
CHAPTER 15

Securing JAX-RS

Many RESTful web services will want secure access to the data and functionality they provide. This is especially true for services that will be performing updates. They will want to prevent sniffers on the network from reading their messages. They may also want to fine-tune which users are allowed to interact with a specific service and disallow certain actions for specific users. The Web and the umbrella specification for JAX-RS, Java EE, provide a core set of security services and protocols that you can leverage from within your RESTful web services. These include:

Authentication

Authentication is about validating the identity of a client that is trying to access your services. It usually involves checking to see if the client has provided an existing user with valid credentials, such as a password. The Web has a few standardized protocols you can use for authentication. Java EE, specifically your servlet container, has facilities to understand and configure these Internet security authentication protocols.

Authorization

Once a client is authenticated, it will want to interact with your RESTful web service. Authorization is about deciding whether or not a certain user is allowed to access and invoke on a specific URI. For example, you may want to allow write access (PUT/POST/DELETE operations) for one set of users and disallow it for others. Authorization is not part of any Internet protocol and is really the domain of your servlet container and Java EE.

Encryption

When a client is interacting with a RESTful web service, it is possible for hostile individuals to intercept network packets and read requests and responses if your HTTP connection is not secure. Sensitive data should be protected with cryptographic services like SSL. The Web defines the HTTPS protocol to leverage SSL and encryption.

JAX-RS has a small programmatic API for interacting with servlet and Java EE security, but enabling security in a JAX-RS environment is usually an exercise in configuration and applying annotation metadata.

Beyond Java EE, servlet, and JAX-RS security configuration and APIs, there's a few areas these standards don't cover. One area is digital signatures and encryption of the HTTP message body. Your representations may be passing through untrusted intermediaries and signing or encrypting the message body may add some extra protection for your data. There's also advanced authentication protocols like OAuth, which allow you to make invocations on services on behalf of other users.

This chapter first focuses on the various web protocols used for authentication in a standard, vanilla Java EE, and servlet environment. You'll learn how to configure your JAX-RS applications to use standard authentication, authorization, and encryption. Next you'll learn about various formats you can use to digitally sign or encrypt message bodies. Finally, we'll talk about the OAuth protocol and how you can use it within your applications.

Authentication

When you want to enforce authentication for your RESTful web services, the first thing you have to do is decide which authentication protocol you want to use. Internet protocols for authentication vary in their complexity and their perceived reliability. In Java land, most servlet containers support the protocols of Basic Authentication, Digest Authentication, and authentication using X.509 certificates. Let's look into how each of these protocols works.

Basic Authentication

Basic Authentication is the simplest protocol available for performing authentication over HTTP. It involves sending a Base 64-encoded username and password within a request header to the server. The server checks to see if the username exists within its system and verifies the sent password. To understand the details of this protocol, let's look at an example.

Say an unauthorized client tries to access one of our secure RESTful web services:

```
GET /customers/333 HTTP/1.1
```

Since the request does not contain any authentication information, the server would reply with an HTTP response of:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="CustomerDB Realm"
```

The 401 response tells the client that it is not authorized to access the URI it tried to invoke on. The `WWW-Authenticate` header specifies which authentication protocol the

client should use. In this case, Basic means Basic Authentication should be used. The realm attribute identifies a collection of secured resources on a website. The client can use the realm information to match against a username and password that is required for this specific URI.

To perform authentication, the client must send a request with the Authorization header set to a Base 64-encoded string of our username and a colon character, followed by the password. If our username is bburke and our password geheim, the Base 64-encoded string of bburke:geheim will be YmJ1cmtlOmdlaGVpbQ==. Put all this together, and our authenticated GET request would look like this:

