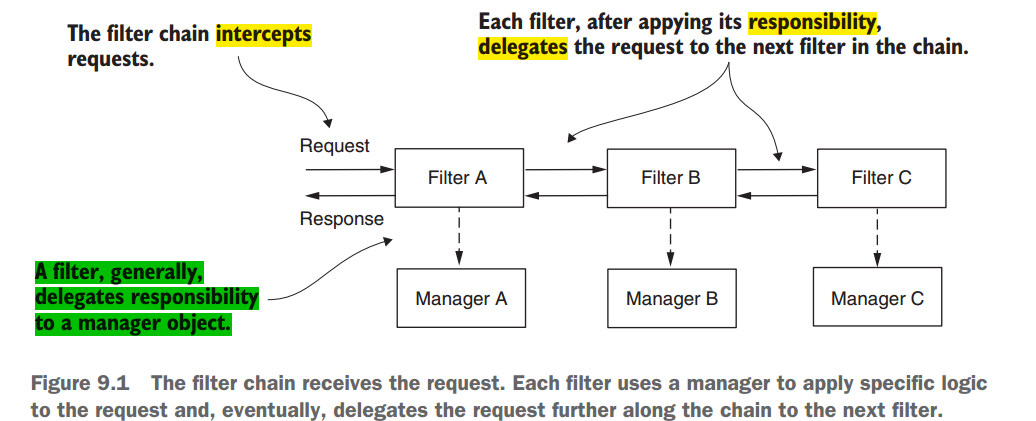
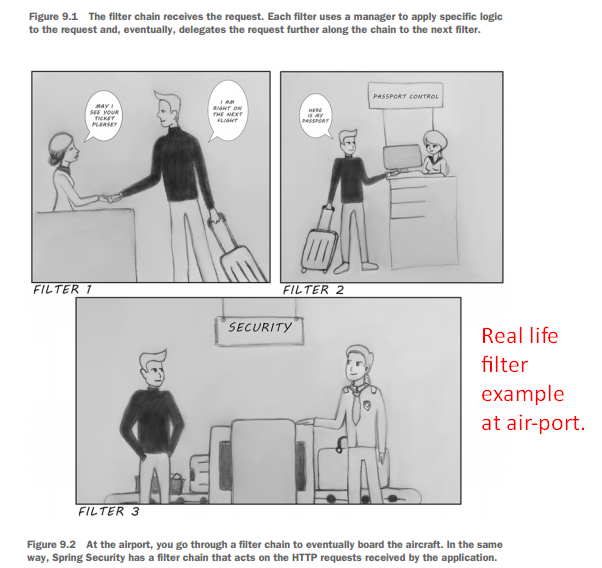
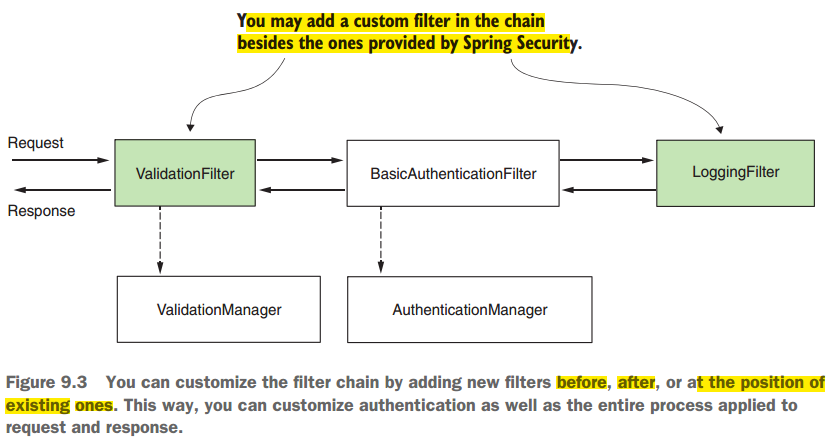
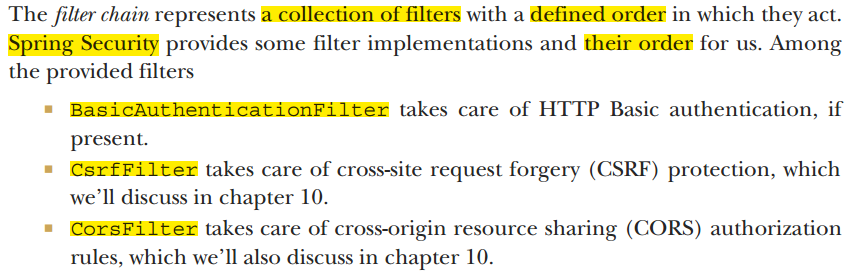
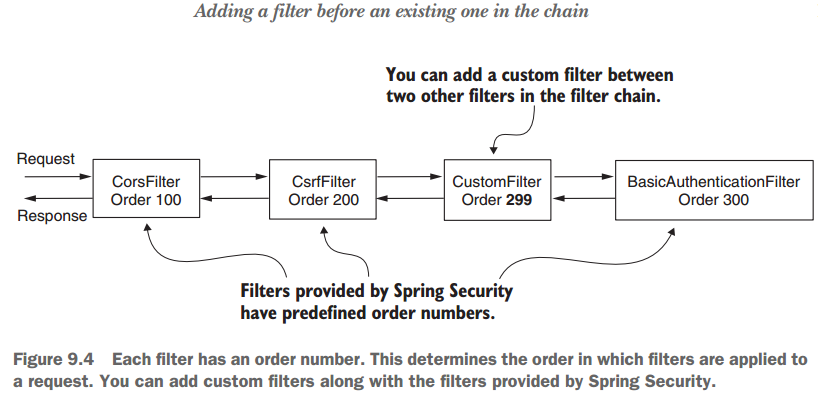
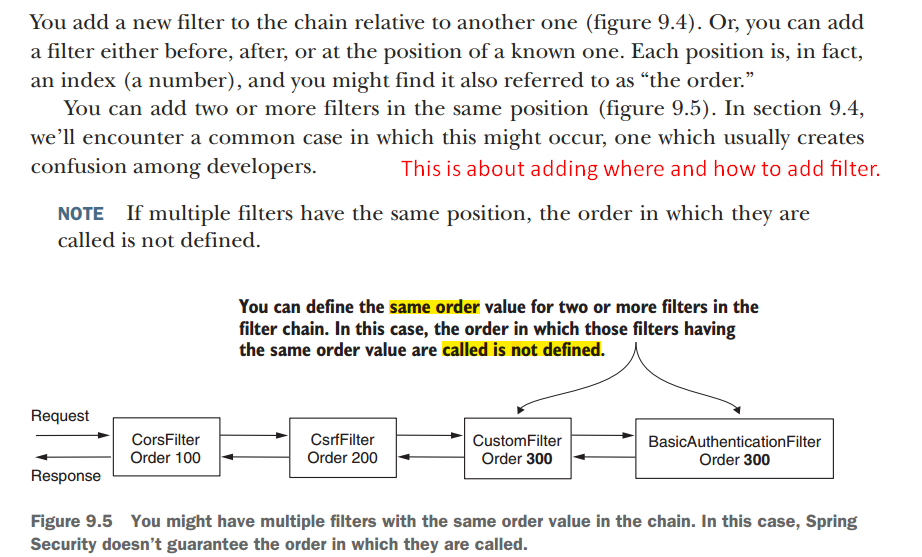
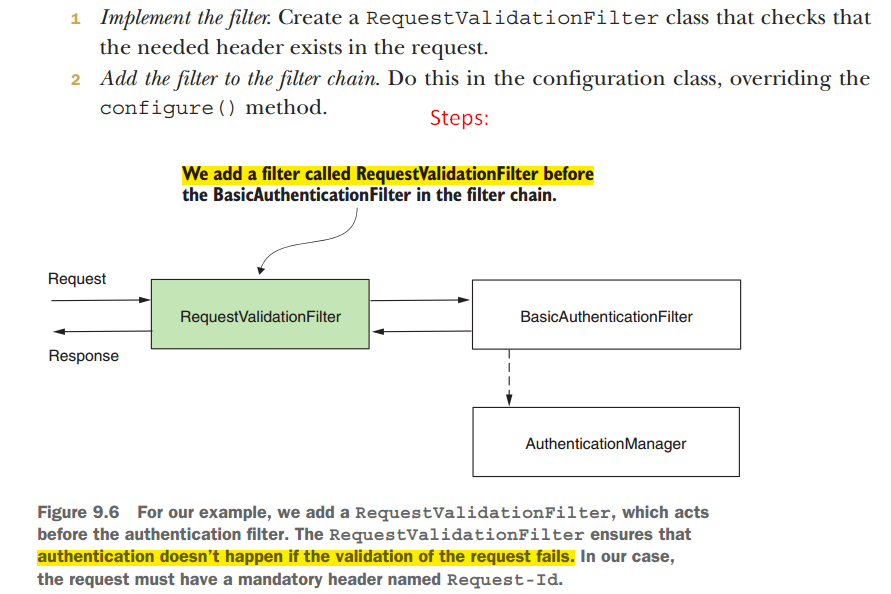
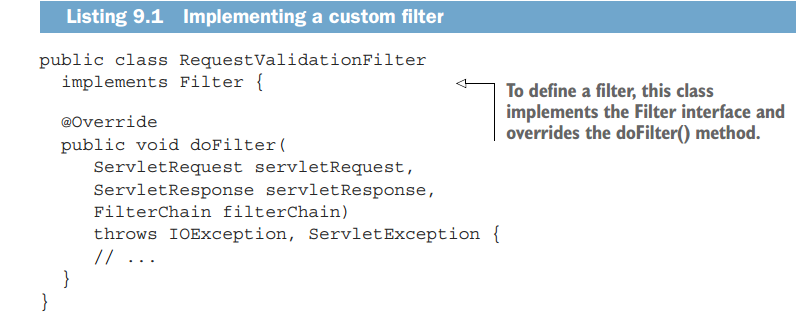
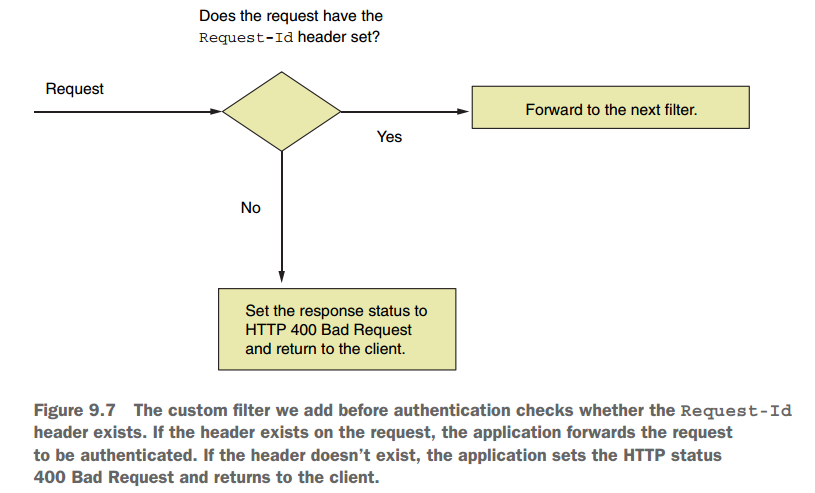
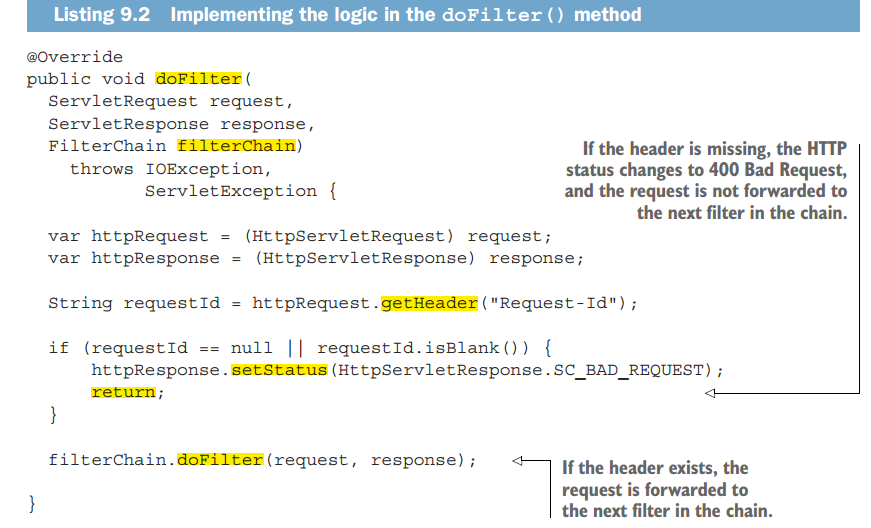
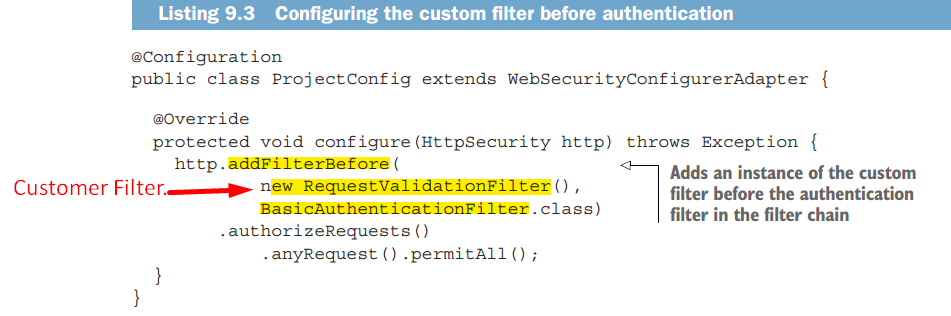
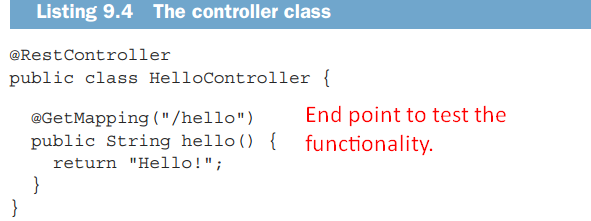
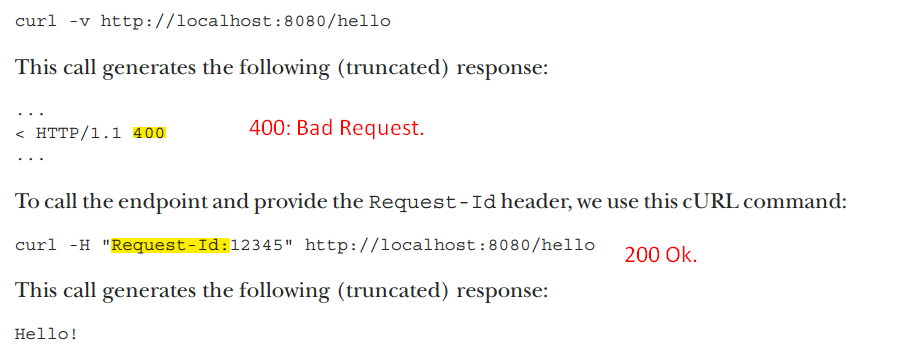
Ch: 09 🡺 Implementing Filters

1. 
2. In Spring Security, HTTP filters delegate the different responsibilities that apply to an HTTP request.
3. We saw that authentication filter delegates authentication responsibility to authentication manager.
   1. NOTE: This is general way where one filter delegates responsibility to its manager.
4. We also learn that a certain filter that takes care of authentication configuration after successful authentication.
5. In Spring Security, in general, HTTP Filters manage each responsibility that must be applied to a request.
6. The filters form a chain of responsibilities.
7. **Real Life Example**:
   1. In airport, you have to go through many filters.
8. 
9. 
10. **Agenda**:
    1. We will discuss how to customize filters that are part of the authentication and authorization architecture in Spring Security.
    2. **For example**, you might want to augment authentication by adding one more step for the user, like checking their email address or using a one-time password.  
       You can audit authentication events.   
       You will find various scenarios where applications use auditing authentication events: from debugging purposes to identifying user’s behavior.
11. Learning how to custom filter chain as default configuration may not fit in many situations.  
    **Such as default authentication filter rely on a username and password but in real scenario, you need more than this.**
12. 

