Spring Security is a powerful and highly customisable framework for implementing security in Java applications. It provides a comprehensive set of tools and components for handling various security concerns, such as authentication and authorisation. This article will delve into the core concepts of Spring Security, how it works, and how to use it effectively, drawing on examples and illustrations to enhance understanding.

Core Concepts of Spring Security

- Filters: At the heart of Spring Security is the concept of a filter chain. Filters are Java classes that intercept incoming requests and outgoing responses. They sit between the web container and the dispatcher servlet. These filters act as a series of components that process each request before it reaches your application's code, applying security at each stage.
  - Functionality: Filters can perform actions such as authentication, authorization, and logging.
  - Filter Chain: A collection of filters is called a filter chain.
  - Request Blocking: Any filter in the chain can block a request and return early, typically with an error such as unauthorised or forbidden.
  - Servlet Concept: It is worth noting that a filter is a web servlet container concept applicable to any application running in a web container such as Tomcat or JBoss.
- Authentication: This is the process of verifying a user's identity.
  - Authentication Filters: These filters are responsible for extracting credentials from requests and creating an authentication token. For example, the **`UsernamePasswordAuthenticationFilter`** processes login credentials from a standard HTML form, extracting the username and password to create an authentication token.
  - Authentication Provider: An `AuthenticationProvider` is responsible for authenticating a user using a specific authentication mechanism. Spring Security uses various `AuthenticationProvider` implementations to support different methods of verifying user credentials, such as database lookups, LDAP connections, or in-memory stores. Each provider handles a specific authentication mechanism, keeping the authentication logic modular.
  - Authentication Manager: The `AuthenticationManager` acts as an intermediary between authentication filters and providers. A common implementation is the **`ProviderManager`**, which iterates through a list of providers to find the one that supports the specific authentication method. The `ProviderManager` uses a basic for loop to iterate through all providers to find the one which it can use for the current authentication strategy.
  - UserDetailsService: The `UserDetailsService` is an interface with a single method, **`loadUserByUsername`**, used to load user details (including password)

  from a persistent data store, like a database. This allows Spring Security to work with custom user structures and data stores.
  - PasswordEncoder: A `PasswordEncoder` is used to hash passwords before storing them in a database. When a user attempts to log in, the clear text password is also hashed using the same encoder, and the resulting hash is then compared to the one stored. This ensures that passwords are not stored in plain text, which is a security risk.
- Authorization: This is the process of determining what resources an authenticated user is allowed to access.
- Security Context: The `SecurityContext` stores the authentication details of the current user, accessible during a request. The security context holds the details of the currently authenticated user.
  - Security Context Holder: The `SecurityContextHolder` provides a static way to access the current security context and user throughout the application without passing the context as a parameter.
- SecurityContextHolderFilter: This filter checks if a user is already authenticated in a session or token and attempts to load it. If it succeeds, the user does not need to re-authenticate, thus saving time by avoiding the full authentication process.
- ExceptionTranslationFilter: This filter is responsible for catching any exceptions thrown during authentication or authorisation. It converts these exceptions into appropriate HTTP error responses (like 401 or 403), which are then sent back to the user.

How Spring Security Works

1. Request Interception: When a client sends a request, it first goes through the filter chain.
2. Authentication:
   - An authentication filter extracts credentials from the request and creates an authentication token. For example, if a user submits a login form with a username and password, the `UsernamePasswordAuthenticationFilter` will extract these values and create a `UsernamePasswordAuthenticationToken`.
   - The authentication token is passed to the `AuthenticationManager`.
   - The `AuthenticationManager` selects an appropriate `AuthenticationProvider` using the provider's support method. For example, if the authentication token is a `UsernamePasswordAuthenticationToken`, the `ProviderManager` might select a `DaoAuthenticationProvider`.
   - The `AuthenticationProvider` uses the `UserDetailsService` to load user details and the `PasswordEncoder` to verify the password.
   - If authentication is successful, the `AuthenticationProvider` returns an authentication object.

3. **Security Context**: The authentication object is stored in the `SecurityContext`. The currently authenticated user is also referred to as a **principal**.
4. **Authorization**: Subsequent filters can use the `SecurityContext` to verify the user's access rights.
5. **Response**: The request is processed, and the response is sent back to the client.
6. **Subsequent Requests**: The `SecurityContextHolderFilter` checks if the user is already authenticated and if so, loads the user from the session or token, avoiding the need to re-authenticate.