```
GET /customers/333 HTTP/1.1
Authorization: Basic YmJ1cmtlOmdlaGVpbQ==
```

The client needs to send this Authorization header with each and every request it makes to the server.

The problem with this approach is that if this request is intercepted by a hostile entity on the network, the hacker can easily obtain the username and password and use it to invoke its own requests. Using an encrypted HTTP connection, HTTPS, solves this problem. With an encrypted connection, a rogue programmer on the network will be unable to decode the transmission and get at the Authorization header. Still, security-paranoid network administrators are very squeamish about sending passwords over the network, even if they are encrypted within SSL packets.

Digest Authentication

Although not used much anymore, Digest Authentication was invented so that clients would not have to send clear-text passwords over HTTP. It involves exchanging a set of secure MD5 hashes of the username, password, operation, URI, and optionally the hash of the message body itself. The protocol starts off with the client invoking an insecure request on the server:

```
GET /customers/333 HTTP/1.1
```

Since the request does not contain any authentication information, the server replies with an HTTP response of:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="CustomerDB Realm",
                  qop="auth,auth-int",
                  nonce="12dcde223152321ab99cd",
                  opaque="aa9321534253bcd00121"
```

Like before, a 401 error code is returned along with a WWW-Authenticate header. The nonce and opaque attributes are special server-generated keys that will be used to build the subsequent authenticated request.

Like Basic Authentication, the client uses the `Authorization` header, but with digest-specific attributes. Here's a request example:

```
GET /customers/333 HTTP/1.1
Authorization: Digest username="bburke",
                    realm="CustomerDB Realm",
                    nonce="12dcde223152321ab99cd",
                    uri="/customers/333",
                    qop="auth",
                    nc=00000001,
                    cnonce="43fea",
                    response="11132fffdeab993421",
                    opaque="aa9321534253bcd00121"
```

The `nonce` and `opaque` attributes are a copy of the values sent with the earlier `WWW-Authenticate` header. The `uri` attribute is the base URI you are invoking on. The `nc` attribute is a request counter that should be incremented by the client with each request. This prevents hostile clients from replaying a request. The `cnonce` attribute is a unique key generated by the client and can be anything the client wants. The `response` attribute is where all the meat is. It is a hash value generated with the following pseudocode:

```
H1 = md5("username:realm:password")
H2 = md5("httpmethod:uri")
response = md5(H1 + ":nonce:nc:cnonce:qop:" + H2)
```

If our username is `bburke` and our password `geheim`, the algorithm will resolve to this pseudocode:

```
H1 = md5("bburke:CustomerDB Realm:geheim")
H2 = md5("GET:/customers/333")
response = md5(H1 + ":12dcde223152321ab99cd:00000001:43fea:auth:" + H2)
```

When the server receives this request, it builds its own version of the response hash using its stored, secret values of the username and password. If the hashes match, the user and its credentials are valid.

One advantage of this approach is that the password is never used directly by the protocol. For example, the server doesn't even need to store clear-text passwords. It can instead initialize its authorization store with prehashed values. Also, since request hashes are built with a `nonce` value, the server can expire these `nonce` values over time. This, combined with a request counter, can greatly reduce replay attacks.

The disadvantage to this approach is that unless you use HTTPS, you are still vulnerable to man-in-the-middle attacks, where the middleman can tell a client to use Basic Authentication to obtain a password.

Client Certificate Authentication

When you buy things or trade stocks on the Internet, you use the HTTPS protocol to obtain a secure connection with the server. HTTPS isn't only an encryption mechanism—it can also be used for authentication. When you first interact with a secure website, your browser receives a digitally signed certificate from the server that identifies it. Your browser verifies this certificate with a central authority like VeriSign. This is how you guarantee the identity of the server you are interacting with and make sure you're not dealing with some man-in-the-middle security breach.

HTTPS can also perform two-way authentication. In addition to the client receiving a signed digital certificate representing the server, the server can receive a certificate that represents and identifies the client. When a client initially connects to a server, it exchanges its certificate and the server matches it against its internal store. Once this link is established, there is no further need for user authentication, since the certificate has already positively identified the user.

Client Certificate Authentication is perhaps the most secure way to perform authentication on the Web. The only disadvantage of this approach is the managing of the certificates themselves. The server must create a unique certificate for each client that wants to connect to the service. From the browser/human perspective, this can be a pain, as the user has to do some extra configuration to interact with the server.

Authorization

While authentication is about establishing and verifying user identity, authorization is about permissions. Is my user allowed to perform the operation it is invoking? None of the standards-based Internet authorization protocols discussed so far deals with authorization. The server and application know the permissions for each user and do not need to share this information over a communication protocol. This is why authorization is the domain of the server and application.