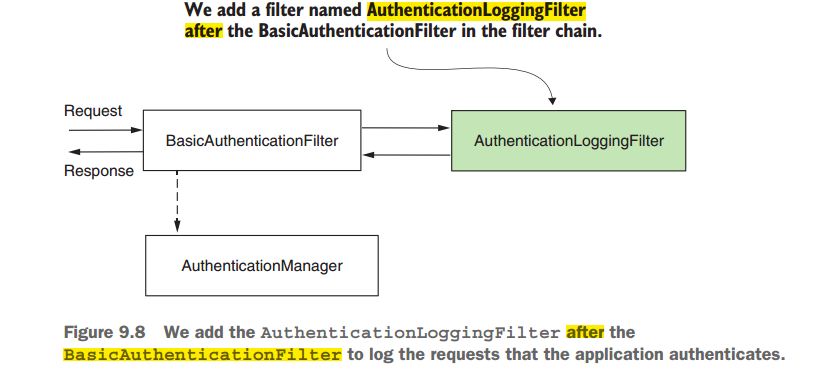
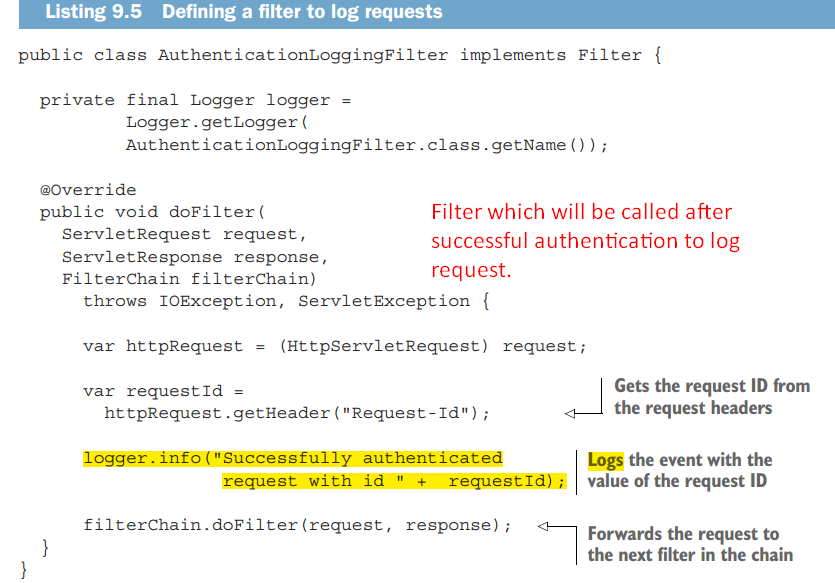
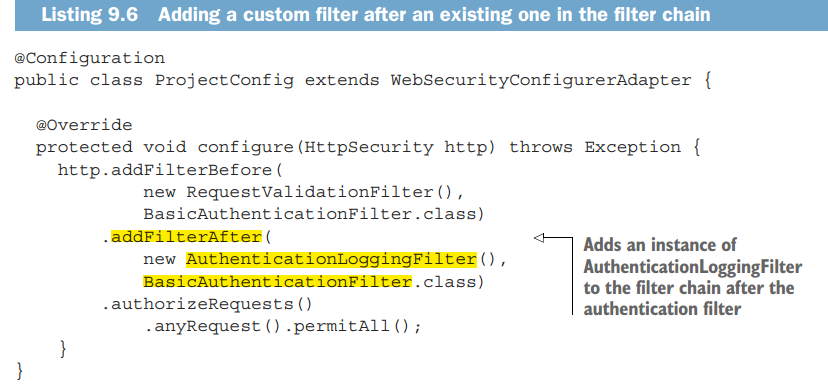
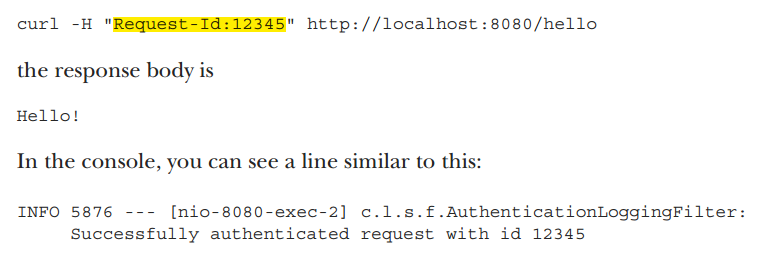
9.1 Implementing filters in Spring Security Architecture

1. **Agenda**:
   1. How filter and filter chain works?
2. We learnt in previous chapters that the authentication filter intercepts the request and delegates the authentication responsibility to authentication manager.
3. The filters in Spring Security are typically **HTTP Filters**.   
   We can create filters by implementing javax.servlet.Filter interface.  
   Override doFilter() to implement its logic.
4. 
5. See when you call **HttpSecurity.httpBasic()**, **BasicAuthenticationFilter** is added to the Spring Security Filter Chain.
6. So, depending on the configuration, filter chain gets affected.
7. 
8.   
   “**At the position of a known one**” means that you can say that I want to add a custom filter at BasicAuthenticationFilter which is standard Spring Security filter. Now there are two cases.
   1. **Case 01**: You didn’t call **HttpSecurity.httpBasic() so BasicAuthenticationFilter** will not be added and your custom filter is all at the place of **BasicAuthenticationFilter.**
   2. **Case 02**: You called above method so BasicAuthenticationFilter would be present in filter chain and then your custom filter would be at the same priority level as the **BasicAuthenticationFilter** so which one would be called first is undefined.   
