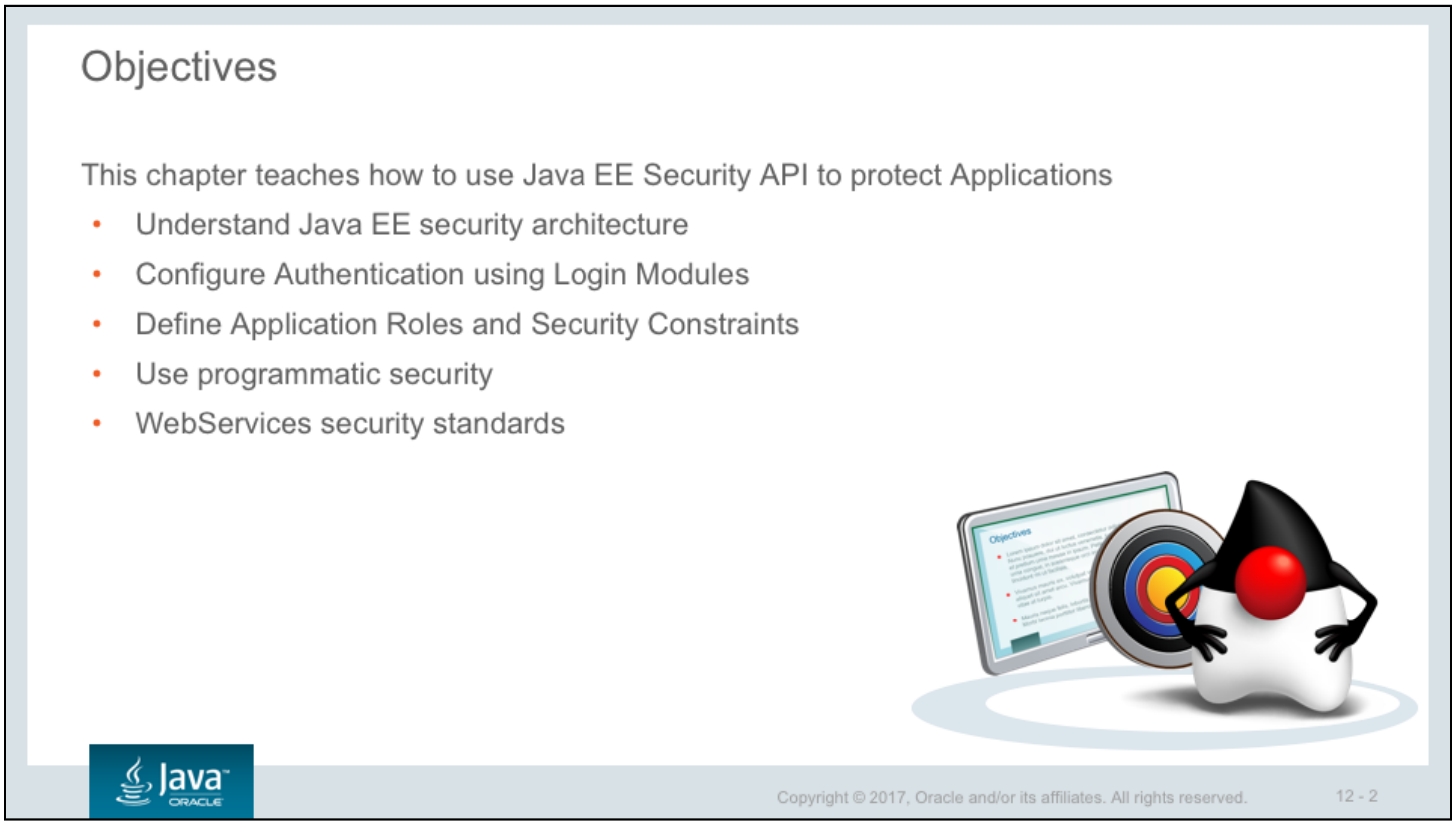
Securing Java EE Applications

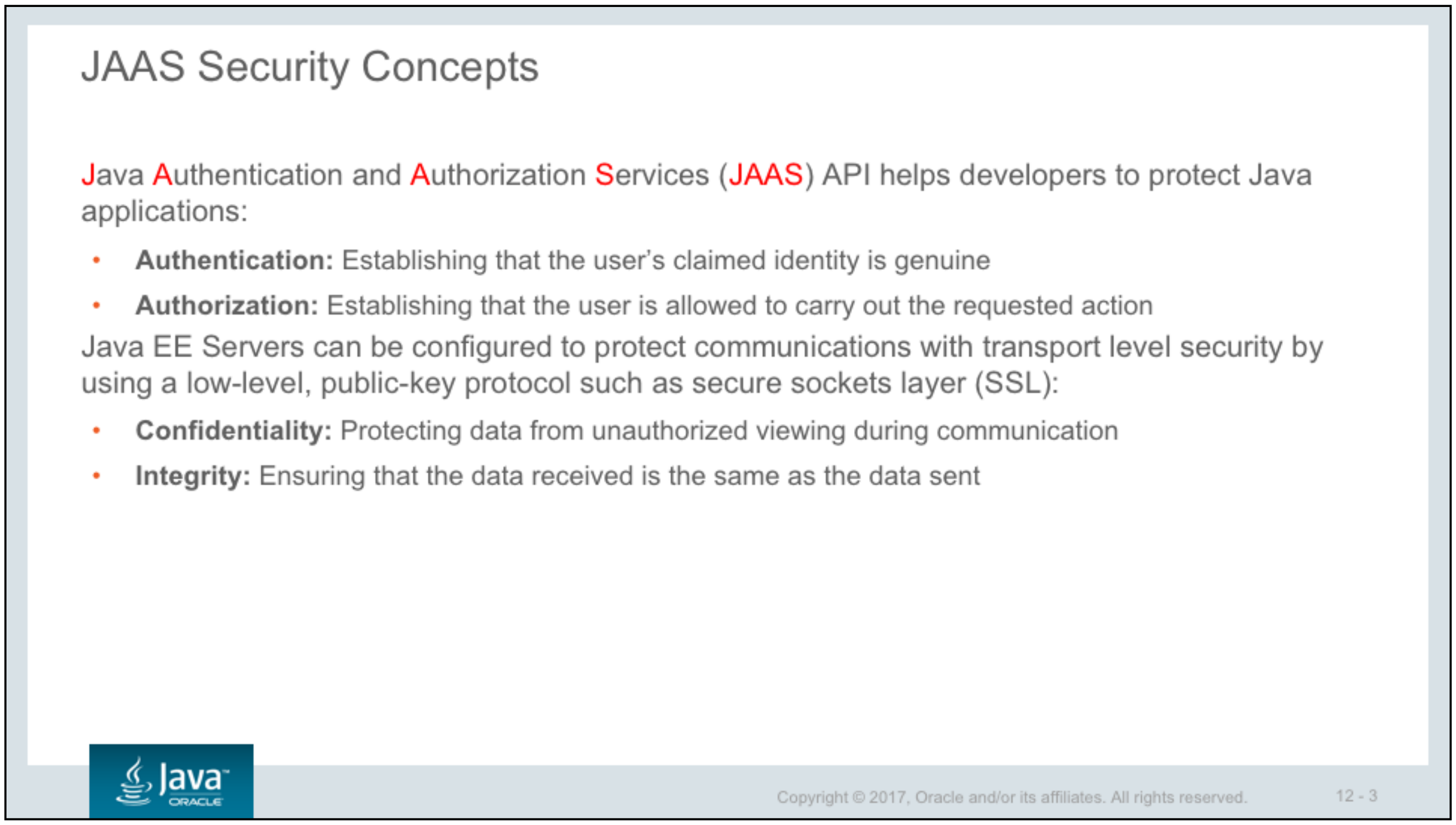
12



Objectives

This chapter teaches how to use Java EE Security API to protect Applications

- Understand Java EE security architecture
- Configure Authentication using Login Modules
- Define Application Roles and Security Constraints
- Use programmatic security
- WebServices security standards



JAAS Security Concepts

Java Authentication and Authorization Services (JAAS) API helps developers to protect Java applications:

- **Authentication:** Establishing that the user's claimed identity is genuine
- **Authorization:** Establishing that the user is allowed to carry out the requested action

Java EE Servers can be configured to protect communications with transport level security by using a low-level, public-key protocol such as secure sockets layer (SSL):

- **Confidentiality:** Protecting data from unauthorized viewing during communication
- **Integrity:** Ensuring that the data received is the same as the data sent

12 - 3

Every application that is accessible over the web or in business-to-business environments must consider security. Your application must be protected from attack; the private data of your site's users must be kept confidential; and your services must also protect the clients and other servers.