JAX-RS relies on the servlet and Java EE specifications to define how authorization works. Authorization is performed in Java EE by associating one or more roles with a given user and then assigning permissions based on that role. While an example of a user might be “Bill” or “Monica,” roles are used to identify a group of users—for instance, “administrator,” “manager,” or “employee.” You do not assign access control on a per-user basis, but rather on a per-role basis.

Authentication and Authorization in JAX-RS

To enable authentication, you need to modify the *WEB-INF/web.xml* deployment descriptor of the WAR file your JAX-RS application is deployed in. You enable authorization through XML or by applying annotations to your JAX-RS resource classes. To

see how all this is put together, let's do a simple example. We have a customer database that allows us to create new customers by posting an XML document to the JAX-RS resource located by the `@Path("/customers")` annotation. This service is deployed by a scanned Application class annotated with `@ApplicationPath("/services")` so the full URI is `/services/customers`. We want to secure our customer service so that only administrators are allowed to create new customers. Let's look at a full XML-based implementation of this example:

```
<?xml version="1.0"?>
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>customer creation</web-resource-name>
      <url-pattern>/services/customers</url-pattern>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>jaxrs</realm-name>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
</web-app>
```

The `<login-config>` element defines how we want our HTTP requests to be authenticated for our entire deployment. The `<auth-method>` subelement can be BASIC, DIGEST, or CLIENT_CERT. These values correspond to Basic, Digest, and Client Certificate Authentication, respectively.

The `<login-config>` element doesn't turn on authentication. By default, any client can access any URL provided by your web application with no constraints. To enforce authentication, you must specify a URL pattern you want to secure. In our example, we use the `<url-pattern>` element to specify that we want to secure the `/services/customers` URL. The `<http-method>` element says that we only want to secure POST requests to this URL. If we leave out the `<http-method>` element, all HTTP methods are secured. In our example, we only want to secure POST requests, so we must define the `<http-method>` element.

Next, we have to specify which roles are allowed to POST to `/services/customers`. In the `web.xml` file example, we define an `<auth-constraint>` element within a

`<security-constraint>`. This element has one or more `<role-name>` elements that define which roles are allowed to access the defined constraint. In our example, applying this XML only gives the `admin` role permission to access the `/services/customers` URL.

If you set a `<role-name>` of `*` instead, any user would be able to access the constrained URL. Authentication with a valid user would still be required, though. In other words, a `<role-name>` of `*` means anybody who is able to log in can access the resource.

Finally, there's an additional bit of syntactic sugar we need to specify in *web.xml*. For every `<role-name>` we use in our `<auth-constraints>` declarations, we must define a corresponding `<security-role>` in the deployment descriptor.

There is a minor limitation when you're declaring `<security-constraints>` for JAX-RS resources. The `<url-pattern>` element does not have as rich an expression syntax as JAX-RS `@Path` annotation values. In fact, it is extremely limited. It supports only simple wildcard matches via the `*` character. No regular expressions are supported. For example:

- `/*`
- `/foo/*`
- `*.txt`

The wildcard pattern can only be used at the end of a URL pattern or to match file extensions. When used at the end of a URL pattern, the wildcard matches every character in the incoming URL. For example, `/foo/*` would match any URL that starts with `/foo`. To match file extensions, you use the format `*.<suffix>`. For example, the `*.txt` pattern matches any URL that ends with `.txt`. No other uses of the wildcard character are permitted in URL patterns. For example, here are some illegal expressions:

- `/foo*/bar`
- `/foo/*.txt`

Enforcing Encryption

By default, the servlet specification will not require access over HTTPS to any user constraints you declare in your *web.xml* file. If you want to enforce HTTPS access for these constraints, you can specify a `<user-data-constraint>` within your `<security-constraint>` definitions. Let's modify our previous example to enforce HTTPS:

```
<web-app>
...

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

