**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**

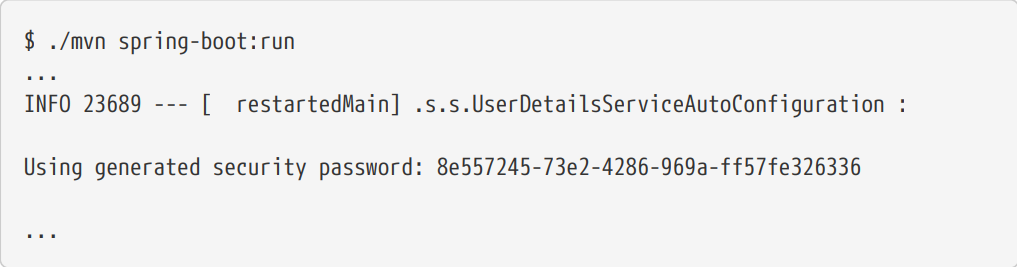
O único passo necessário é atualizar as dependências usando Maven ou Gradle.

**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:

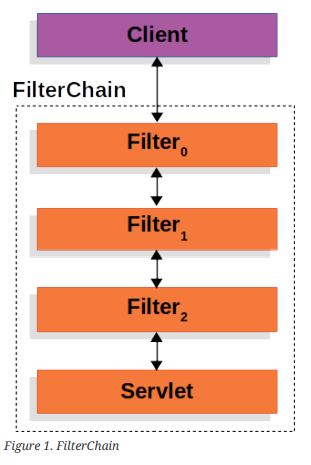
* Exige um usuário autenticado para qualquer interação com a aplicação.
* Gera um formulário de login padrão para você.
* Permite que o usuário com o nome de usuário user e uma senha registrada no console se autentique por meio de autenticação baseada em formulário (no exemplo anterior, a senha é 8e557245-73e2-4286-969a-ff57fe326336).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

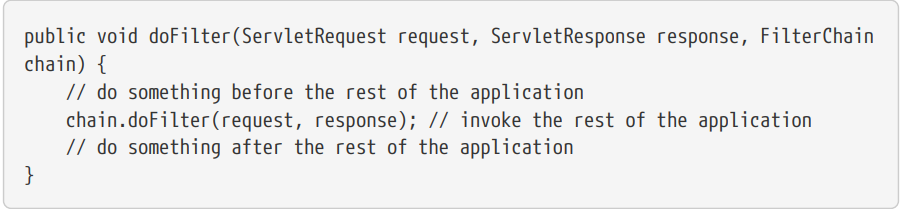


O cliente envia uma solicitação para a aplicação, e o contêiner cria um **FilterChain** que contém os **Filtros** e o **Servlet** que devem processar o HttpServletRequest com base no caminho da URI da solicitação. Em uma aplicação Spring MVC, o Servlet é uma instância do DispatcherServlet. No máximo,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

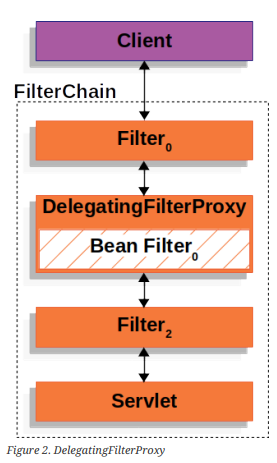


Como um Filtro só afeta os Filtros e o Servlet subsequentes, a ordem em que cada Filtro é invocado é extremamente importante.

**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**.



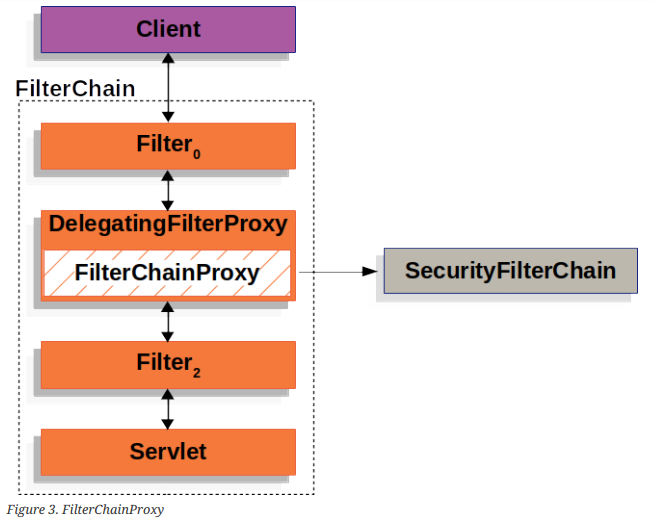
O **DelegatingFilterProxy** procura o Bean Filter0 no **ApplicationContext** e, em seguida, invoca o Bean Filter0. O pseudocódigo do **DelegatingFilterProxy** pode ser visto abaixo.