Unfortunately, these considerations are never complete and security must always be viewed as a level of protection, not absolute protection. Recognizing this, system administration should include monitoring, so that when an attack occurs, it is noticed and can be stopped or the system disconnected before any major damage occurs.

Because of the sheer complexity, scope, and importance of the security problem, consider using specialist security experts in the creation of enterprise applications. However, each team member should also have as much understanding of major issues as is practical.

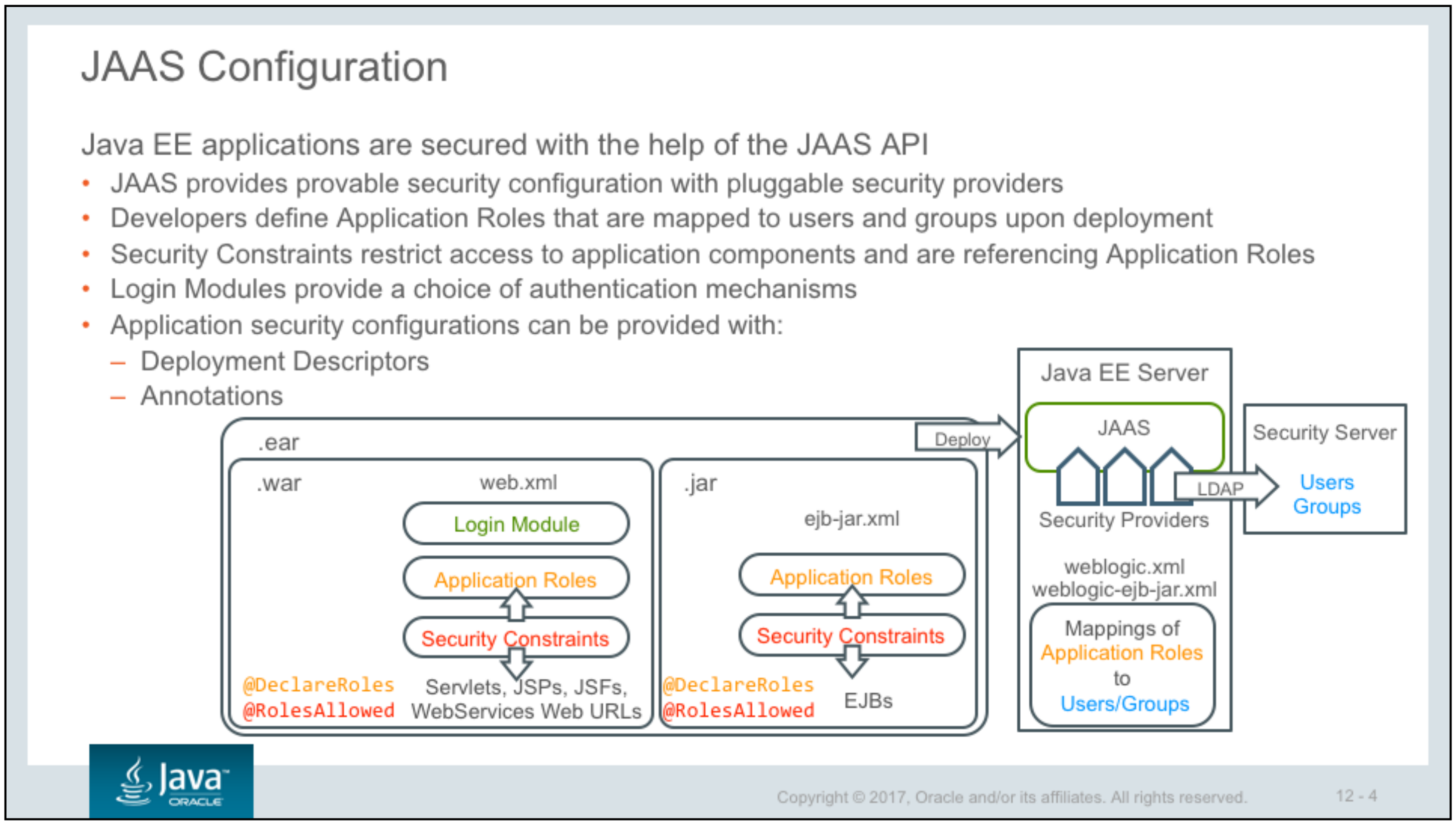
Authentication can be as simple as prompting for a username and password. More complex systems might make use of a public-key infrastructure to manage the certificate exchange. In the Java EE model, authentication is carried out by containers and not by components.

Normally, it is necessary to authenticate a user before authorization can be checked, although an unauthenticated user might be allowed to carry out certain operations. The Java EE model primarily uses declarative authorization.

For confidentiality, the normal approach is to use encryption that is usually based on public-key techniques.

Integrity is normally achieved by signing the data. That is, the security model encrypts a checksum of the data and adds the encrypted checksum to the message to ensure that the message has not been modified during transmission.

SSL is now formally known as transport layer security (TLS). The use of TLS for component interactions is frequently standard in Java EE applications. Configuration of the transport is part of the administration of the application server and is not normally visible to the application developer.



JAAS Configuration

Java EE applications are secured with the help of the JAAS API

- JAAS provides provable security configuration with pluggable security providers
- Developers define Application Roles that are mapped to users and groups upon deployment
- Security Constraints restrict access to application components and are referencing Application Roles
- Login Modules provide a choice of authentication mechanisms
- Application security configurations can be provided with:
  – Deployment Descriptors
  – Annotations

12 - 4

Java EE security contains no reference to the real security infrastructure, so that the application may be portable. JAAS (Java Authentication and Authorization Service) maps the application security policies to the security domain of the deployment environment. Thus, platform-independence is essentially preserved even though it becomes deployment-specific. JAAS implementations can provide for simple username-password pairs that are stored in a database, up to more complex systems such as Kerberos or Active Directory. In any case, the verification interface is standardized, although the means of adding new users is not. JAAS provides end-to-end security, so that security credentials are gathered in one part of the system, but are available to all parts. For example, if the user is using a web browser to interact with a servlet-based application and that servlet application calls enterprise beans, all the components (servlets and enterprise beans) that are invoked in a particular user interaction see the same user.

