**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**

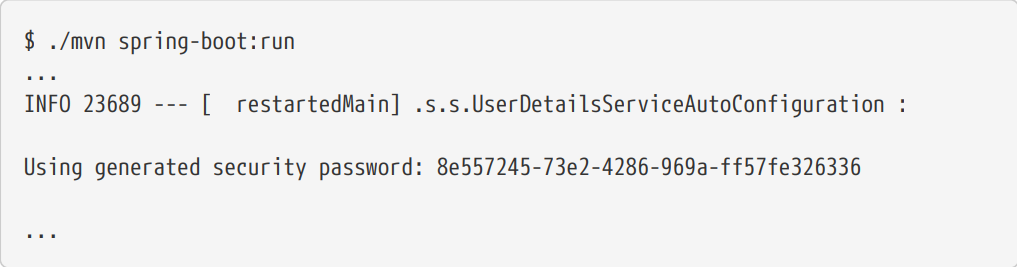
O único passo necessário é atualizar as dependências usando Maven ou Gradle.

**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**



**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:

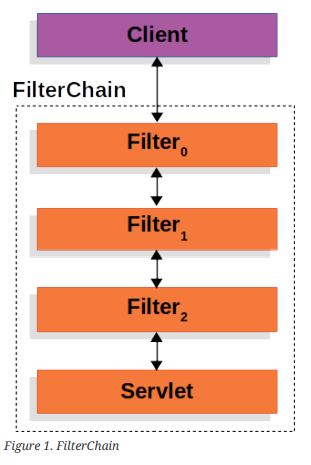
* Exige um usuário autenticado para qualquer interação com a aplicação.
* Gera um formulário de login padrão para você.
* Permite que o usuário com o nome de usuário user e uma senha registrada no console se autentique por meio de autenticação baseada em formulário (no exemplo anterior, a senha é 8e557245-73e2-4286-969a-ff57fe326336).
* 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.

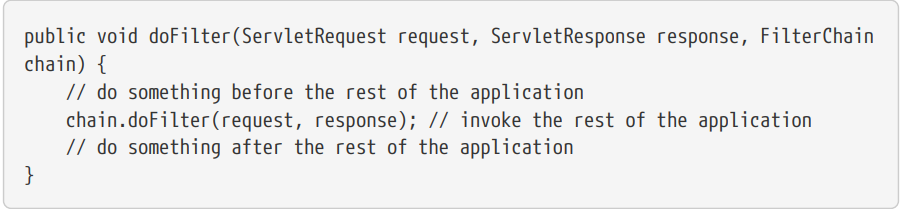


O cliente envia uma solicitação para a aplicação, e o contêiner cria um **FilterChain** que contém os **Filtros** e o **Servlet** que devem processar o HttpServletRequest com base no caminho da URI da solicitação. Em uma aplicação Spring MVC, o Servlet é uma instância do DispatcherServlet. No máximo, 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**

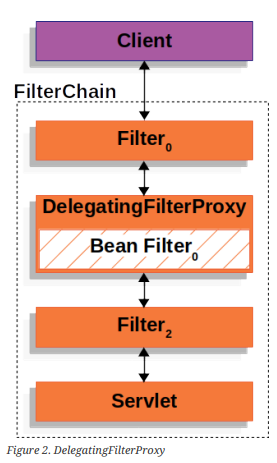


Como um Filtro só afeta os Filtros e o Servlet subsequentes, a ordem em que cada Filtro é invocado é extremamente importante.

**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**.



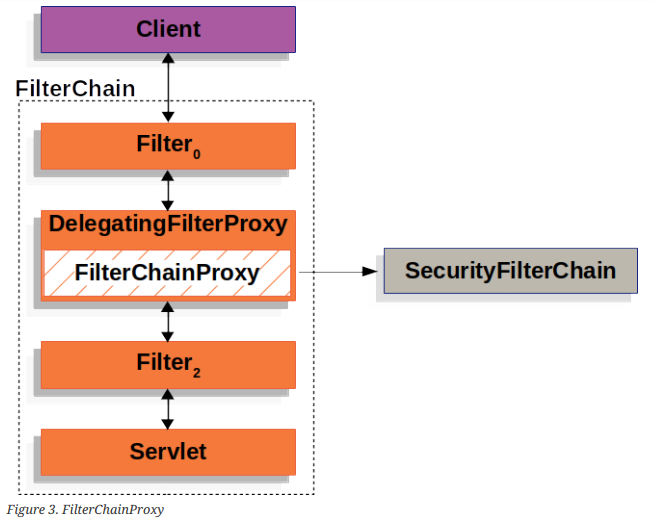
O **DelegatingFilterProxy** procura o Bean Filter0 no **ApplicationContext** e, em seguida, invoca o Bean Filter0. O pseudocódigo do **DelegatingFilterProxy** pode ser visto abaixo.

