# Java EE 8 Security API 入門, 第 4 回: SecurityContext を使用して呼び出し側のデータを調査する

## サーブレット・コンテナーと EJB コンテナーをまたがってユーザー・アクセスを認証・許可する

Alex Theedom

2018年 10月 18日

Java EE Security API について解説するシリーズの最終回では、SecurityContext API を紹介します。この API を使用すると、サーブレット・コンテナーと EJB コンテナーとで一貫した方法によって、呼び出し側のデータを調査できます。SecurityContext で HttpAuthenticationMechanism の宣言型機能を拡張する方法を確認してから、サーブレット・コンテナーを例に、この API を使用して呼び出し側のデータを調査する方法を説明します。

**この連載について**
新しくリリースされた待望の Java EE Security API (JSR 375) によって、クラウドとマイクロサービス・コンピューティング時代に初めて Java エンタープライズ・セキュリティーが導入されます。このシリーズでは、この新しいセキュリティー・メカニズムがあらゆる Java EE コンテナーの実装をまたがってセキュリティー処理を単純化および標準化する仕組みを紹介した上で、クラウド対応のプロジェクト内で実際にこのセキュリティー・メカニズムを活用する方法を説明します。

このシリーズの前回の記事では、`IdentityStore` を紹介しました。IdentityStore は、Java Web アプリケーション内でユーザー資格情報へのセキュアなアクセスをセットアップして構成するために使用する、アイデンティティー・ストアの抽象化です。開発者は `IdentityStore` を `HttpAuthenticationMechanism` と組み合わせることで、強力な組み込み認証・許可メカニズムを実装できますが、セキュリティーのニーズによっては、`HttpAuthenticationMechanism` の宣言型セキュリティー・モデルでは対応しきれません。そこで必要となるのが、`SecurityContext` API です。

この記事では、`SecurityContext` を使用してプログラムによって HttpAuthenticationMechanism を拡張し、Web アプリケーションでアプリケーション・リソースへのアクセスを拒否または許可できるようにする方法を説明します。この記事で取り上げる例は、サーブレット・コンテナーに基づいていることに注意してください。

### コードを入手する

## Soteria をインストールする

`SecurityContext` インターフェースについて探るために、Java EE 8 Security API のリファレンス実装となっている Soteria を使用します。Soteria を取り込むには、以下に説明する 2 つの方法があります。

### 1. POM 内に明示的に Soteria を指定する

POM 内で以下の Maven コーディネートを使用して Soteria を指定します。

### リスト 1. Soteria プロジェクトの Maven コーディネート

```
<dependency>
  <groupId>org.glassfish.soteria</groupId>
  <artifactId>javax.security.enterprise</artifactId>
  <version>1.0</version>
</dependency>
```

### 2. 組み込み Java EE 8 コーディネートを使用する

Java EE 8 対応のサーバーは、新しい Java EE 8 Security API の独自の実装を使用するか、Sotoria の実装に依存することになります。いずれにしても、必要となるのは以下の Java EE 8 コーディネートだけです。

### リスト 2. Java EE 8 の Maven コーディネート

```
<dependency>
 <groupId>javax</groupId>
 <artifactId>javaee-api</artifactId>
 <version>8.0</version>
 <scope>provided</scope>
</dependency>
```

`SecurityContext` インターフェースは、`javax.security.enterprise` パッケージ内にあります。

## SecurityContext の機能

`SecurityContext` API が作成された目的は、サーブレット・コンテナーと EJB コンテナーの両方をまたがって一貫したアプリケーション・セキュリティー手法を提供するためです。セキュリティー・コンテキストが、現在認証対象となっているユーザーに関連付けられているセキュリティー関連の情報にアクセスできるようにします。これにより、プログラムによって Web ベースの認証プロセスの開始をトリガーすることができます。

サーブレット・コンテナーと EJB コンテナーでのセキュリティー・コンテキスト・オブジェクトの実装方法は類似していますが、違いもあります。例えば、サーブレット・コンテナー内でユーザーのアイデンティティーを取得するには、`HttpServletRequest` インスタンスに対して `getUserPrincipal()` メソッドを呼び出すことで、UserPrincipal オブジェクトを取得します。一方、EJB コンテナー内では同じ名前のメソッドを `EJBContext` インスタンスに対して呼び出します。同様に、ユーザーが特定の役割に属しているかどうかをテストする場合も、サーブレット・コンテナー内では `HttpServletRequest` インスタンスに対して `isUserRole()` メソッドを呼び出す一方、EJB コンテナー内では `EJBContext` インスタンスに対して `isCallerInRole()` メソッドを呼び出します。