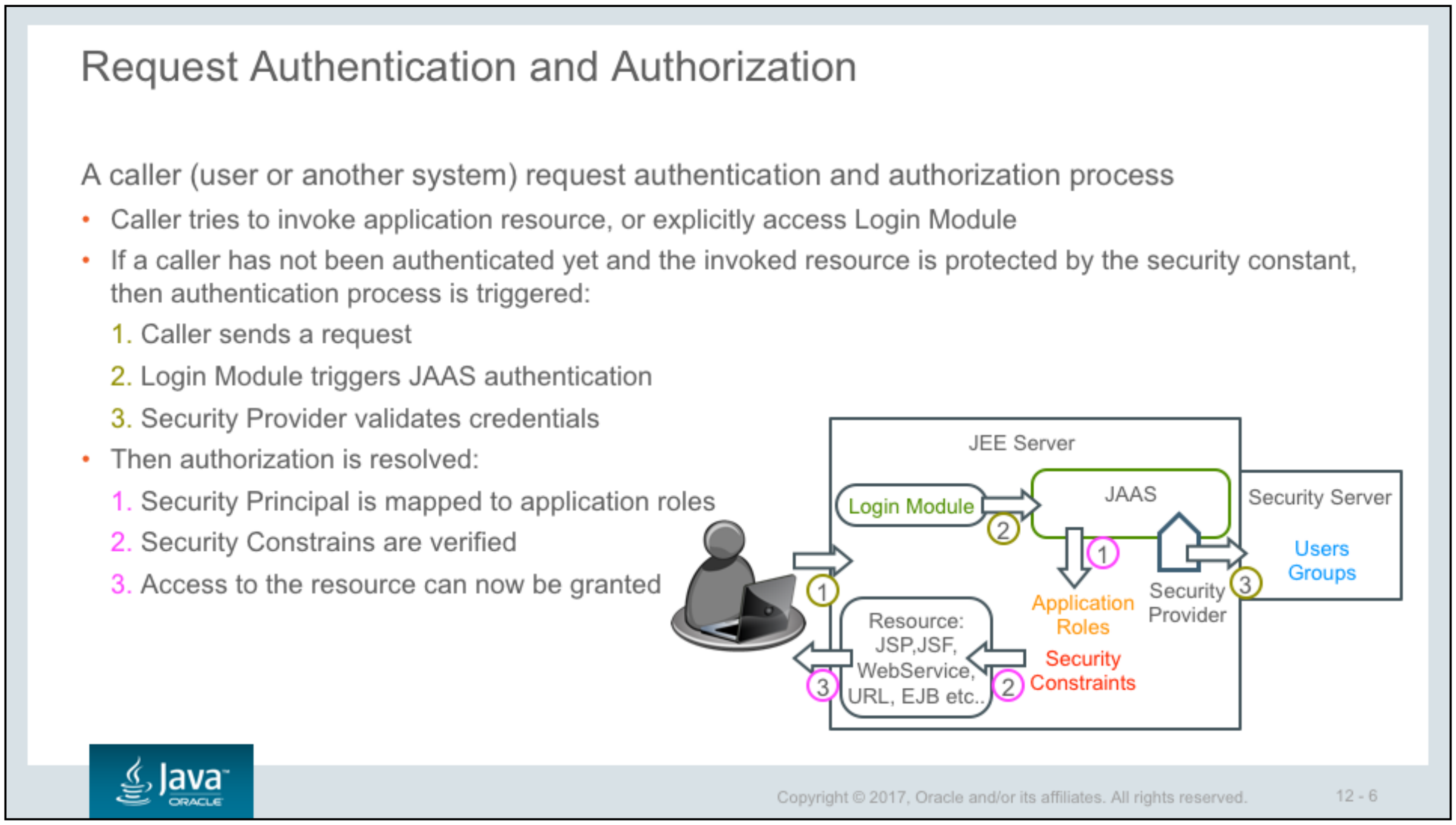
This portable configuration is achieved with following steps:

- Java EE server administrators configure security policy domains based on pluggable security service provider architecture, which enables various security providers, such as LDAP servers.
- Security Server administrators manage information about users and groups.
- Java application developers define Application Roles and describe security constrains that reference these Application Roles.
- Upon deployment Application Roles are mapped to actual users and groups defined by the security provider configured with security policy domain.
- Application developers select a Login Module, which defines a mechanism that allows to challenge the invoker to produce authentication.

To access an application, a user or an application may have to respond to the authentication challenge made by the Login Module.

- Security Principal is a user or system that is authenticated with such society provider.
- Once Security Principal is established it can then be authorized to use application resources based on the mappings between this security principal and application roles that are referenced by application security constraints.
- Unauthenticated invoker may also be granted access to application resources, if security constraints allow that.

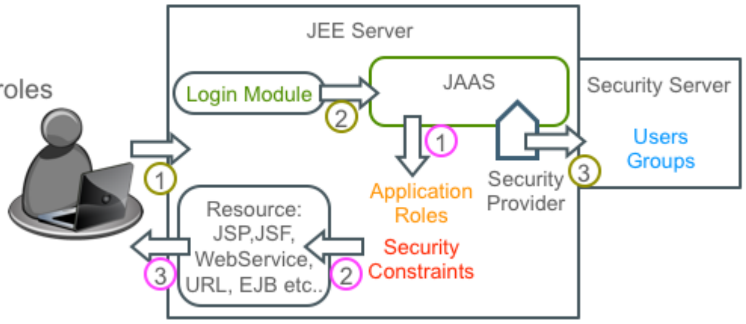
In addition to configuring authentication and authorization declaratively, developer can also use programmatic authorization controls provisioned by JAAS API.



Request Authentication and Authorization

A caller (user or another system) request authentication and authorization process

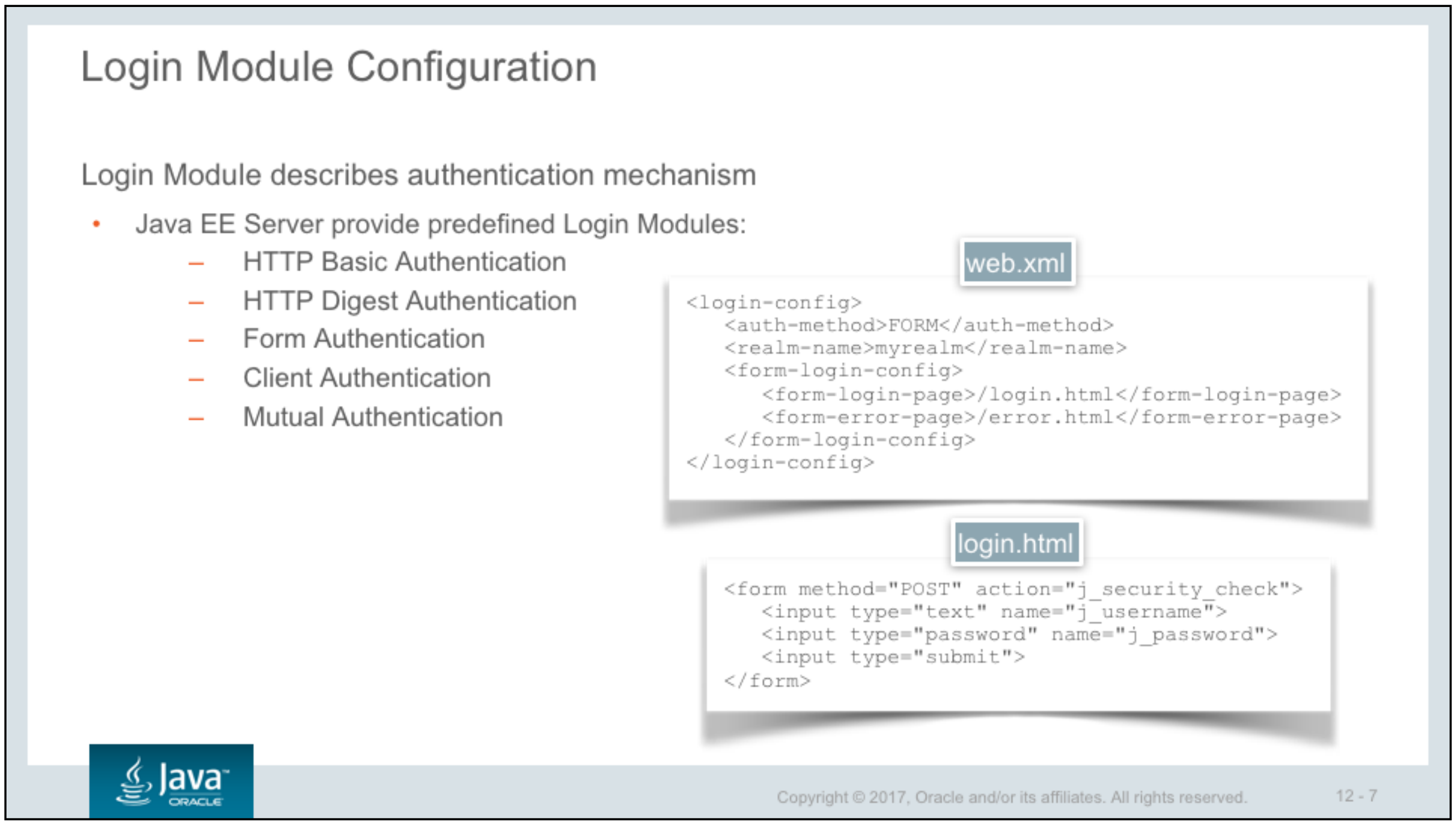
- Caller tries to invoke application resource, or explicitly access Login Module
- If a caller has not been authenticated yet and the invoked resource is protected by the security constant, then authentication process is triggered:
  1. Caller sends a request
  2. Login Module triggers JAAS authentication
  3. Security Provider validates credentials
- Then authorization is resolved:
  1. Security Principal is mapped to application roles
  2. Security Constrains are verified
  3. Access to the resource can now be granted



12 - 6

Caller may access Login Module directly, or invocation of the Login Module can be triggered by the server when caller tries to access a resource protected with security constraints.