**Exemplo 48. Pseudocódigo do DelegatingFilterProxy**

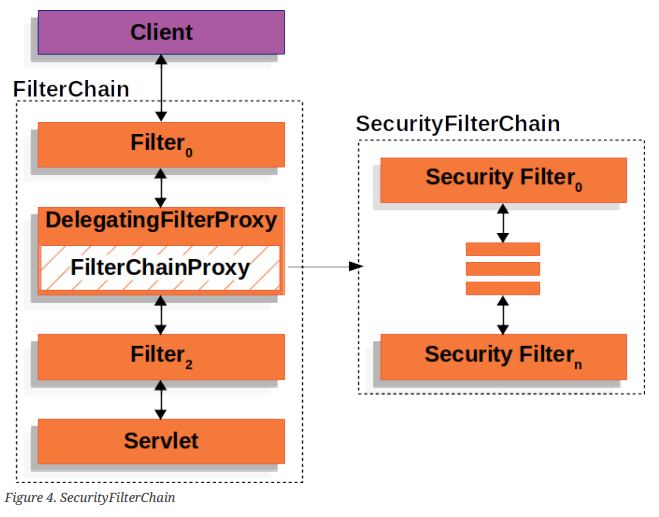


Outro benefício do **DelegatingFilterProxy** é que ele permite adiar a busca pelas instâncias de Beans de Filtro. Isso é importante porque o contêiner precisa registrar as instâncias de Filtro antes de iniciar. No entanto, o Spring normalmente usa o **ContextLoaderListener** para carregar os Beans do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



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

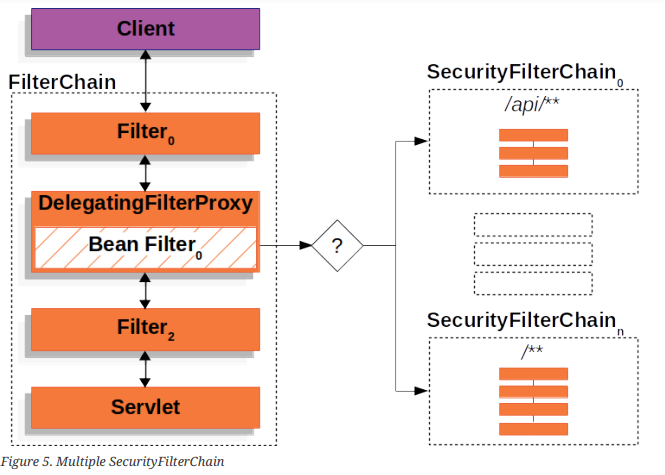


Os Filtros de Segurança no **SecurityFilterChain** são normalmente Beans, mas são registrados com o **FilterChainProxy** em vez de com o **DelegatingFilterProxy**. O **FilterChainProxy** oferece várias vantagens em relação ao registro direto no contêiner Servlet ou ao **DelegatingFilterProxy**. Primeiramente, 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.



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.

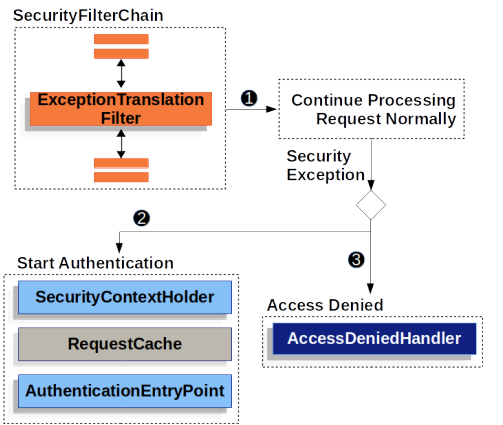
Abaixo está uma lista abrangente da ordem dos Filtros de Segurança do Spring Security:

* **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**

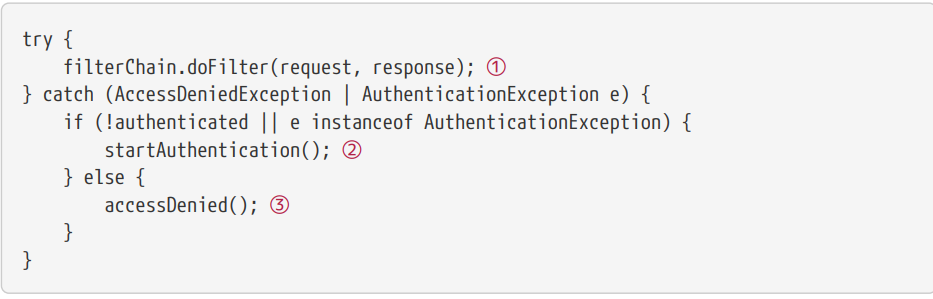
O **ExceptionTranslationFilter** permite a tradução das exceções **AccessDeniedException** e **AuthenticationException** em respostas HTTP.

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:



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.

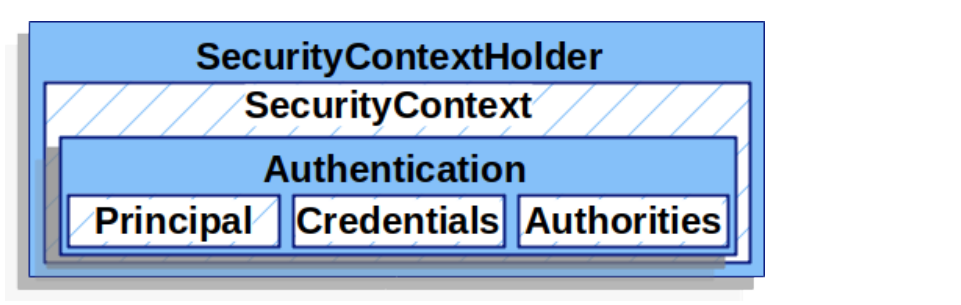
* **SecurityContextHolder** – O **SecurityContextHolder** é onde o Spring Security armazena os detalhes de quem está autenticado.
* **SecurityContext** – Obtido a partir do **SecurityContextHolder**, contém a **Authentication** do usuário atualmente autenticado.
* **Authentication** – Pode ser a entrada 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**



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**



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

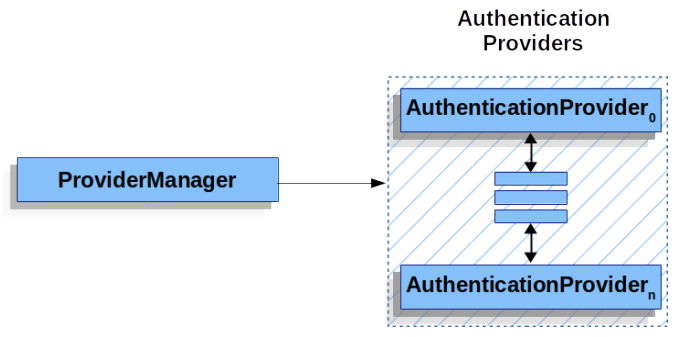
**AuthenticationManager** é a API que define como os filtros do **Spring Security** realizam a autenticação. A **Authentication** que é retornada é então definida no **SecurityContextHolder** pelo controlador (ou seja, pelos filtros do **Spring Security**) que invocou o **AuthenticationManager**. Se 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

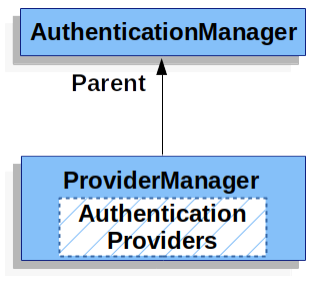
**ProviderManager** é a implementação mais comumente usada do **AuthenticationManager**.

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.

