Sun Certified Web Component Developer Study Companion

SCWCD J2EE 1.4 (Exams 310-081 and 310-082)

Charles Lyons

Published by Garner Press (http://www.garnerpress.com)

Copyright © 2006 Charles E. Lyons, Garner Press

All rights reserved. This book, or parts thereof, may not be reproduced or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system now known or to be invented, without written permission from the Publisher.

ISBN: 0-9551603-1-6
ISBN: 978-0-9551603-1-8 (From 1st January 2007)

18

Security

Almost all Web application developers will, at some stage in their career, be concerned about the safety of their application and the data transmitted between it and the client. The immediate concerns which spring to mind would likely cover a broad range of possible infiltration targets: the client and server machines, the transmission channel between the server and client, how data is stored on the server, and how secure passwords are for critical resources like databases. Most of these issues already outlined are the responsibility of the server or network administrators, and as such are not our concern. As application developers, we are primarily interested in the following aspects of security:

- How do we prove that a user really is who they claim to be? This process is called **authentication**.
- Is a user permitted to access a certain resource – are they logged in with a valid username and password? This is known as **authorisation**: is a user authorised to access a resource?
- Can we make sure that data is transmitted without corruption or modification by a third party? This ensures that **data integrity** is maintained.
- Can we secure communications with the client using, for example, Secure HTTP (HTTPS)? This is taking steps to ensure **confidentiality** (**data privacy**) – a third party is unable eavesdrop on our secret or private communications.

These goals lie at the software-level of an application, not with the hardware issues like open ports and the correct use of firewalls. The J2EE platform is incredibly helpful with security issues: using the deployment descriptor for an application, we can specify **declarative security** requirements; being able to do this is the only security-based exam objective. For completeness, we will look briefly at how a finer-grained security model can be built using **programmatic security** – i.e. programming security requirements ourselves using Java code rather than relying on the container's declarative security model, which may not suit every application's needs.

Methods for Authentication

J2EE 1.4 provides four ways in which to authenticate a user. Three of these rely on a username and password, while the fourth uses more sophisticated encryption technology. Thankfully, we don't need to know anything about the underlying mechanisms for the exam, and I don't propose to discuss the mathematics behind today's cryptography here!

The data used to authenticate a user are known as the **credentials** of that user; **valid credentials** result in a user being authenticated and 'logged in', while **invalid credentials** represent their failure to provide sufficient or correct information. The provision of invalid credentials will prevent the user from being authenticated, and as such they will not be authorised to access restricted resources.

Basic Authentication

At its most basic, authentication relies on the client to ask for a username and password. When using a browser, this normally results in a pop-up window with the requisite fields as shown in Figure 18.1.

This method of authentication is simple to implement, but provides a platform- and browser-dependent dialog window which may not suit the style of your application. This type of authentication uses HTTP headers to request authentication data from the client and to transmit the input data back to the server.

Figure 18.1

Sample Basic Authentication Dialog

Digest Authentication

One of the problems with Basic Authentication is that it transmits data in plain, unencrypted, base 64 format. This makes the interception of data by an unauthorised party a relatively simple task: eavesdrop on the HTTP packets and simply read the submitted username and password.

The alternative is to use a form of cryptography known as **hashing**, or more correctly using a **message digest**. A digest is a numerical summary of the transmitted data's contents; the calculations are performed in such a way that every string has a different digest value. What makes this method strong is that hashing a message is a one-way operation only: a digest can be created from a message, but it is computationally infeasible to reconstruct the original message from the digest. The strength of the digest depends on the type of algorithm used, but on the whole the default digests are strong enough for most applications.

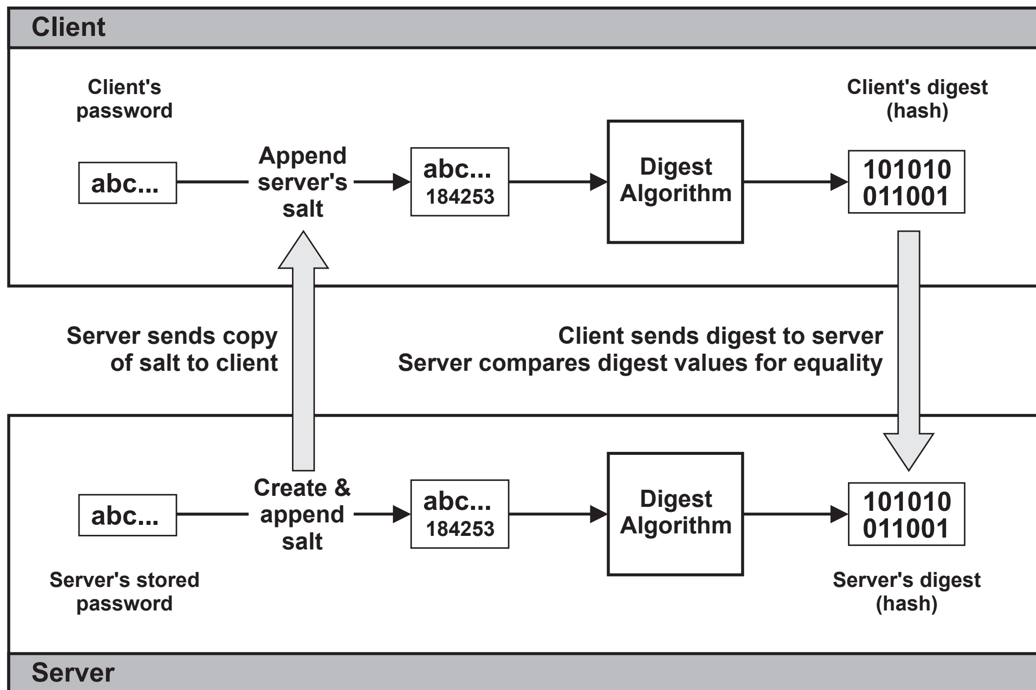
Using Digest Authentication causes the password to be hashed by the client (a message digest created from it) before it is sent back to the server. Since the digest cannot be reversed to obtain the original password string, the server must also hash the password it stores using the same algorithm and compare the hashes rather than the original passwords. Since each string hashes to its own unique digest, if the client and server copies of the password are equal, then by definition their hashes must also be equal. Notice that hashing is only performed on passwords: usernames are always sent in plain text (as unencrypted strings).

This still might not seem terribly secure: surely an unauthorised party could eavesdrop on the digest for the password and use that to authenticate themselves, in the same way the real client does? To overcome this, the server passes a **salt** to the client when it requests authentication; this usually comes in the form of the current date and time (or the numerical timestamp). The client incorporates this timestamp in the password string before hashing it. The timestamp will be different for every request for authentication, because the server's clock is constantly changing, so each new password digest created by the client will be different even if the password is the same. Both the server and client use the same salt, and the same combination of salt and password, so if both the salt and password are equal then the client's digest will equal the server's digest and they will be successfully authenticated. Using a different salt for each digest means that using the same password/salt hash in the future will result in a failed authentication. Only someone who knows the original password and the current salt can be authenticated. Figure 18.2 shows a summary of the Digest Authentication process, including how the salt created by the server is used.