      **NOTE**: If this is not clear no problem. Later on, in this chapter, you will see this with practical example.

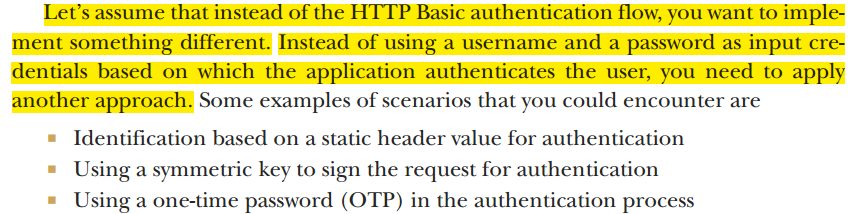
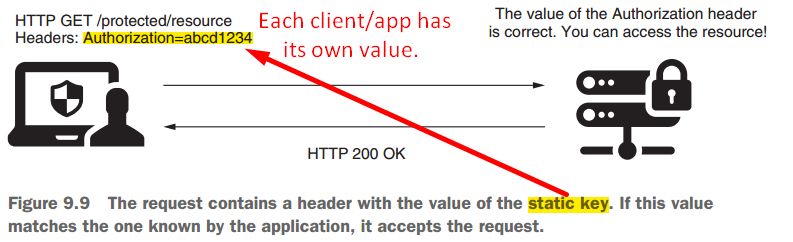
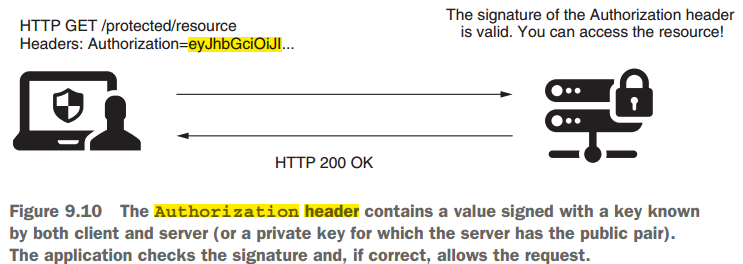
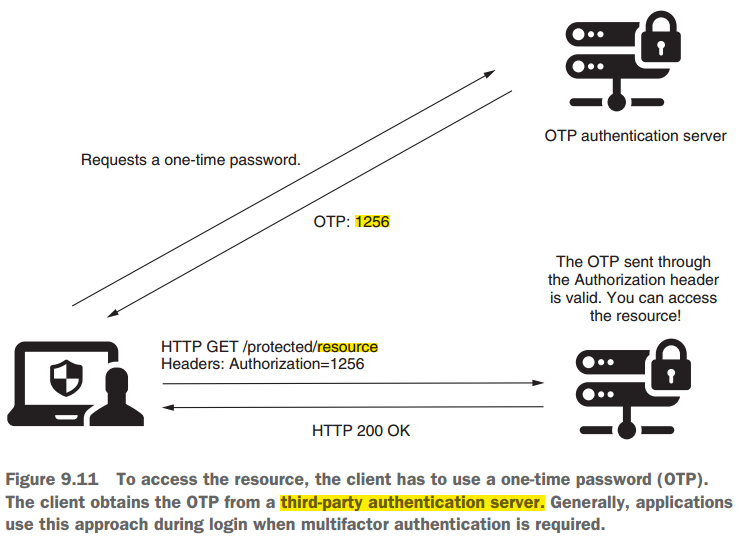
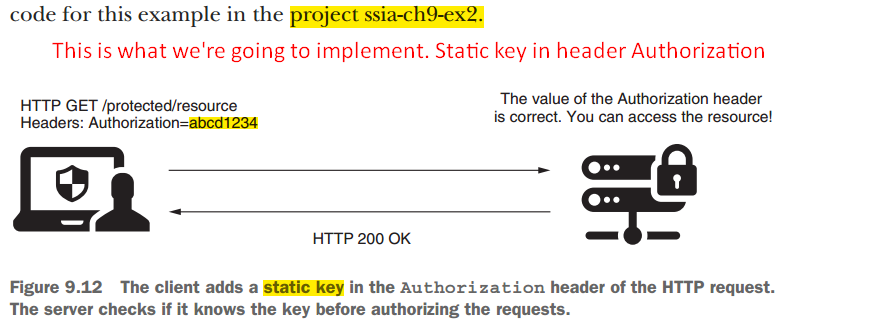
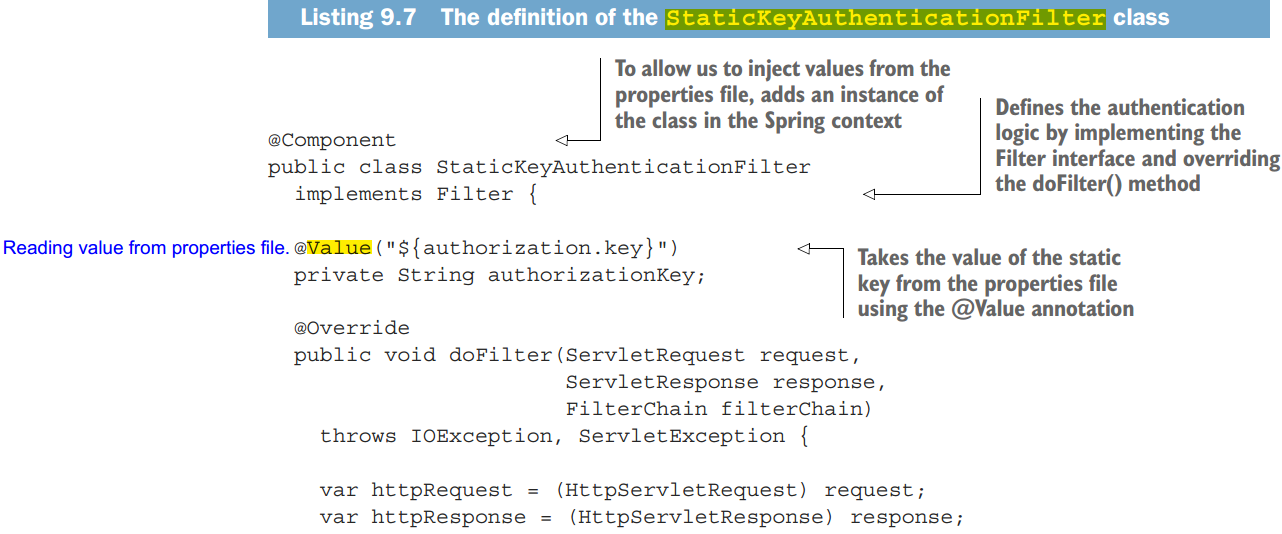
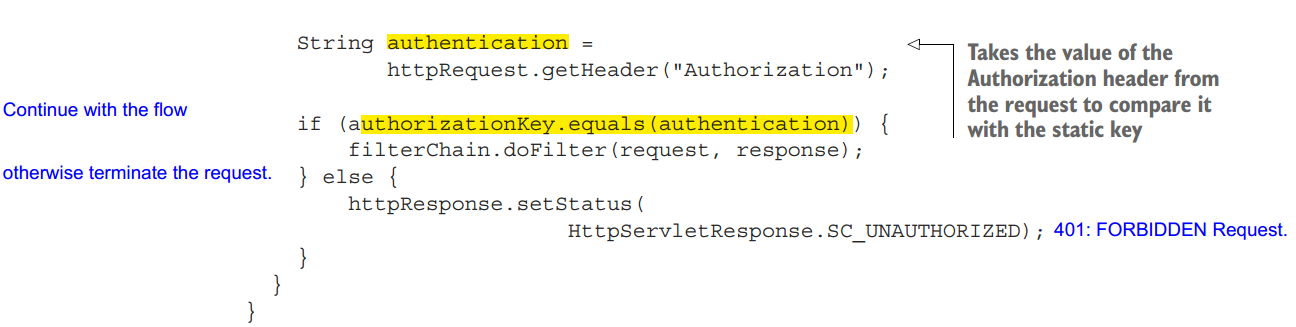
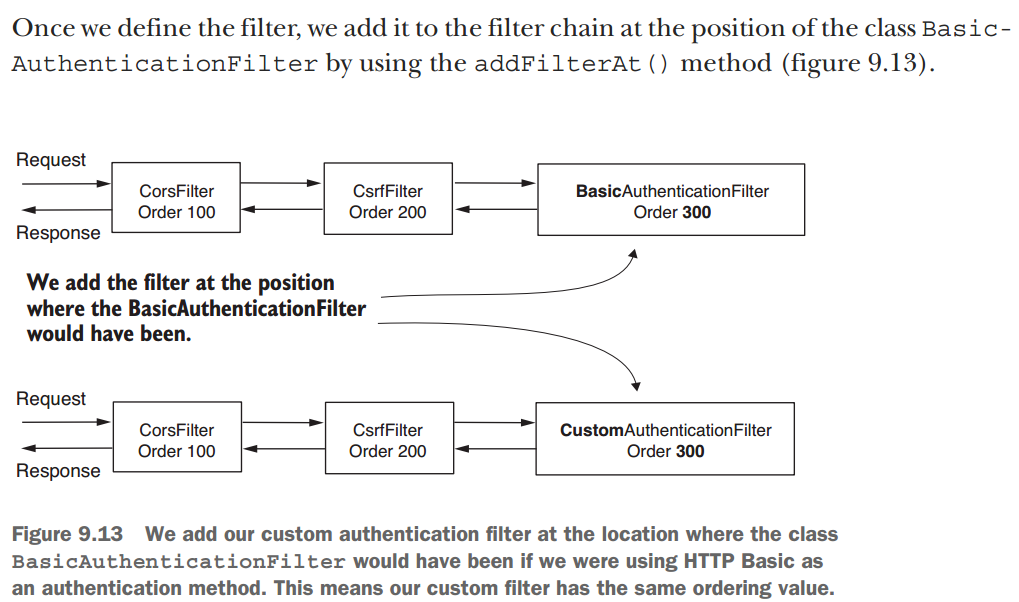
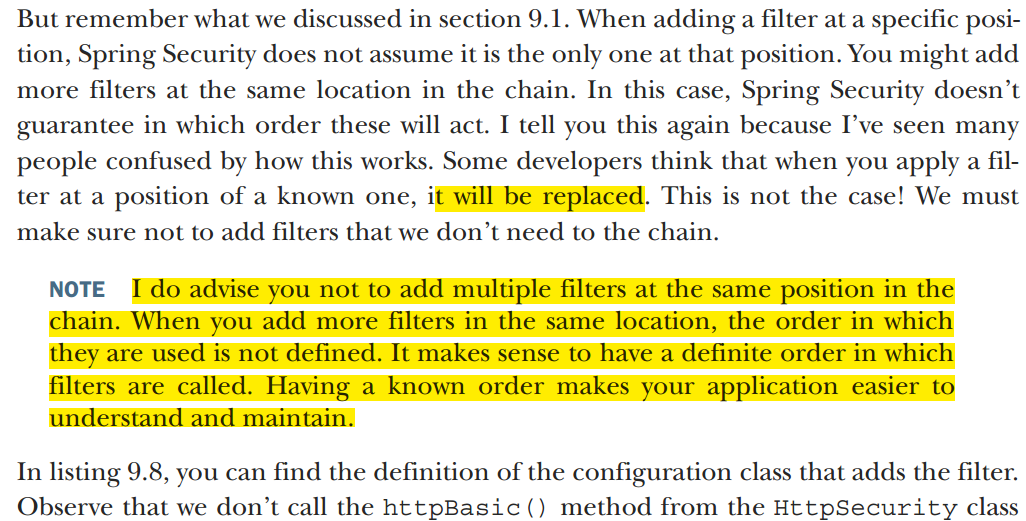
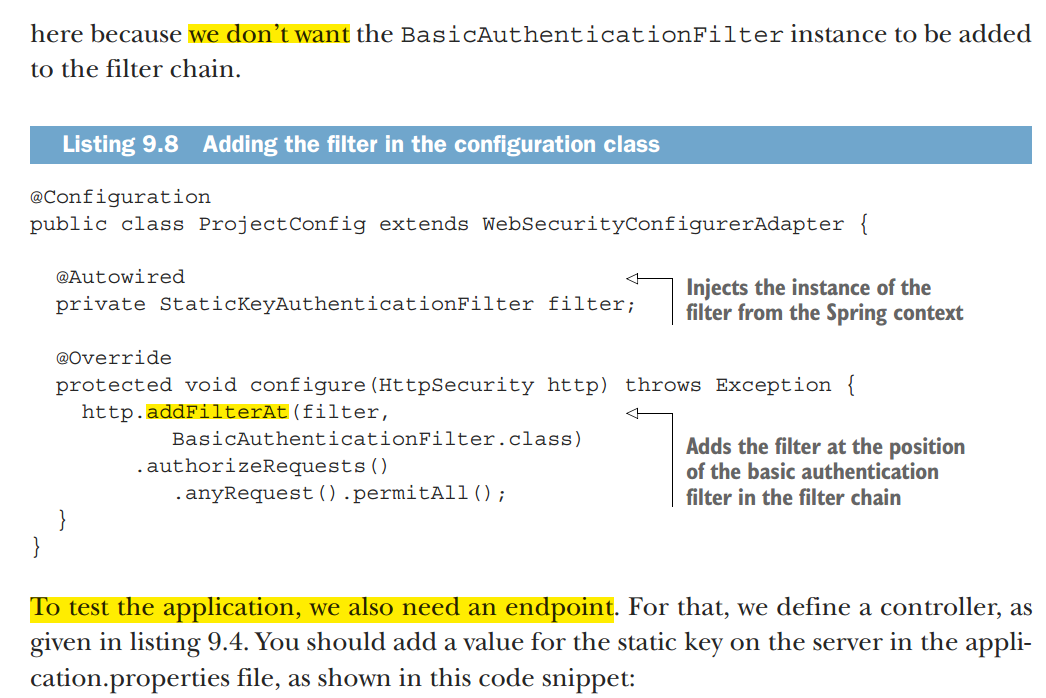
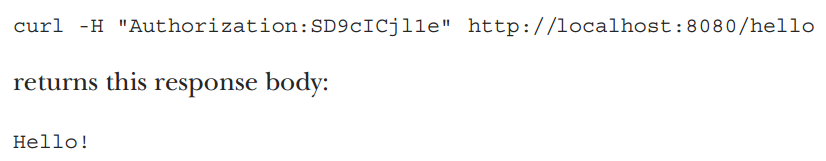
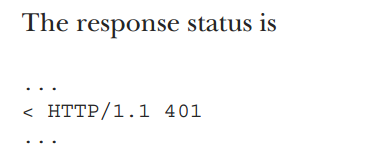
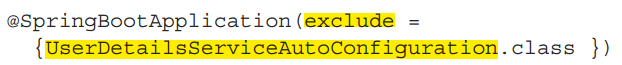
9.2 Adding a filter before an existing one in the chain

1. **Agenda**:
   1. How to add **HTTP custom filters** before an existing one in the filter chain.
2. **Requirement**:
   1. Implement filter checking for existence for Request-Id header and add it to Spring Security filter chain.
   2. Before application performs authentication, we want to check whether this header is present or not.
3. **See project** 🡺 ssia-ch9-ex1
4. 
5. 
6. 
7. 
8. 
9. 
10. **Two cases**:
    1. If Request-Id header is present then HTTP Status 200 Ok.
    2. Otherwise HTTP Status 400 Bad Request.
11. 

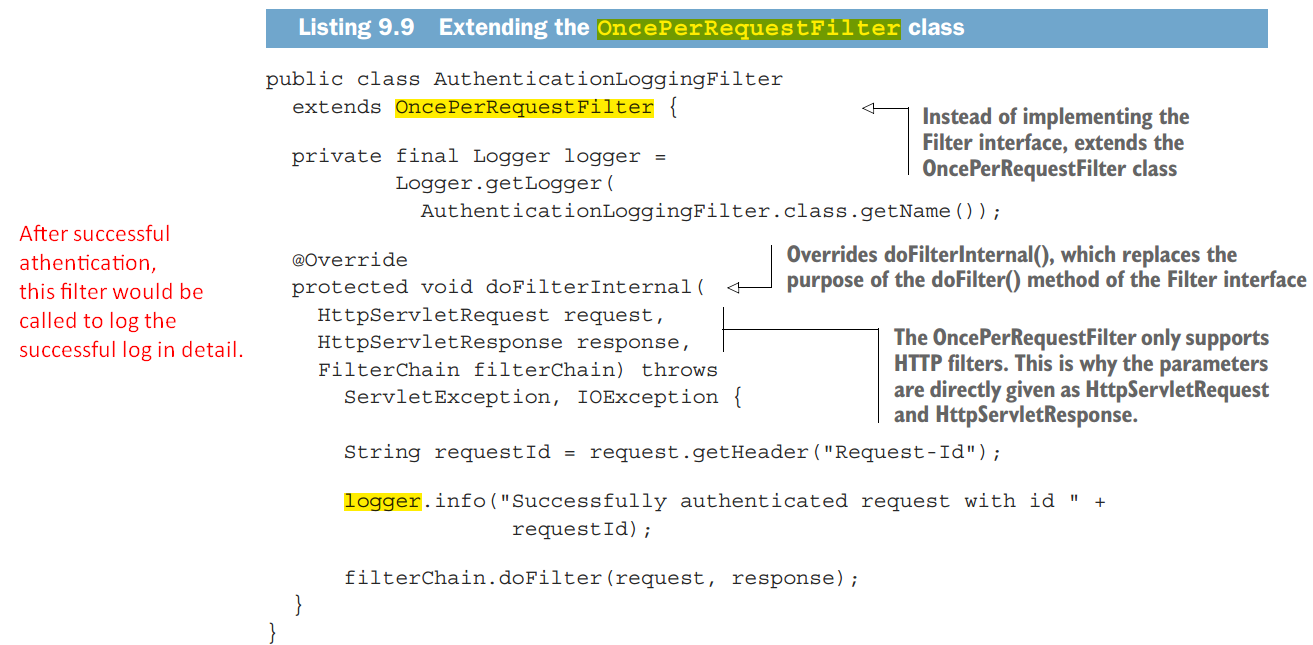
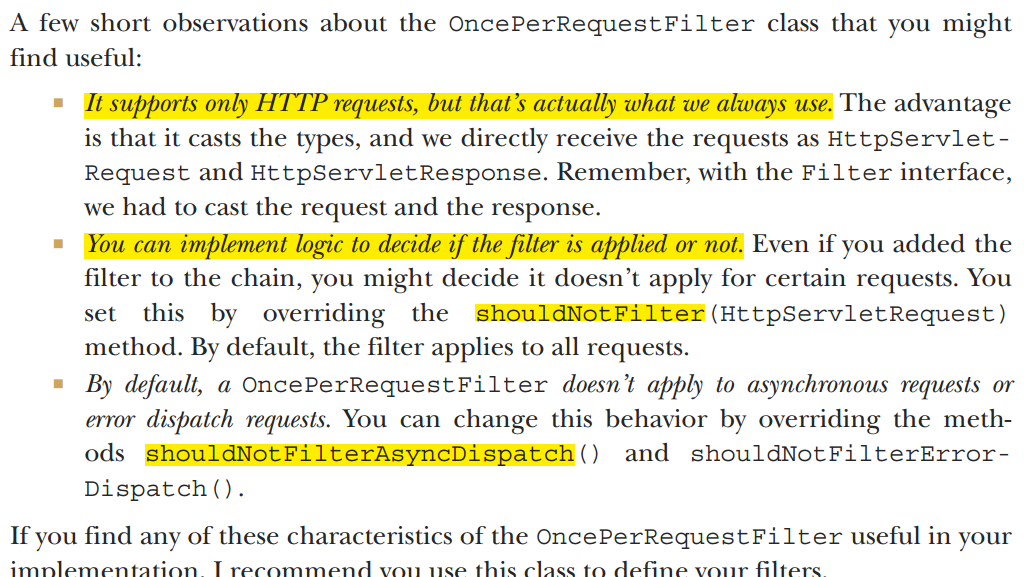
9.3 Adding a filter after an existing one in the chain

1. **Agenda**:
   1. Adding a filter after an existing one in the filter chain.
2. **Application:**
   1. When you want to execute some logic after something already existing in the filter chain.  