Unsuccessful authentication will prevent caller from accessing resources. Even if a caller has been successfully authenticated access to a resource can still be denied if the established Security Principal has no permissions to access resources protected by security constrains.



Login Module Configuration

Login Module describes authentication mechanism

- Java EE Server provide predefined Login Modules:
  - HTTP Basic Authentication
  - HTTP Digest Authentication
  - Form Authentication
  - Client Authentication
  - Mutual Authentication

web.xml

```
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>myrealm</realm-name>
   <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/error.html</form-error-page>
   </form-login-config>
</login-config>
```

login.html

```
<form method="POST" action="j_security_check">
   <input type="text" name="j_username">
   <input type="password" name="j_password">
   <input type="submit">
</form>
```

HTTP basic authentication requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a configured security provider. Basic authentication is the default when you do not specify an authentication mechanism.

- A client requests access to a protected resource.
- The web server returns a dialog box that requests the user name and password.
- The client submits the user name and password to the server.
- The server authenticates the user in the specified realm and, if successful, returns the requested resource.

Like basic authentication, digest authentication authenticates a user based on a user name and a password. However, unlike basic authentication, digest authentication does not send user passwords over the network. Instead, the client sends a one-way cryptographic hash of the password and additional data. Although passwords are not sent on the wire, digest authentication requires that clear-text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

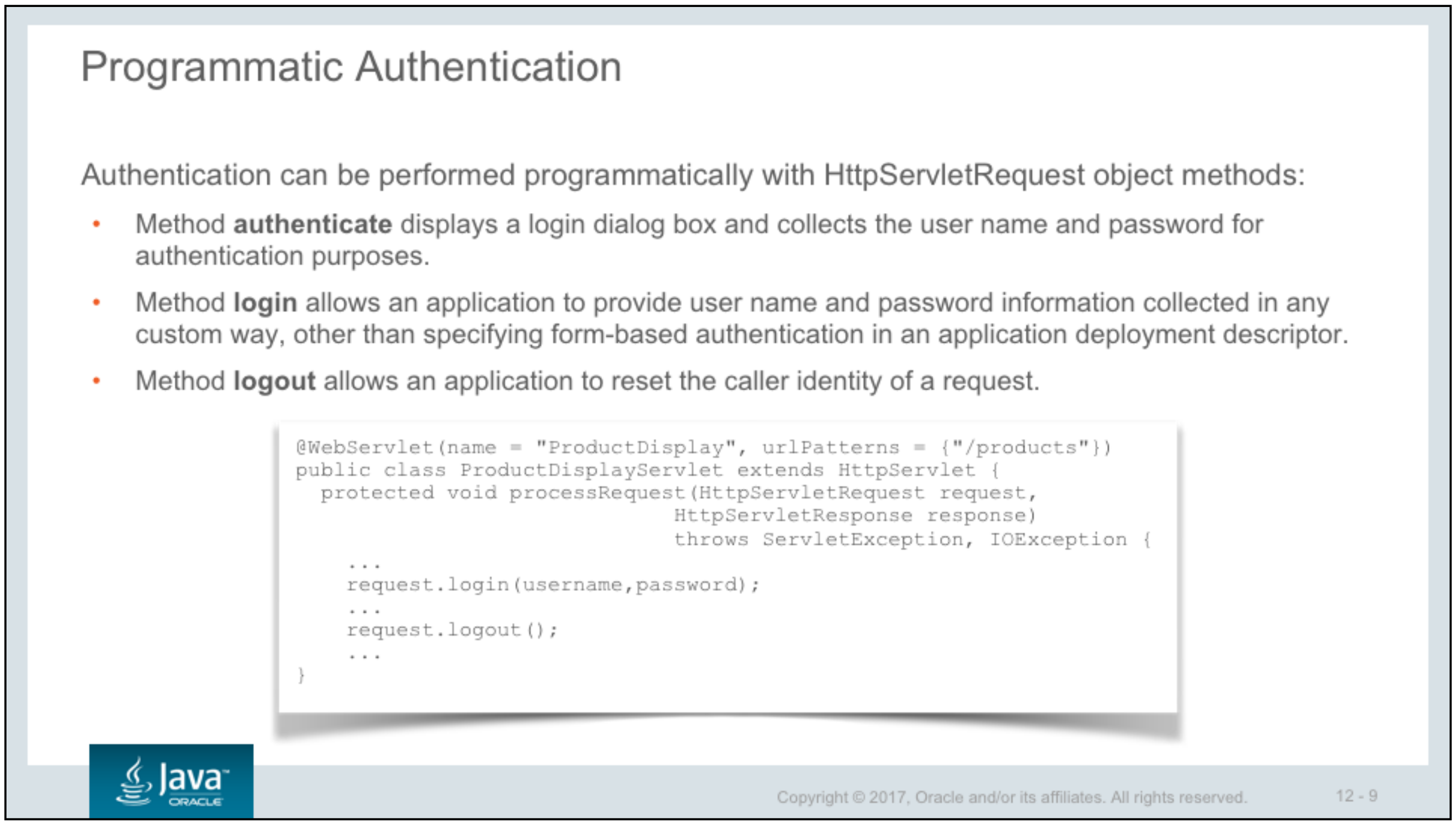
Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur.

- A client requests access to a protected resource.
- If the client is unauthenticated, the server redirects the client to a login page.
- The client submits the login form to the server.
- The server attempts to authenticate the user.
    - If authentication succeeds, the authenticated user's principal is checked to ensure that it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource by using the stored URL path.
    - If authentication fails, the client is forwarded or redirected to an error page.

With client authentication, the web server authenticates the client by using the client's public key certificate. Client authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's public key certificate. SSL technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. The certificate is issued by a trusted organization, a certificate authority (CA), and provides identification for the bearer.

With mutual authentication, the server and the client authenticate each other. Mutual authentication is of two types:

- Certificate-based:
    - A client requests access to a protected resource.
    - The web server presents its certificate to the client.
    - The client verifies the server's certificate.
    - If successful, the client sends its certificate to the server.
    - The server verifies the client's credentials.
    - If successful, the server grants access to the protected resource requested by the client.
- User name/password-based:
    - A client requests access to a protected resource.
    - The web server presents its certificate to the client.
    - The client verifies the server's certificate.
    - If successful, the client sends its user name and password to the server.
    - The server verifies the client's credentials
    - If the verification is successful, the server grants access to the protected resource requested by the client.