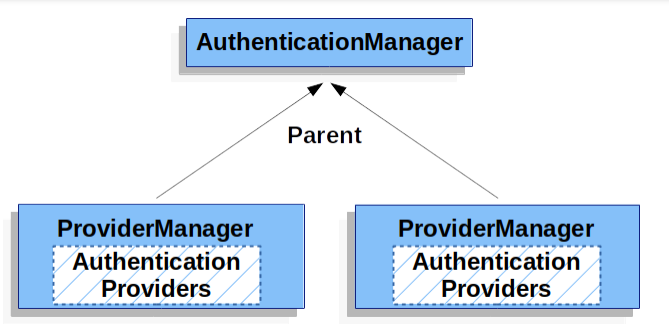


Na prática, cada **AuthenticationProvider** sabe como realizar um tipo específico de autenticação. Por exemplo, um **AuthenticationProvider** pode ser capaz de validar um nome de usuário/senha, enquanto outro pode ser capaz de autenticar uma asserção SAML. Isso permite que cada **AuthenticationProvider** 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.

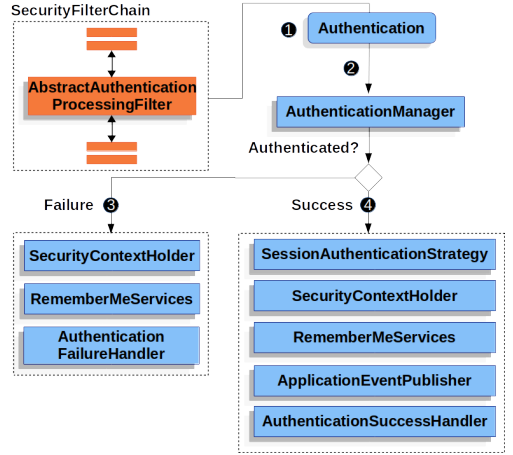
**10.7. AuthenticationProvider**  
Múltiplos **AuthenticationProviders** podem ser injetados no **ProviderManager**. Cada **AuthenticationProvider** realiza um tipo específico de autenticação. Por exemplo, o **DaoAuthenticationProvider** suporta autenticação baseada em nome de usuário/senha, enquanto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

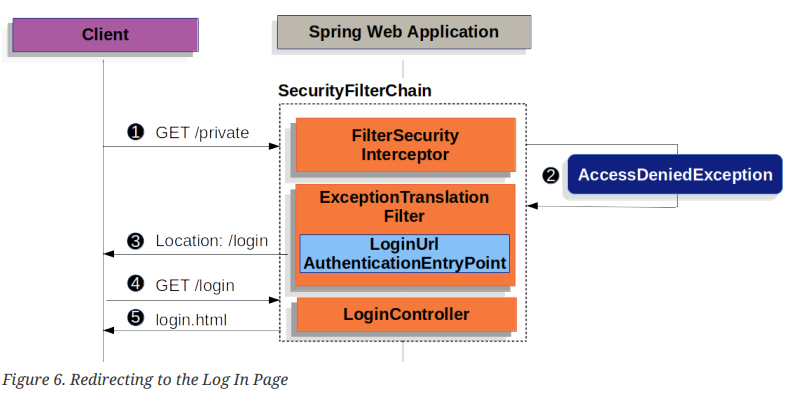
**Lendo o Nome de Usuário e a Senha**  
O **Spring Security** fornece os seguintes mecanismos integrados para ler um nome de usuário e senha a partir do **HttpServletRequest**:

* 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.



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.

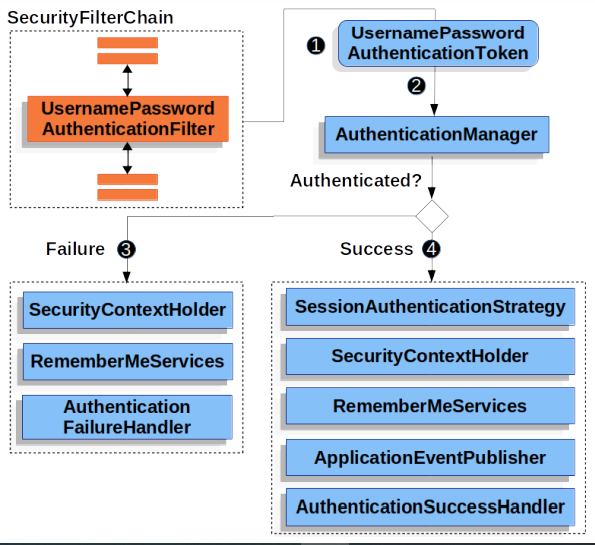


Figura 7. Autenticação de Nome de Usuário e Senha

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.