J2EE-compliant containers are not required to support this authentication method, but must support the other three.

Figure 18.2

Using Password Digests



Form Authentication

If you want to integrate the authentication process with the rest of your website, you'll want to use Form Based Authentication. This uses an (X)HTML form, embedded in a web page designed by you, to request the username and password. Since this is just another web page, it can be designed to fit with the rest of the site–it needn't be obtrusive like Basic Authentication.

You could program your own Form Based Authentication mechanism, but the container already provides a template for the layout of the form. Using this default layout allows the container to do declarative authentication, which saves you having to write your own application code. The container's template requires a username and password; it obtains these using (X)HTML input fields:

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

The key to this template lies not in the HTML code, but the `action` destination which must be `j_security_check`, and the names of the input fields, `j_username` and `j_password`. The action attribute of the HTML `<form>` element is the URL to which the form's data will be submitted; `j_security_check` alerts the container that this is the default authentication form. The container will look for the values of the `j_username` and `j_password` request parameters in the body of the request (since the request uses POST); it uses these as the credentials.

The advantage of this method is that it easily integrates with the rest of the website. The disadvantage is that the data is still sent *unencrypted*, as with Basic Authentication. For true security, you would need to use Form Based Authentication over a Secure HTTP channel, so that all submitted data (the username, and password in particular) are encrypted. This is declared as part of the *confidentiality* requirements of an application, so a discussion is deferred until later in the chapter.

SSL Certificates

As part of the process of establishing a secure channel with the client, both the client and server are required to identify themselves by providing appropriate credentials. This typically occurs by comparing the client's Public Key Certificate (PKC) with a copy stored on the server. This is the most secure form of authentication since it is difficult to fake a certificate.

While J2EE allows us to use SSL certificates for authentication simply by making the appropriate declarations in the deployment descriptor, the theory behind SSL and asymmetric encryption is too involved to discuss here. Bear in mind, however, that there are drawbacks: each client authenticated using this method must have its own PKC, which is not something most of the general public possess. Moreover, the server must already own a copy of the certificate for comparison purposes, which implies that a server administrator has taken the time to establish a database of known certificates. Hence, the use of SSL certificates for authentication is really limited to internal networks and intranets where *both* the client and server machines are under the control of a single authoritative body.

Authentication and Sessions

The container must have some way to track users and their authentication status. It typically does this using either HTTP sessions or SSL sessions. While the latter provide a secure, undisputable, identification of the client through the use of unique certificates, using unencrypted sessions over HTTP can be problematic. For example, what if an unauthorised user were able to obtain (or guess) the `jsessionid` for an authorised user? This could very easily lead to unauthorised (and even unauthenticated) users acting as though they are authorised to access restricted resources. In many

cases, applications don't require a totally secure form of authentication, and the use of session timeouts can help to reduce security problems, but for online banking and eCommerce especially, the use of SSL authentication should be a design priority.

Declarative Authentication

Basic, Digest and Client-Cert Authentication

The type of authentication you wish to use, if you don't want to create your own programmatic authentication mechanism, must be configured in the deployment descriptor for the application, using the `<login-config>` element:

```
<web-app>
  ...
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Restricted zone</realm-name>
  </login-config>
  ...
</web-app>
```

As you can see, `<login-config>` is a direct child of the descriptor's root `<web-app>` element. There may only be one `<login-config>` element per deployment descriptor, so the authentication method you use must be consistent across the *entire Web application*.

The example given above is suitable when HTTP Basic Authentication is required, using the client's or browser's own pop-up dialog window. The `<auth-method>` specifies the name of the authentication type being used: here `BASIC`. `<realm-name>` is specific to the `BASIC` method, declaring the realm name to be displayed in the pop-up dialog window. Realm names are human-readable textual reminders which can help to remind a user what facility they are attempting to access. In Figure 18.1, this name is blank; if that dialog had been produced from this example, the value of `<realm-name>` would be displayed next to the Realm label (above the input boxes).

The declaration for Digest Authentication is the same as for Basic Authentication, since it uses the same HTTP mechanism and client dialog, but specifies the `DIGEST` authentication method flag:

```
<web-app>
  ...
  <login-config>
```

```
    <auth-method>DIGEST</auth-method>
    <realm-name>Restricted zone</realm-name>
  </login-config>
  ...
</web-app>
```

Specifying that the client's SSL certificate should be used for authentication is even simpler since it cannot include the optional `<realm-name>` element anyway (the use of a realm name with certificates is meaningless):

```
<web-app>
  ...
  <login-config>
    <auth-method>CLIENT-CERT</auth-method>
  </login-config>
  ...
</web-app>
```

Form Authentication

The configuration for Form Based Authentication is necessarily a little more complex:

```
<web-app>
  ...
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/errors/403.html</form-error-page>
    </form-login-config>
  </login-config>
  ...
</web-app>
```

The `<auth-method>` value must now be the `FORM` flag; notice that `<realm-name>` does not apply to Form Based Authentications so we've omitted it. The key to the configuration lies in the `<form-login-config>` element and its subelements. `<form-login-page>` specifies the *context-relative* URL (beginning with a `/`), to the page containing the login form. If the page is authored using (X)HTML, the form should use the template we saw when we first discussed Form Based Authentication in this chapter. `<form-error-page>` specifies the *context-relative* URL of the error page to use if authentication fails. Typically this will point to a 'login failed' page.

An interesting idea which I've implemented in the past is to effectively point the error page to the login page – i.e. the error page and login page are (very nearly) one and the same. This means that if login fails, the user gets another chance to enter their details and try to authenticate again. By storing the username in an appropriate scope (namely the session), the form can be auto-completed with the username last used, should authentication have failed. Optionally, by using an appropriately scoped integer variable, you can also keep track of how many times authentication has already failed; after three or so times you could respond with the 'login failed' page instead of another presentation of the login page. If a user has mistyped or forgotten their password, this allows them to attempt login at least three times before receiving an error. This is quite often preferable to jumping straight to an error page, and frequently implemented on websites of varying complexity. HTTP Basic Authentication also characteristically asks for credentials three times before returning an error.

Performing Authorisation

As we've already said, authorisation is about deciding whether a user is permitted to access a resource or not. The question is how do we make this decision? Clearly, authentication is the first step towards authorisation; once a user is 'logged in' we know who they are, and can identify them amongst other users in other sessions. To be useful, we need to make use of authentication for the purposes of authorisation. The design which springs to mind is to maintain a database of users' usernames against the resources they're permitted to access on the server. This allows us to look up the authenticated user's database record to see if the page requested is in the record: if so, then the user is authorised to access the resource, and if not access is denied. Indeed, this is quite possible using programmatic security which we look at briefly at the end of the chapter.