Programmatic Authentication

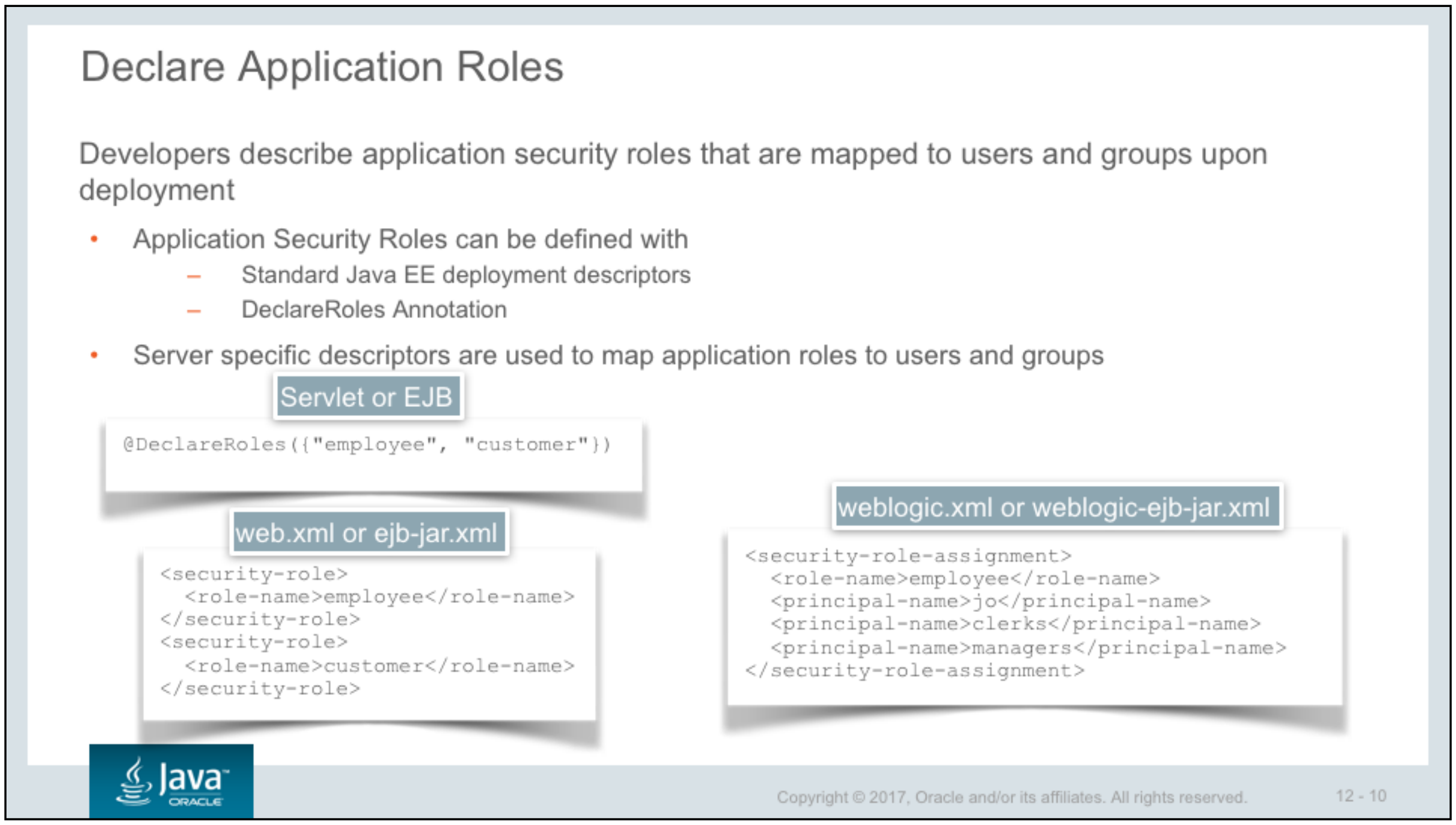
Authentication can be performed programmatically with HttpServletRequest object methods:

- Method **authenticate** displays a login dialog box and collects the user name and password for authentication purposes.
- Method **login** allows an application to provide user name and password information collected in any custom way, other than specifying form-based authentication in an application deployment descriptor.
- Method **logout** allows an application to reset the caller identity of a request.

```
@WebServlet(name = "ProductDisplay", urlPatterns = {"/products"})
public class ProductDisplayServlet extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response)
                                throws ServletException, IOException {
    ...
    request.login(username,password);
    ...
    request.logout();
    ...
}
```

An example of authenticate method of the HttpServletRequest object:

```
package demos;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AuthTestServlet extends HttpServlet {
  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      request.authenticate(response);
      out.println("Authenticate Successful");
     } finally {
      out.close();
     }
   }
}
```



Declare Application Roles

Developers describe application security roles that are mapped to users and groups upon deployment

- Application Security Roles can be defined with
  - Standard Java EE deployment descriptors
  - DeclareRoles Annotation
- Server specific descriptors are used to map application roles to users and groups

Servlet or EJB

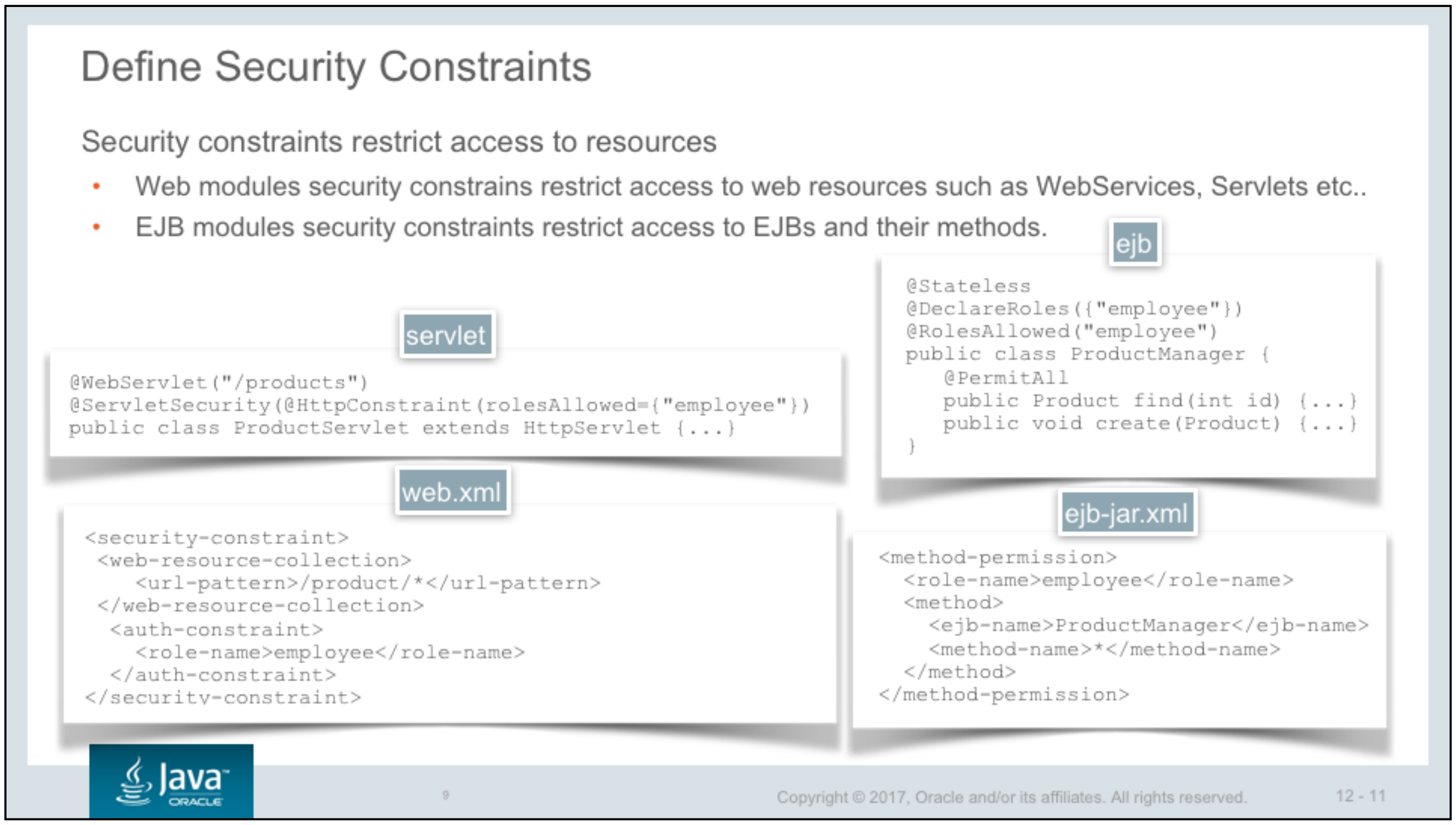
```
@DeclareRoles({"employee", "customer"})
```

web.xml or ejb-jar.xml

```
<security-role>
  <role-name>employee</role-name>
</security-role>
<security-role>
  <role-name>customer</role-name>
</security-role>
```

weblogic.xml or weblogic-ejb-jar.xml

```
<security-role-assignment>
  <role-name>employee</role-name>
  <principal-name>jo</principal-name>
  <principal-name>clerks</principal-name>
  <principal-name>managers</principal-name>
</security-role-assignment>
```



Define Security Constraints

Security constraints restrict access to resources

- Web modules security constrains restrict access to web resources such as WebServices, Servlets etc..
- EJB modules security constraints restrict access to EJBs and their methods.