こうした一貫性のなさを解消するために、新しい `SecurityContext` では、プログラムによって認証・許可情報を取得するメカニズムを、サーブレット・コンテナーと EJB コンテナーの両方をまたがって 1 つに統一しています。新たな Java EE 8 Security API 仕様では、Java EE 8 との互換性を持つサーブレット・コンテナーおよび EJB コンテナーの内部で `SecurityContext` を使用可能にすることを規定しています。一部のサーバー・ベンダーでは、他のコンテナー内でも `SecurityContext` を使用可能にすることが考えられます。

## SecurityContext の仕組み

SecurityContext インターフェースはプログラムによるセキュリティーのエントリー・ポイントであり、注入することが可能な型です。このインターフェースは 5 つのメソッドで構成されています。これらのメソッドには、いずれもデフォルトの実装がありません。

### 呼び出し側のデータを処理するメソッド

- **`getCallerPrincipal()`。**現在認証対象となっているユーザーの名前を表す、コンテナー固有のプリンシパルを取得するメソッドです。現在の呼び出し側が認証されていない場合、このメソッドは `null` を返します。返されるプリンシパルの型は、当初 `HttpAuthenticationMechanism` によって設定された型とは異なる場合があります。この `getCallerPrincipal()` メソッドと `EJBContext` インターフェース上の同じ名前のメソッドとの間には、重要な違いがあります。それは、認証されていないユーザーに対しては、このメソッドは名前を null に設定した `Principal` インスタンスを返すことです。
- **`getPrincipalsByType()`。**認証対象の呼び出し側の `Subject` から、指定された型のすべての `Principal` を返すメソッドです。指定された型が見つからない場合、または現在のユーザーが認証されていない場合は、空の `Set` を返します。このメソッドを使用することになるのは、コンテナー呼び出し側のプリンシパルがアプリケーション呼び出し側のプリンシパルと異なる場合や、アプリケーションに、アプリケーション呼び出し側のプリンシパルからでないと入手できない情報が必要な場合です。
- **`isCallerInRole()`。**呼び出し側が String として渡された役割に属しているかどうかを判別するメソッドです。該当する役割がユーザーに割り当てられている場合は `true` を返し、そうでなければ `false` を返します。このメソッドを呼び出して返される結果は、コンテナー固有の呼び出しを行ったとしても同じです。`SecurityContext.isUserInRole()` が `true` を返す場合、`HttpServletRequest.isUserInRole()` または `EJBContext.isCallerInRole()` を呼び出すと true が返されます。

### その他のメソッド

- **`hasAccessToWebResource()`。**現在のアプリケーション内で、指定された HTTP メソッドの特定の Web リソースに呼び出し側がアクセスできるかどうかを判別するメソッドです。アプリケーションのセキュリティー・コンテナー内では、このメソッドが Servlet 4.0 のセキュリティー制約に関する仕様に従って構成されます。
- **`authenticate()`。**プログラムによってコンテナーをトリガーして、クライアントがリソースにアクセスするための呼び出しを行ったかのように、呼び出し側との HTTP ベースの認証カンバセーションを開始または続行させるメソッドです。このメソッドは `HttpServletRequest` インスタンスおよび `HttpServletResponse` インスタンスを必要とすることから、有効なサーブレット・コンテキストに依存します。このメソッドが機能するのは、サーブレット・コンテナー内のみです。

メソッドとその機能の概要を把握したところで、いくつかのサンプル・コードを見ていきましょう。以降で取り上げる例は、Servlet 4.0 Web アプリケーション内での `SecurityContext` のメソッドに関するものです。

## 例 1: サーブレット内で呼び出し側のデータをテストする

### SecurityContext の getCallerPrincipal()、getPrincipalsByType()、isCallerInRole() メソッド

リスト 3 では、呼び出し側のデータをテストする `SecurityContext` の 3 つのメソッドの使い方をデモするために、これらのメソッドを 1 つのサーブレットに結合しています。以下の例では `SecurityContext` を CDI として使用できるため、コンテキストを認識するインスタンスであれば、どのインスタンスにでも SecurityContext を注入できます。

### リスト 3. サーブレット・コンテナー内での呼び出し側データに関するメソッドの例

```
@WebServlet("/securityContextServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class SecurityContextServlet extends HttpServlet {

   @Inject
   private SecurityContext securityContext;

   @Override
   public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                              throws IOException {

       // Example 1: Is the caller is one of the three roles: admin, user and demo
       PrintWriter pw = response.getWriter();

       boolean role = securityContext.isCallerInRole("admin");
       pw.write("User has role 'admin': " + role + "\n");

       role = securityContext.isCallerInRole("user");
       pw.write("User has role 'user': " + role + "\n");

       role = securityContext.isCallerInRole("demo");
       pw.write("User has role 'demo': " + role + "\n");


       // Example 2: What is the caller principal name
       String contextName = null;
       if (securityContext.getCallerPrincipal() != null) {
           contextName = securityContext.getCallerPrincipal().getName();
       }
       response.getWriter().write("context username: " + contextName + "\n");

       // Example 3: Retrieve all CustomPrincipal
       Set<CustomPrincipal> customPrincipals = securityContext
                                       .getPrincipalsByType(CustomPrincipal.class);
       for (CustomPrincipal customPrincipal : customPrincipals) {
           response.getWriter().write((customPrincipal.getName()));
       }

   }

}
```