**Exemplo 48. Pseudocódigo do DelegatingFilterProxy**

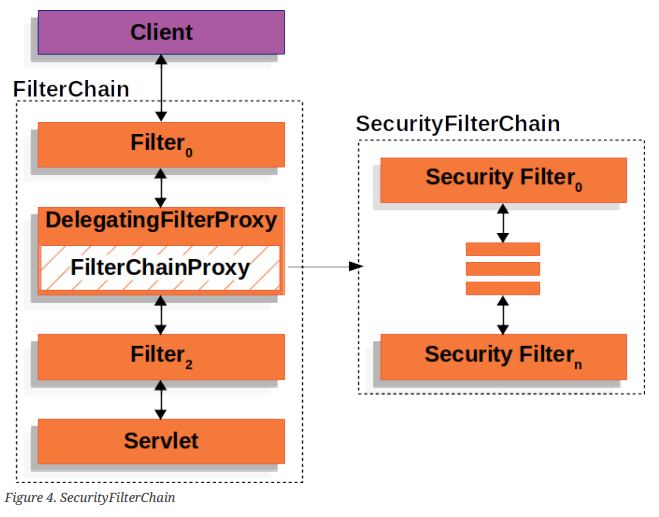


Outro benefício do **DelegatingFilterProxy** é que ele permite adiar a busca pelas instâncias de Beans de Filtro. Isso é importante porque o contêiner precisa registrar as instâncias de Filtro antes de iniciar. No entanto, o Spring normalmente usa o **ContextLoaderListener** para carregar os Beans do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

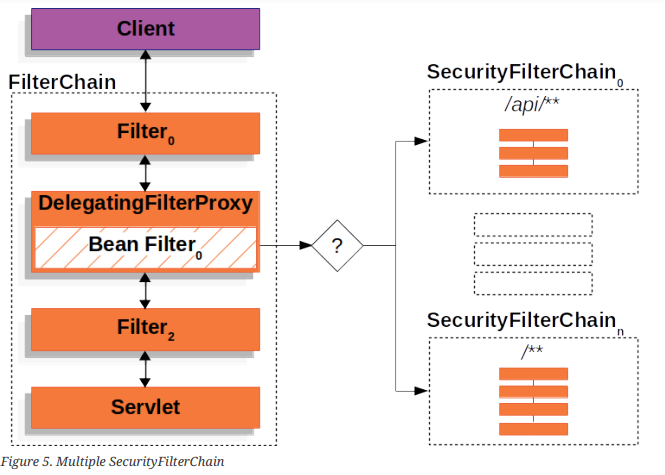


Os Filtros de Segurança no **SecurityFilterChain** são normalmente Beans, mas são registrados com o **FilterChainProxy** em vez de com o **DelegatingFilterProxy**. O **FilterChainProxy** oferece várias vantagens em relação ao registro direto no contêiner Servlet ou ao **DelegatingFilterProxy**. Primeiramente,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

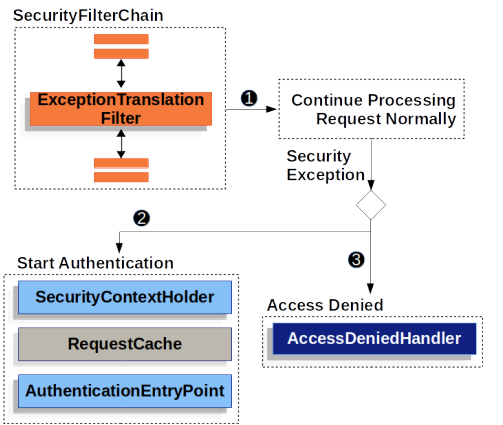
Abaixo está uma lista abrangente da ordem dos Filtros de Segurança do Spring Security:

* **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**

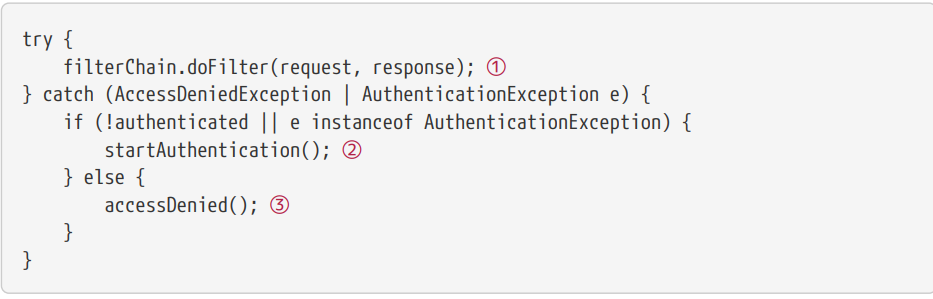
O **ExceptionTranslationFilter** permite a tradução das exceções **AccessDeniedException** e **AuthenticationException** em respostas HTTP.

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**, onde invocar FilterChain.doFilter(request, response) é equivalente a invocar o restante da aplicação. Isso significa que, se outra parte da aplicação (por exemplo, **FilterSecurityInterceptor** ou segurança por método) lançar uma **AuthenticationException** ou **AccessDeniedException**, ela será capturada e tratada aqui.
2. **Se o usuário não estiver autenticado ou ocorrer uma AuthenticationException**, então o processo de autenticação é iniciado:
   * O **SecurityContextHolder** é limpo.
   * A **HttpServletRequest** é salva no **RequestCache**.
   * O **AuthenticationEntryPoint** solicita as credenciais ao cliente (redirecionando para a página de login ou enviando um cabeçalho **WWW-Authenticate**).**Caso contrário, se for uma AccessDeniedException**, o acesso será negado, e o **AccessDeniedHandler** será invocado para tratar o acesso negado.