```

        <web-resource-name>customer creation</web-resource-name>
        <url-pattern>/services/customers</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
...
</web-app>

```

All you have to do is declare a `<transport-guarantee>` element within a `<user-data-constraint>` that has a value of `CONFIDENTIAL`. If a user tries to access the URL pattern with HTTP, she will be redirected to an HTTPS-based URL.

Authorization Annotations

Java EE defines a common set of annotations that can define authorization metadata. The JAX-RS specification suggests, but does not require, vendor implementations to support these annotations in a non-Java EE 6 environment. These annotations live in the `javax.annotation.security` package and are `@RolesAllowed`, `@DenyAll`, `@PermitAll`, and `@RunAs`.

The `@RolesAllowed` annotation defines the roles permitted to execute a specific operation. When placed on a JAX-RS annotated class, it defines the default access control list for all HTTP operations defined in the JAX-RS class. If placed on a JAX-RS method, the constraint applies only to the method that is annotated.

The `@PermitAll` annotation specifies that any authenticated user is permitted to invoke your operation. As with `@RolesAllowed`, you can use this annotation on the class to define the default for the entire class or you can use it on a per-method basis. Let's look at an example:

```

@Path("/customers")
@RolesAllowed({"ADMIN", "CUSTOMER"})
public class CustomerResource {

    @GET
    @Path("/{id}")
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id) {...}

    @RolesAllowed("ADMIN")
    @POST
    @Consumes("application/xml")
    public void createCustomer(Customer cust) {...}
}

```

```

        @PermitAll
        @GET
        @Produces("application/xml")
        public Customer[] getCustomers() {}
    }

```

Our `CustomerResource` class is annotated with `@RolesAllowed` to specify that, by default, only ADMIN and CUSTOMER users can execute HTTP operations and paths defined in that class. The `getCustomer()` method is not annotated with any security annotations, so it inherits this default behavior. The `createCustomer()` method is annotated with `@RolesAllowed` to override the default behavior. For this method, we only want to allow ADMIN access. The `getCustomers()` method is annotated with `@PermitAll`. This overrides the default behavior so that any authenticated user can access that URI and operation.

In practice, I don't like to specify security metadata using annotations. Security generally does not affect the behavior of the business logic being executed and falls more under the domain of configuration. Administrators may want to add or remove role constraints periodically. You don't want to have to recompile your whole application when they want to make a simple change. So, if I can avoid it, I usually use *web.xml* to define my authorization metadata.

There are some advantages to using annotations, though. For one, it is a workaround for doing fine-grained constraints that are just not possible in *web.xml* because of the limited expression capabilities of `<url-pattern>`. Also, because you can apply constraints per method using these annotations, you can fine-tune authorization per media type. For example:

```

@Path("/customers")
public class CustomerService {

    @GET
    @Produces("application/xml")
    @RolesAllowed("XML-USERS")
    public Customer getXmlCustomers() {}

    @GET
    @Produces("application/json")
    @RolesAllowed("JSON-USERS")
    public Customer getJsonCustomers() {}
}

```

Here we only allow XML-USERS to obtain `application/xml` content and JSON-USERS to obtain `application/json` content. This might be useful for limiting users in getting data formats that are expensive to create.

Programmatic Security

The security features defined in this chapter have so far focused on declarative security metadata, or metadata that is statically defined before an application even runs. JAX-RS also has a small programmatic API for gathering security information about a secured request. Specifically, the `javax.ws.rs.core.SecurityContext` interface has a method for determining the identity of the user making the secured HTTP invocation. It also has a method that allows you to check whether or not the current user belongs to a certain role:

```
public interface SecurityContext {  
  
    public Principal getUserPrincipal();  
    public boolean isUserInRole(String role);  
    public boolean isSecure();  
    public String getAuthenticationScheme();  
}
```

The `getUserPrincipal()` method returns a standard Java Standard Edition (SE) `javax.security.Principal` security interface. A `Principal` object represents the individual user who is currently invoking the HTTP request. The `isUserInRole()` method allows you to determine whether the current calling user belongs to a certain role. The `isSecure()` method returns true if the current request is a secure connection. The `getAuthenticationScheme()` tells you which authentication mechanism was used to secure the request. BASIC, DIGEST, CLIENT_CERT, and FORM are typical values returned by this method. You get access to a `SecurityContext` instance by injecting it into a field, setter method, or resource method parameter using the `@Context` annotation.

Let's examine this security interface with an example. Let's say we want to have a security log of all access to a customer database by users who are not administrators. Here is how it might look:

```
@Path("/customers")  
public class CustomerService {  
  
    @GET  
    @Produces("application/xml")  
    public Customer[] getCustomers(@Context SecurityContext sec) {  
  
        if (sec.isSecure() && !sec.isUserInRole("ADMIN")) {  
            logger.log(sec.getUserPrincipal() +  
                " accessed customer database.");  
        }  
        ...  
    }  
}
```

In this example, we inject the `SecurityContext` as a parameter to our `getCustomer()` JAX-RS resource method. We use the method `SecurityContext.isSecure()` to determine whether or not this is an authenticated request. We then use the method `SecurityContext.isUserInRole()` to find out if the caller is an ADMIN or not. Finally, we print out to our audit log.

With the introduction of the filter API in JAX-RS 2.0, you can implement the `SecurityContext` interface and override the current request's `SecurityContext` via the `ContainerRequestContext.setSecurityContext()` method. What's interesting about this is that you can implement your own custom security protocols. Here's an example:

```
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.HttpHeaders;

@PreMatching
public class CustomAuth implements ContainerRequestFilter {
    protected MyCustomerProtocolHandler customProtocol = ...;

    public void filter(ContainerRequestContext requestContext) throws IOException
    {
        String authHeader = request.getHeaderString(HttpHeaders.AUTHORIZATION);
        SecurityContext newSecurityContext = customProtocol.validate(authHeader);
        requestContext.setSecurityContext(authHeader);
    }
}
```

This filter leaves out a ton of detail, but hopefully you get the idea. It extracts the Authorization header from the request and passes it to the `customProtocol` service that you have written. This returns an implementation of `SecurityContext`. You override the default `SecurityContext` with this variable.

Client Security

The JAX-RS 2.0 specification didn't do much to define a common client security API. What's weird is that while it has a standard API for rarely used protocols like two-way SSL with client certificates, it doesn't define one for simple protocols like . Instead, you have to rely on the vendor implementation of JAX-RS to provide these security features. For example, the RESTEasy framework provides a `ContainerRequestFilter` you can use to enable Basic Authentication:

```
import org.jboss.resteasy.client.jaxrs.BasicAuthentication;

Client client = Client.newClient();
client.register(new BasicAuthentication("username", "password"));
```

You construct the `BasicAuthentication` filter with the username and password you want to authenticate with. That's it. Other JAX-RS implementations might have other mechanisms for doing this.

JAX-RS 2.0 does have an API for enabling two-way SSL with client certificates. The `ClientBuilder` class allows you to specify a `java.security.KeyStore` that contains the client certificate you want to use to authenticate:

```
abstract class ClientBuilder {  
    public ClientBuilder keyStore(final KeyStore keyStore, final String password)  
}
```

Alternatively, it has methods to create your own `SSLContext`, but creating one is quite complicated and beyond the scope of this book.

Verifying the Server

HTTPS isn't only about encrypting your network connection, it is also about establishing trust. One aspect of this on the client side is verifying that the server you are talking to is the actual server you want to talk to and not some middleman on the network that is spoofing it. With most secure Internet servers, you do not have to worry about establishing trust because the server's certificates are signed by a trusted authority like VeriSign, and your JAX-RS client implementation will know how to verify certificates signed by these authorities.

In some cases, though, especially in test environments, you may be dealing with servers whose certificates are self-signed or signed by an unknown authority. In this case, you must obtain a truststore that contains the server certificates you trust and register them with the Client API. The `ClientBuilder` has a method for this:

```
abstract class ClientBuilder {  
    public abstract ClientBuilder trustStore(final KeyStore trustStore);  
}
```

How you initialize and populate the `KeyStore` is beyond the scope of this book.

OAuth 2.0

OAuth 2.0 is an authentication protocol that allows an entity to gain access to a user's data in a secure manner without having to know the user's credentials.¹ A typical example is a news site like `cnn.com`. You're reading an interesting political editorial and want to voice your opinion on the article in its comment section. To do this, though, you have to tell CNN who you are and what your email address is. It gives you the option of logging in via your Google or Facebook account. You are forwarded to Google and

1. For more information, see [the OAuth 2.0 Authorization Framework](#).

log in there. You grant CNN permission to ask Google who you are and what your email address is, and then you are forwarded back to `cnn.com` so that you can enter in your comment. Through this interaction CNN is granted an access token, which it then uses to obtain information about you via a separate HTTP request.

Here's how it works:

1. The CNN website redirects your browser to Google's login page. This redirect sets a special `cnn.com` session cookie that contains a randomly generated value. The redirect URL contains `client_id`, `state`, and `redirect_uri`. The `client_id` is the Google username CNN has registered with Google.com. The `state` parameter is the same value that was set in the session cookie. The `redirect_uri` is a URL you want Google to redirect the browser back to after authentication. A possible redirect URL in this scenario thus would be `http://googleapis.com/oauth?client_id=cnn&state=23423423123412352314&redirect_uri=http%3A%2F%2Fcnn.com`.
2. You enter your username and password on Google's login page. You then are asked if you will grant CNN access to your personal information.
3. If you say yes, Google generates an access code and remembers the `client_id` and `redirect_uri` that was sent in the original browser redirect.
4. Google redirects back to CNN.com using the `redirect_uri` sent by CNN's initial redirect. The redirect URL contains the original `state` parameter you forwarded along with a `code` parameter that contains the access code: `http://cnn.com/state=23423423123412352314&code=0002222`.
5. With this redirection, CNN will also get the value of the special cookie that it set in step 1. It checks the value of this cookie with the `state` query parameter to see if they match. It does this check to make sure that it initiated the request and not some rogue site.
6. The CNN server then extracts the `code` query parameter from the redirect URL. In a separate authenticated HTTP request to Google, it posts this access code. Google.com authenticates that CNN is sending the request and looks up the access code that was sent. If everything matches up, it sends back an access token in the HTTP response.
7. CNN can now make HTTP requests to other Google services to obtain information it wants. It does this by passing the token in an `Authorization` header with a value of `Bearer` plus the access token. For example:

```
GET /contacts?user=billburke
Host: contacts.google.com
Authorization: Bearer 2a2345234236122342341bc234123612341234123412adff
```

In reality, sites like Google, Facebook, and Twitter don't use this protocol exactly. They all put their own spin on it and all have a little bit different way of implementing this protocol. The same is true of OAuth libraries. While the core of what they do will follow the protocol, there will be many custom attributes to each library. This is because the OAuth specification is more a set of detailed guidelines rather than a specific protocol set in stone. It leaves out details like how a user or OAuth client authenticates or what additional parameters must be sent. So using OAuth may take a bunch of integration work on your part.

There are many different Java frameworks out there that can help you turn your applications into OAuth providers or help you integrate with servers that support OAuth authentication. This is where I make my own personal plug. In 2013, I started a new project at Red Hat called Keycloak. It is a complete end-to-end solution for OAuth and SSO. It can also act as a social broker with social media sites like Google and Facebook to make leveraging social media easier. Please check us out at <http://www.keycloak.org>.

Signing and Encrypting Message Bodies

Sometimes you have RESTful clients or services that may have to send or receive HTTP messages from unknown or untrusted intermediaries. A great example of an intermediary is Twitter. You post tweets to Twitter through the Twitter REST API, and one or more people can receive these tweets via Twitter. What if a tweet receiver wanted to verify that the tweet originator is who he says he is? Or what if you wanted to post encrypted tweets through Twitter that only trusted receivers could decode and read? This interaction is different from HTTPS in that HTTPS is a trusted SSL socket connection between one client and one server. For the Twitter example, we're sending a representation that is retransmitted via a totally different HTTP request involving different clients and servers. Digitally signing or encrypting the representation gives you the protection you need in this retransmission scenario.

Digital Signatures

Java developers are intimately familiar with the `HashMap` class. The way maps work is that a semi-unique hash code is generated for the key you are storing in the map. The key's hash code is used as an array index to quickly look up a value in the map. Under the covers, a digital signature is simply an encrypted hash code of the piece of data you are transmitting.

While a shared secret can be used to generate and verify a digital signature, the best approach is to use an asymmetric key pair: in other words, a private and public key. The signer creates the digital signature of the message using its private key. It then publishes its public key to the world. The receiver of the message uses the public key to verify the signature. If you use the right hash and encryption algorithms, it is virtually impossible

to derive the private key of the sender or fake the signatures. I'm going to go over two methods you can use to leverage digital signatures in your RESTful web services.

DKIM/DOSETA

DomainKeys Identified Mail (DKIM)² is a digital signature protocol that was designed for email. Work is also being done to apply this header to protocols other than email (e.g., HTTP) through the DOSETA³ specifications. DKIM is simply a request or response header that contains a digital signature of one or more headers of the message and the content. What's nice about DKIM is that its header is self-contained and not part of the transmitted representation. So if the receiver of an HTTP message doesn't care about digital signatures, it can just ignore the header.

The format of a DKIM header is a semicolon-delimited list of name/value pairs. Here's an example:

```
DKIM-Signature: v=1;
                a=rsa-sha256;
                d=example.com;
                s=burke;
                c=simple/simple;
                h=Content-Type;
                x=0023423111111;
                bh=2342322111;
                b=M232234=
```

While it's not *that* important to know the structure of the header, here's an explanation of each parameter:

v

Protocol version. Always 1.

a

Algorithm used to hash and sign the message. RSA signing and SHA256 hashing is the only supported algorithm at the moment by RESTEasy.

d

Domain of the signer. This is used to identify the signer as well as discover the public key to use to verify the signature.

s

Selector of the domain. Also used to identify the signer and discover the public key.

2. For more information, see <http://dkim.org>

3. For more information, see [the DomainKeys Security Tagging](#).

- c Canonical algorithm. Only simple/simple is supported at the moment. Basically, this allows you to transform the message body before calculating the hash.
- h Semicolon-delimited list of headers that are included in the signature calculation.
- x When the signature expires. This is a numeric long value of the time in seconds since epoch. Allows the signer to control when a signed message's signature expires.
- t Timestamp of signature. Numeric long value of the time in seconds since epoch. Allows the verifier to control when a signature expires.
- bh Base 64-encoded hash of the message body.
- b Base 64-encoded signature.

What's nice about DKIM is that you can include individual headers within your digital signature of the message. Usually Content-Type is included.

To verify a signature, you need a public key. DKIM uses DNS text records to discover a public key. To find a public key, the verifier concatenates the selector (s parameter) with the domain (d parameter):

```
<selector>._domainKey.<domain>
```

It then takes that string and does a DNS request to retrieve a TXT record under that entry. In our previous example, `burke._domainKey.example.com` would be used as the lookup string.

This is a very interesting way to publish public keys. For one, it becomes very easy for verifiers to find public keys, as there's no real central store that is needed. Second, DNS is an infrastructure IT knows how to deploy. Third, signature verifiers can choose which domains they allow requests from. If you do not want to be dependent on DNS, most DKIM frameworks allow you to define your own mechanisms for discovering public keys.

Right now, support for DKIM in the Java world is quite limited. The RESTEasy framework does have an API, though, if you're interested in using it.

JOSE JWS

JOSE JSON Web Signature is a self-contained signature format that contains both the message you want to sign as well as the digital signature of the message.⁴ The format is completely text-based and very compact. It consists of three Base 64-encoded strings delimited by a . character. The three encoded strings in the JOSE JWS format are a JSON header describing the message, the actual message that is being transmitted, and finally the digital signature of the message. The media type for JOSE JWS is `application/jose+json`. Here's what a full HTTP response containing JWS might look like:

```
HTTP/1.1 200 OK
Content-Type: application/jose+json

eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxLLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3X0iZj5RZmh7
AAuHIm4Bh-0Qc_LF5YKt_08W2Fp5jujGbdS9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB--f_nZLgrnbyTyWz075vRK5h6xBARLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383Lc0Ln6_d0--xi12jzDwusC-e0kHWEsqtfZESc6BfI7no0Pqv
hJ1phCnvWh6IeYI2w9Q0YEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywLVmtVrB
p0igcN_IoypG1UPQGe77Rw
```

Let's break down how an encoded JWS is created. The first encoded part of the format is a JSON header document that describes the message. Minimally, it has an `alg` value that describes the algorithm used to sign the message. It also often has a `cty` header that describes the Content-Type of the message signed. For example:

```
{
  "alg" : "RS256",
  "cty" : "application/xml"
}
```

The second encoded part of the JWS format is the actual content you are sending. It can be anything you want, like a simple text message, a JSON or XML document, or even an image or audio file; really, it can be any set of bytes or formats you want to transmit.

Finally, the third encoded part of the JWS format is the encoded digital signature of the content. The algorithm used to create this signature should match what was described in the header part of the JWS message.

What I like about JOSE JWS is that it is HTTP-header-friendly. Since it is a simple ASCII string, you can include it within HTTP header values. This allows you to send JSON or even binary values within an HTTP header quite easily.

4. For more information, see [the JSON Web Signature](#).

Encrypting Representations

While you can rely on HTTPS to encrypt your HTTP requests and responses, I noted earlier that you may have some scenarios where you want to encrypt the HTTP message body of your requests and responses. Specifically, consider scenarios where you are sending messages to a public or untrusted intermediary. While there are a few standard ways to encrypt your representations, my favorite is JOSE JSON Web Encryption.⁵

JWE is a compact text format. It consists of five Base 64–encoded strings delimited by a . character. The first encoded string is a JSON header describing what is being transmitted. The second encoded string is an encrypted key used to encrypt the message. The third is the initialization vector used to encrypt the first block of bytes. The fourth is the actual encrypted message. And finally, the fifth is some extra metadata used to validate the message. The media type for JOSE JWE is `application/jose+json`. So here's what a full an HTTP response containing JWE might look like:

```
HTTP/1.1 200 OK
Content-Type: application/jose+json

eyJhbGciOiJSU0ExXzUuLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIQguC7IuEvf_NPVaXsGmOL0mwvc1GyqLIK0K1nN94nHPoltGRhWhw7Zx0-kFm
1N3n8LE9XSH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzo0egEI-8E66jX2E4zyJKx-YxzzIItrZC5hLRirb6Y5CL_p-ko3YvkkysZIF
NPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng80tvzLV7eLprCbuPhcCdZ6XDP0_F8
rkXds2vE4X-nc0IM8hAYHHi29NX0mckIRaD0-D-ljQTP-cFPgwCp6X-nZZd90HBv
-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGLjb3RoZQ.
KDLTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9w0GY.
9hH0vgRFYgPnAH0d8stkvw
```

Like JSON Web Signatures, the encoded header for JWE is a simple JSON document that describes the message. Minimally, it has an `alg` value that describes the algorithm used to encrypt the message and a `enc` value that describes the encryption method. It often has a `cty` header that describes the Content-Type of the message signed. For example:

```
{
  "alg": "RSA1_5",
  "enc": "A128CBC-HS256",
  "cty": "application/xml"
}
```

The algorithms you can use for encryption come in two flavors. You can use a shared secret (i.e., a password) to encrypt the data, or you can use an asymmetric key pair (i.e., a public and private key).

5. For more information, see [the JSON Web Encryption](#).

As for the other encoded parts of the JWE format, these are really specific to the algorithm you are using and something I'm not going to go over.

As with JWS, the reason I like JWE is that it is HTTP-header-friendly. If you want to encrypt an HTTP header value, JWE works quite nicely.

Wrapping Up

In this chapter, we discussed a few of the authentication protocols used on the Internet—specifically, Basic, Digest, and Client Certificate Authentication. You learned how to configure your JAX-RS applications to be secure using the metadata provided by the servlet and Java EE specifications. You also learned about OAuth as well as digital signatures and encryption of HTTP messages. [Chapter 29](#) contains some code you can use to test-drive many of the concepts in this chapter.