Unfortunately, using usernames per se for authentication is too fine-grained to be practical for the container's model of declarative security. Another suggestion is quite often more realistic; it uses the concept of **roles**, which are *groups* into which users are categorised. Now, instead of permitting access to each individual user in turn, we can assign permissions based on the roles into which users 'fit'. This is easiest seen with an example: take a banking application which is accessed by system-level administrators, bank clerks and customers. Each of these categories represents a distinct role: `administrators`, `clerks` and `customers`. Each role has different levels of access permissions: `administrators` maintain the application so they have unrestricted access, `clerks` have permission to make bank-level transactions and access bank funds as well as check and update customers' accounts, which is the only function `customers` can perform. Most authorisations here can be performed based on the user's role alone, but it would obviously be necessary to restrict `customers` access to their *own account* only and not the collection of all customers' accounts. This

would be a job for programmatic security: controlling access to account details based on the *individual user's credentials* rather than the entire role.

Declaring Security Roles

In order to use roles as part of the container's security, we need to do two things: declare the set of roles that exist in the application, and assign individual users to those roles.

Security Roles in Applications

We use the `<security-role>` element (a direct child of the `<web-app>` root) in the deployment descriptor to declare a role which exists in the application. *All roles used in the application must be declared* in order to be used as identifiers in the rest of the deployment descriptor and in component code itself. Declaring a role name using this syntax is equivalent to notifying the container up front about the roles it is required to recognise and support:

```
<web-app>
  ...
  <security-role>
    <role-name>customers</role-name>
  </security-role>
  ...
</web-app>
```

While `<security-role>` may only contain one mandatory `<role-name>` subelement, the declaration of multiple `<security-role>` elements is permitted; thus, from the previous example:

```
<web-app>
  ...
  <security-role>
    <role-name>administrators</role-name>
  </security-role>
  <security-role>
    <role-name>clerks</role-name>
  </security-role>
  <security-role>
    <role-name>customers</role-name>
  </security-role>
  ...
</web-app>
```

Assigning Users to Roles

Users are assigned to roles in a way *specific to each J2EE server*. There's no one generic way to do this assignment, it depends purely on what J2EE server software you've got running. Common methods for the authentication of users include a dedicated file stored in the J2EE server environment, querying a directory service and interrogating the lists of usernames, passwords and user groups native to the operating system on which the J2EE server is running.

You should check your J2EE server's specific documentation before considering implementing security. If you require authentication based on data which is operating-system dependent, or which references an external database or file store, you will most likely need to program your own authentication modules. Your server should provide an API and documentation to enable you to do this. See the section on programmatic security later in the chapter for more details.

Sun's AppServer 8, which at the time of writing was the J2EE 1.4 Reference Implementation, includes support for two methods of authentication:

- File storage based on a file named `keyfile`, which contains a list of usernames and hashed (digested) passwords, as well as a list of role names for each username.

- Retrieval using LDAP from a named directory service.

AppServer 8 allows the usernames and passwords for its file-based storage to be set through its Web-based administration console, which is ideal if the data doesn't change very often, but it doesn't allow changes to be made by anyone other than administrators; the use of a directory promotes scalability and extensibility, but you might still find yourself programming your own modules. Thankfully, AppServer 8 comes with a useful 'Developer's Guide' which explains the API you should use to program custom authentication and authorisation mechanisms.

The standard distribution of the Apache Tomcat Web container includes XML configuration files for lists of users, passwords and their roles, and ships with a JDBC database realm for authentication.

Web Security and EJBs

Using any of the declarative methods for authentication, or indeed creating a correct programmatic authentication procedure, allows security to extend across the entire J2EE platform and not just within the current Web application. In particular, the EJB container is able to use the authentication details for the current user to determine whether they are permitted to access EJB components

as well. This presents a two-tier security system: not only can we prevent unauthorised users from *accessing Web resources*, but also from *using services* provided by the business logic layer of a J2EE application.

The deployment descriptor provides a way to interface security between Web and EJB applications and containers. It is important that every developer realise that security, whether declarative or programmatic, but especially the latter, *must* be implemented correctly for this work. A makeshift login system which does not interoperate with the rest of the J2EE platform or the container could just be the weakest link in an application. Declarative security is always implemented correctly by the container.

Another important property of J2EE security is that authentication and the user's credentials are tracked at the *container level*, and not individually at the application level. This means that users authenticated by one Web application are automatically authenticated in all other applications running in the same container. This may be advantageous or not: in some cases it is useful because some Web applications run together and are designed to be interoperable. It can be an inconvenience, as well as a possible security issue, if two Web applications contain *the same role names* for authorisation, but are intended to function separately. A user logged in to one of the applications with a role name will be able to access all the resources constrained by that role name in the other application(s) as well, which means unauthorised access in those other applications. To avoid this, you should either run completely separate applications in separate containers (for example, on different physical servers or in different J2EE virtual servers, realms or domains), or ensure that all role names in each deployed application are independent of every other application's.

Note that all invocations of EJBs must be accompanied by a security identity, or principal; by default, the user's principal is used. However, when a servlet grants access to unauthenticated users, or indeed the servlet incorporates its own programmatic security overriding that of the declarative role-based system, using the user's principal becomes inadequate. If this is the case, you can use the `<run-as>` subelement of `<servlet>` in the deployment descriptor to place the *servlet itself* into a role. It is this role which will then be used for security when invoking an EJB from the servlet. This allows the Web developer to build yet another layer of role-based security into the application. Note that if the user is not authenticated and `<run-as>` is not declared for the servlet making the EJB invocation, the 'unauthenticated principal' used for invoking the EJB is undefined by the J2EE 1.4 specification and will depend on the vendor-specific J2EE implementation.

Declarative Authorisation and Confidentiality

For the exam, you will need to know all the deployment syntax which appears in this section, and it isn't that detailed so my best advice is to learn as much of it as possible.

All authorisation and confidentiality security requirements for an application are contained in one or more `<security-constraint>` elements in the deployment descriptor:

```
<web-app>
  ...
  <security-constraint>
    <!-- put the constraint details in here -->
  </security-constraint>
  ...
</web-app>
```

Each `<security-constraint>` element declares security properties only for a specified collection of Web resources, much like `<jsp-property-group>` declares properties for a collection of JSPs. If different security constraints are required for different components in the application, they can each be configured in separate `<security-constraint>` elements. Each `<security-constraint>` may contain zero or more `<display-name>` elements, which specifies human-readable names given to this constraint (if any):

```
<security-constraint>
  <display-name>My Constraint 1</display-name>
</security-constraint>
```

We will now look at the other child elements grouped by their function; a summary of all the security-related descriptor elements can be found at the end of this chapter.