Implementing Industry Standard Security

- **Component-Based Architecture**: Spring Security's component-based architecture allows developers to customise the framework by implementing only the required parts.
- **Multiple Authentication Mechanisms**: Spring Security supports multiple authentication methods. This includes login forms, basic authentication, and JWT. The framework provides multiple authentication filters for handling different ways of sending login credentials, including HTML forms, basic authentication headers, or a URL. Each filter extracts the credentials and creates an authentication token in a uniform way.
- **Password Hashing**: Using the `PasswordEncoder` is a key part of securing user passwords and adhering to security best practices.
- **Session Management**: Spring Security uses sessions to store user information, avoiding the need to re-authenticate on every request. It also supports stateless authentication with JWT.
- **Error Handling**: The `ExceptionTranslationFilter` ensures that authentication and authorization errors are handled gracefully.
- **Detailed Logging:** Spring Security provides detailed logging functionality, including a trace logging level, that shows what happens at each stage of the authentication process. This can be very useful for troubleshooting. By adding a trace logging level for a Spring Security package in the application.yaml file, the console will log each filter invoked during the authentication process.

Illustrative Examples

1. **Custom Filter Implementation:**
    - To create a custom filter, you need to annotate it with `@Component` so Spring can pick it up.
    - You will also need to implement the `jakarta.servlet.Filter` interface (or `javax.servlet.Filter` in older Spring versions).
    - The most important method to implement is the `doFilter` method, which takes a servlet request, a servlet response, and a filter chain.

- Within the `doFilter` method, you can perform your custom logic and then pass the request and response down the filter chain to other filters by calling `chain.doFilter(request, response)`.
- Alternatively you can extend the `OncePerRequestFilter` class, which provides a simpler interface with the `doFilterInternal` method.
- Here's a basic example of a filter that logs request headers using `OncePerRequestFilter`:

```
@Component
public class HeadersLoggingFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        Enumeration<String> headerNames =
request.getHeaderNames();
        List<String> headers = Collections.list(headerNames);
        headers.forEach(headerName -> {
            System.out.println("Header: " + headerName + ",
Value: " + request.getHeader(headerName));
        });
        filterChain.doFilter(request, response);
    }
}
```

- This filter logs all request headers and then calls `filterChain.doFilter()` to continue processing the request.

2. **Filter Ordering**
    - Filters can be ordered using the `@Order` annotation or by implementing the `Ordered` interface.
    - The `@Order` annotation is straightforward to use but has the downside that it cannot be controlled programmatically.
    - Implementing the `Ordered` interface allows the filter's order to be controlled programmatically using the `getOrder` method.

```
@Component
public class CustomHeaderFilter extends OncePerRequestFilter
implements Ordered {

    private final ApplicationContext applicationContext;
```

```
     public CustomHeaderFilter(ApplicationContext
applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        response.setHeader("X-Custom-Header", "Custom Value");
        filterChain.doFilter(request, response);
    }

    @Override
    public int getOrder() {
    if(applicationContext.containsBean("headersLoggingFilter")){
        return -1;
    }
    return Ordered.LOWEST_PRECEDENCE;
}
}
```

- A lower integer value places the filter higher in the chain (closer to the top) and vice versa. The `Ordered.HIGHEST_PRECEDENCE` constant represents the top of the chain, and `Ordered.LOWEST_PRECEDENCE` is the bottom.
- If two filters have the same order value, the order of execution is not guaranteed.
- An alternative to implementing the `Ordered` interface is to use a `FilterRegistrationBean` which allows you to externally control the properties of a filter such as setting the order of the filter.

```
@Bean
public FilterRegistrationBean<CustomHeaderFilter>
myCustomHeaderFilterRegistration(CustomHeaderFilter
customHeaderFilter, ApplicationContext applicationContext) {
   FilterRegistrationBean<CustomHeaderFilter> registrationBean =
new FilterRegistrationBean<>(customHeaderFilter);

  if (applicationContext.containsBean("headersLoggingFilter")) {
     registrationBean.setOrder(-1);
  } else {
     registrationBean.setOrder(Ordered.LOWEST_PRECEDENCE);
   }
```

```
    return registrationBean;
}
```

- The `FilterRegistrationBean` also allows you to specify the URL patterns that a filter should apply to.

Additional Notes

- The Jakarta namespace is used in Spring 3.0 and higher, while older versions of Spring use Java X.
- It is important to understand that a filter is a Java class that is executed for each incoming request and outgoing response. It sits between the web container and the dispatcher servlet.

By understanding these concepts and components, developers can leverage Spring Security to build secure and robust applications that adhere to industry standards. Spring Security's flexible architecture allows you to customise and extend the default behaviour to meet your specific security requirements.