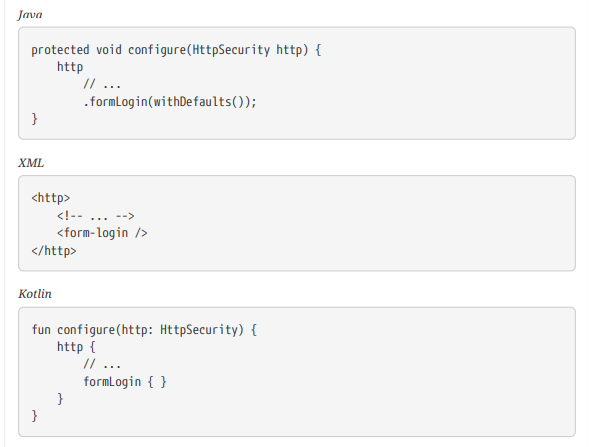
Se a autenticação falhar, então **Falha**:

* 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:



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

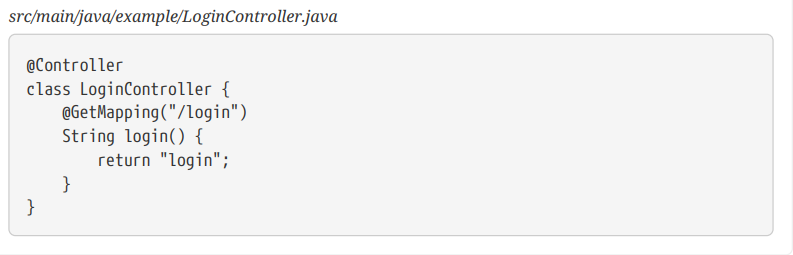
A configuração abaixo demonstra como fornecer um formulário de login personalizado.



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**:



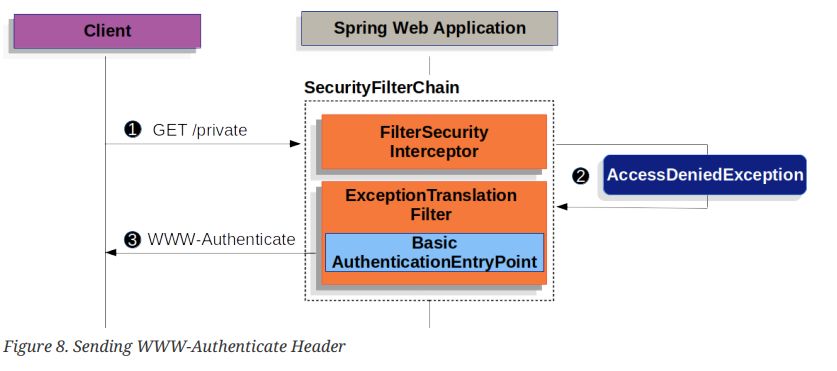
Há alguns pontos chave sobre o formulário HTML padrão: • O formulário deve realizar um post para /login • O formulário precisará incluir um Token CSRF, que é incluído automaticamente pelo Thymeleaf. • O formulário deve especificar o nome de usuário em um parâmetro chamado username • O formulário deve 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:



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.



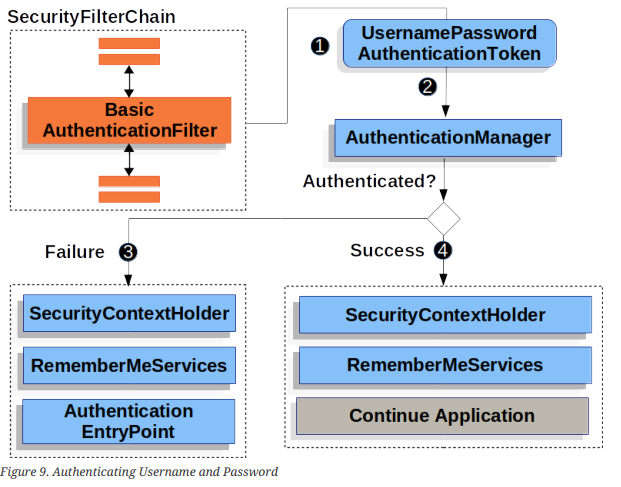
A figura é baseada no nosso diagrama de SecurityFilterChain.

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.