servlet

```
@WebServlet("/products")
@ServletSecurity(@HttpConstraint(rolesAllowed={"employee"})
public class ProductServlet extends HttpServlet {...}
```

web.xml

```
<security-constraint>
 <web-resource-collection>
    <url-pattern>/product/*</url-pattern>
 </web-resource-collection>
  <auth-constraint>
    <role-name>employee</role-name>
  </auth-constraint>
</security-constraint>
```

ejb

```
@Stateless
@DeclareRoles({"employee"})
@RolesAllowed("employee")
public class ProductManager {
   @PermitAll
   public Product find(int id) {...}
   public void create(Product) {...}
}
```

ejb-jar.xml

```
<method-permission>
  <role-name>employee</role-name>
  <method>
    <ejb-name>ProductManager</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

9

12 - 11

Java defines a number of security annotations:

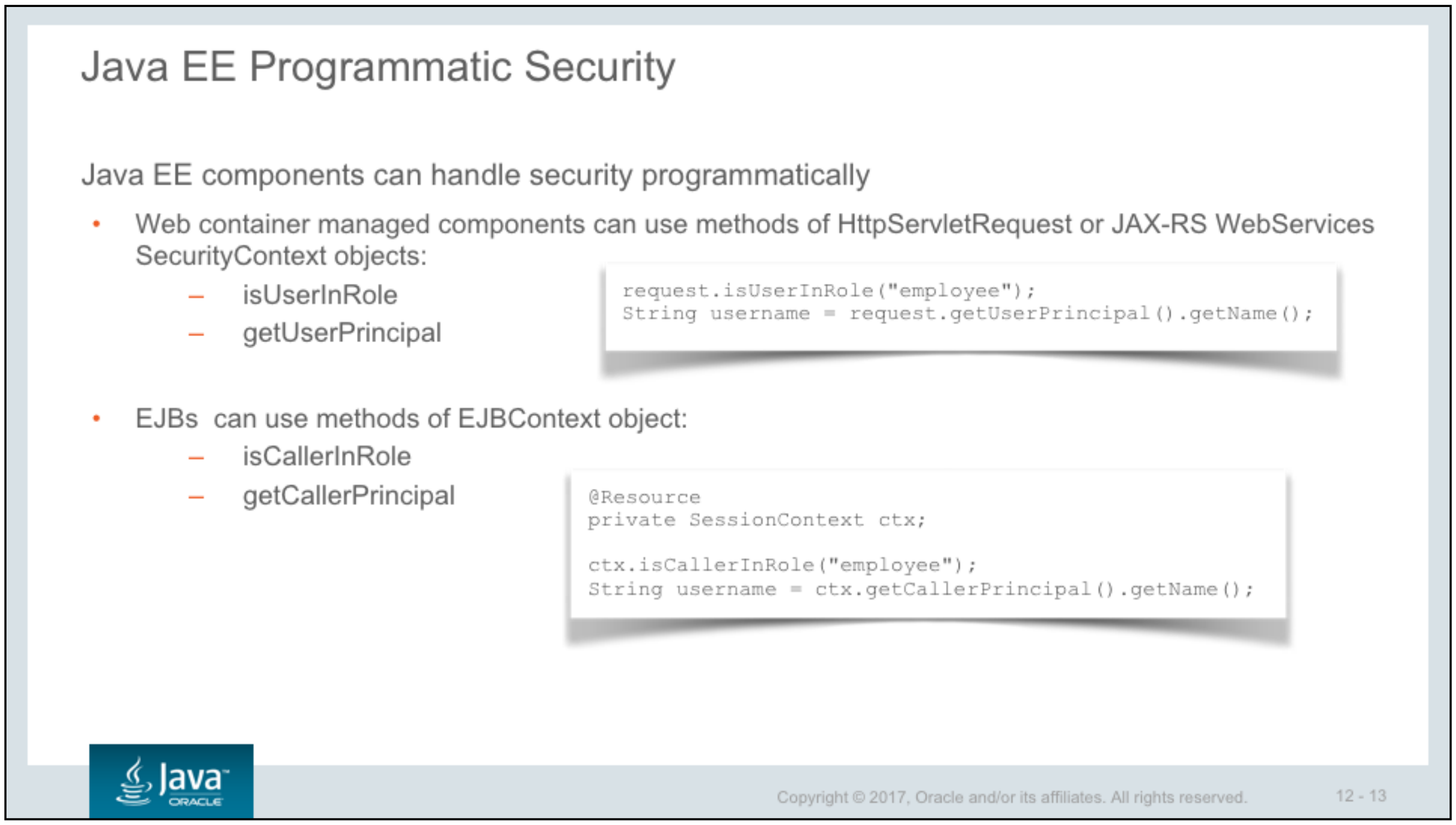
- PermitAll - indicates that the given method or all business methods of an EJB are accessible by everyone
- DenyAll - indicates that the given method in the EJB cannot be accessed by anyone
- RolesAllowed - Indicates that the given method or all business methods in the EJB can be accessed by users associated with the list of roles
- DeclareRoles - Used by both Servlets and EJBs to define application security roles
- RunAs - Used by both Servlets and EJBs to specify the run-as role for the given components. By default components assume they are executed with the identity of the invoker

A web container managed resources can describe security constraints restarted to specific HTTP Methods. In the following example, only the POST method is restricted to the role customer:

```
<security-constraint>
  <display-name>Customer Product Permission</display-name>
  <web-resource-collection>
    <url-pattern>/product/*</url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>customer</role-name>
  </auth-constraint>
</security-constraint>
```

Or in this example annotations are configured to allow GET Method to be invoked by everyone and POST Method only by members of the employee role:

```
@WebServlet("/products")
@ServletSecurity(
   httpMethodConstraints={@HttpMethodConstraint("GET"),
                                    @HttpMethodConstraint(value="POST", rolesAllowed={"employee"}))
public class ProductServlet extends HttpServlet {...}
```



Java EE Programmatic Security

Java EE components can handle security programmatically

- Web container managed components can use methods of HttpServletRequest or JAX-RS WebServices SecurityContext objects:
  - isUserInRole
  - getUserPrincipal

```
request.isUserInRole("employee");
String username = request.getUserPrincipal().getName();
```

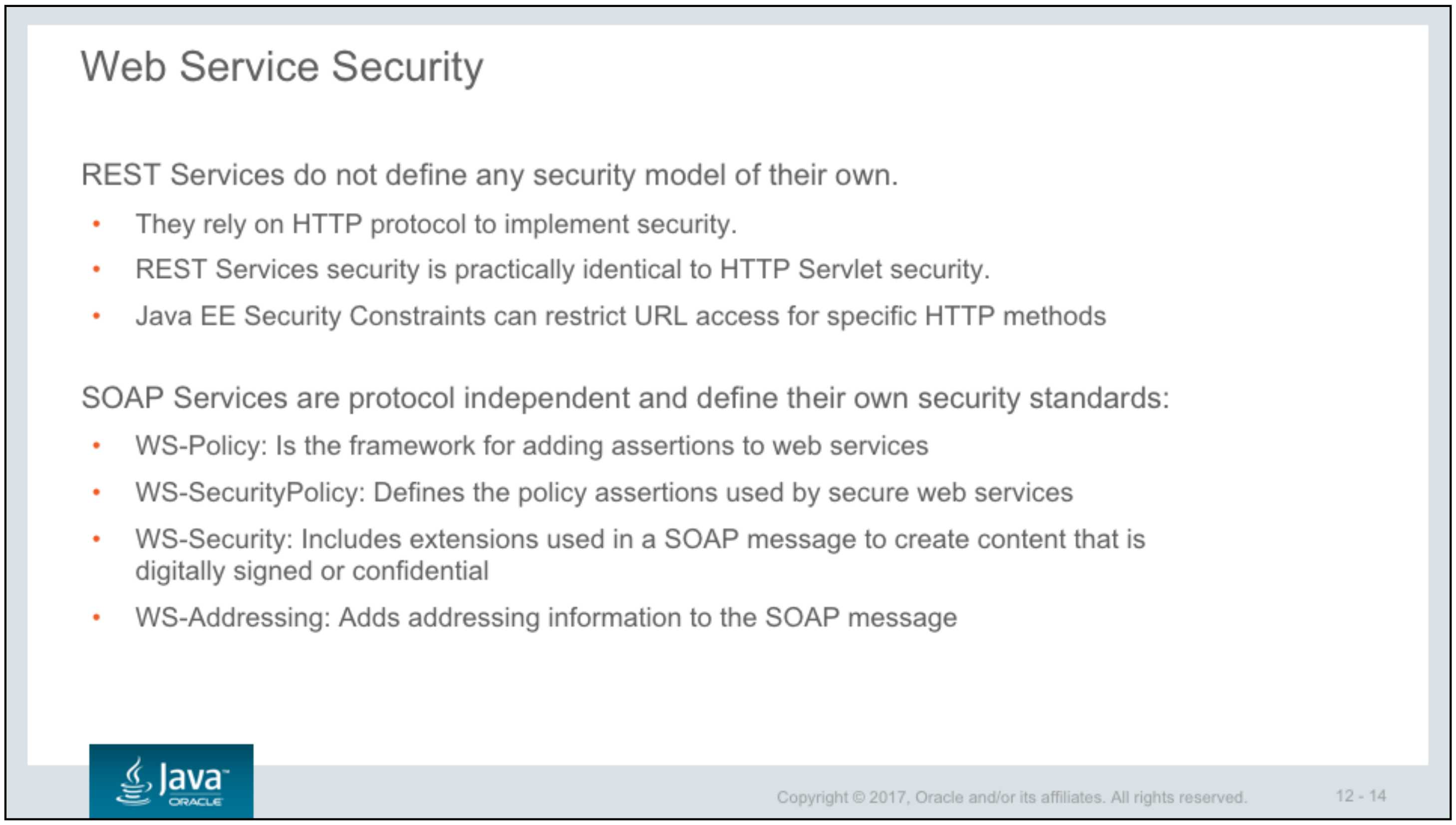
- EJBs can use methods of EJBContext object:
  - isCallerInRole
  - getCallerPrincipal