Defining Resource Collections

We said that each `<security-constraint>` applies only to a certain, specified, collection of resources and that this model allows us to have different security constraints for different resources.

Resource collections are specified using one or more `<web-resource-collection>` elements and their children:

```
<security-constraint>
  ...
  <web-resource-collection>
```

```
    <web-resource-name>shoppingbasket1</web-resource-name>
    <url-pattern>/basket/*</url-pattern>
    ...
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    ...
  </web-resource-collection>
  ...
</security-constraint>
```

There may only be one `<web-resource-name>` element per collection; this element specifies the *logical* descriptor name given to this collection.

There may be one or more `<url-pattern>` elements, each specifying a pattern to which every incoming request will be matched; if the request matches the pattern, the container will consider the resource to be in this collection and impose the declared security constraints.

Requests may also be filtered by what HTTP method is being used to access the resource, allowing some methods to be blocked while others are allowed through. A typical requirement is to impose access restrictions on the sensitive `PUT` and `DELETE` methods (if these are actually implemented) while allowing `GET` requests to pass through unchallenged. The optional `<http-method>` is used to specify each HTTP method which should fall under this security constraint for the given URL patterns.

There are therefore two ways to specify a collection:

- Using only the URL pattern(s); every request which matches this pattern(s), regardless of its HTTP method type, will have the specified constraints applied to it.
- Using both the URL pattern and one or more `<http-method>` elements will apply security constraints only if the request matches both the URL pattern(s) and the declared HTTP method(s).

A resource may belong to more than one `<web-resource-collection>` in more than one `<security-constraint>` simultaneously. When this is the case, the container matches each incoming URL to *every* resource collection encountered in the descriptor, and not just the first found. The container then amalgamates all `<security-constraint>` properties for the requested resource.

Authorisation Constraints

Now that we have selected some (or perhaps all) resources in the application to which the container's security model will be imposed, we have a choice of what security is required: authorisation for these resources only, confidentiality only, or both.

Authorisation is declared using the `<auth-constraint>` child element of `<security-constraint>`. Aside from the generic `<description>` element, this declares only one child element:

```
<security-constraint>
  ...
  <auth-constraint>
    <role-name>role1</role-name>
    <role-name>role2</role-name>
    ...
  </auth-constraint>
  ...
</security-constraint>
```

Each `<role-name>` declares the name of a role into which the authenticated user can fit if they are authorised to access the resources in the collection. If the user fits into the role name specified here, they are granted access. Otherwise, access is denied. There are some special cases:

- A declaration of the form `<role-name>*</role-name>` means 'grant access to an authenticated user of any role'. This does imply that the user *has already been authenticated*, but you don't care to which exact role(s) they belong. *Never* mistake the wildcard to mean 'grant anyone access, even if they're not authenticated'. Declaring any other `<role-name>` elements is redundant since the `*` wildcard encompasses all roles anyway.

- Omitting all `<role-name>` elements, or indeed the parent `<auth-constraint>`, means 'grant access to anyone, authenticated or not'. The emphasis here is that we are allowing *unrestricted* access to the resources by not specifying any roles; the user doesn't even have to have been authenticated.

In the case that a resource is mapped to more than one `<security-constraint>` at one time, by virtue of being declared in multiple `<web-resource-collection>` elements, all role names will be amalgamated at run time for that resource. For example:

```
<security-constraint>
  ...
  <web-resource-collection>
    <web-resource-name>View page</web-resource-name>
    <url-pattern>/basket/view.jsp</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>role1</role-name>
    <role-name>role2</role-name>
  </auth-constraint>
  ...
</security-constraint>
<security-constraint>
  ...
  <web-resource-collection>
    <web-resource-name>View page</web-resource-name>
    <url-pattern>/basket/*.jsp</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>role3</role-name>
  </auth-constraint>
  ...
</security-constraint>
```

The resource `/basket/view.jsp` falls into both declared security constraints, since its URL matches both the `<url-pattern>`s specified. If the user were to request this resource, the authorised role name set is the union of both `<security-constraint>` instances: `role1`, `role2` and `role3`. If the user is in *any of these three* roles, they will be authorised.

Confidentiality and Data Integrity

The `<security-constraint>` element is also the place for declarations relating to securing communication channels. Confidentiality and data integrity are configured using the `<user-data-constraint>` element as a child of the `<security-constraint>` to which this communication restriction applies. In turn, the `<user-data-constraint>` element only has one child of its own, `<transport-guarantee>` which is mandatory and singular (i.e. may only appear once):

```
<user-data-constraint>
  <transport-guarantee>NONE</transport-guarantee>
</user-data-constraint>
```

The `<transport-guarantee>` element must contain one of the values:

- **NONE**
  Specifies that no security precautions are required on the channel.

- **INTEGRAL**
  Specifies that data integrity should be preserved on the channel by creating a digest for each message sent between the client and server. This digest is normally appended to the message being transmitted as supplementary information.

- **CONFIDENTIAL**
  Declares that complete encryption is required on the channel; on the Web, this is usually implemented using Secure HTTP (HTTPS), which in turn uses Secure Socket Layer (SSL) encryption. Note that encryption systems such as SSL include message digests (hashing) as an inherent part of their cryptography, to ensure the encrypted transmitted data is not modified during transmission. Using a value of `CONFIDENTIAL` often implies that `INTEGRAL` is being used as well, but as part of a more sophisticated form of encryption.

It is only legal to declare one `<transport-guarantee>` and one of the values given above, but as we said `CONFIDENTIAL` implies `INTEGRAL`, and `NONE` stands alone, so there shouldn't be a need to use more than one value anyway.

Servlets and Role References

When developers write servlets which incorporate programmatic security, they may query the roles to which an authenticated user belongs, but may not actually know the role name at the time of development. In a large enterprise, this is a common problem since lists of users are maintained by the Administrators while the application is written by the Component Developers and deployed by the Deployer. These three parties may not talk to each other very often, so it can be difficult for Component Developers to know what roles the Administrators are going to be using. This problem is particularly prevalent when your application is to be deployed to *different clients'* systems – for example, if your application or its components are sold under licence.

The deployment descriptor allows role names referenced in servlets to be linked to the proper role names stored in the deployment environment. Servlets do not need to be coupled directly to the environment, which makes them portable. It is the duty of the Deployer to liaise with the Administrator and ensure that the correct role associations are made in the deployment descriptor for the application. This declarative approach decouples the servlet code from the environment,

allowing changes to be made to the environment's role names and the descriptor, without having to modify, recompile or redeploy servlet code.

We link up the two via one or more instances of the `<security-role-ref>` subelement of the `<servlet>` element (which it itself covered in detail in Chapter 9):

```
<web-app>
  ...
  <servlet>
    ...
    <security-role-ref>
      <role-name>servlet_role</role-name>
      <role-link>environment_role</role-link>
    </security-role-ref>
    ...
  </servlet>
  ...
</web-app>
```

There may be zero or more occurrences of `<security-role-ref>` in every `<servlet>` element, corresponding to zero or more role names used in the servlet code. The `<role-name>` element is used to declare the name as it appears in the servlet code; the `<role-link>` specifies the associated role name as found in the server environment.

