma classe ou interface) limita o acesso a esse método de acordo. O suporte nativo de anotações do Spring Security define um conjunto de atributos para o método. Esses atributos serão passados para o `AccessDecisionManager` para que ele tome a decisão real:

```
public interface BankService {
  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
}
```

O suporte para anotações JSR-250 pode ser habilitado com:

```
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
  // ...
}
```

Essas anotações são baseadas em padrões e permitem aplicar restrições simples baseadas em funções, mas não possuem o poder das anotações nativas do Spring Security. Para usar a nova sintaxe baseada em expressões, você deve usar:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)

public class MethodSecurityConfig {

  // ...

}
```

E o código equivalente em Java seria:

```
public interface BankService {
  @PreAuthorize("isAnonymous()")
  public Account readAccount(Long id);

  @PreAuthorize("isAnonymous()")
  public Account[] findAccounts();

  @PreAuthorize("hasAuthority('ROLE_TELLER')")
  public Account post(Account account, double amount);
}
```

11.5.2. GlobalMethodSecurityConfiguration

Às vezes, você pode precisar realizar operações mais complicadas do que as possíveis com a anotação `@EnableGlobalMethodSecurity`. Para esses casos, você pode estender o `GlobalMethodSecurityConfiguration`, garantindo que a anotação `@EnableGlobalMethodSecurity` esteja presente em sua subclasse. Por exemplo, se você quiser fornecer um `MethodSecurityExpressionHandler` personalizado, pode usar a seguinte configuração:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
  @Override
  protected MethodSecurityExpressionHandler createExpressionHandler() {
    // ... criar e retornar um MethodSecurityExpressionHandler personalizado ...
    return expressionHandler;
  }
}
```

Para informações adicionais sobre os métodos que podem ser sobrescritos, consulte a Javadoc do `GlobalMethodSecurityConfiguration`.

11.5.3. O Elemento `<global-method-security>`

Esse elemento é usado para habilitar a segurança baseada em anotações em sua aplicação (definindo os atributos apropriados no elemento) e também para agrupar declarações de pointcuts de segurança que serão aplicadas em todo o contexto da sua aplicação. Você deve declarar apenas um elemento `<global-method-security>`. A seguinte declaração habilitaria o suporte para `@Secured` do Spring Security:

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

Adicionar uma anotação a um método (em uma classe ou interface) limita o acesso a esse método de acordo. O suporte nativo de anotações do Spring Security define um conjunto de atributos para o método. Esses atributos serão passados para o AccessDecisionManager para que ele tome a decisão real:

```
public interface BankService {
  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();
```

```
  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
}
```

O suporte para anotações JSR-250 pode ser habilitado com:

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

Essas anotações são baseadas em padrões e permitem aplicar restrições simples baseadas em funções, mas não possuem o poder das anotações nativas do Spring Security. Para usar a nova sintaxe baseada em expressões, você deve usar:

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

E o código equivalente em Java seria:

```
public interface BankService {
  @PreAuthorize("isAnonymous()")
  public Account readAccount(Long id);

  @PreAuthorize("isAnonymous()")
  public Account[] findAccounts();

  @PreAuthorize("hasAuthority('ROLE_TELLER')")
  public Account post(Account account, double amount);
}
```

Anotações baseadas em expressões são uma boa escolha se você precisar definir regras simples que vão além da verificação de nomes de funções contra a lista de autoridades do usuário.

Observação
Os métodos anotados serão protegidos apenas para instâncias que são definidas como beans do Spring (no mesmo contexto de aplicação onde a segurança de método está habilitada). Se você quiser proteger instâncias que não são criadas pelo Spring (por exemplo, usando o operador `new`), então você precisará usar AspectJ.

Observação
Você pode habilitar mais de um tipo de anotação na mesma aplicação, mas apenas um tipo deve ser usado para qualquer interface ou classe, caso contrário, o comportamento não será bem definido. Se duas anotações forem encontradas que se aplicam a um método específico, apenas uma delas será aplicada.