最初の例ではセキュリティー・コンテキストを使用して、現在認証対象となっているユーザーが属する論理的役割をテストしています。テストする役割は、admin、user、demo です。

2 番目の例では、`getCallerPrincipal()` メソッドを使用して、認証対象のユーザーの名前を表す、プラットフォーム固有の呼び出し側プリンシパルを取得する方法を示しています。このメソッドは、現在のユーザーが認証されていない場合は `null` を返します。したがって、適切な `null` チェックを行う必要があります。

最後の例では、`getPrincipalsByType()` メソッドを使用して型別にプリンシパルのセットを取得する方法を説明しています。

リスト 4 に、`Principal` インターフェースを実装するカスタム・プリンシパルを記載します。`getPrincipalsByType()` メソッドを呼び出すと、指定した型のプリンシパルのセットが返されます。

### リスト 4. カスタム・プリンシパル

```
public class CustomPrincipal implements Principal {

   private final String name;

   public CustomPrincipal(String name) {
       this.name = name;
   }

   @Override
   public String getName() {
       return name;
   }
}
```

## 例 2: Web リソースに対する呼び出し側のアクセス権をテストする

### SecurityContext の hasAccessToWebResources() メソッド

リスト 5 に、`hasAccessToWebResource()` を使用して、呼び出し側が、指定した HTTP メソッドの特定のリソースに対するアクセス権を持っているかどうかをテストする方法を示します。この例では、`SecurityContext` インスタンスをサーブレットに注入して `hasAccessToWebResource()` を呼び出しています。ここでテストするのは、URI `/secretServlet` にあるリソースに対して呼び出し側が `GET` アクセス権を持っているかどうかです (リスト 6 を参照)。したがって、以下に示されている引数をメソッドに渡します。呼び出し側に admin 役割が割り当てられている場合、このメソッドは `true` を返します。そうでなければ `false` を返します。

### リスト 5. SecurityContext の hasAccessToWebResource()

```
@WebServlet("/hasAccessServlet")
public class HasAccessServlet extends HttpServlet {

   @Inject
   private SecurityContext securityContext;

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse res)
                                        throws ServletException, IOException {

       boolean hasAccess = securityContext.hasAccessToWebResource("/secretServlet", "GET");

   }
}
```

### リスト 6. アクセス権をテストするリソース

```
@WebServlet("/secretServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
public class SecretServlet extends HttpServlet { }
```

## 例 3: 呼び出し側のアクセスを認証する

### SecurityContext の authenticate() メソッド

最後の例では、`authenticate()` メソッドを使用して、ユーザーが入力した資格情報を検証する方法を説明します。まず、ユーザーが JSE にユーザー名とパスワードを入力します (リスト 7 を参照)。このログイン・フォームが送信されると、`LoginBean` がその資格情報を処理して認証します (リスト 8 を参照)。

### リスト 7. ログイン・フォーム

```
<form jsf:id="form">
 <p>
   <strong>Username </strong>
   <input jsf:id="username" type="text"    jsf:value="#{loginBean.username}" />
 </p>
 <p>
   <strong>Password </strong>
   <input jsf:id="password" type="password" jsf:value="#{loginBean.password}" />
 </p>
 <p>
   <input type="submit" value="Login" jsf:action="#{loginBean.login}" />
 </p>
</form>
```

Credential インスタンスを生成するために、入力された `username` と `password` を `LoginBean` 上に設定します (リスト 8 を参照)。この資格情報を使用して、`AuthenticationParameters` インスタンスを生成します。このインスタンスを、`FacesContext` から取得した `HttpServletRequest` および `HttpServletResponse` と併せて `authenticate()` メソッドに渡します。これにより、`AuthenticationParameters` が `AuthenticationStatus` 列挙体の値を返します。

`AuthenticationStatus` 列挙体は認証プロセスのステータスを示します。値は以下のいずれかになります。

- **`NOT_DONE`**: 認証メカニズムは呼び出されたものの、認証メカニズムが認証を行わないと決定したことを意味します。通常、このステータスはプリエンプティブ・セキュリティーの使用時に返されます。
- **`SEND_CONTINUE`**: 認証メカニズムが呼び出され、呼び出し側との多段階認証カンバセーションが開始されたことを意味します。
- **`SUCCESS`**: 認証メカニズムが呼び出され、呼び出し側が正常に認証されたことを意味します。したがって、呼び出し側のプリンシパルを使用できます。
- **`SEND_FAILURE`**: 認証メカニズムは呼び出されたものの、呼び出し側が正常に認証されなかったため、呼び出し側のプリンシパルを使用できないことを意味します。