The `<role-link>` element is optional; if omitted, it is the responsibility of the application Deployer to provide the association. The Component Developer is only required to declare the list of all role names *used in servlet code*, using the mandatory `<role-name>` element.

The only servlet code which this actually affects is invocations of the `isUserInRole()` method on the `HttpServletRequest` object. Should a `<security-role-ref>` for the supplied role name not be declared for the servlet, the container will check the `<security-role>` elements for the application and attempt to find a match. Thus `isUserInRole()` is implemented as follows:

1. `isUserInRole()` checks for a `<security-role-ref>` mapping for this servlet in the deployment descriptor; if it exists, it performs the mapping and checks to see if the user is in the `<role-link>` role supplied. If the user is in the role, return `true`; otherwise return `false`.

2. `isUserInRole()` checks for a `<security-role>` declaration for the entire application in the deployment descriptor; if it exists and the user is in this role, this method returns `true`.

3. Otherwise return `false`.

NOTE: For each `<role-link>` name supplied, there must be a corresponding `<security-role>` declared (with a `<role-name>` child which has the same value as `<role-link>`) for the application as a whole.

For example, suppose the deployment descriptor contained the following code:

```
<web-app>
  <servlet>
    ...
    <security-role-ref>
      <role-name>shelf_stacker</role-name>
      <role-link>shelf_replenishment_technician</role-link>
    </security-role-ref>
    ...
  </servlet>
  ...
  <!--must also declare the role for the entire application -->
  <security-role>
    <role-name>shelf_replenishment_technician</role-name>
  </security-role>
  ...
</web-app>
```

then the following method invocation in that servlet would return `true` due to the `<security-role-ref>` mapping:

```
request.isUserInRole("shelf_stacker")
```

This next method invocation would also return `true` since the supplied role is declared as a `<security-role>` for the entire application:

```
request.isUserInRole("shelf_replenishment_technician")
```

Deployment Descriptor Syntax Summary

We've covered the entire J2EE security model, but in bits and pieces (rather than bytes and nibbles!) as we've gone along. Figure 18.3 is a collective summary of all the Web application deployment descriptor's syntax relating to security. All other deployment descriptor syntax can be found in Chapter 9.

For the exam, just learn this diagram; there aren't that many elements, and most are named sensibly anyway. Make sure you know all the important security concepts, what they achieve and how to declare them, and not only will you be able to build robust and secure software applications, but you'll be closer to passing the exam as well!

Getting Started with Programmatic Security (Not on the Exam)

To go down the road of investigating ways to program custom authentication and authorisation methods would take us too far into the realms of the APIs exposed by specific J2EE servers, which is outside the scope of this book. Instead, I would refer you to the following technologies and resources available from Sun:

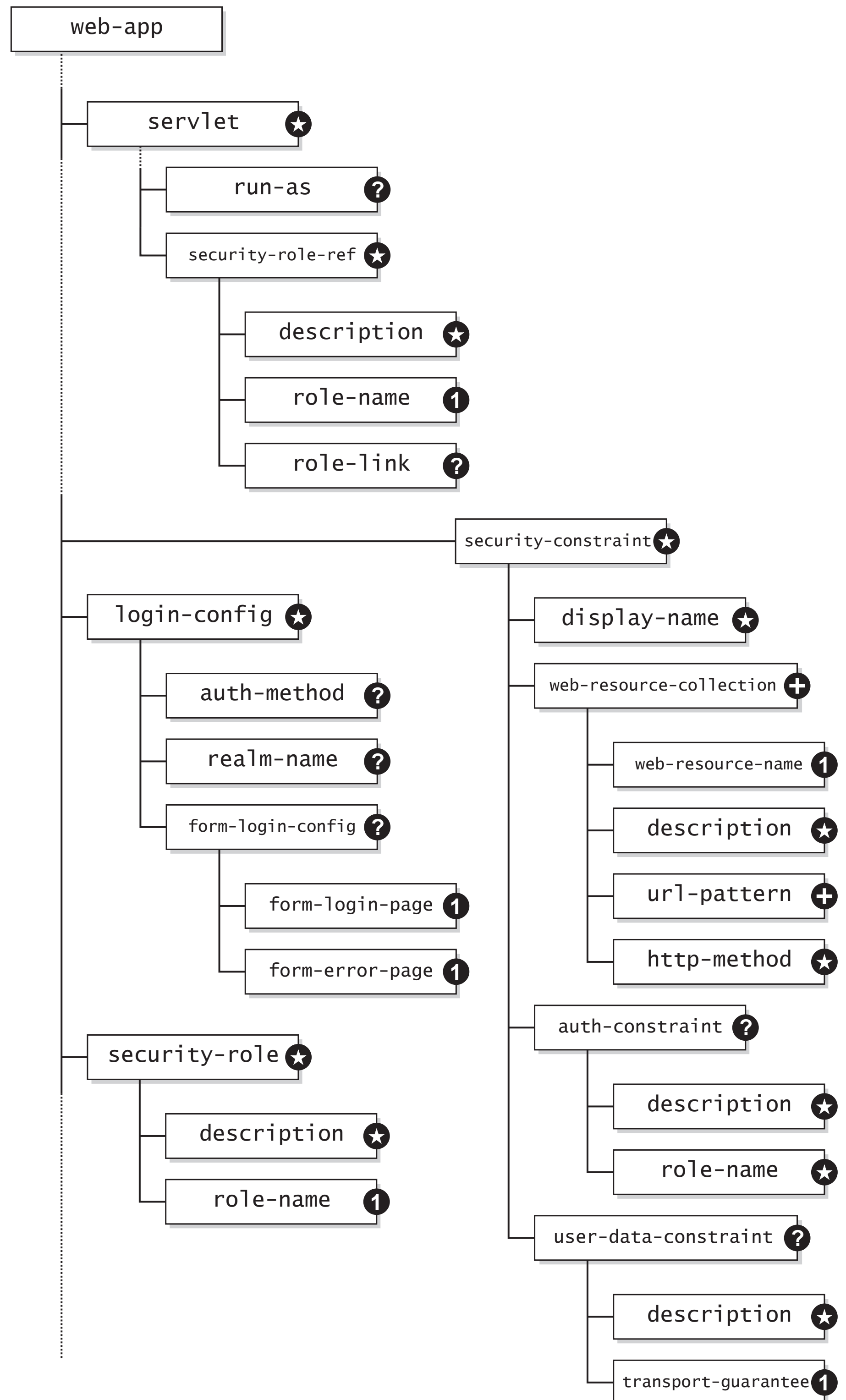
- Java Authentication and Authorization Service (JAAS) which since J2SE 1.4 comes bundled with the J2SE platform,
- Java Authorization Contract for Containers (JACC) defined in JSR-115,
- Sun Java System Application Server 8 Developer's Guide.

