Working with Threads Outline

- SecurityContext – I
- SecurityContext – II
- AuthenticationPrincipal
- Authentication
- Principal
- Processing Secure Methods Asynchronously
- AsyncConfigurerSupport
- AsyncConfigurer

SecurityContext - I

- Spring Security is fundamentally **thread-bound** because it needs to make the current **authenticated principal** available to a wide variety of downstream consumers.
- SecurityContext contains an Authentication.

- You can access and manipulate the SecurityContext through static convenience methods in SecurityContextHolder, which, in turn, manipulate a ThreadLocal.

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
assert(authentication.isAuthenticated);
```

- If you need access to the **currently authenticated user** in a web endpoint, you can use a method parameter in a @RequestMapping, annotated by @AuthenticationPrincipal.
  - pulls the current Authentication out of the SecurityContext and calls the getPrincipal() method on it to yield the method parameter.

```
@RequestMapping(“/foo”)
public String foo(@AuthenticationPrincipal User user) {
    // do stuff with user
}
```

SecurityContext - II

- The type of the Principal in an Authentication is dependent on the AuthenticationManager used to validate the authentication, so this can be a useful little trick to get a **type-safe** reference to user data.

- If Spring Security is in use, the Principal from the HttpServletRequest is of type Authentication, so you can also use that directly.

```
@RequestMapping(“/foo”)
public String foo(Principal principal) {
    Authentication authentication = (Authentication) principal;
    User = (User) authentication.getPrincipal();
    // do stuff with user
}
```

AuthenticationPrincipal

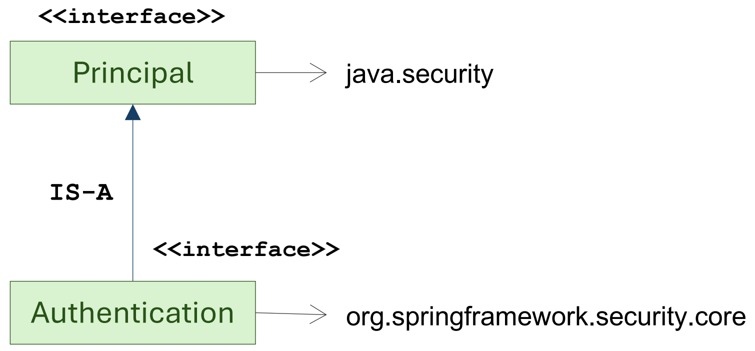
- Annotation that is used to resolve Authentication.**getPrincipal**() to a method argument.

```
package org.springframework.security.core.annotation;

@Target({ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface AuthenticationPrincipal {
    // True if a ClassCastException should be thrown when
    // the current Authentication.getPrincipal() is the incorrect type.
    boolean errorOnInvalidType() default false;
    String expression() default "";
}
```

Authentication

- Represents the **token** for an **authentication request** or for an **authenticated principal** once the request has been processed by the AuthenticationManager.**authenticate**(Authentication) method.
- Once the request has been authenticated, the Authentication will usually be stored in a **thread-local** SecurityContext managed by the SecurityContextHolder by the authentication mechanism which is being used.



```
package org.springframework.security.core;

public interface Authentication extends Principal, Serializable {
    // returns the Principal being authenticated or the authenticated principal after authentication.
    //
    // In the case of an authentication request with username and password, this would be the username.
    //
    // The AuthenticationManager implementation will often return an Authentication containing richer
    // information as the principal for use by the application.
    //
    // Many of the authentication providers will create a UserDetails object as the principal.
    Object getPrincipal();

    // other APIs were intentionally skipped

    Collection<? extends GrantedAuthority> getAuthorities();
    boolean isAuthenticated();
}
```

Principal

- Interface represents the abstract notion of a Principal, which can be used to represent any entity, such as an individual, a corporation, and a login id.

```
package java.security;

public interface Principal {
    boolean equals(Object another);
    String toString();
    int hashCode();
    String getName();
}
```

Processing Secure Methods Asynchronously

- Since the SecurityContext is **thread-bound**, if you want to do any background processing that calls secure methods, i.e., with @Async, you need to ensure that the context is propagated.
    - This boils down to **wrapping** the SecurityContext with the task (Runnable, Callable, and so on) that is executed in the background.
- To propagate the SecurityContext to @Async methods, you need to supply an AsyncConfigurer and ensure the Executor is of the correct type

```
@Configuration
public class ApplicationConfiguration extends AsyncConfigurerSupport {
    @Override
    public Executor getAsyncExecutor() {
        return new DelegatingSecurityContextExecutorService(Executors.newFixedThreadPool(5));
    }
}
```

AsyncConfigurerSupport

- A convenience AsyncConfigurer that implements all methods so that the defaults are used.
    - Provides a **backward compatible alternative** of implementing AsyncConfigurer directly.
- Deprecated
    - as of 6.0 in favor of implementing AsyncConfigurer directly.

```java
package org.springframework.scheduling.annotation;

@Deprecated(since = “6.0”)
public class AsyncConfigurerSupport implements AsyncConfigurer {
    @Override
    public Executor getAsyncExecutor() {
         return null;
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return null;
    }
}
```

AsyncConfigurer

- Interface to be implemented by classes annotated with @EnableAsync and @Configuration that wish to customize
  - the Executor instance used when **processing async method invocations** or
  - the AsyncUncaughtExceptionHandler instance used to process **exceptions thrown from async method** with void return type.

```java
package org.springframework.scheduling.annotation;

public interface AsyncConfigurer {

    default Executor getAsyncExecutor() {
        return null;
    }

    default AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return null;
    }
}
```