11.5.4. Adicionando Pointcuts de Segurança com `protect-pointcut`

O uso de `protect-pointcut` é particularmente poderoso, pois permite aplicar segurança a muitos beans com uma declaração simples. Considere o seguinte exemplo:

```
<global-method-security>
  <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))" access="ROLE_USER"/>
</global-method-security>
```

Isso protegerá todos os métodos dos beans declarados no contexto de aplicação cujas classes estão no pacote com.mycompany e cujos nomes de classe terminam em "Service". Apenas usuários com o papel ROLE_USER poderão invocar esses métodos. Como na correspondência de URLs, as correspondências mais específicas devem vir primeiro na lista de pointcuts, pois a primeira expressão correspondente será usada. As anotações de segurança têm precedência sobre os pointcuts.

11.6. Segurança de Objetos de Domínio (ACLs)

11.6.1. Visão Geral

Aplicações complexas frequentemente precisam definir permissões de acesso não apenas em um nível de requisição web ou invocação de método. Em vez disso, as decisões de segurança devem considerar quem (Autenticação), onde (Invocação de Método) e o que (Algum Objeto de Domínio). Em outras palavras, as decisões de autorização também precisam considerar a instância real do objeto de domínio que está sendo invocado no método.

Imagine que você está projetando uma aplicação para uma clínica veterinária. Haverá dois grupos principais de usuários em sua aplicação baseada em Spring: a equipe da clínica veterinária e os clientes da clínica. A equipe terá acesso a todos os dados, enquanto seus clientes só poderão ver seus próprios registros. Para tornar as coisas mais interessantes, seus clientes poderão permitir que outros usuários vejam seus registros, como o mentor do "pré-escola de filhotes" ou o presidente do "Clube do Pônei" local.

Usando o Spring Security como base, você tem várias abordagens que podem ser usadas:
• Escrever seus métodos de negócios para aplicar a segurança. Você poderia consultar uma coleção dentro da instância do objeto de domínio Customer para determinar quais usuários têm acesso. Usando SecurityContextHolder.getContext().getAuthentication(), você poderá acessar o objeto Authentication.
• Escrever um AccessDecisionVoter para aplicar a segurança a partir do array GrantedAuthority[] armazenado no objeto Authentication. Isso significaria que seu AuthenticationManager precisaria preencher o Authentication com GrantedAuthority[] personalizados representando cada uma das instâncias do objeto Customer que o principal tem acesso.
• Escrever um AccessDecisionVoter para aplicar a segurança e acessar diretamente o objeto de domínio Customer. Isso significaria que seu voter precisaria de acesso a um DAO que permita recuperar o objeto Customer. Ele então acessaria a coleção de usuários aprovados do objeto Customer e tomaria a decisão apropriada.

Cada uma dessas abordagens é perfeitamente legítima. No entanto, a primeira associa sua verificação de autorização ao seu código de negócios. Os principais problemas com isso incluem a dificuldade aumentada de testar as unidades e o fato de que seria mais difícil reutilizar a lógica de autorização de Customer em outros lugares. Obter o array GrantedAuthority[] do objeto Authentication também é viável, mas não escalaria bem para um grande número de Customers. Se um usuário puder acessar 5.000 Customers (o que seria improvável, mas imagine se fosse um veterinário popular para um grande "Clube do Pônei"), a quantidade de memória consumida e o tempo necessário para construir o objeto Authentication seria indesejável. O método final, acessar diretamente o Customer de código externo, provavelmente é o melhor dos três. Ele alcança a separação de preocupações e não desperdiça memória ou ciclos de CPU, mas ainda é ineficiente, pois tanto o AccessDecisionVoter quanto o método de negócios final precisarão realizar uma chamada ao DAO responsável por recuperar o objeto Customer. Duas acessos por invocação de método são claramente indesejáveis. Além disso, com qualquer uma das abordagens listadas, você precisará escrever sua própria lógica de persistência e controle de lista de acesso (ACL) do zero.

Felizmente, existe uma alternativa, que discutiremos abaixo.

11.6.2. Conceitos Chave

Os serviços ACL do Spring Security são fornecidos no arquivo spring-security-acl-xxx.jar. Você precisará adicionar esse JAR ao seu classpath para usar as capacidades de segurança de instância de objetos de domínio do Spring Security.
As capacidades de segurança de instância de objetos de domínio do Spring Security se concentram no conceito de uma lista de controle de acesso (ACL). Cada instância de objeto de domínio no seu sistema tem sua própria ACL, e a ACL registra os detalhes de quem pode ou não trabalhar com aquele objeto de domínio. Com isso em mente, o Spring Security oferece três capacidades principais relacionadas ao ACL para sua aplicação:
• Uma maneira eficiente de recuperar entradas ACL para todos os seus objetos de domínio (e modificar essas ACLs)
• Uma maneira de garantir que um determinado principal tenha permissão para trabalhar com seus objetos antes que os métodos sejam chamados
• Uma maneira de garantir que um determinado principal tenha permissão para trabalhar com seus objetos (ou algo que eles retornem) depois que os métodos forem chamados