Whilst the latter is of course platform-specific, based on the Reference Implementation, it provides a readable insight into the methodology behind how similar servers may implement custom security. If you are already running a server from a different vendor, consult their documentation instead.

The APIs exposed by server platforms allow for the complete customisation of authentication and authorisation mechanisms using plug-ins which you create. While these are by far the best way to manage users and authorisation permissions across an entire J2EE application, there is an alternative available to individual servlets (and filters) for authorisation only. Since this interface is part of the core J2EE API, we will investigate it briefly now.

Figure 18.3

Security-Related Deployment Descriptor Schema



Programmatic Authorisation with Servlets

When we discussed the request mechanism used by the container, we came across four methods in the `HttpServletRequest` interface which at the time might not have made much sense; recapping, these were:

- `String getAuthType()`
- `String getRemoteUser()`
- `Principle getUserPrinciple()`
- `boolean isUserInRole(String)`

The first retrieves a string which describes the method used to authenticate the user. Unless you've programmed custom authentication, this will return one of the values given by these constants, also contained in the `HttpServletRequest` interface:

- **`BASIC_AUTH`**
  This has the value `BASIC`.
- **`CLIENT_CERT_AUTH`**
  This has the value `CLIENT_CERT`.
- **`DIGEST_AUTH`**
  This has the value `DIGEST`.
- **`FORM_AUTH`**
  This has the value `FORM`.

The second method, `getRemoteUser()`, returns the username of the authenticated user; if they are not logged in (i.e. have not yet been authenticated), this method returns `null`.

The third method returns a `java.security.Principal` instance which represents an entity capable of logging into a system – for example, an individual, corporation or another computer terminal. This method returns `null` if the user has not already been authenticated.

The last method tests if the currently authenticated user is in the role specified as the `String` parameter. For example:

```
isUserInRole("admins")
```

will return `true` only if the current user is configured as being in the `admins` role, or `false` otherwise. This method returns `false` if the current user has not yet been authenticated. The role name used as the parameter to this method must be declared in the deployment descriptor inside the `<role-name>` subelement of `<security-role-ref>` for the servlet(s) in which `isUserInRole()` is called, as we've already discussed.

Of these methods, the second and forth will be of most interest to us. The second, `getRemoteUser()`, can be used to find the username of the user. We can then match this against a database of user entries and perform fine-grained authorisation based on that username. The forth, `isUserInRole()`, allows us to make decisions about authorisation based on whether the user is in the specified role. Authorisation by roles is likely to be less useful to us when building additional programmatic security, since these could be declared in the deployment descriptor.

Let's look at the outline for a very simple servlet which does some authorisation based on the username:

```
public class MyAuthorisationServlet extends HttpServlet {
  /* An array of usernames which we permit access this resource */
  private static final String[] allowedUsers = new String[]{
                                              "administrator",
                                              "customer1",
                                              "clerk" };

  public void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String username = req.getRemoteUser();

    /* If the user isn't logged in, then leave now with a 401 SC */
    if(username == null) {
      resp.sendError(HttpServletResponse.SC_UNAUTHORIZED);
      return;
    }

    /* Check to see if the user is in our "allowedUsers" category */
    boolean permission = false;
    for(String au : allowedUsers) {
      if(au.equals(username)) {
        permission = true;
        break;
      }
    }
```

```
    /* If permission is still false, then the user isn't authorised */
    if(!permission) {
      resp.sendError(HttpServletResponse.SC_FORBIDDEN);
      return;
    }

    /* If we get here, the user must be authorised */
    resp.getWriter().write("This user has been authenticated " +
                           "and is authorised for this resource");
  }
}
```

The logic behind this code is explained in the comments. If the user isn't already authenticated and we want to deny them access based on this fact, then we check for `getRemoteUser()` returning `null`, return the 401 ('Unauthorised') status code and terminate processing the rest of the servlet. Otherwise we iterate through the array of permitted usernames. If any matches the current user's username, then we allow them access straight away, and execution effectively jumps to the output writer line which simply sends one line of plain text back to the client telling the user they are authorised. Otherwise, if by the time the loop finishes still no matches have been made, we send the 403 ('Forbidden') status code response and terminate processing of the servlet.

This servlet actually makes use of *both* declarative *and* programmatic security; you'll notice that nowhere have we provided a method to allow a user to login–instead, we've assumed that the container has already authenticated the user by some declarative method. All we're doing is imposing *additional programmatic* security on top of the declarative model provided by the container, since the declarative model based on roles isn't quite up to the requirements of our application. By using the basic declarative security model and adding programmatic security *only where necessary*, we keep all the benefits of container-managed security (such as single sign-on compatibility with EJBs) while being able to extend the model as appropriate.

Obviously this is a crude implementation: having usernames hard-coded in a static array ties the code to a particular application and environment, and should these names change, the servlet would need to be recompiled (I'm sure you can think of other bad practises here as well!). A much better implementation would reference persistent storage such as a database. However, this example should illustrate one way to do programmatic authorisation. If this servlet were to be implemented in the Front Controller pattern or as an Intercepting Filter, both of which we look at in the next chapter, this has the potential to be a very robust application indeed.

Revision Questions

1 **What are the benefits of container-managed security?** **(choose two)**

- A Makes an application more portable.
- B Provides authentication across multiple Web servers.
- C Provides authentication across multiple containers in the same Web server.
- D Reduces programming time.
- E The container always uses a secure connection to process login information.

2 **Which of the following are correct statements?** **(choose two)**

- A Basic Authentication uses plain text for transmission of the username and password.
- B HTTPS Client Authentication is the only mechanism not required to be supported by a J2EE-compliant Web container.
- C Every website should provide a secure login mechanism.
- D Digest Authentication can be seamlessly integrated with existing Web pages.
- E Digest Authentication uses an encrypted password for increased security.

3 **3. Match the security terms to their correct descriptions.**

Term	Description
	The system used to ensure information is consistent between two endpoints, in particular that the information isn't altered by a middleman.
Authentication	Ensures information is readable only by the intended recipients.
Authorisation	The process of determining the validity of the client's credentials
Data integrity	The process of determining whether a user has permission to access a resource.
Confidentiality	

4 **What statement can be used to authorise the user only if their role name starts with 'admin'?** **(choose one)**

A
```
if(request.getUserRoles().startsWith("admin")) {
  // authorise
}
```
B
```
if(request.getRemoteRole().startsWith("admin")) {
  // authorise
}
```
C
```
String[] roles = request.getUserRoles();
for(String role : roles) {
   if(role.startsWith("admin")) {
      // authorise
   }
}
```
D
```
if(request.isUserInRole("admin")) {
   // authorise
}
```
E None of the above.