```
@Resource
private SessionContext ctx;

ctx.isCallerInRole("employee");
String username = ctx.getCallerPrincipal().getName();
```

12 - 13

Generally it is preferable to utilize declarative security in Java EE applications. However, with declarative approach security constraint may be too coarse gained. In the EJB module you may wish to programmatically access information about caller identity to provide audit information, such as recording user audit data into the database. When declaring security constraints for the Web UI, you may use a programmatic approach to define security permission for a specific fragment of the page or JSF component.



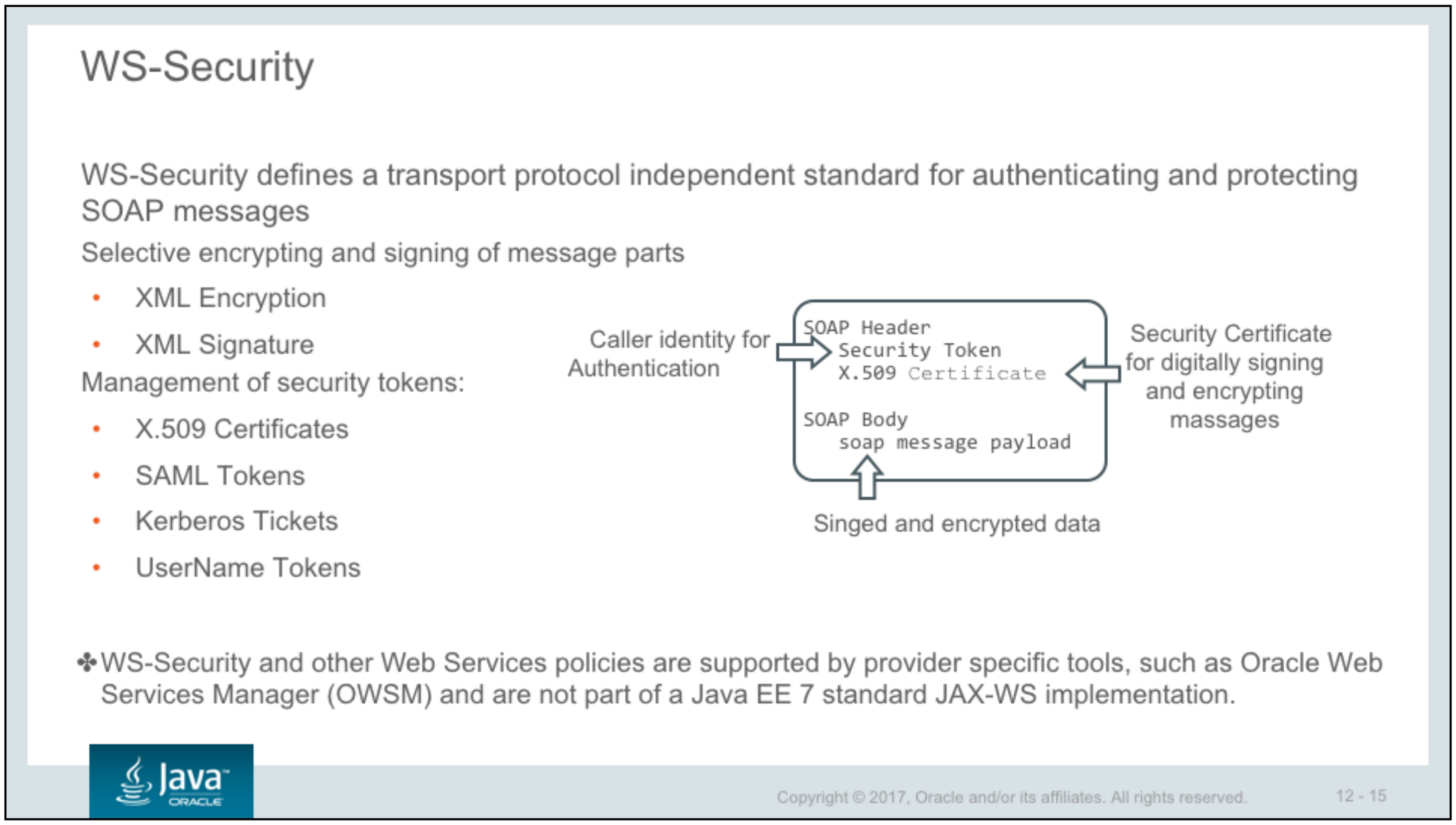
Web Service Security

REST Services do not define any security model of their own.

- They rely on HTTP protocol to implement security.
- REST Services security is practically identical to HTTP Servlet security.
- Java EE Security Constraints can restrict URL access for specific HTTP methods

SOAP Services are protocol independent and define their own security standards:

- WS-Policy: Is the framework for adding assertions to web services
- WS-SecurityPolicy: Defines the policy assertions used by secure web services
- WS-Security: Includes extensions used in a SOAP message to create content that is digitally signed or confidential
- WS-Addressing: Adds addressing information to the SOAP message



WS-Security

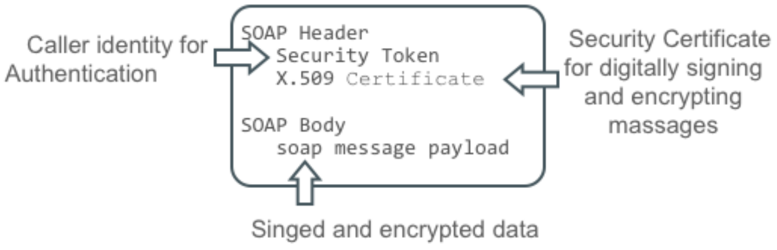
WS-Security defines a transport protocol independent standard for authenticating and protecting SOAP messages

Selective encrypting and signing of message parts