Como indicado no primeiro ponto, uma das principais capacidades do módulo ACL do Spring Security é fornecer uma maneira de alto desempenho para recuperar ACLs. Essa capacidade de repositório de ACL é extremamente importante, pois cada instância de objeto de domínio no seu sistema pode ter várias entradas de controle de acesso, e cada ACL pode herdar de outras ACLs em uma estrutura hierárquica (isso é suportado nativamente pelo Spring Security e é muito comum). A capacidade ACL do Spring Security foi projetada cuidadosamente para fornecer recuperação de ACLs de alto desempenho, juntamente com cache pluggable, atualizações de banco de dados minimizando deadlock, independência de frameworks ORM (usamos JDBC diretamente), encapsulamento adequado e atualização transparente do banco de dados.

Como os bancos de dados são essenciais para o funcionamento do módulo ACL, vamos explorar as quatro principais tabelas usadas por padrão na implementação. As tabelas são apresentadas abaixo na ordem do tamanho em uma implantação típica do Spring Security ACL, com a tabela que possui mais linhas listada por último:
• ACL_SID nos permite identificar de maneira única qualquer principal ou autoridade no sistema ("SID" significa

"identidade de segurança"). As únicas colunas são o ID, uma representação textual do SID e uma flag que indica se a representação textual se refere a um nome de principal ou a um GrantedAuthority. Assim, há uma única linha para cada principal ou GrantedAuthority único. Quando usado no contexto de recebimento de permissão, um SID é geralmente chamado de "destinatário".
• ACL_CLASS nos permite identificar de maneira única qualquer classe de objeto de domínio no sistema. As únicas colunas são o ID e o nome da classe Java. Assim, há uma única linha para cada classe única para a qual desejamos armazenar permissões ACL.
• ACL_OBJECT_IDENTITY armazena informações para cada instância única de objeto de domínio no sistema. As colunas incluem o ID, uma chave estrangeira para a tabela ACL_CLASS, um identificador único para saber qual instância da ACL_CLASS estamos fornecendo informações, o pai, uma chave estrangeira para a tabela ACL_SID representando o proprietário da instância do objeto de domínio e se permitimos que as entradas ACL herdem de qualquer ACL pai. Temos uma única linha para cada instância de objeto de domínio para a qual estamos armazenando permissões ACL.
• Finalmente, ACL_ENTRY armazena as permissões individuais atribuídas a cada destinatário. As colunas incluem uma chave estrangeira para ACL_OBJECT_IDENTITY, o destinatário (ou seja, uma chave estrangeira para ACL_SID), se estamos auditando ou não, e a máscara de bits inteira que representa a permissão real concedida ou negada. Temos uma única linha para cada destinatário que recebe uma permissão para trabalhar com um objeto de domínio.

Como mencionado no último parágrafo, o sistema ACL usa mascaramento de bits inteiros. Não se preocupe, você não precisa entender os detalhes do deslocamento de bits para usar o sistema ACL, mas basta saber que temos 32 bits que podemos ligar ou desligar. Cada um desses bits representa uma permissão, e por padrão as permissões são ler (bit 0), escrever (bit 1), criar (bit 2), excluir (bit 3) e administrar (bit 4). É fácil implementar sua própria instância de Permission se desejar usar outras permissões, e o restante da estrutura ACL funcionará sem conhecimento de suas extensões.

É importante entender que o número de objetos de domínio em seu sistema não tem absolutamente nenhum impacto no fato de termos escolhido usar mascaramento de bits inteiros. Embora você tenha 32 bits disponíveis para permissões, pode ter bilhões de instâncias de objetos de domínio (o que significará bilhões de linhas nas tabelas ACL_OBJECT_IDENTITY e provavelmente em ACL_ENTRY). Fazemos esse ponto porque às vezes as pessoas erroneamente acreditam que precisam de um bit para cada objeto de domínio potencial, o que não é o caso.