5 **Which of the following configures Basic Authentication for use in a Web application?** **(choose one)**

A
```
<login-config>
  <method>BASIC</method>
  <realm>Private</realm>
</login-config>
```
B
```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm>Private</realm>
</login-config>
```
C
```
<login-config>
  <method-name>BASIC</method-name>
  <realm-name>Private</realm-name>
</login-config>
```
D
```
<login-config>
  <auth-method>BASIC</method-name>
  <realm-name>Private</realm-name>
</login-config>
```
E
```
<login-config>
  <method>BASIC</method >
  <realm-name>Private</realm-name>
</login-config>
```

6 **Arrange the following tags in the order they appear inside the deployment descriptor for Form Authentication:**

<auth-method>FORM</auth-method>

<form-error-page>/403.html</form-error-page>

</login-config>

</form-login-config>

<form-login-config>

<login-config>

<form-login-page>/login.html</form-login-page>

7 **Which of the following is the correct HTML form template to use when the application is using Form Authentication?** **(choose one)**

A
```
<form method="POST" action="security_check">
  <input type="text" name="user" />
  <input type="password" name="password" />
</form>
```

B
```
<form method="POST" action="security_check">
  <input type="text" name="username" />
  <input type="password" name="password" />
</form>
```

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

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

8 **What element combination inserted directly into <web-app> in the deployment descriptor declares the security role 'customer' as being used by the Web application?** **(choose one)**

A
```
<security-config>
  <role>customer</role>
</security-config>
```

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

C
```
<security-role-group>
  <role-name>customer</role-name>
</security-role-group>
```

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

E
```
<security-roles>
  <role-name>customer</role-name>
  …
</security-roles>
```

9 **Which element combination inside a <servlet> declares a mapping between the 'admin' role as declared in the servlet (to invocations of the HttpServletRequest.isUserInRole method), and the 'administrator' role declared in the application?** **(choose one)**

A
```
<role-ref>
  <name>admin</name>
  <link>administrator</link>
</role-ref>
```

B
```
<security-role-ref>
  <role-name>admin</role-name>
  <role-ref>administrator</role-ref>
</security-role-ref>
```

C
```
<security-role-ref>
  <role-name>admin</role-name>
  <role-link>administrator</role-link>
</security-role-ref>
```

D
```
<security-role-ref>
  <servlet-role>admin</servlet-role>
  <app-role>administrator</app-role>
</security-role-ref>
```

E
```
<role-ref>
  <servlet-role>admin</servlet-role>
  <app-role>administrator</app-role>
</ role-ref>
```

10 **Which of the following statements in the deployment descriptor configures the /secure path to be inaccessible to all unauthenticated users?** **(choose one)**

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

B
```
<security-constraint>
  <url-pattern>/secure/*</url-pattern>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

C
```
<security-constraint>
  <url-pattern>/secure/*</url-pattern>
  <constraint>
    <role-name>*</role-name>
  </constraint>
</security-constraint>
```

D
```
<security-constraint>
  <url-pattern>/secure/*</url-pattern>
  <role-name>*</role-name>
</security-constraint>
```

E Omit any security constraint for this resource.

11 **A website needs to integrate the design of an HTML login page with the rest of its pages. Which of the following authentication methods would be the best one to choose?** **(choose one)**

A Basic Authentication

B Digest Authentication

C Form Authentication

D HTTPS Client Authentication

12 **Which of the following methods in which classes can be used to perform programmatic security?** **(choose two)**

A getRemoteUser() in HttpSession

B isUserInRole(String) in ServletRequest

C getUserPrincipal() in HttpServletRequest

D getAuthType() in ServletRequest

E getRemoteUser() in HttpServletRequest

13 Which of the following statements in the deployment descriptor configures all resources under the /admin path to be accessible only to users in the 'admin' role? **(choose one)**

A
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/admin</url-pattern>
    <role-name>admin</role-name>
  </web-resource-collection>
</security-constraint>
```

B
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/admin</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

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

D
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <role-name>admin</role-name>
</security-constraint>
```

E
```
<security-constraint>
  <url-pattern>/admin/*</url-pattern>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

14 **Which of the following statements in the deployment descriptor configures the container to accept only secure connections when a client accesses a resource under the /filestore path?** **(choose one)**

A
```
<security-constraint>
  <url-pattern>/filestore/*</url-pattern>
  <auth-constraint>
    <transport-guarantee>SECURE</transport-guarantee>
  </auth-constraint>
</security-constraint>
```

B
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/filestore/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <transport-guarantee>SECURE</transport-guarantee>
  </auth-constraint>
</security-constraint>
```

C
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/filestore/*</url-pattern>
  </web-resource-collection>
  <data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </data-constraint>
</security-constraint>
```

D
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/filestore/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>SECURE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

E
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/filestore/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

15 **Which of the following statements in the deployment descriptor inside <security-constraint> will allow all users, even unauthenticated ones, to access a resource?** **(choose one)**

A
```
<auth-constraint>
  <role>*</role>
</auth-constraint>
```
B
```
<auth-constraint>
  <role-name>*</role-name>
</auth-constraint>
```
C
```
<auth-constraint>
  <role-name>NONE</role-name>
</auth-constraint>
```
D Omit any declarations.

16 **The exhibit shows a servlet used for authorisation. If the user's name is not equal to 'administrator', the servlet must return SC_NOT_AUTHORIZED, and must otherwise forward to the /welcome.jsp page. What is the best statement that should be used on line 7 to achieve this?** **(choose one)**

EXHIBIT

A boolean auth = req.isUserInRole("administrator");

B boolean auth = req.getUsername().equals("administrator");

C boolean auth = req.getRemoteUser().equals("administrator");

D boolean auth = (req.getRemoteUser() == null ? false : req.getRemoteUser().equals("administrator"));

E boolean auth = (req.getUserPrincipal() == null ? false : req.getUserPrincipal().getUsername().equals("administrator"));

17 **The exhibit shows a declaration in the deployment descriptor. What is the effect of this declaration?** **(choose two)**

EXHIBIT

A Provides a constraint on all resources located in the /files directory, preventing any PUT or DELETE operations from occurring.

B Restricts access to all resources in /files/* to users in the 'admin' role.

C Ensures data integrity is maintained.

D Prevents any users other than those in the 'admin' role from using the PUT or DELETE methods.

E Enforces the use of a secure connection.