      Say, you have to execute some logic after the authentication process.  
      Examples for this could be notifying a different system after certain authentication events or logging and tracing purposes.
3. **Agenda:** **We will do**
   1. Log all successful **authentication events** by adding a filter after the authentication filter.
4. 
5. 
6. 
7. 

9.4 Adding a filter at the location of another in the chain

1. **Agenda**:
   1. How to add a custom filter in place of another filter in the Spring Security filter chain
2. **Application**:
   1. You can use this approach when providing a **different implement** for a responsibility that is already assumed by one of the filters known by Spring Security.  
      A typical scenario is authentication.
3. 
4. **Identification based on a static header value for authentication**:  
   
   1. **Procedure**:
      1. The client sends a string to the app in the header of HTTP request, which is always the same.  
         **NOTE**: App stores all these values already in some DB or secrets vault.
      2. Based on this static value, the app identifies the client.
   2. **Disadvantage**: This approach offers weak security related to authentication
   3. **Application**: Architects and developers often choose it in calls b/w backend applications for its simplicity and its fast execution as its implementation doesn’t do complex calculations.
   4. This way, **static keys** used for authentication represent a compromise where developers rely more on the infrastructure level in terms of security and also don’t leave the endpoints wholly unprotected.
5. **Using a Symmetric Key to sign the request for authentication**:  
   
   1. In this scenario using symmetric key to sign and validate requests, both client and server know the value of a key (client and server share the key).
   2. **Procedure**:
      1. Client uses this key to sign a part of the request (for example, to sign the value of specific headers).
      2. The server checks if the signature is valid using the same secrete key.  
         **NOTE**: Server stores individual keys for each client in DB or secrets vault.  
         **NOTE**: Similarly, you can use a pair of asymmetric keys.
6. **Using a one-time password (OTP) in the authentication process:  
   **
   1. User receives OTP in the authentication process via a message or by using an authentication provider app like **Google Authenticator**.
7. Let’s implement an example to demonstrate how to apply a custom filter.
8. **Scenario**:
   1. To be authenticated, the user must add the correct value of the static key in the Authorization Header as presented in figure 9.12.
9. 
10. We will start with implementing the filter, named **StaticKeyAuthenticationFilter**.
11. 
12. 
13. 
14. 
15. 
16.   
    **NOTE**: Use secrete vault to store such critical info.
17. 
18. 
19. 
20. Because we didn’t configure **UserDetailsService** so Spring Boot configures it automatically.   
    But we don’t need it as we don’t have the concepts of Users.  
    You can disable this **autoconfiguration**.  
    To disable the configuration of the default UserDetailsService, you can use the **exclude** attribute of the @**SpringBootApplication** annotation on the main class like this:  
    

9.5 Filter Implementations provided by Spring Security

1. **Agenda**:
   1. We will discuss classes provided by Spring Security which implement the Filter interface.
2. An idea how to create your own filter classes:
   1. Spring security offers a few **abstract classes** that implement the Filter interface and for which you can extend your filter definitions.
   2. These classes also add functionality your implementations could benefit from when you extend them.
   3. **For example**:
      1. Extending **GenericFilterBean** class which allows you to use initialization parameters that you would define in a web.xml descriptor file where applicable.  
         A more useful class that extends the **GenericFilterBean** is **OncePerRequestFilter**.   
         When adding a filter to the chain, the framework doesn’t guarantee it will be called only once per request.  
         **OncePerRequestFilter** as the name suggests, implements logic to make sure that the filter’s doFilter() method is executed only one time per request.
   4. If you don’t need any existing filter functionality then implement Filter interface.
3. **Where to use OncePerRequestFilter:**
   1. We want to avoid logging the same requests multiple times.
   2. See this project 🡺 ssia-ch9-ex3
   3. 
   4. 
   5. 