- XML Encryption
- XML Signature

Management of security tokens:

- X.509 Certificates
- SAML Tokens
- Kerberos Tickets
- UserName Tokens

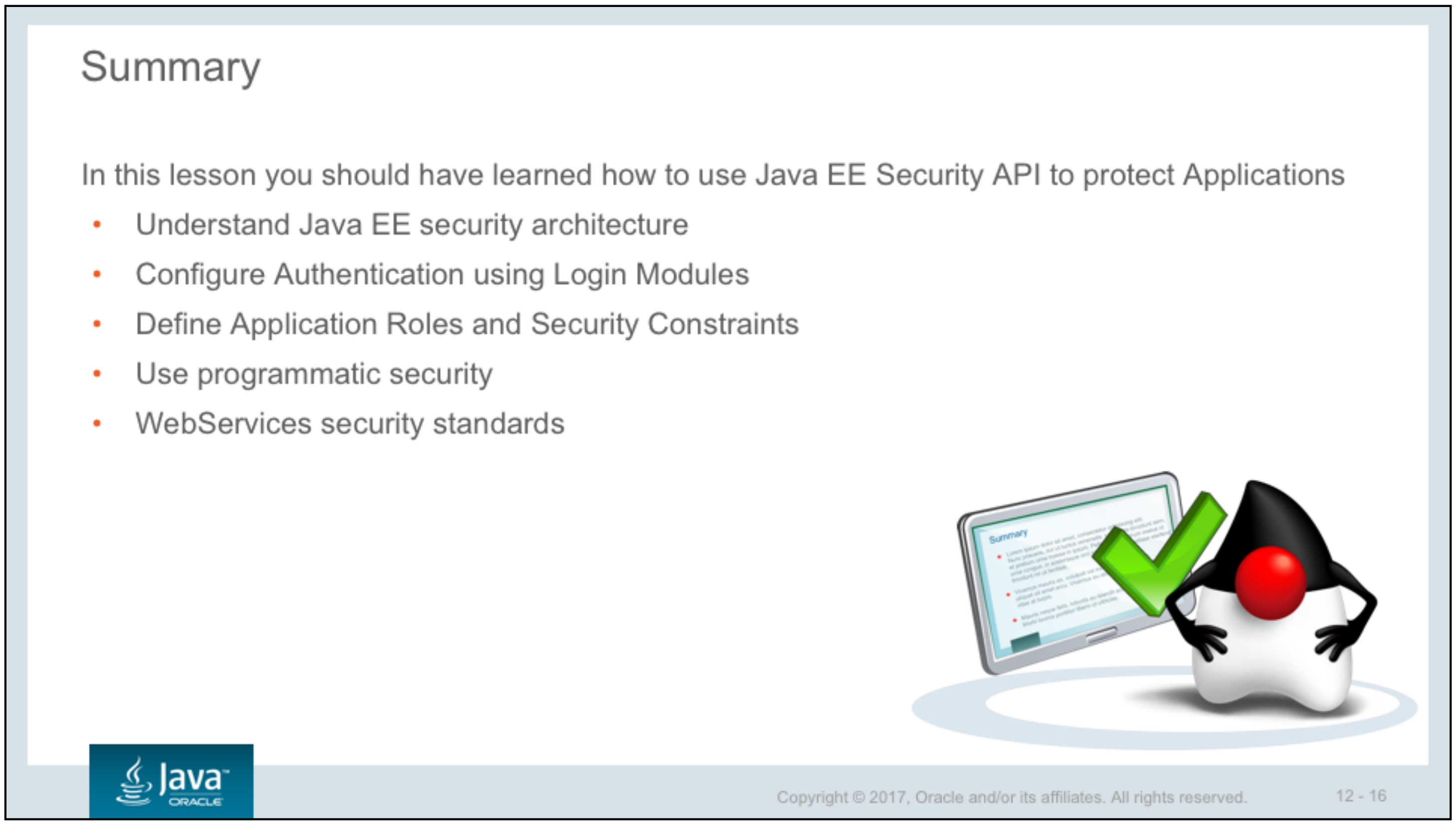

Caller identity for Authentication
SOAP Header
Security Token
X.509 Certificate
SOAP Body
soap message payload
Security Certificate for digitally signing and encrypting massages
Singed and encrypted data

✤WS-Security and other Web Services policies are supported by provider specific tools, such as Oracle Web Services Manager (OWSM) and are not part of a Java EE 7 standard JAX-WS implementation.

OASIS group is a consortium that defines standards used by web services, including WS-Security specification. For more information see: https://www.oasis-open.org

W3C XML Signature specification http://www.w3.org/Signature/

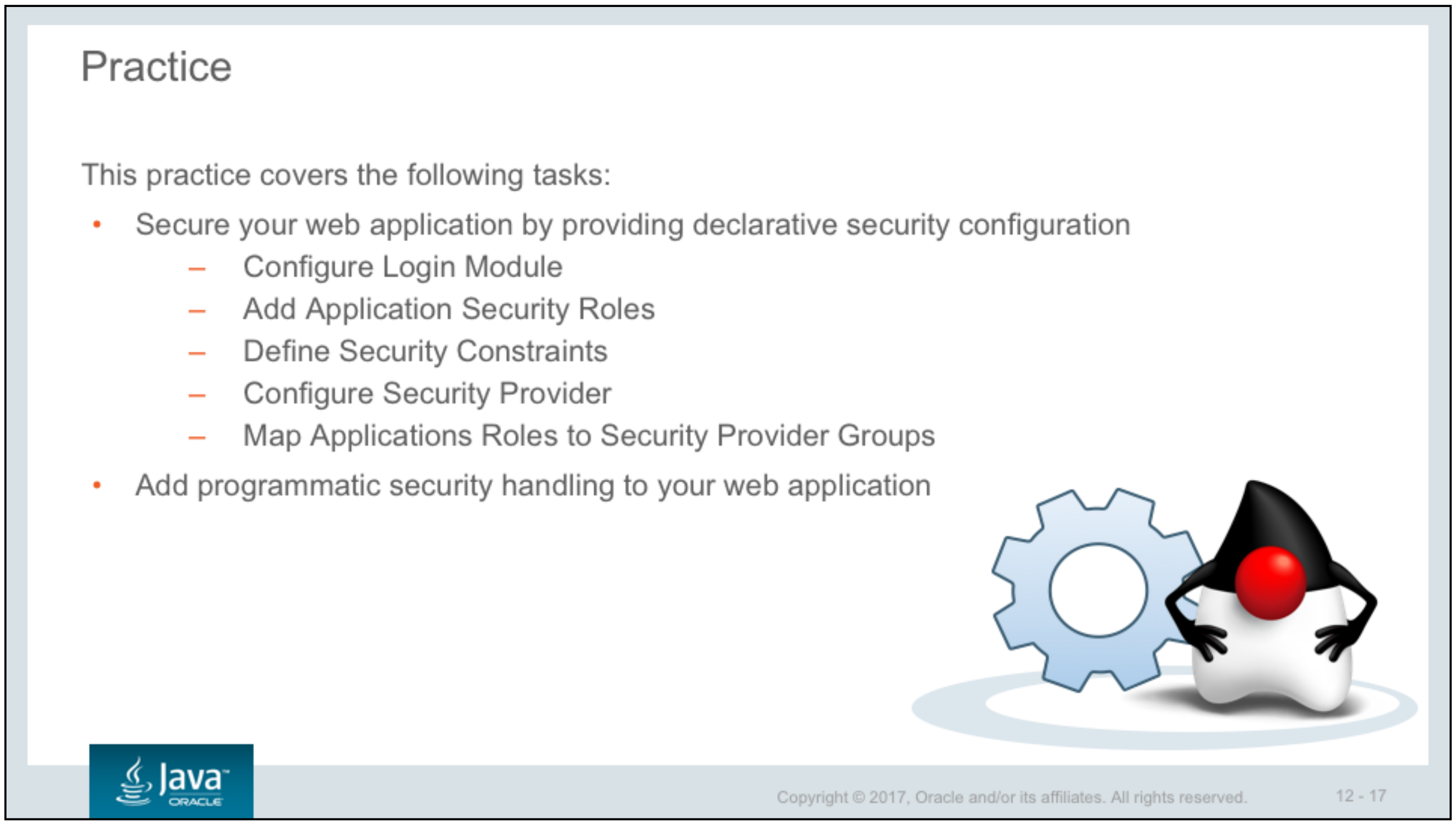
W3C XML Encryption specification http://www.w3.org/Encryption/2001/



Summary

In this lesson you should have learned how to use Java EE Security API to protect Applications

- Understand Java EE security architecture
- Configure Authentication using Login Modules
- Define Application Roles and Security Constraints
- Use programmatic security
- WebServices security standards



Practice

This practice covers the following tasks:

- Secure your web application by providing declarative security configuration
  - Configure Login Module
  - Add Application Security Roles
  - Define Security Constraints
  - Configure Security Provider
  - Map Applications Roles to Security Provider Groups
- Add programmatic security handling to your web application