18 **Which of the following allows only authenticated users to use POST requests for all resources in the application?** **(choose one)**

A
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>*</url-pattern>
    <method>POST</method>
  </web-resource-collection>
  <auth-constraint>
    <role>*</role>
  </auth-constraint>
</security-constraint>
```

B
```
<security-constraint>
  <web-resource-collection>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

C
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/*</url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

D
```
<security-constraint>
  <web-resource-collection>
    <url-pattern>*</url-pattern>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
```

Exhibits

Q.16

```
1. import javax.servlet.*;
2. import javax.servlet.http.*;
3.
4. public class MyServlet extends HttpServlet {
5.
6.     public void doGet(HttpServletRequest req, HttpServletResponse resp) {
7.         // Insert line here
8.         if(auth) {
9.             resp.sendError(HttpServletResponse.SC_NOT_AUTHORIZED);
10.            return;
11.        }
12.
13.        /* Otherwise forward */
14.        RequestDispatcher rd = req.getRequestDispatcher("/welcome.jsp");
15.        if(rd != null) {
16.            rd.forward(req,resp);
17.        }
18.    }
19.
20.}
```

Q.17

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/files/*</url-pattern>
    <http-method>PUT</http-method>
    <http-method>DELETE </http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>INTEGRAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Answers to Revision Questions

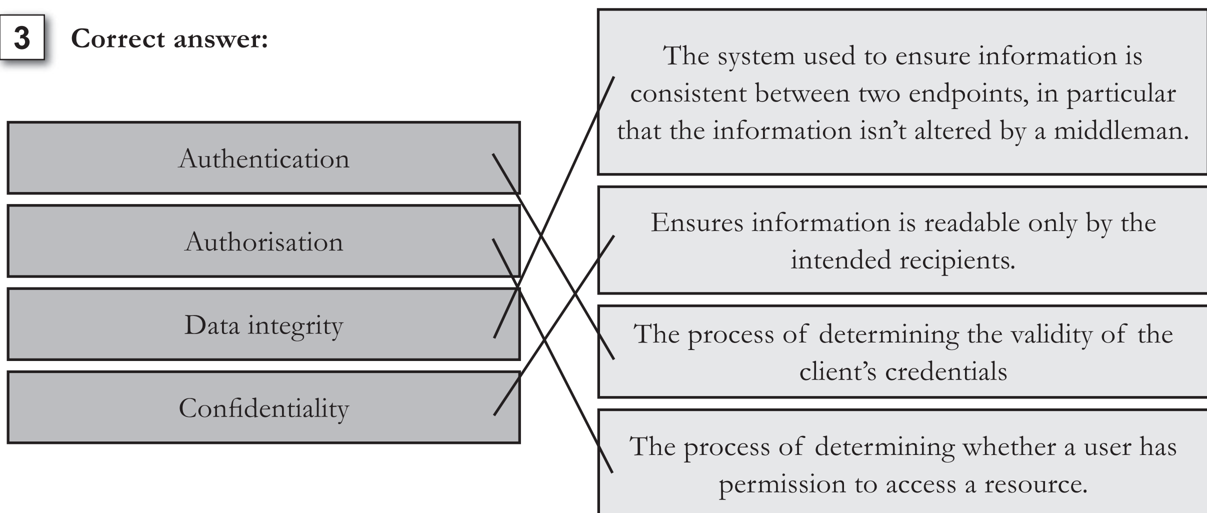
1 **Correct answers: C, D**

Note that A isn't necessarily true; only certain features list in the specification must be implemented by containers, others are optional, and there are many proprietary custom authentication mechanisms. In fact, an application developed with its own internal authentication mechanism is more portable than custom authentication mechanisms which integrate with specific containers. B is false, in general, as authentication isn't carried across different Web servers for security reasons. C is true, allowing a Web application to do authentication and an EJB container to use that authentication information. D is certainly true; using declarative security takes only a few minutes to configure in the deployment descriptor and in your server's configuration. E is incorrect; you can *opt* to have a secure connection but it is not mandatory.

2 **Correct answers: A, E**

B is incorrect because all J2EE-compliant containers must support HTTPS Client Authentication; Digest Authentication is the only optional mechanism. C is incorrect because often it is inappropriate to use SSL certificates – for example, a public forum might use a login only to track which members create each thread, and to encourage people to register; since this data isn't 'sensitive', it is unlikely that the author would want to go to the trouble and expense of obtaining a Public Key Certificate and configuring SSL. D is incorrect because this method causes a browser-dependent pop-up window to be used; Form Authentication is the only mechanism which can be integrated with an existing website design.

3 **Correct answer:**



4 **Correct answer: E**

In fact, there is no portable way to determine the roles the user belongs to; the best hope is to access the Principal returned from getUserPrincipal(), but the exact object returned and the methods available on it will be container-dependent. Note that D would be correct if we just wanted to test if the user was in the role 'admin', rather than testing if any of their roles start with that token.

5 **Correct answer: D**

6 **Correct answer:**

<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
<form-login-page>/login.html</form-login-page>
<form-error-page>/403.html</form-error-page>
</form-login-config>
</login-config>

7 **Correct answer: C**

Note that the path to post the form to, j_security_check, is relative to the current path and not to an absolute resource. When the request is submitted by the client, the container checks to see if the request URI ends with j_security_check; if so its authentication mechanism is automatically engaged. All parameter names start with a 'j_'.

8 **Correct answer: D**

Note that only one <role-name> may be declared per <security-role>, and it is the only useful child element–unless additional subelements of <security-role> are added at a later date, there is practically always vast redundancy here.

9 **Correct answer: C**

10 **Correct answer: A**

A is the correct syntax; B to D are all incorrect. Note that E is incorrect as this would allow *all* users, even unauthenticated ones, to gain access to /secure.

11 **Correct answer: C**

12 **Correct answers: C, E**

Note that all the programmatic security methods are found in HttpServletRequest.

13 **Correct answer: C**

Note that only B and C have the correct syntax, but B maps only to the single /admin resource, whereas C maps to all resources matching /admin/*, which are all resources inside the admin directory.

14 **Correct answer: E**

15 **Correct answer: D**

The only way to allow all users through is to *not* declare any elements; declaring a <role-name>*</role-name> is equivalent to allowing *any authenticated* user to access the resource.

16 **Correct answer: D**

Note the requirement to check whether getRemoteUser() returns null, otherwise trying to invoke equals() on that will cause a NullPointerException. A is incorrect, as it attempts to check the role the user is in, and not the username. B is incorrect because the method doesn't exist. C isn't the best statement because it could throw the NullPointerException. E is incorrect; there is no method called getUsername() on java.security.Principal.

17 **Correct answers: C, D**

A is incorrect, as this constraint allows PUT or DELETE operations *from users in the admin role* (it doesn't prevent all operations from occurring); B is incorrect, as it doesn't actually restrict access – clients using GET or POST could still access the resources; this constraint is designed to prevent the use of the potentially more dangerous HTTP methods. E is incorrect, as INTEGRAL ensures that some form of data integrity test (such as digests) is being used, but doesn't mandate that a completely secure connection such as SSL is required.

18 **Correct answer: C**

Note that using '*' for a <url-pattern> is invalid; you should use /*. In addition, note that there must be one or more <url-pattern>s per <web-resource-collection>; it is insufficient to provide only an <http-method>.