Agora que fornecemos uma visão básica do que o sistema ACL faz e como ele se parece em termos de estrutura de tabela, vamos explorar as interfaces principais. As interfaces principais são:
• Acl: Cada objeto de domínio tem um único objeto Acl, que internamente mantém as entradas de controle de acesso (AccessControlEntry) e também conhece o proprietário do Acl. Um Acl não se refere diretamente ao objeto de domínio, mas sim a um ObjectIdentity. O Acl é armazenado na tabela ACL_OBJECT_IDENTITY.
• AccessControlEntry: Um Acl contém várias entradas de controle de acesso (AccessControlEntry), que muitas vezes são abreviadas como ACE no framework. Cada ACE se refere a uma tupla específica de Permission, Sid e Acl. Um ACE também pode ser concedente ou não concedente e pode conter configurações de auditoria. O ACE é armazenado na tabela ACL_ENTRY.
• Permission: Uma permissão representa uma máscara de bits imutável e oferece funções convenientes para mascaramento de bits e saída de informações. As permissões básicas apresentadas acima (bits de 0 a 4) estão contidas na classe BasePermission.
• Sid: O módulo ACL precisa se referir a principais e GrantedAuthority[]. Um nível de indireção é fornecido pela interface Sid, que é uma abreviação de "identidade de segurança". As classes comuns incluem PrincipalSid (para representar o principal dentro de um objeto Authentication) e GrantedAuthoritySid. As informações de identidade de segurança são armazenadas na tabela ACL_SID.
• ObjectIdentity: Cada objeto de domínio é representado internamente dentro do módulo ACL por um ObjectIdentity. A implementação padrão é chamada de ObjectIdentityImpl.
• AclService: Recupera o Acl aplicável para um dado ObjectIdentity. Na implementação incluída (JdbcAclService), as operações de recuperação são delegadas a um LookupStrategy. O `LookupStrategy

```
// Prepare the information we'd like in our access control entry (ACE)
ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new Long(44));
Sid sid = new PrincipalSid("Samantha");
Permission p = BasePermission.ADMINISTRATION;
// Create or update the relevant ACL
MutableAcl acl = null;
try {
```

```
acl = (MutableAcl) aclService.readAclById(oi);
} catch (NotFoundException nfe) {
acl = aclService.createAcl(oi);
}
// Now grant some permissions via an access control entry (ACE)
acl.insertAce(acl.getEntries().length, p, sid, true);
aclService.updateAcl(acl);
```

No exemplo acima, estamos recuperando o ACL associado ao objeto de domínio "Foo" com o identificador número 44. Em seguida, estamos adicionando uma ACE (Access Control Entry) para que um principal chamado "Samantha" possa "administrar" o objeto. O código é autoexplicativo, exceto pelo método insertAce. O primeiro argumento do método insertAce define a posição onde a nova entrada será inserida no ACL. No exemplo acima, estamos apenas adicionando a nova ACE no final das ACEs existentes. O último argumento é um valor booleano indicando se a ACE está concedendo ou negando permissão. Na maioria das vezes, será concedendo (true), mas se for negando (false), as permissões são efetivamente bloqueadas.

O Spring Security não fornece integração especial para criar, atualizar ou excluir ACLs como parte de suas operações de DAO ou repositório. Em vez disso, você precisará escrever código como o mostrado acima para seus objetos de domínio individuais. Uma abordagem interessante seria considerar o uso de AOP (Programação Orientada a Aspectos) na camada de serviços para integrar automaticamente as informações de ACL com as operações dessa camada, o que tem mostrado ser eficaz em várias situações.

Depois de usar as técnicas acima para armazenar informações de ACL no banco de dados, o próximo passo é utilizar essas informações de ACL como parte da lógica de decisão de autorização. Você tem várias opções aqui. Você poderia escrever seu próprio AccessDecisionVoter ou AfterInvocationProvider, que são executados respectivamente antes ou depois da invocação de um método. Essas classes usariam o AclService para recuperar o ACL relevante e chamariam o método Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode) para decidir se a permissão é concedida ou negada. Alternativamente, você poderia usar nossas classes AclEntryVoter, AclEntryAfterInvocationProvider ou AclEntryAfterInvocationCollectionFilteringProvider. Essas classes oferecem uma abordagem declarativa para avaliar as informações de ACL em tempo de execução, liberando você da necessidade de escrever código. Consulte os aplicativos de exemplo para aprender como usar essas classes.

Continua na página 164, capitulo 12, oauth2