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**



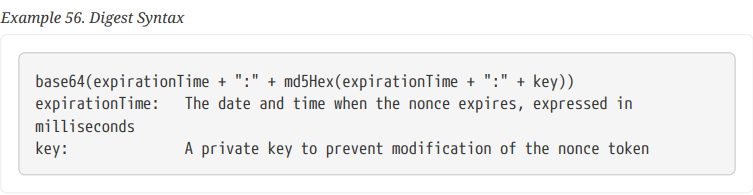
10.10.3. Autenticação Digest  
Esta seção fornece detalhes sobre como o Spring Security oferece suporte à Autenticação Digest, que é fornecida pelo **DigestAuthenticationFilter**.

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:



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:



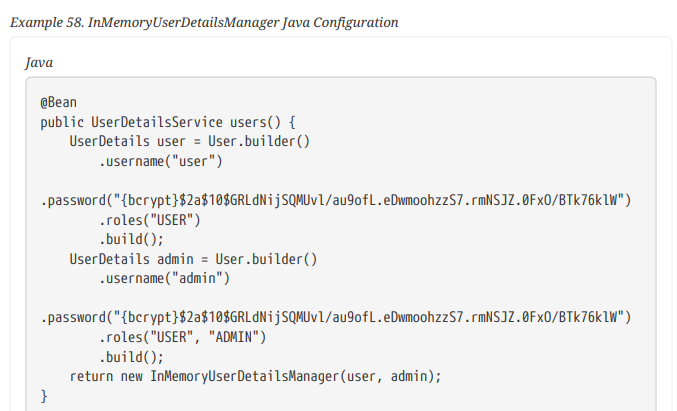


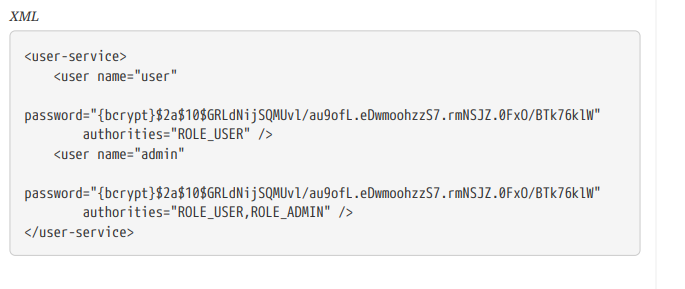
10.10.4. **Autenticação em Memória**

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.

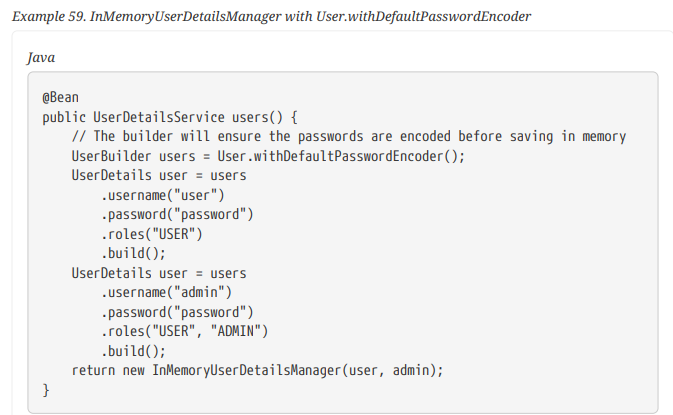


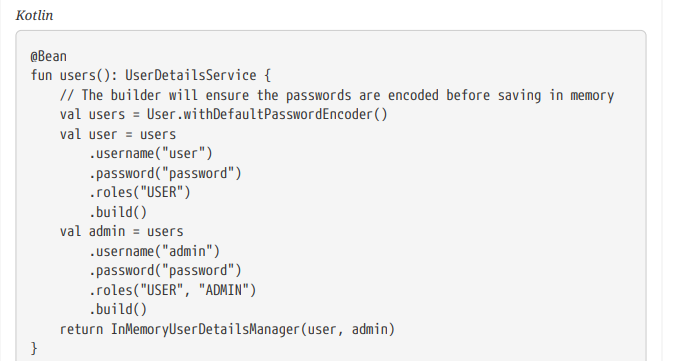




Os exemplos acima armazenam as senhas em um formato seguro, mas deixam a desejar em termos de experiência para começar.

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.





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.



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**.

... continuar a partir da pagina 78