EE 8 の `FacesContext` は、JSF 2.3 で注入可能になりました。

### リスト 8. ログイン Bean による呼び出し側資格情報の処理と認証

```
@Named
@RequestScoped
@FacesConfig(version = JSF_2_3)
public class LoginBean {

   @Inject
   private SecurityContext securityContext;

   @Inject
   private FacesContext facesContext;

   private String username, password;

   public void login() {

       Credential credential = new UsernamePasswordCredential(username, new Password(password));

       AuthenticationStatus status = securityContext.authenticate(
           getRequestFrom(facesContext),
           getResponseFrom(facesContext),
           withParams().credential(credential));

       if (status.equals(SEND_CONTINUE)) {
           facesContext.responseComplete();
       } else if (status.equals(SEND_FAILURE)) {
           addError(facesContext, "Authentication failed");
       }

   }

   private static HttpServletResponse getResponseFrom(FacesContext context) {
       return (HttpServletResponse) context
           .getExternalContext()
           .getResponse();
   }

   private static HttpServletRequest getRequestFrom(FacesContext context) {
       return (HttpServletRequest) context
           .getExternalContext()
           .getRequest();
   }


   // Getter and setters omitted
}
```

## まとめ

このシリーズでは、新しい Java EE 8 Security API が一般的なエンタープライズ認証・許可ルーチンに最もよく使われている、確かな Java EE テクノロジーのいくつかを統合するしくみを説明してきました。なかでも Java 開発者コミュニティーで長年求められていたのはサーブレット・コンテナーと EJB コンテナーとで一貫した単純なセキュリティー・モデルですが、`SecurityContext` はまさにこのモデルを実現しています。

この記事で取り上げた例はサーブレット・コンテナーに基づくものですが、`SecurityContext` では呼び出し側プリンシパルをサーブレット・コンテナーでも EJB コンテナーでも一貫した方法で容易に調査できるようになっています。最新の Java EE アプリに対応するために XML とアノテーション・ベースの構成を結合しなければならなかった開発者は、純粋なアノテーション・フレー

ムワークへの移行を歓迎するはずです。新しい Security API では XML 宣言もサポートしているため、セキュリティー構成を早急に変更する必要はありません。したがって、古いプロジェクトでも比較的簡単に手間をかけずに Java EE 8 に移行できます。

このシリーズを楽しんでいただけたこと、そして新しく得た知識を実際に活用できるようになっていただけたことを願います。以下の最終クイズで理解度をテストしてください。

## 学んだ知識をテストしてください

1. 次のメソッドのうち、`SecurityContext` インターフェースに属しているものはどれですか？
    a. `getCallerPrincipal()`
    b. `isUserRole()`
    c. `getPrincipalsByType()`
    d. `isCallerInRole()`
    e. `isCallerPrincipal()`
2. 次のうち、`hasAccessToWebResource()` メソッドがテストする内容はどれですか？
    a. 指定されたユーザーに、指定されたリソースに対するアクセス権があるかどうか
    b. サーブレットにリソースに対するアクセス権があるかどうか
    c. 呼び出し側に、指定されたリソースに対するアクセス権があるかどうか
    d. 呼び出し側に、リモート Web リソースに対するアクセス権があるかどうか
3. 次のうち、`getPrincipalsByType()` メソッドから返されるものはどれですか？
    a. 呼び出し側の `Subject` に含まれる特定の型の `Principal` のセット
    b. コンテキストに含まれる特定の型の `Principal`
    c. 特定の型の `Principal` のリスト
    d. 呼び出し側が許可されていない場合、`null`
    e. 呼び出し側が許可されていない場合、空のセット
4. 次のうち、`getCallerPrincipal()` メソッドの動作はどれですか？
    a. 認証対象の呼び出し側の名前を返す
    b. 現在の呼び出し側が認証されていない場合、`null` を返す
    c. 呼び出し側の `Principal` のセットを返す
    d. 認証対象の呼び出し側のプラットフォーム固有の `Principal` を返す

このリンク先のページで正解を確認してください。

## 関連トピック

- Java EE Security API 仕様
- Java EE Security API 仕様の GitHub ページ
- Soteria 参照実装
- Java EE Security API 実装
- Devoxx 2017 での、新しい Java Security API の暫定版のプレゼンテーション
- Alex の著書: Java EE 8: Only What's New